Texture synthesis using label assignment over a graph

Jack Caron, David Mould

Carleton University
Ottawa, Canada

Abstract

Partition of unity parametrics (*PUPs*) are a recent framework designed for geometric modeling. We propose employing PUPs for procedural texture synthesis, taking advantage of the framework's guarantees of high continuity and local support. Using PUPs to interpolate among data values distributed through the plane, the problem of texture synthesis can be approached from the perspective of point placement and attribute assignment; a graph over the points, such as the Delaunay triangulation, allows us to apply attributes or labels to the nodes in a structured way. We present several alternative mechanisms for point distribution and demonstrate how the system is able to produce a variety of distinct classes of texture, including analogs to cellular texture, Perlin noise, and progressively-variant textures. We further describe structured textures obtained by assigning label clusters using queries over the graph, such as breadth-first or depth-first traversal.

Keywords: Computer Graphics, I.3.3, Picture/Image Generation, Line and curve generation

1. Introduction

Procedural texture has been a central aspect of content creation since the earliest days of texture. Probably the single most widely used texture synthesis tool is Perlin noise [11, 12]; however, many other techniques exist, including spot noise [17], cellular texture [19], reaction-diffusion texture [15, 18], and more. Modern approaches tend to be spectral techniques, such as wavelet noise [4] and Gabor noise [8], or nonparametric [5, 6].

At the same time, methods for point distribution are an ongoing area of research and have become very sophisticated. Methods for Poisson disk distributions abound, and recent methods such as capacity-constrained distributions [1] and differential domain distributions [16] allow high-quality sampling of arbitrary fields.

Email addresses: jack.caron@gmail.com (Jack Caron), mould@scs.carleton.ca (David Mould)

Point distributions and texture synthesis are linked by the notion of *discrete element textures* [7], where discrete textons are distributed over the plane in some fashion (in the cited work, by nonparametric synthesis based on a sample distribution). In our case, we begin with a discrete arrangement obtained by direct procedural methods but then use an interpolation scheme to create a continuous-tone texture.

We previously described a proposal to employ partition of unity parametrics (PUPs) for texture synthesis [3]. The PUPs framework establishes local, high-continuity interpolation, allowing texture synthesis tasks to become point distribution tasks. We have direct spatial control over texture features by manipulating point positions and values, as in conventional geometric modeling activities. We can also make use of known techniques for automatically distributing points, whether in Poisson disk [9, 10] or other distributions. In our previous work, we relied heavily on *labels* associated with the points in order to coax a texture from a point distribution. In this extended version of the paper, we explore label assignment using operations over the graph of a point distribution, rather than using point placement directly. Given a set of points, we find the Delaunay triangulation, and use the information in the edges to decide on label assignments. For example, spotted textures akin to animal coat markings can be obtained by choosing several starting seeds and expanding clusters from the seeds along the graph edges. The clusters and background have different labels. Different patterns emerge depending on whether the cluster expansion uses breadth-first order, depth-first order, or some other order. Our experiments in this regard are recounted in Section 5.

PUPs-based textures are flexible and versatile, able to imitate Perlin noise, cellular textures, and other texture types. In addition, our technique is well-suited to nonstationary or progressive distributions, where the size and shape of texture elements vary spatially in controlled ways. We use the term *stationary* to refer to textures where the statistical properties do not vary over the image, and *nonstationary* when the properties vary with location.

This paper is organized as follows. First, we discuss existing work in texture synthesis and point distribution, and we give an overview of PUPs. Next, we describe our texture synthesis algorithm: given an input point set and associated values, we create the corresponding continuous-tone texture. We next show a suite of textures generated by variations in point distribution and attribute assignment, demonstrating the versatility of the approach. Lastly, we conclude with some summarizing remarks and suggestions for possible future research directions.

2. Related Work

The single most widely used texture synthesis primitive is Perlin noise [11, 12]. In Perlin noise, random orientations are set at lattice nodes and splines used to interpolate intermediate values. The ease of implementation, continuity, reproduceability, and ability to evaluate at arbitrary coordinates have been huge advantages for Perlin noise. Our proposed method is similar, but our use of the

PUPs framework allows arbitrary interpolation functions and does not require a regular lattice; as we will see, the freedom to create an arbitrary graph allows progressive textures which would be difficult to create using Perlin noise, as well as avoiding lattice artifacts.

For biologically inspired textures such as spots and stripes, reaction-diffusion textures [15, 18] were proposed. Such textures have extremely high quality and the methods can synthesize nonstationary patterns, but have generally been deemed slow and difficult to control. Cellular texture [19] is an alternative for some animal markings such as giraffe spots, but the range of possible textures is more limited. Progressively-variant textures [21] use example-based techniques on a texton map to create textures that change gradually from one type to another, with intriguing results.

In recent years most research on texture synthesis has been based on non-parametric synthesis [6], where pixels or patches are copied in a structured way from an example texture to a destination. Such methods have achieved high quality, but have the drawback that the exemplar texture is needed. Another strand of recent work has been spectrally controlled noise such as wavelet noise [4] or Gabor noise [8]. These are texture synthesis primitives and allow a variety of textures to be created, and the proposed work is in the same spirit.

The proposed texture synthesis process is based on *partition of unity parametrics*, a recent meta-modeling framework proposed by Runions and Samavati [13]. In the version of the PUPs framework used here, a triangle mesh stores data at nodes; the values are interpolated across triangle faces using projection and normalization. Originally proposed for geometry, here we adapt it to texture synthesis; since the PUPs structure is already well suited to texture synthesis, we concentrate on showing how point distributions and point labels can be assigned in such a way as to create different types of texture. We show textures resembling Perlin noise, Worley noise, and reaction-diffusion spots, plus demonstrate how to create progressively-variant textures with controllable feature size and nonstationary structures. We discuss PUPs, and our modifications thereto, in detail in section 3.

3. Algorithm

PUPs uses a set of control points $\mathcal{P} \subset \mathbb{R}^3$ to create an interpolated surface, which is a weighted sum of the control points. While in PUPs the points stored no information beyond their geometry, points can be assigned additional attributes which are then interpolated: in our case, we store an orientation, used to adjust the interpolation weight, and a value, typically a scalar intensity value or a vector color value. Since we are creating 2D textures, our control points lie in $\mathbb{R}^2$; without loss of generality, we created textures where $\mathcal{P} \in [-1, 1]^2$.

At a point in space u, the texture value is a weighted sum of the values at the nearby control points. The weight is constructed with reference to the edges of a graph connecting the control points; in our case, we used the Delaunay triangulation [2] (*see fig. 1*). A control point c is said to have a set of *axes*, α_i, where each axis is an edge originating at c and terminating at its neighbour.

The point u is projected onto each axis, and the projected length is used to determine an axial weight, A_{α_r}. The axial weights are given by cubic polynomial interpolants, decreasing from a maximum of 1 (exactly on the control point) to a minimum of zero (beyond the axis length); use of the cubic allows zero gradient at both ends, preventing discontinuities :

$$A_{\alpha_r}(p) = (1-p)^2(1+2p) \tag{1}$$

Finally, a given control point has a weight W_i (*see fig. 2a*), which is the product of the axial weights:

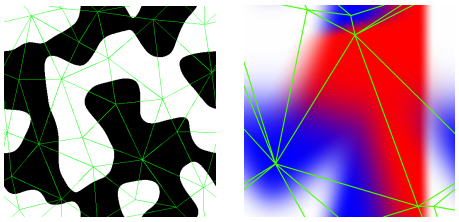
$$W_i(u) = \Pi_{r=0}^{|\alpha_i|} A_{\alpha_r}(proj_{\alpha_r}(u)). \tag{2}$$

Note that $|\alpha_i|$ denotes the number of axes for control point i and the projection operation $proj_{\alpha_r}$ returns a normalized projection in the range 0-1; the projected length is given as a fraction of the axis length, and clamped so that points beyond the axis end project to 1, while points behind the axis project to 0. It is also possible to control the smoothness of the intensity interpolation by changing the length of the axis (*see fig. 3*). Longer axes overlap and produce smoother outcomes, albeit with approximating interpolations rather than exact ones.

The texture value for a point in space is then the weighted sum of the attribute values at the nearby control points[1] $\mathcal{P}' \subseteq \mathcal{P}$, normalized by the weights:

$$T(u) = \frac{\sum_{i=0}^{|\mathcal{P}'|} T_i W_i(u)}{\sum_{i=0}^{|\mathcal{P}'|} W_i(u)} \tag{3}$$

where T_i is a value, a scalar intensity or a color, stored with point $p_i \in \mathcal{P}'$.

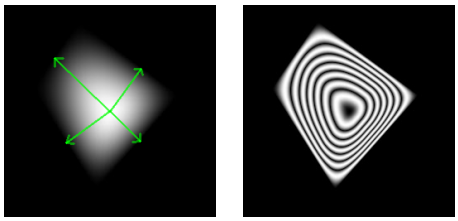


(a) (b)

Figure 1: Delaunay triangulations of $\mathcal{P}$.

We can further adjust the weights using an *orientation* attribute associated with each control point. Orientation is a unit vector in $\mathbb{R}^3$, used to shift the weight of its point towards a preferred direction; call the orientation v. Suppose we are computing the shifted weight for a point $p \in \mathbb{R}^2$ with respect to a control point at $c \in \mathbb{R}^2$. We first compute $\eta = (p-c) \cdot v/|p-c|$. Then we compute

[1] control points close enough to have a non-zero weight with u

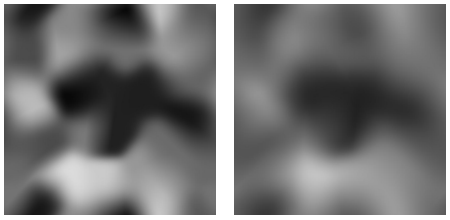


(a) (b)

Figure 2: (a) A visual representation of the weight of a point and its axis (*brighter means more weight*) and (b) its corresponding isovalues in its scalar field.

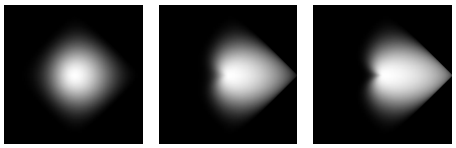
the shifted weight w', using equation 4, where the original weight $w \in [0, 1]$ is the weight of the point from equation 2; $\epsilon \in [0, 1]$ is a parameter governing the strength of the shift and $K = \epsilon\eta$ (by default, $\epsilon = 0.5$). In equation 4, the exponent is in $[0, 1]$ if $K \geq 0$, which increases the weight, and greater than 1 otherwise, which decreases the weight. The asymmetry of the exponent in the two parts of equation 4 allows more drastic changes on control points facing away from v. Figure 4 shows the effects of an orientation on the weight distribution.

$$w' = \begin{cases} w^{1.0-K} & \text{if } K \geq 0 \\ w^{\frac{1.0}{1.0+K}} & \text{otherwise} \end{cases} \tag{4}$$



(a) (b)

Figure 3: The same point set with (a) normal axis length and (b) axis twice as long.



(a) (b) (c)

Figure 4: The weight shifting as the orientation is (a) almost aligned with the z-axis; (c) almost aligned with the x-axis. Throughout, $\epsilon = 0.1$.

We can adapt the intensity interpolation to produce basic cellular textures, similar to those of Worley [19]. Each control point is given a *label* attribute; pixels are given the label of the control point with the highest weight. Because

of the way weights are computed, the resulting cells approximate but are not exactly the same as Voronoi cells; in particular, they are not bounded by straight edges. We consider the appearance of the cells to be more organic and lively than Voronoi cells. Using orientations, we are able to curve the cell boundaries even further; this is illustrated in Figure 5.

(a) (b) (c)

Figure 5: Effect of different orientations on the same point set; (a) all orientations aligned with the z-axis, (b) all orientations pointing to the right, and (c) all orientations pointing away from the center.

4. Texture Variations

In this section, we present algorithms to distribute control points and assign their attributes so as to create a wide variety of textures.

4.1. Noise

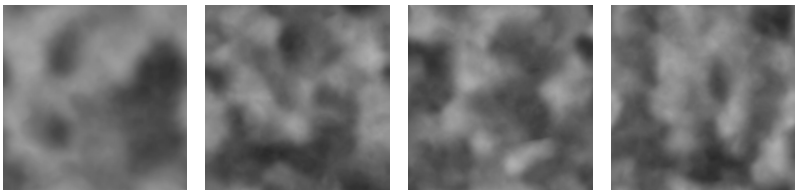
Using the basic interpolation scheme over random data, we can create a smooth scalar field similar to Perlin noise. The details of the process are as follows. First, we place points over the plane in a Poisson disc distribution. Then, we randomly assign an intensity value to each point. We create a graph over the distribution by finding the Delaunay triangulation. Then, we construct the noise field by taking the PUPs interpolation as described in the previous section. Four octaves of texture akin to Perlin noise are shown in Figure 6.

Multiscale Perlin noise can be created by summing octaves of with different point densities, and examples are shown in Figure 6. These results use the equation

$$T(x) = \sum_{i=0}^{i=3} P_i(x) 2^{-i} \tag{5}$$

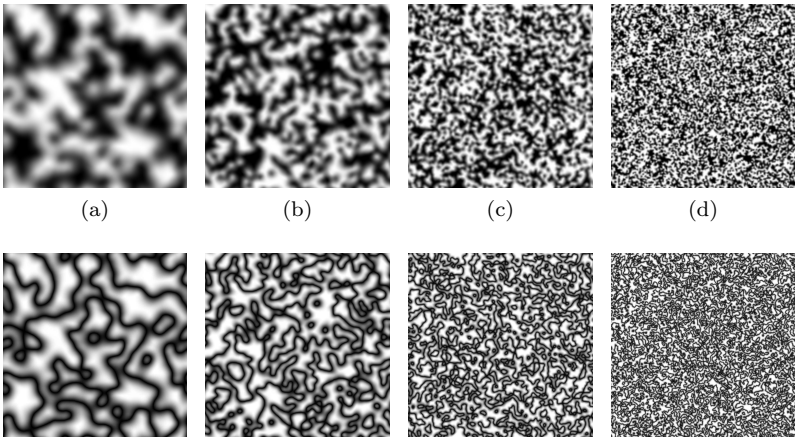
where P_i are Perlin-like textures with Poisson disk distributions of control points; the minimum interpoint distance is 0.1×2^{-i}. Note that we used separate textures for the different octaves, but because our textures are tileable, we could have used a single texture and changed the scale.

We can also reproduce Perlin turbulence [11], applying the folding transform $T'(x) = 2|T(x) - 0.5|$ to each octave; intensity is in the range 0 to 1. The foldings of the individual octaves are shown in Figure 7 and the resulting turbulence texture appears in Figure 8. Unlike regular Perlin turbulence, there are no lattice artifacts: the Poisson disc control point distribution frees us from any such defects.



(a) (b) (c) (d)

Figure 6: Multiscale Perlin noise textures (four octaves).



(a) (b) (c) (d)

(e) (f) (g) (h)

Figure 7: Multiple scales of Perlin-noise type textures, before and after folding.

4.2. Labels

We previously discussed how to create cellular textures by organizing points into groups with common labels. Here, we demonstrate some of the different textures that can be achieved with this process.

Figure 9a shows a simple Voronoi-like cellular texture, obtained by assigning each point to a separate group. It is not identical to the Voronoi diagram; the use of orientation in the weight calculation disturbs the Voronoi boundaries. More interesting possibilities emerge when there are multiple points per group: Figure 9b shows a texture created by grouping points into clusters of three or four points. In this case, the regions are even further from Voronoi regions.

Multiscale cellular textures are another straightforward possibility. Two examples are shown in Figure 10. To obtain these, we created 4 layers of cellular textures of increasing density and summed them; higher densities have lower amplitude, as in multiscale Perlin noise.

An entirely different type of structure can be created by assigning groups binary labels. In Figure 11 we see a texture resembling a stereotypical cow pattern, obtained by randomly assigning labels to points. The simplicity and unpredictability of this binary texture give it a distinctly organic character.

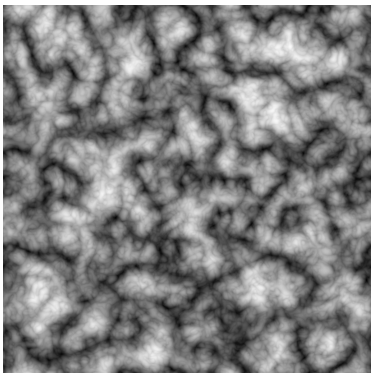
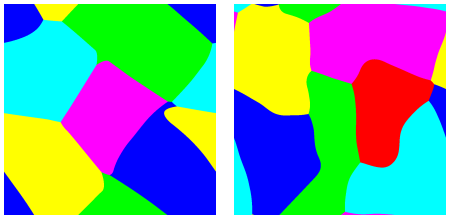


Figure 8: Perlin turbulence: sum of folded textures from fig. 7

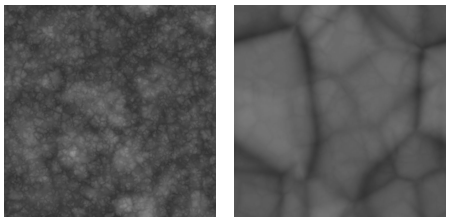


(a) (b)

Figure 9: Cells made of (a) one point per cell and (b) a few points per cell.

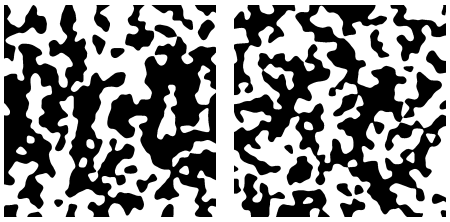
A more elaborate example with two groups is called the *space giraffe*, following the reaction-diffusion texture by Witkin and Kass [18]. The principal objective in crafting this texture was to control the irregular shape of the boundaries of the cells; a secondary objective was to prevent two cells from merging. The space giraffe texture is shown in Figure 12; we next describe the process used to create it.

We distribute cell centres with a Poisson disk distribution. For each centre, we then find its nearest neighbour; the cell's radius r is taken to be half the distance to the nearest neighbour. We then add further points surrounding the



(a) (b)

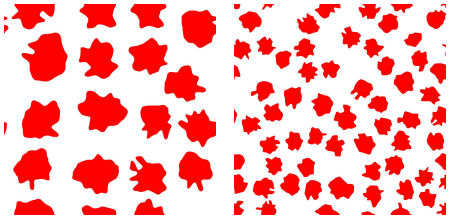
Figure 10: Cellular texture results.



(a) (b)

Figure 11: Two "cow texture" examples: Poisson disc distribution of control points with random assignment of binary labels.

cell centre, as follows: choose a random number n, say between 10 and 20, and place pairs of points at angular intervals of $2\pi/n$. The points are placed at radii randomly chosen from the range 0 to r. At each angle, there is an inner point (smaller radius) and an outer point (larger radius); assign to the inner point the spot label and to the outer point the background label. Create a new triangulation on this new point set and apply the cellular texture generation process: the result is the space giraffe pattern, examples of which are shown in Figure 12.



(a) (b)

Figure 12: Space giraffe patterns.

More elaborate structures, and in particular nonstationary structures, can be obtained by using a reference image to decide group membership. In this variant, we distribute points using a Poisson disk distribution and for each of these points, we check the intensity of the corresponding pixel in the image. We select a random value, and if that value is less than the reference intensity, we assign that point to the white group, otherwise to the black group. Figure 13 shows images created with the approach; with this reference image, the only gray regions are near the region boundaries, so we get a few variations on the input texture with different feature sizes. Note that for this process to be effective, the reference image must include a range of intensity levels.

In another variant, we propose to use an iterative version of the previous process. The only difference is that we use the intensity interpolation to create the textures, ensuring that there are regions of intermediate intensity near region boundaries. At each iteration, we double the point density, ultimately creating a fractal boundary layer; four iterations are shown in Figure 14. Restricting the

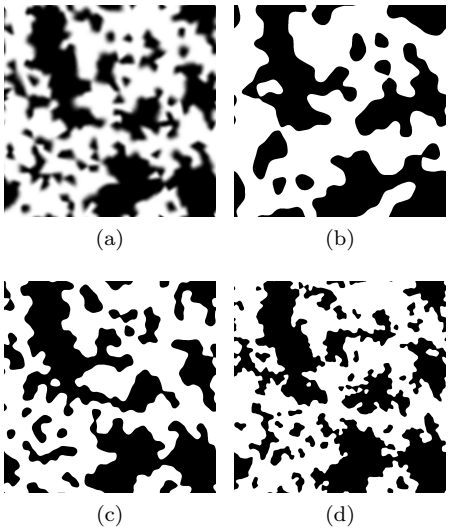


Figure 13: Labels governed by reference image. (a) Input image; (b-d) results with different densities of points.

regions where control point labels are chosen randomly is a powerful technique, producing results that would not be straightforward to make using other existing texture synthesis techniques. This approach could be used to produce Masai giraffe patterns such as those seen in Figure 18.

4.3. Progression

One of the main strengths of our method is its ability to use nonstationary spatial distributions of points or spatially aware label assignments to produce progressively-varying textures in a systematic way. The simplest example is that of Figure 15. This image shows two examples of progressive cow textures, where the point density becomes greater as you move to the right. The same random binary labeling strategy is employed. The resulting texture shows an obvious but continuous decrease in feature size as the point density increases.

A more elaborate example appears in Figure 16, where again the point density increases left to right. Here, we have used the space giraffe process on an initial distribution. The method is robust enough to produce irregular spots smoothly varying in size.

Changing the point distribution is one of the two mechanisms for synthesizing nonstationary textures. Even with a stationary distribution of control points, we can create nonstationary textures by applying labels using logic that depends on spatial location. For example, we can create a texture ramp by assigning black or white labels with a probability that depends on the point's x-coordinate. See Figure 17 for an example. Of course, other distributions are also possible: this example only hints at the range of possibilities.

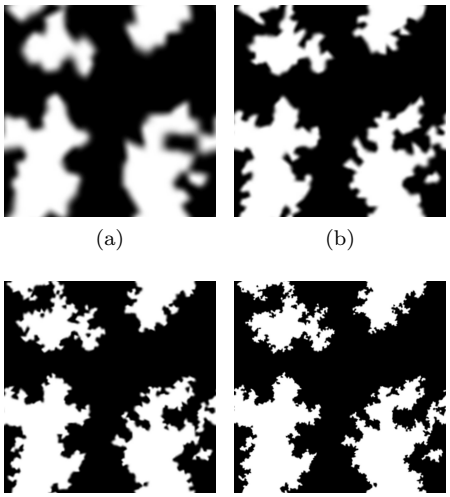


Figure 14: Iterative process to assign points to groups.

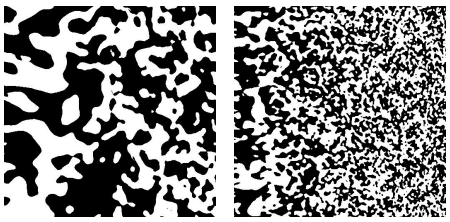


Figure 15: Progressive textures.

5. Reasoning over graphs

The following results exploit the structure of the graph to assign labels to the nodes. By explicitly considering the graph structure, we can enforce larger-scale structure into our textures. Images are an alternative mechanism, as seen in the previous section, but we also wanted to present purely algorithmic methods that require only abstract parameters as input.

The most basic feature of graphs is the edges that form connections between nodes. Every point connected to point p is said to be at 1-hop from p. If there are 2-hops between points p and q, it means the two points lack a direct connection, but there exists a third point r at 1-hop from both p and q. These hops represent a path between the original pair of points. We will use the notion of *hop* in the following subsections.

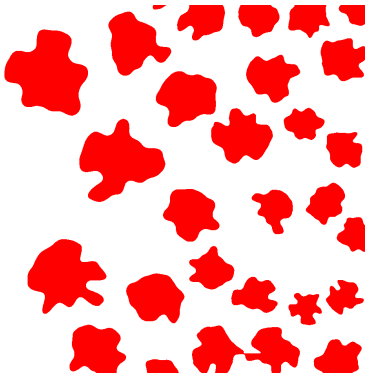
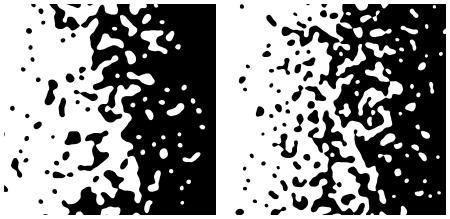


Figure 16: A progressive space giraffe.



(a) (b)

Figure 17: Points assigned to groups according to their location.

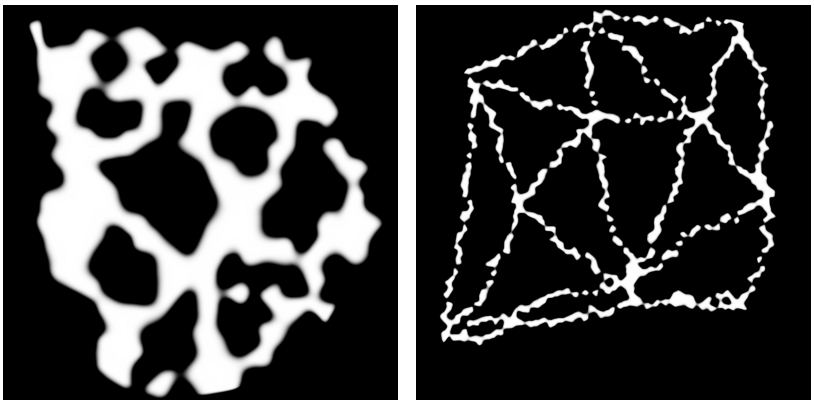
5.1. Clustering

In this section, we demonstrate some algorithms that use the graph structure to build clusters, a texture type that includes giraffe and leopard spots, markings on amphibians, and other natural patterns. Two examples of such patterns are shown in Figure 18. Initially, we sought simply to exploit the connectivity of the graph to create clusters. To begin with, we distribute points uniformly and assign all of them to the black group. Then some number of points, a proportion selected by the user, are reassigned to the white group. We compute the Delaunay triangulation only for the white points, $\mathcal{T}_w$; we will use this to enforce connectivity among all white points. Subsequently, for each edge in $\mathcal{T}_w$, we find a short path between its endpoints in the original point distribution and reassign every point on the path to the white group.

To visually have the impression that the result is indeed connected (*no isolated white point*), the initial size of the white group must be quite large; empirically, we found a proportion around 40% to be suitable, as is used in Figure 19a. With an initial white group too small, we end up with a disconnected, excessively structured set of paths, as seen in Figure 19b. A stronger visual impression of connectivity amongst white points can be achieved by extending the white labels: following the path creation, we assigned the 1-hop of all white points to also be in the white group. This was used to generate the textures

Figure 18: Textured animal coats with irregular clusters.

in fig. 20. But even if the white group is strongly connected, due to the way the pixel intensities are computed, we can still have isolated points; examples appear in fig. 20a.

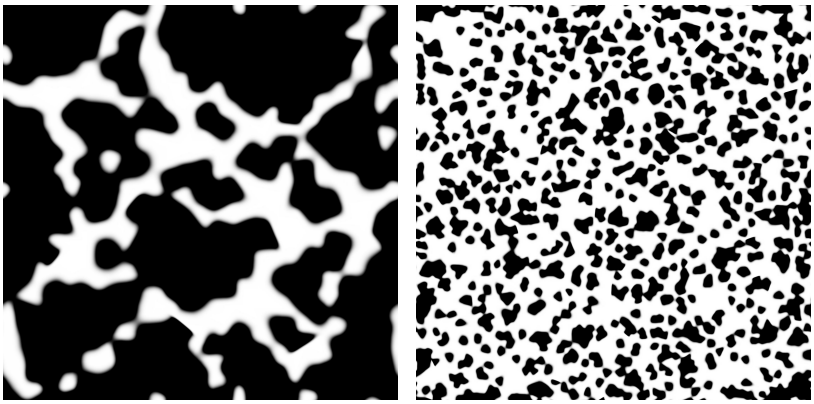


(a) (b)

Figure 19: Strongly connected, (a) the initial white group has a proper size, (b) the initial white group is too small.

This leads to an important observation about the behavior of PUPs for textures. Connections in the subgraph do not necessarily imply connected clusters in the output texture: i.e., when two nodes linked by an edge share a label, we cannot conclude that there will be a contiguous set of pixels with the corresponding color. (For an arbitrary graph we would not expect this to be the case, but it is not true even when the graph is the Delaunay triangulation.)

One possible solution might be to put a limit on the length of an edge directly connecting two points. Beyond this limit, the edge ceases to exist; the two points might still be mutually reachable by a different series of shorter edges. With this approach, care must be taken to ensure that the overall connectivity of the graph is not compromised. A second approach could be to check every edge and add a middle point to long edges whose endpoints share the same label. An edge



(a) (b)

Figure 20: Strongly connected with 1-hop.

exceeding the threshold would be replaced with a point and the triangulation recalculated. Note that these solutions privilege one label over the other; we can enforce connectivity of one color in this way, but not more.

5.2. Clustering from graph traversal

Here we suggest a general clustering approach based on graph traversal. The basic idea is to initiate a cluster at some arbitrary location, traverse the nearby nodes of the graph in some order to obtain the cluster, and repeat until no nodes remain unallocated to any cluster. Study of graphs has produced numerous algorithms for graph traversal: various techniques exist to visit all points or edges. Perhaps the two most common methods are breadth-first traversal and depth-first traversal.

To create textures using these traversal methods, we perform the following operations. We first distribute points in a Poisson disk distribution and compute the triangulation. Then, we select randomly a point c and a number k for the cluster size, say 20. Starting at c, using either breadth-first or depth-first, the exploration consists of selecting the next unlabelled point q, assign it in the black group and add it to the visiting list (*according to the exploration method*). After exploring k points, we enumerate the unlabelled neighbours of the cluster and assign them to the white group. We repeat this process until no points remain unlabelled. A step-by-step description of the process is shown in Figure 21.

Figure 22 shows examples of clustering resulting from breadth-first traversal. Each starting point anchors a cluster, and the clusters are reasonably compact around the starting point. Sometimes the textures are quite close to those made with the customized space-giraffe approach, shown in figure 12; figure 22d shows a very similar example. This highlights the effectiveness of using reasoning over a graph rather than attempting to achieve effects with direct point placement.

Results from our implementation of depth-first traversal are shown in figure 23. Our objective was to create long thin clusters; sometimes, long thin

Parameters:
cluster size k (number of nodes)
Note, a node p has property l, its label.

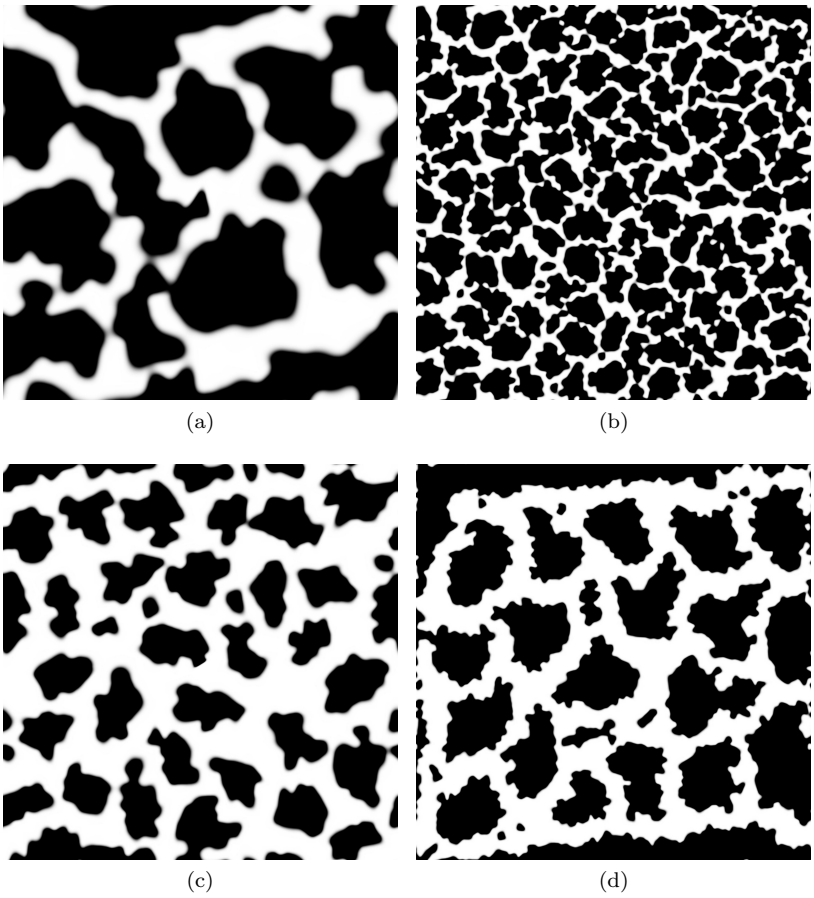
1. Distribute points p over plane, yielding point set P.
2. Compute Delaunay triangulation over P.
3. For all $p \in P, p.l \leftarrow 0$.
4. While $\exists c \in P$ st $c.l = 0$, create new cluster.
5. [Create new cluster]
5a. Choose cluster centre $c \in P$ st $c.l = 0$.
5b. Current cluster (set of nodes) contains only centre: $C \leftarrow \{c\}$.
5c. Set order of nodes relative to c based on traversal type (e.g., BFS).
5d. Grow cluster: for $i = 1$ to $k - 1$ do
5d-i. find next node in order, say q, st $q.l = 0$
5d-ii. $q.l \leftarrow$ black
5d-iii. Add q to cluster: $C \leftarrow \{C, q\}$.
5d-iv. Note, must terminate loop if no unlabeled neighbours exist.
5e. For all nodes s adjacent to C, $s.l \leftarrow$ white.
6. Done: all nodes now have a label.

Figure 21: Process for basic clustering algorithm.

clusters do appear, but many of the clusters cannot be so described. The process works as follows. From our starting point, we impose an order over the neighbouring points, visiting them in turn. The visited point is marked as belonging to the cluster. Then, recursively, we visit each of the unmarked neighbours of the most recently opened point. We used the CGAL library [14] to create and manipulate $\mathcal{T}$; the neighbours of a point are listed in counterclockwise order, starting with an arbitrary neighbour. Note that we do not exclude any unmarked points from being visited; this means that sometimes the search does not expand outward quickly, when points from the 1-ring of the previous point are visited rather than more distant points. Nonetheless, this created interesting results where a diversity of cluster shapes coexist within the same texture. Examples are shown in Figure 23; clusters are shown in black. Notice the peculiar clusters consisting of larger clumps linked by irregular narrow structures.

5.3. Round-robin cluster expansion

We here present *round-robin* clustering, where clusters take turns claiming adjacent points. The structures arising from round-robin clustering are organized but irregular, with both compact and non-compact elements. We make use of both the information in the graph and the spatial positions of the points in order to achieve our effects. Round-robin clustering is in some sense the opposite of the traversal-based clustering approaches: while traversal-based clustering



(a) (b) (c) (d)

Figure 22: Breadth first visit.

adds clusters of known size one by one, round-robin clustering starts with a known cluster count and expands the clusters in turn one node at a time.

More formally, the method operates as follows. First, we randomly select n centres from the points in the graph. Clusters then extend outward from the centres, as follows. One after another, each centre examines its unclaimed boundary points, selects the one with Euclidean distance closest to the centre, and claims it. A centre with no unclaimed boundary points must halt its expansion. This process is repeated until no points remain unclaimed, i.e., all points are associated with some centre. Detailed instructions for the process are given in Figure 25.

Finally we convert the labeling to a binary texture: if a point's 1-hop points all have the same owner, we assign this point to the black group, otherwise it goes in the white one. Figure 24 shows textures resulting from this algorithm. Optionally, to have thicker borders, if a point is assigned to the white group, its

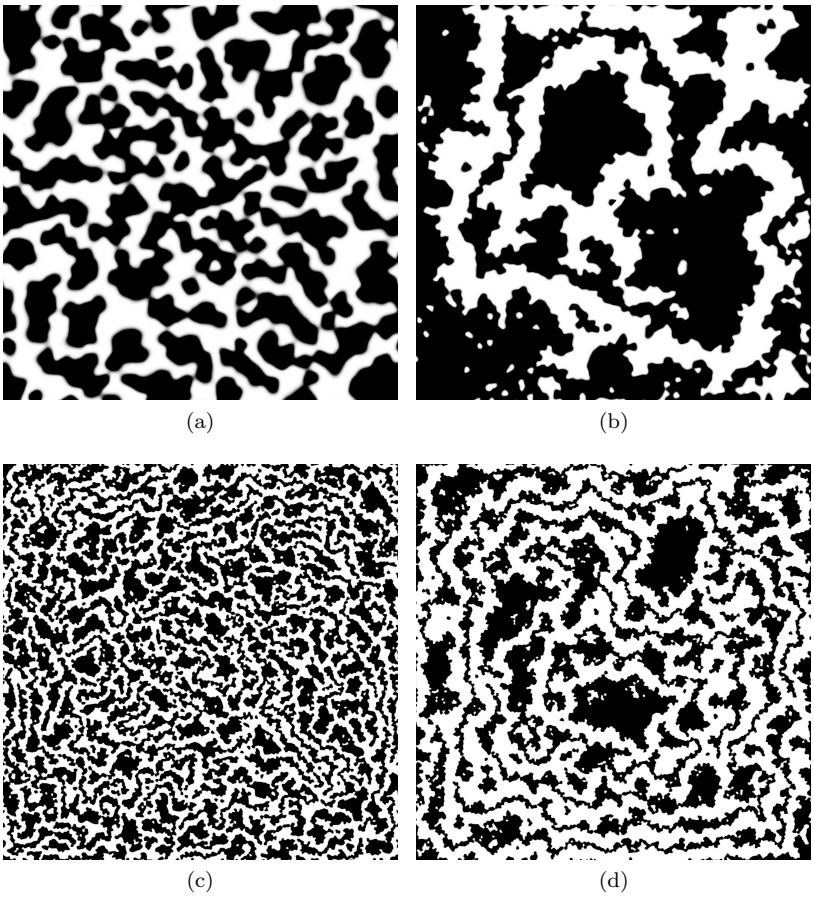


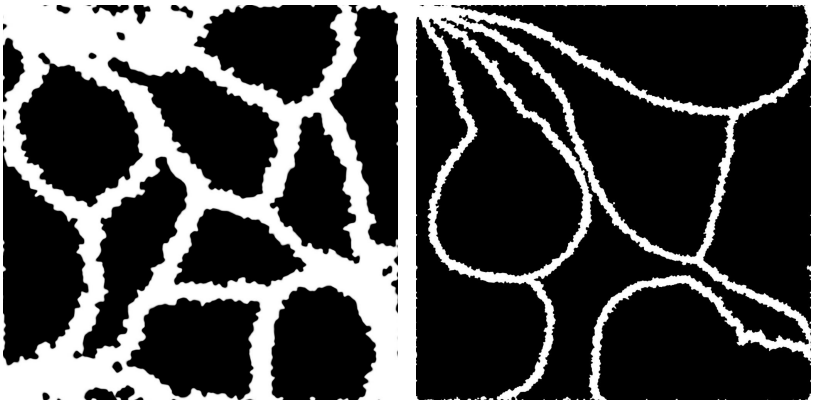
Figure 23: Depth first traversal.

1-hop is also assigned to the white group; figure 24a gives an example.

Notice that this process results in tessellations quite distinct from the Voronoi diagram. Voronoi diagrams result from uniform anisotropic expansion. In this algorithm, clusters will begin to expand unequally once their growth in a particular direction becomes restricted: each cluster takes its turn, and while a cluster with no neighbours will indeed grow at an equal rate in all directions, a cluster blocked in one direction will grow with greater speed in other directions. Figure 24b shows the consequences, with highly non-convex regions resulting from applying the algorithm.

6. Discussion

The images in the previous sections demonstrate some of the textures that can be generated using this method. We demonstrated analogs to Perlin noise,



(a) (b)

Figure 24: Cell structures from round-robin clustering.

Worley noise, some biological structures, and some multiscale and progressive structures. We also showed elaborate clusters that are unlike those produced by previously existing methods.The versatility of the method is one of its chief strengths.

The PUPs framework allows arbitrary control net topology while maintaining local support of feature interpolation. We are able to exploit this in making progressively variant textures such as the cow textures in Figure 11, where the feature density varies spatially. An alternative approach for building progressive textures is to vary the attributes in a systematic way, e.g., as seen in Figure 15. The control points provide discrete locations where the attributes are set, and the PUPs interpolation propagates the values to the rest of the plane.

Figure 14 shows a process where the distribution of intensities on the control points are distributed according to the intensity in the previous image at the same location. At each iteration, more points are used (in our case, we doubled the number of points used in the previous image) allowing the creation of a sharper and more detailed image each time.

We can use the control net more directly to organize our attribute assignments. For example, Figure 22 shows a texture created by propagating labels across graph edges. We have explored cluster propagation using breadth-first and depth-first traversal, demonstrating structures similar to some animal coat markings, and we also suggested round-robin clustering, capable of generating unusual patterns not previously seen. The cluster propagation methods, including round-robin clustering, are examples of what we believe to be a broad class of label propagation algorithms: the programmer-designer needs to specify rules for adding nodes to clusters and different patterns can arise. The use of the graph for label propagation is independent from our previous suggestions of changing node density locally or using an input image to guide label assignment; we have explored it only lightly in this paper and believe that logic over the graph is a powerful source of structure for procedural texture synthesis.

Parameters:
cluster count n (number of separate clusters)
Note, a node p has property l, its label.

1. Distribute points p over plane, yielding point set P.
2. Compute Delaunay triangulation over P.
3. For all $p \in P, p.l \leftarrow 0$.
4. for $i = 1$ to n:
4a. [Initialize cluster]
4b. choose random cluster centre $c \in P$.
4c. store centre in array: $c[i] \leftarrow i$.
4d. $c[i].l \leftarrow i$.
4e. Set ith cluster set: $C[i] = \{c\}$.
5. $i \leftarrow 0$ [counter for current cluster]
5. while $\exists p \in P$ st $p.l = 0$
5a. [increment next cluster]
5b. $i \leftarrow ((i + 1)\%n) + 1$ [values from 1 to n]
5c. Find S: all s st $s.l = 0, s$ adjacent to some $q \in C[i]$.
5d. If $|S| > 0$: [only proceed if there is at least one unlabeled neighbour]
5d-i. Choose $s \in S$ minimizing $||s - c[i]||$.
5d-ii. $s.l \leftarrow i$.
5d-iii. $C[i] \leftarrow \{C[i], s\}$.
6. Done: all nodes now have a nonzero label.

Figure 25: Process for round-robin clustering algorithm.

Overall, the approach we presented still has some drawbacks. We have not much investigated color; while color can be interpolated using the same process we used to interpolate intensity or group membership, color assignment is a more challenging problem. In our current prototype implementation, optimized for ease of experimentation rather than for speed, rendering is slow: for a typical case, a 1000×1000 texture with 700 control points, rendering requires approximately 30 seconds on a Quad-Core 1.3 MHz computer with 8 GB RAM.

We have cited the control net as an advantage of our method, but it has some disadvantages also. We used the Delaunay triangulation to provide our graph; small changes in the position of points can cause sudden changes in network topology, with correspondingly sudden changes in the texture, albeit only local. For static textures this is not an issue, but for manual editing of textures, or for animated textures, this would be a concern. Another drawback is the need to match the control net resolution to the feature resolution; arbitrary topology and local support allow us to do the matching adaptively, but very high-frequency features are still costly. Exploring different weight functions – for example, functions with oscillatory behaviour – might reveal a way to break that limitation.

7. Conclusion and Future Work

In this paper we described an adaptation of the "partition of unity parametrics" metamodelling framework for texture synthesis. We showed how to create textures reminiscent of classic textures such as Perlin noise and cellular texture, as well as novel textures including intricate patterns with multiscale boundaries. Because the PUPs interpolation method can propagate attribute values to the space between the control points, we can concentrate on procedurally distributing control points (for example, according to a Poisson distribution) and on assigning attributes to the control points (for example, randomly, according to spatial location or graph connectivity, or according to a reference image).

The method is able to generate stationary textures, but is not restricted to doing so. Progressive textures, where the characteristics vary smoothly from one image location to another, are possible. Also, use of reference images allows us to create highly irregular textures; we also demonstrated an iterative point placement and attribute assignment process, producing fractal boundaries reminiscent of coastlines. The ability to mix large-scale and small-scale features is extremely powerful.

Opportunities for future work are numerous. The rendering speed can be dramatically improved with a GPU implementation. Further investigation along the lines we have already begun will reveal more types of patterns to be described procedurally. We are interested in trying other triangulations or even non-planar control net topologies, such as the Yao graph [20], and in performing direct synthesis over polygonal meshes.

Animated textures are a direction to which this method seems especially suited. The motion of the texture can be governed by the motion of the under-

lying discrete structure. The main difficulty here lies in smoothing out sudden changes in the texture arising from sudden changes in the control net topology.

Finally, we are optimistic about the usability of the approach for manual texture creation and editing. Building a tool for editing PUPs-based texture is another area of future work.

Acknowledgements

We would like to thank Adam Runions and Faramarz Samavati for helpful discussions about PUPs. Financial support for this work was provided by NSERC, GRAND, and Carleton University. Photos of animal markings were obtained from Flickr; thanks to Flickr users redteam (giraffe) and ucumari (frog).

References

[1] Balzer, M., Schlömer, T., Deussen, O., 2009. Capacity-constrained point distributions: a variant of Lloyd's method. ACM Trans. Graph. 28, 86:1–86:8. URL: http://doi.acm.org/10.1145/1531326.1531392, doi:10.1145/1531326.1531392.

[2] Berg, M.d., Cheong, O., Kreveld, M.v., Overmars, M., 2008. Computational Geometry: Algorithms and Applications. 3rd ed. ed., Springer-Verlag TELOS, Santa Clara, CA, USA.

[3] Caron, J., Mould, D., 2013. Partition of unity parametrics for texture synthesis, in: Proc. Graphics Interface, pp. 173–179.

[4] Cook, R.L., DeRose, T., 2005. Wavelet noise. ACM Trans. Graph. 24, 803–811. URL: http://doi.acm.org/10.1145/1073204.1073264, doi:10.1145/1073204.1073264.

[5] Efros, A.A., Freeman, W.T., 2001. Image Quilting for Texture Synthesis and Transfer. Proceedings of SIGGRAPH 2001 , 341–346.

[6] Efros, A.A., Leung, T.K., 1999. Texture Synthesis by Non-Parametric Sampling, in: Proceedings of the International Conference on Computer Vision-Volume 2 - Volume 2, IEEE Computer Society, Washington, DC, USA. pp. 1033–. URL: http://dl.acm.org/citation.cfm?id=850924.851569.

[7] Ijiri, T., Mech, R., Igarashi, T., Miller, G.S.P., 2008. An Example-based Procedural System for Element Arrangement. Comput. Graph. Forum 27, 429–436.

[8] Lagae, A., Lefebvre, S., Drettakis, G., Dutré, P., 2009. Procedural noise using sparse Gabor convolution. ACM Trans. Graph. 28, 54:1–54:10. URL: http://doi.acm.org/10.1145/1531326.1531360, doi:10.1145/1531326.1531360.

[9] McCool, M., Fiume, E., 1992. Hierarchical Poisson disk sampling distributions, in: Proceedings of the conference on Graphics interface '92, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. pp. 94–105. URL: http://dl.acm.org/citation.cfm?id=155294.155306.

[10] Mitchell, D.P., 1987. Generating antialiased images at low sampling densities. SIGGRAPH Comput. Graph. 21, 65–72. URL: http://doi.acm.org/10.1145/37402.37410, doi:10.1145/37402.37410.

[11] Perlin, K., 1985. An image synthesizer. SIGGRAPH Comput. Graph. 19, 287–296. URL: http://doi.acm.org/10.1145/325165.325247, doi:10.1145/325165.325247.

[12] Perlin, K., 2002. Improving noise. ACM Trans. Graph. 21, 681–682. URL: http://doi.acm.org/10.1145/566654.566636, doi:10.1145/566654.566636.

[13] Runions, A., Samavati, F.F., 2011. Partition of unity parametrics: a framework for meta-modeling. The Visual Computer 27, 495–505.

[14] The CGAL Project, 2013. CGAL User and Reference Manual. 4.2 ed., CGAL Editorial Board. Http://www.cgal.org/Manual/4.2/doc_html/cgal_manual/packages.html.

[15] Turk, G., 1991. Generating textures on arbitrary surfaces using reaction-diffusion, in: Proceedings of the 18th annual conference on Computer graphics and interactive techniques, ACM, New York, NY, USA. pp. 289–298. URL: http://doi.acm.org/10.1145/122718.122749, doi:10.1145/122718.122749.

[16] Wei, L.Y., Wang, R., 2011. Differential domain analysis for non-uniform sampling. ACM Trans. Graph. 30, 50:1–50:10. URL: http://doi.acm.org/10.1145/2010324.1964945, doi:10.1145/2010324.1964945.

[17] van Wijk, J.J., 1991. Spot noise texture synthesis for data visualization, in: Proceedings of the 18th annual conference on Computer graphics and interactive techniques, ACM, New York, NY, USA. pp. 309–318. URL: http://doi.acm.org/10.1145/122718.122751, doi:10.1145/122718.122751.

[18] Witkin, A., Kass, M., 1991. Reaction-diffusion textures. SIGGRAPH Comput. Graph. 25, 299–308. URL: http://doi.acm.org/10.1145/127719.122750, doi:10.1145/127719.122750.

[19] Worley, S., 1996. A cellular texture basis function, in: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, ACM, New York, NY, USA. pp.

291–294. URL: http://doi.acm.org/10.1145/237170.237267, doi:10.1145/237170.237267.

[20] Yao, A.C.C., 1982. On Constructing Minimum Spanning Trees in k-Dimensional Spaces and Related Problems. SIAM J. Comput. 11, 721–736.

[21] Zhang, J., Zhou, K., Velho, L., Guo, B., Shum, H.Y., 2003. Synthesis of progressively-variant textures on arbitrary surfaces. ACM Trans. Graph. 22, 295–302. URL: http://doi.acm.org/10.1145/882262.882266, doi:10.1145/882262.882266.