Texture Synthesis over Arbitrary Manifold Surfaces

Li-Yi Wei Marc Levoy

Stanford University *

Abstract

Algorithms exist for synthesizing a wide variety of textures over rectangular domains. However, it remains difficult to synthesize general textures over arbitrary manifold surfaces. In this paper, we present a solution to this problem for surfaces defined by dense polygon meshes. Our solution extends Wei and Levoy's texture synthesis method [25] by generalizing their definition of search neighborhoods. For each mesh vertex, we establish a local parameterization surrounding the vertex, use this parameterization to create a small rectangular neighborhood with the vertex at its center, and search a sample texture for similar neighborhoods. Our algorithm requires as input only a sample texture and a target model. Notably, it does not require specification of a global tangent vector field; it computes one as it goes - either randomly or via a relaxation process. Despite this, the synthesized texture contains no discontinuities, exhibits low distortion, and is perceived to be similar to the sample texture. We demonstrate that our solution is robust and is applicable to a wide range of textures.

Keywords: Texture Synthesis, Texture Mapping, Curves & Surfaces

1 Introduction

Computer graphics applications often use textures to decorate virtual objects without modeling geometric details. These textures can be generated from sample images using texture synthesis algorithms. However, most existing texture synthesis algorithms are designed for rectangular domains and can not be easily extended to general surfaces. One solution is to paste textures onto such surfaces using texture mapping. However, because general surfaces lack a continuous parameterization, this type of texture mapping usually causes distortions or discontinuities. An alternative approach that minimizes distortion is to generate textures directly over the surface. However, since we can not apply traditional image processing operations to surfaces, most existing methods for surface texture synthesis work only for limited classes of textures.

This paper presents a method for synthesizing textures directly over 3D meshes. Given a texture sample and a mesh model, our algorithm first uniformly distributes the mesh vertices using Turk's method [23]. It then assigns texture colors to individual mesh vertices so that the appearance of the surface appears to be the same as the input texture (Figure 1). It does this using a non-trivial extension of Wei and Levoy's approach [25]. Specifically, given a sample texture image, their algorithm synthesizes a new texture pixel

(a) Texture (b) Model (c) Synthesis result

Figure 1: *Surface texture synthesis. Given a texture sample (a) and a model (b), we synthesize a similar texture directly over the model surface (c).*

by pixel in a scanline order. To determine the value of a particular output pixel, its spatial neighborhood is compared against all possible neighborhoods from the input image. The input pixel with the most similar neighborhood is then assigned to the output pixel. This neighborhood search process constitutes the core of [25] and is inspired by the pioneering work of Efros and Leung [6] and Popat and Picard [19]. The primary differences between [25] and [6, 19] are that [25] uses neighborhoods with fixed shapes and conducts the search deterministically; therefore it can be accelerated by tree-structured vector quantization.

Although [25] can synthesize a wide variety of textures, there are several difficulties in extending it to general meshes:

Connectivity Vertices on meshed surfaces are irregularly distributed, with varying inter-vertex distances and angles. As a result, the scanline order used in [25] cannot be applied.

Geometry Most surfaces are curved and cannot be flattened without cutting or distortion. This presents difficulties for defining the spatial neighborhoods that characterize textures.

Topology Because the surface of a general object cannot be mapped to a rectangle, it can not be parameterized using a rectangular grid. Most texture synthesis methods require the specification of a local texture orientation.

In this paper, we present two modifications of [25] to address those challenges. First, we relax that algorithm's scanline order, instead visiting vertices in random order, to allow texture synthesis over surfaces with arbitrary topology. Second, we replace the rectangular parameterization of the output domain that is implicit in [25] with tangent directions at each mesh vertex, coupled with a scale factor derived from the mesh vertex density. Based on this new parameterization we generalize the definition of search neighborhoods in [25] to meshes, and we show that this generalization works over a wide variety of textures. Specifically, for textures that are moderately isotropic, we use random tangent directions, and for anisotropic textures, we use tangent directions that are either user-specified or automatically assigned by our relaxation procedure.

The rest of the paper is organized as follows. In Section 2, we review previous work. In Section 3, we present the algorithm. In Section 4, we demonstrate synthesis results. In Section 5, we conclude the paper and discuss future work.

* Email: {liyiwei | levoy}@graphics.stanford.edu

2 Previous Work

Texture Synthesis: Recent statistical texture synthesis algorithms [11, 22, 4, 25, 6] have achieved success in modeling image textures. Since these algorithms rely on planar grids, it is not clear how they can be extended to arbitrary surfaces. A different class of methods generate textures through specialized procedures [5]. These techniques produce textures directly over 3D surfaces, so the texture distortion problem is largely eliminated. However, procedural synthesis is capable of modeling only a limited class of textures.

There have been several attempts to extend statistical texture synthesis to surfaces [7] or 3D volumes [8, 11]. Based on second-order statistics, [7] relates pairs of mesh vertices via their geodesic curves. However, second-order statistics are unable to capture significant structures that occur in may textures [22]. Volumetric synthesis [8, 11] avoids this texture distortion. However, these algorithms begin from multiple 2D textures and require consistent statistics over these multiple views; therefore they can model only textures without large-scale structures.

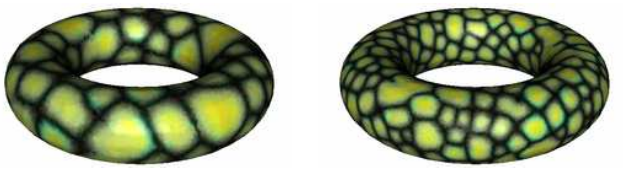
Texture Mapping: Another body of related work is texture mapping algorithms. However, globally consistent texture mapping [14] is difficult. Often, either distortions or discontinuities, or both, will be introduced. [17] addressed this problem by patching the object with continuously textured triangles. However, this approach works only for isotropic textures, and it requires careful preparation of input texture triangles obeying specific boundary conditions. In addition, since it employs relatively large triangles, the approach is less effective for texturing narrow features. Our algorithm performs moderately well on semi-anisotropic textures, and it does not require extensive preparation. Another method that has been suggested is to cover a model with irregular overlapping patches [20]. This approach works well for some but not all kinds of textures. Also, the discontinuity between adjacent texture instances are evident if the textured model is seen close up. The local parameterization method used in [20] inspired the parameterization of the algorithm presented here.

Mesh Signal Processing: In principle, we could directly generalize [25] for meshes if there existed a toolkit of general mesh signal processing operations. Unfortunately, despite promising recent efforts [10, 21], mesh signal processing still remains largely an open problem; [21] works only for spheres and [10] is designed for filtering geometries and functions over meshes, not for general mesh signal processing operations such as convolution.

3 Algorithm

Symbol	Meaning
I_a	Input texture image
I_s	Output texture image
M_s	Output textured mesh
G_a	Gaussian pyramid built from I_a
G_s	Gaussian pyramid built from I_s or M_s
p_i	An input pixel in I_a or G_a
p	An output pixel/vertex in I_s/G_s
$P_s(p)$	Flattened patches around p
$N(p)$	Neighborhood around the pixel p
$G(L)$	Lth level of pyramid G
$G(L,p)$	Pixel p at level $G(L)$
$\vec{s}, \vec{t}, \vec{n}$	Local texture coordinate system: texture right, texture up, and surface normal
{RxC,k}	neighborhood containing k levels, with sample density RxC pixels at the top level

Table 1: *Table of symbols*



(a) 24576 vertices (b) 73728 vertices

Figure 2: *The retiling vertex density determines the scale for texture synthesis. Textured torus with (a) 24576 vertices and (b) 73728 vertices.*

Our algorithm uses the same framework as [25]. To make the exposition clear, we first summarize that algorithm in Table 2. We then describe our extensions. The core of [25] uses spatial neighborhoods defined on rectangular grids to characterize image textures. In this paper, we generalize the definition of spatial neighborhood so that it can be used for producing textures over general surfaces. We parameterize mesh surfaces using local coordinate orientations defined for each mesh vertex and a scale factor derived from vertex density. We also change the codes for building/reconstructing mesh pyramids, as well as the order for traversing output pixels. For clarity, we mark a $*$ at the the beginning of each line in Table 2 that needs to be extended or replaced.

In the rest of this section, we present our extensions following the order in the pseudo-code in Table 2. For easy comparison we also summarize our new algorithm in Table 3.

```
function I_s ← ImageTextureSynthesis(I_a, I_s)
1*   InitializeColors(I_s);
2    G_a ← BuildImagePyramid(I_a);
3*   G_s ← BuildImagePyramid(I_s);
4    foreach level L from lower to higher resolutions of G_s
5*     loop through all pixels p of G_s(L) in scanline order
6        C ← FindBestMatch(G_a, G_s, L, p);
7        G_s(L, p) ← C;
8*   I_s ← ReconPyramid(G_s);
9    return I_s;

function C ← FindBestMatch(G_a, G_s, L, p)
10*  N_s ← BuildImageNeighborhood(G_s, L, p);
11   N_a^best ← null;   C ← null;
12   loop through all pixels p_i of G_a(L)
13     N_a ← BuildImageNeighborhood(G_a, L, p_i);
14     if Match(N_a, N_s) > Match(N_a^best, N_s)
15       N_a^best ← N_a;   C ← G_a(L, p_i);
16   return C;
```

Table 2: *Pseudocode of [25]. Lines marked with a $*$ need to be replaced or extended for synthesizing surface textures.*

3.1 Preprocessing

The preprocessing stage consists of building multiresolution pyramids and initializing output texture colors (Table 2, line 1 to 3, and Table 3, line 1 to 5). For texturing a surface we add two more steps to this stage: retiling meshes and assigning a local texture orientation. Let us consider each step in this stage.

In Table 2, an image pyramid is built for both the input and output texture image. In the present algorithm, we build the image pyramid G_a via standard image processing routines, as in [25]. However, for output mesh M_s, we construct the corresponding pyramid G_s using mesh simplification algorithms [23]. Note that at this stage G_s only contains a sequence of simplifications of the geometry of M_s; the vertex colors are not yet assigned.

After building the mesh pyramid G_s, we retile the surfaces on each level using Turk's algorithm [23]. This retiling serves two pur-

function $M_s \leftarrow$ SurfaceTextureSynthesis(I_a, M_s)

1 $G_a \leftarrow$ BuildImagePyramid(I_a);
2* $G_s \leftarrow$ BuildMeshPyramid(M_s);
3* RetileMeshes(G_s);
4* AssignTextureOrientation(G_s);
5* InitializeColor(G_s);
6 **foreach** level L from lower to higher resolutions of G_s
7* **loop** through all pixels p of $G_s(L)$ in random order
8 $C \leftarrow$ FindBestMatch(G_a, G_s, L, p);
9 $G_s(L, p) \leftarrow C$;
10* $M_s \leftarrow$ ReconMeshPyramid(G_s);
11 **return** M_s;

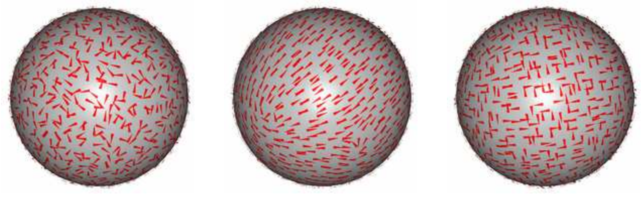
function $C \leftarrow$ FindBestMatch(G_a, G_s, L, p)

12* $N_s \leftarrow$ BuildMeshNeighborhood(G_s, L, p);
13 $N_a^{best} \leftarrow$ null; $C \leftarrow$ null;
14 **loop** through all pixels p_i of $G_a(L)$
15 $N_a \leftarrow$ BuildImageNeighborhood(G_a, L, p_i);
16 **if** Match(N_a, N_s) > Match(N_a^{best}, N_s)
17 $N_a^{best} \leftarrow N_a$; $C \leftarrow G_a(L, p_i)$;
18 **return** C;

function $N_s \leftarrow$ BuildMeshNeighborhood(G_s, L, p)

19* $P_s(p) \leftarrow$ FlattenLocalPatch(G_s, L, p, $\vec{s}$, $\vec{t}$, $\vec{n}$);
20* $N_s \leftarrow$ ResampleNeighborhood($P_s(p)$);
21 **return** N_s;

Table 3: *Pseudocode of our algorithm. Lines marked with a $*$ indicate our extensions from the algorithm in Table 2. Note that in our current implementation we only use Gaussian pyramids for meshes; therefore line 10 simply extracts the highest resolution from G_s.*



(a) (b) (c)

Figure 3: *Orienting textures via relaxation. The red arrows illustrate the $\vec{s}$ directions over the mesh vertices: (a) random (b) 2-way symmetry (c) 4-way symmetry.*

poses: 1) it uniformly distributes the mesh vertices, and 2) the retiling vertex density, a user-selectable parameter, determines the scale of the synthesized texture relative to the mesh geometry (Figure 2, see Section 3.3 for details). The retiling progresses from higher to lower resolutions, and we retile each lower resolution mesh with one quarter of the number of vertices of the immediate higher resolution so that the relative sample densities of adjacent pyramid levels relative to one another are compatible between image pyramid G_a and mesh pyramid G_s.

After retiling, we initialize colors of each level of G_s by assigning random colors from the corresponding level in G_a. This initialization method naturally equalizes the color histograms between G_a and G_s, thereby improving the resulting texture.

The next step is to assign a local coordinate frame for each vertex in the mesh pyramid. This coordinate frame, which determines the texture orientation, consists of three orthogonal axes $\vec{s}$ (texture right), $\vec{t}$ (texture up), and $\vec{n}$ (surface normal). These three axes are tacitly assumed to be $\vec{x}$, $\vec{y}$, $\vec{z}$ for planar image grids. For general surfaces it is usually impossible to assign a globally consistent local orientation (e.g. a sphere). In other words, singularities are unavoidable.

Our solution to this problem is to assign the $\vec{s}$ vectors randomly, at least for isotropic textures. One of the contributions of this paper is the recognition that, in the context of a texture synthesis algorithm that searches a texture sample for matching neighborhoods, rotating the $\vec{s}$ and $\vec{t}$ between the searches conducted at adjacent mesh vertices does not significantly degrade the quality of the match found as long as the input texture is reasonably isotropic. (Although isotropic textures are by definition rotationally invariant, this does not immediately imply that we can generate isotropic textures by matching neighborhoods in a rotationally invariant way.)

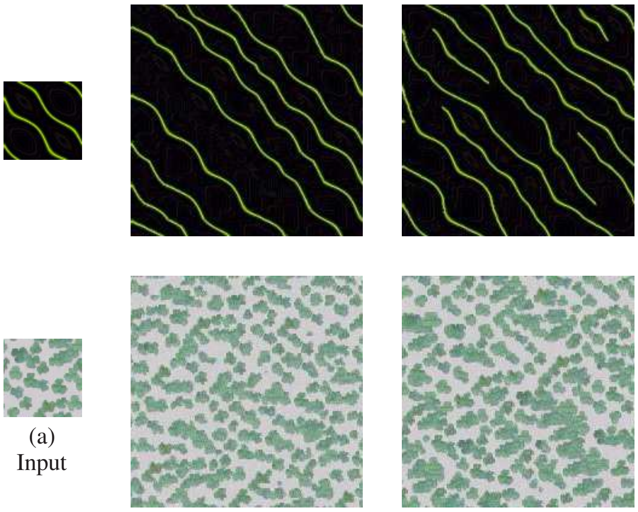
For anisotropic textures this solution does not work. Therefore, we either let the user specify the texture direction as in [20], or we automatically assign $\vec{s}$ and $\vec{t}$ using a relaxation procedure. The goal of this relaxation procedure is to determine the local texture orientation from the directionality of the input texture. That is, given an n-way symmetric texture, we orient $\vec{s}$ vectors so that to the extent possible, adjacent $\vec{s}$ vectors form angles of integer multiples of $\frac{360}{n}$ degrees. The relaxation algorithm begins by assigning random orientations for the lowest resolution level of G_s. It then proceeds from lower to higher resolutions of G_s, and at each resolution it first initializes $\vec{s}$ vectors by interpolating from the immediate lower resolution. Each $\vec{s}$ is then aligned, iteratively, with respect to its spatial neighbors (at the current and lower resolutions) so that the sum of individual mis-registration is minimized. The amount of mis-registration for each $\vec{s}$ at vertex p is calculated by the following error function:

$$E = \sum_{q \text{ near } p} \left| \phi_{s_{qp}} - \text{round}\left(\frac{\phi_{s_{qp}}}{\frac{360}{n}}\right) \times \frac{360}{n} \right|^2,$$

where n is the degree of symmetry of the input texture, and $\phi_{s_{qp}}$ is the angle between $\vec{s_p}$ ($\vec{s}$ of vertex p) and the projection of $\vec{s_q}$ on the local coordinate system of vertex p. The idea of using energy minimization for assigning local directions is not new. A similar function is used in [18], with the following differences to our approach: (1) we set $\vec{s}$ and $\vec{t}$ to be always orthogonal to each other, and (2) we use modular arithmetic in the function so that it favors adjacent $\vec{s}$ vectors forming angles that are multiples of $\frac{360}{n}$ degrees. Our approach is also similar to [12], but we use a slightly different functional, and we do not require the direction fields to align with the principle surface curvatures. Examples of orienting 2-way and 4-way symmetric textures (e.g. stripes and grid) are shown in Figure 3 (b) and (c).

3.2 Synthesis Order

The scanline synthesis order in Table 2 (line 5) cannot be directly applied to mesh pyramid G_s since its vertices do not have rectangular connectivity. One solution might be to use the two-pass algorithm for constrained synthesis [25], growing textures spirally outward from a seed point. However, there is no natural seed point for meshes of arbitrary topology. Surprisingly, we have found that our algorithm works even if we visit pixels of $G_s(L)$ in random order. Thus, we use a modified two-pass algorithm, as follows. During the first pass, we search the input texture using a neighborhood that contains only pixels from the lower resolution pyramid levels (except the lowest resolution where we randomly copy pixels from the input image). This pass uses the lower resolution information to "extrapolate" the higher resolution levels. In the second pass, we use a neighborhood containing pixels from both the current and lower resolution. In both passes, on each level, the neighborhoods used are symmetric (noncausal). We alternate these two passes for each level of the output pyramid, and within each pass we simply visit the vertices in a random order. In our experience this random order works as well as the spiral order used in [25], and it produces slightly worse textures than scanline order only for patterns with scanline dependencies. An example comparing different synthesis orders is shown in Figure 4.

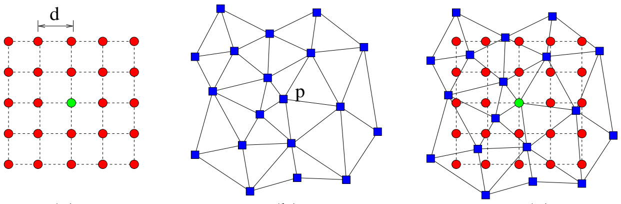


(a) Input

(b) Scanline order

(c) Random order

Figure 4: *Texture synthesis order. (a) Input textures (b) Results with scanline-order synthesis (c) Results with random-order synthesis. For textures without scanline dependencies, we have found that random-order works well.*



(a) (b) (c)

Figure 5: *Mesh neighborhood construction. (a) neighborhood template (b) flattened patch of the mesh (c) neighborhood template embedded in the flattened patch.*

3.3 Neighborhood Construction

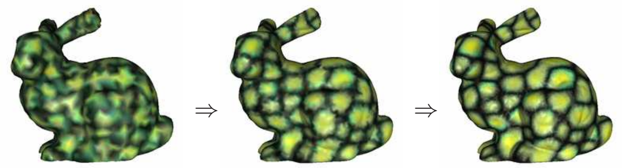
Table 2 characterizes textures using spatial neighborhoods (line 10 and 13). These neighborhoods are planar and coincident with the pyramid grids. For meshes, however, we have to generalize neighborhoods so that they are defined over general surfaces having irregular vertex positions.

We build mesh neighborhoods by flattening and resampling the mesh locally (Figure 5). To build the neighborhood around an output vertex p, we first select and flatten a set of nearby vertices, henceforth called a patch, so that they fully cover the given neighborhood template (Figure 5 (a,b)). We then resample the flattened patch (Figure 5 (c)) by interpolating the color of each neighborhood pixel (red circles) from the vertex colors of the patch triangle (blue squares) that contains that pixel. Before flattening, the neighborhood template is scaled with a constant $d = \sqrt{2 \times A}$, where $A =$ average triangle area of $G_s(L)$, so that the sampling density of the neighborhood and mesh vertices are roughly the same[1]. Leaving d much larger than $\sqrt{2 \times A}$ would either introduce aliasing during resampling or would waste mesh vertices by necessary filtering; if d were too small, the neighborhood would be poorly represented since most of its samples would come from the same triangle.

The method we use for flattening patches is taken from [20]. First, we orthographically project the triangles adjacent to p onto p's local texture coordinate system. Starting from these seed triangles, we grow the flattened patch by adding triangles one at a time until the neighborhood template is fully covered. Triangles are added in order of increasing distance from the seed triangles, and we determine the position of each newly added vertex using the heuristic in [15, Section 3.1.4]. Note that the flattening process can introduce flipped triangles. If this happens, we stop growing patches along the direction of flipping. This might in turn produce patches that only partially cover the neighborhood template. In this case, we assign a default color (the average of I_a) to the uncovered neighborhood pixels. Another solution might be to use smaller neighborhoods for highly curved areas. However, since a new neighborhood size would require a new VQ codebook [25], this implies building multiple codebooks for tree-structured VQ acceleration. Fortunately, since we only use small neighborhoods, flipping rarely happens.

We construct multiresolution neighborhoods in a similar fashion. For each vertex p of pyramid G_s, we first find the corresponding parent faces at lower resolution pyramid levels by intersecting the normal $\vec{n}$ of p with the coarse meshes. We project each parent face orthographically with respect to p's $\vec{s}, \vec{t}, \vec{n}$, and we grow a flattened patch from the parent face as in the single-resolution case. The collection of flattened patches $P_s(p)$ is then resampled to obtain the multiresolution neighborhood $N(p)$[2].

[1] We choose this formula so that if the mesh is a regular planar grid, the neighborhood will be scaled to align exactly with the grid vertices.



(a) 2186 vertices, 4368 faces

(b) 8715 vertices, 17426 faces

(c) 34938 vertices, 69868 faces

Figure 6: *Multi-resolution surface texture synthesis. The synthesis progresses from lower to higher resolutions, and information at lower resolution meshes is used to constrain the growth of textures at higher resolutions.*

4 Results

Our first example, illustrating the multiresolution synthesis pipeline, is shown in Figure 6. The synthesis progresses from lower to higher resolutions, and information at lower resolution meshes is used to constrain the growth of texture patterns at higher resolutions. All synthesis results shown in this paper are generated with 4-level Gaussian pyramids, with neighborhood sizes {1x1,1}, {3x3,2}, {5x5,2}, {7x7,2} (Table 1), respectively, from lower to higher resolutions.

Texture Orientation: Figure 7 demonstrates the performance of our algorithm on textures with varying amounts of anisotropy. The model we use, a sphere, is the simplest non-developable object that has no consistent texture parameterization. Despite this, many textures are sufficiently isotropic that they can be synthesized using random texture orientations (columns (a) and (b)). For highly anisotropic textures (column (c)), a random parameterization may fail, depending on the nature of the textures (column (d)). We can retain the anisotropy by assigning consistent surface orientations either by hand (column (e) and (f)) or using our iterative relaxation procedure (column (g)).

Model Geometry & Topology: Several textured meshes with different topologies and geometries are shown in Figure 9. As shown, the algorithm generates textures without discontinuity

[2] If $\vec{n}$ of p does not intersect a particular coarse mesh (e.g. it lies on a crease), we simply skip flattening at that level. Instead we assign a default color to the neighborhood pixels that are not covered, as in the flipping case.

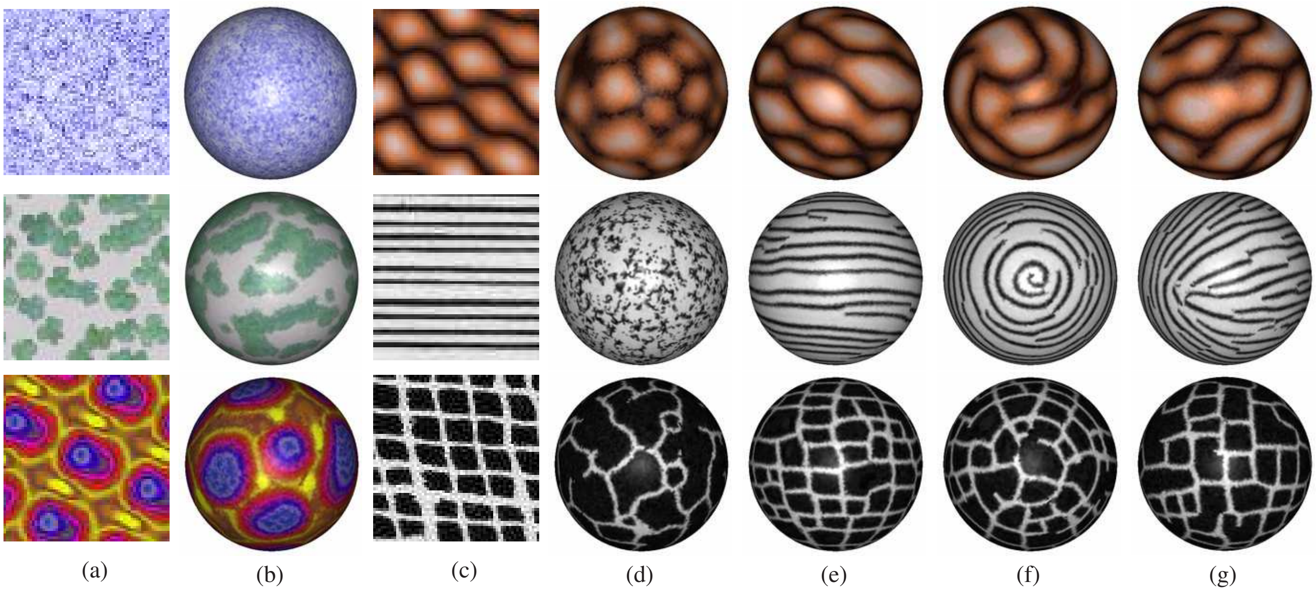


Figure 7: *Texture synthesis over a sphere uniformly tesselated with 24576 vertices and 49148 faces. (a) Isotropic textures of size 64x64. (b) Synthesis with random orientations. (c) Anisotropic textures of size 64x64. (d) Synthesis with random orientations. (e) Synthesis with $\vec{s}$ and $\vec{t}$ vectors at each vertex parallel to longitude and altitude of the sphere. (f) The polar views of (e), showing the singularity. (g) Synthesis with orientation computed by our relaxation procedure (Section 3.1). The top two textures are generated using 2-way symmetry (Figure 3 (b)), while the bottom one is generated using 4-way symmetry (Figure 3 (c)).*

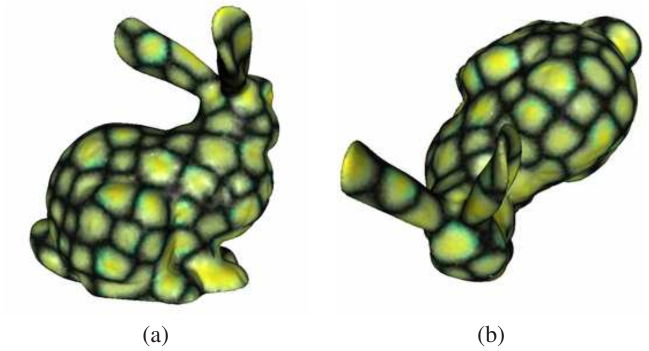


Figure 8: *Different views around the ears of the textured bunny in Figure 1. (a) back view. (b) top view.*

across a variety of surface geometries and topologies, even across fine features such as the bunny ear (Figure 8). The algorithm can also be used to synthesize surface attributes other than colors such as displacement maps (the mannequin model in Figure 9).

Computation Time: By using an efficient data structure for meshes (we use the quad-edge data structure [9], although other approaches are possible), we achieve linear time complexity with respect to the neighborhood sizes for both the flattening and resampling operations. In our C++ implementation running on a 450 MHz Pentium II machine, the timing for texturing the sphere in Figure 7 is as follows: relaxation (30 iterations) - 85 seconds, synthesis with exhaustive search - 695 seconds, and synthesis with tree-structured VQ acceleration - 82 seconds.

5 Conclusions and Future Work

We have presented extensions of [25] that permit us to synthesize textures over surfaces of arbitrary topology, beginning with a rectangular texture sample. The most significant of these extensions are that we traverse output vertices in a random order, thus allowing texture synthesis for general meshes, and we parameterize meshes with a user-selectable scale factor and local tangent directions at each mesh vertex. We define mesh neighborhoods based on this parameterization, and we show that this approach works over a variety of textures. Specifically, we synthesize isotropic textures with random local orientations, while generating anisotropic textures with local directions that are either hand-specified or automatically determined by our relaxation procedure.

Our approach has several limitations. Since it is an extention of [25], it only works for texture images; therefore it is not as general as [20] which can paste any image onto a mesh model. However for the class of textures that can be modeled by [25], our approach usually produces continuous surface textures with less blocky repetitions. In addition, for textures that are not well modeled by [25], we could generate better results by combining our surface-synthesis framework with other improved texture synthesis algorithms such as [1]. Finally, our representation of the output as a retiled polygonal mesh with vertex colors may not be desirable in cases where we would like to preserve the original mesh geometry. In such cases the output can be mapped back onto the original model in a post-process by resampling, such as in [3].

In concurrent work, Turk has developed a similar approach for synthesizing textures over surfaces [24]. The primary differences between [24] and our work are as follows: (1) we have used random as well as symmetric vector fields for certain textures, whereas [24] always creates a smooth vector field, (2) instead of a sweeping order, we visit mesh vertices in random order, (3) the two approaches use different methods for constructing mesh neighborhoods; [24] uses surface marching while we use flattening and resampling, and (4) we do not enforce a explicit parent-child relationship between mesh vertices at adjacent resolutions.

We envision several possible directions for future work. Although our relaxation procedure can assign reasonable local orientations for many anisotropic but symmetric textures, it remains an open problem for which symmetry classes local orientations can be assigned in this way. Another future direction is to use a variant of our algorithm to transfer textures (either colors or displacements) from one scanned model [13] to another mesh model. This could be done by replacing the input image I_a Table 3 with an input mesh

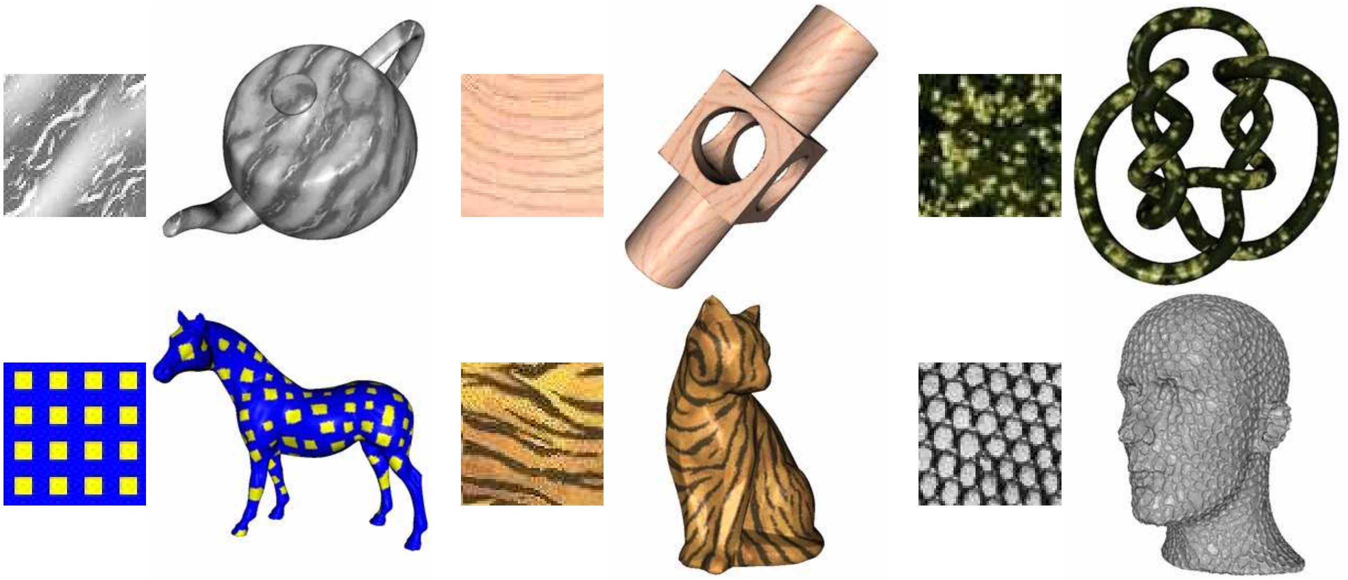


Figure 9: *Surface texture synthesis over different models. The small rectangular patches (size 64x64) are the input textures, and to their right are synthesis results. In all examples the textures are used to modulate the colors, except the last one where the texture is used for displacement mapping. Texture orientation and mesh sizes: teapot (no symmetry, 256155 vertices, 512279 faces), mechanical part (2-way symmetry, 49180 vertices, 98368 faces), knot (random, 49154 vertices, 98308 faces), horse (4-way symmetry, 48917 vertices, 97827 faces), cat (2-way symmetry, 50015 vertices, 100026 faces), and mannequin (no symmetry, 256003 vertices, 512002 faces).*

model, and changing line 1 and 15 in Table 3 to **BuildMeshPyramid** and **BuildMeshNeighborhood**, respectively. Finally, our definition of mesh neighborhoods might be applicable to other signal processing operations over meshes such as convolution, filtering, and pattern matching.

Acknowledgments

We would like to thank Greg Turk for his mesh retiling code and the anonymous reviewers for their comments. The texture thumbnails shown in the paper were acquired from the Brodatz texture album [2], MIT Vision Texture [16], Jeremy De Bonet's webpage, and other anonymous websites. Polygonal models were acquired from Hugues Hoppe's webpage, the Large Geometric Models Archive at Georgia Tech, and the OpenInventor model database. This research was supported by Intel, Interval, and Sony under the Stanford Immersive Television Project.

References

[1] M. Ashikhmin. Synthesizing natural textures. *2001 ACM Symposium on Interactive 3D Graphics*, pages 217–226, March 2001. ISBN 1-58113-292-1.

[2] P. Brodatz. *Textures: A Photographic Album for Artists and Designers*. Dover, New York, 1966.

[3] P. Cignoni, C. Montani, C. Rocchini, R. Scopigno, and M. Tarini. Preserving attribute values on simplified meshes by resampling detail textures. *The Visual Computer*, 15(10):519–539, 1999. ISSN 0178-2789.

[4] J. S. De Bonet. Multiresolution sampling procedure for analysis and synthesis of texture images. In T. Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 361–368. ACM SIGGRAPH, Addison Wesley, Aug. 1997.

[5] D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley. *Texturing and Modeling: A Procedural Approach.* Morgan Kaufmann Publishers, 1998.

[6] A. Efros and T. Leung. Texture synthesis by non-parametric sampling. In *International Conference on Computer Vision*, volume 2, pages 1033–8, Sep 1999.

[7] A. Gagalowicz and Song-Di-Ma. Model driven synthesis of natural textures for 3-D scenes. *Computers and Graphics*, 10(2):161–170, 1986.

[8] D. Ghazanfarpour and J. Dischler. Generation of 3D texture using multiple 2D models analysis. *Computer Graphics Forum*, 15(3):311–324, Aug. 1996. Proceedings of Eurographics '96. ISSN 1067-7055.

[9] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and computation of voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, April 1985.

[10] I. Guskov, W. Sweldens, and P. Schröder. Multiresolution signal processing for meshes. *Proceedings of SIGGRAPH 99*, pages 325–334, August 1999.

[11] D. J. Heeger and J. R. Bergen. Pyramid-Based texture analysis/synthesis. In R. Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 229–238. ACM SIGGRAPH, Addison Wesley, Aug. 1995.

[12] A. Hertzmann and D. Zorin. Illustrating smooth surfaces. *Proceedings of SIGGRAPH 2000*, pages 517–526, July 2000. ISBN 1-58113-208-5.

[13] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The digital michelangelo project: 3d scanning of large statues. *Proceedings of SIGGRAPH 2000*, pages 131–144, 2000.

[14] B. Lévy and J.-L. Mallet. Non-distorted texture mapping for sheared triangulated meshes. *Proceedings of SIGGRAPH 98*, pages 343–352, July 1998.

[15] J. Maillot, H. Yahia, and A. Verroust. Interactive texture mapping. In J. T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 27–34, Aug. 1993.

[16] MIT Media Lab. Vision texture. http://www-white.media.mit.edu/vismod/-imagery/VisionTexture/vistex.html.

[17] F. Neyret and M.-P. Cani. Pattern-based texturing revisited. *Proceedings of SIGGRAPH 99*, pages 235–242, August 1999.

[18] H. K. Pedersen. Decorating implicit surfaces. *Proceedings of SIGGRAPH 95*, pages 291–300, August 1995.

[19] K. Popat and R. Picard. Novel cluster-based probability model for texture synthesis, classification, and compression. In *Visual Communications and Image Processing*, pages 756–68, 1993.

[20] E. Praun, A. Finkelstein, and H. Hoppe. Lapped textures. *Proceedings of SIGGRAPH 2000*, pages 465–470, July 2000.

[21] P. Schröder and W. Sweldens. Spherical wavelets: Efficiently representing functions on the sphere. *Proceedings of SIGGRAPH 95*, pages 161–172, August 1995.

[22] E. Simoncelli and J. Portilla. Texture characterization via joint statistics of wavelet coefficient magnitudes. In *Fifth International Conference on Image Processing*, volume 1, pages 62–66, Oct. 1998.

[23] G. Turk. Re-tiling polygonal surfaces. *Computer Graphics (Proceedings of SIGGRAPH 92)*, 26(2):55–64, July 1992.

[24] G. Turk. Texture synthesis on surfaces. *Proceedings of SIGGRAPH 2001*, August 2001.

[25] L.-Y. Wei and M. Levoy. Fast texture synthesis using tree-structured vector quantization. *Proceedings of SIGGRAPH 2000*, pages 479–488, July 2000.