Texture and Shape Synthesis on Surfaces

Lexing Ying, Aaron Hertzmann, Henning Biermann, Denis Zorin
New York University
http://www.mrl.nyu.edu

Abstract. We present a novel method for texture synthesis on surfaces from examples. We consider a very general type of textures, including color, transparency and displacements. Our method synthesizes the texture *directly* on the surface, rather than synthesizing a texture image and then mapping it to the surface. The synthesized textures have the same qualitative visual appearance as the example texture, and cover the surfaces without the distortion or seams of conventional texture-mapping. We describe two synthesis methods, based on the work of Wei and Levoy and Ashikhmin; our techniques produce similar results, but directly on surfaces.

1 Introduction

Computer graphics applications increasingly require surfaces with highly detailed reflective properties, geometry and transparency. Constructing such detailed appearances manually is a difficult and tedious task. A number of techniques have been proposed to address this problem; procedural synthesis techniques are among the most widely used. A number of recent techniques [4, 13, 2] make it possible to synthesize textures from examples.

Creating a surface with a complex appearance can be viewed as synthesis of a collection of functions on an arbitrary two-dimensional domain. These functions include color, transparency, normals and coordinates of surface points. We will refer to all such functions as textures. The textures can be thought of as continuous and defined directly on surfaces, although they will be represented as samples in implementation. In this view, previously proposed example-based texture synthesis algorithms synthesize attributes for a special kind of surfaces, i.e. flat rectangles.

In this paper, we extend synthesis from examples to arbitrary surfaces. The obvious approach to the problem, synthesizing a texture on a rectangle and mapping it to an arbitrary surface, is likely to result in artifacts (e.g. seams or shape distortion); typically, creating high-quality maps requires considerable user intervention. Performing synthesis directly on a surface avoids many of the these problems.

Existing texture synthesis methods rely on the presence of identical, regular sampling patterns for both the example and the synthesized texture. Therefore, it is impossible to apply such methods directly to surfaces. In this paper, we regard the example and the synthesized texture as continuous functions that happen to be represented by samples, but not necessarily laid out in identical patterns. Whenever necessary, we resample either the example or the synthesized texture on a different pattern.

We describe two specific synthesis methods, based on the methods of Wei and Levoy [13] and Ashikhmin [2]. As with image synthesis, the choice of algorithm depends primarily on the texture.

The main contributions of this paper are: generalizations of existing image texture synthesis methods to synthesis on surfaces; synthesis of surface texture maps indepen-

dent of parameterization; efficient and accurate neighborhood sampling operations; and synthesis of texture, transparency, and displacements.

2 Related Work

Recently, several nearest-neighbor methods have been shown to produce textures of good quality. De Bonet [3] demonstrates a coarse-to-fine procedure that fills in pixels in an output texture image pyramid by copying pixels from the example texture with similar coarse image structure. Efros and Leung [4] create texture using a single-scale neighborhood. Wei and Levoy [13] combine the methods of [3, 4] by using a neighborhood that contains both coarse-scale and same-scale pixel information, and use Tree-structured vector quantization (TSVQ) [5] to accelerate the search for the nearest neighbor. Ashikhmin [2] produces high-quality textures by copying contiguous texture patches when possible.

Neyret and Cani [10] texture a surface isotropically by tiling from a small collection of tileable example texture triangles. Praun et al. [11] extend this by placing oriented texture patches over an independent parameterization of a surface. Although these methods produce high-quality results for many textures, they have some drawbacks: they cannot use texture patches that are large with respect to the surface shape, they cannot capture low-frequency texture without sacrificing high-frequency randomness, and the texture patches do not necessarily line up exactly, which requires careful selection of the patch shapes, and blending to hide discontinuities between patches.

In work concurrent to our own, Wei and Levoy [14] and Turk [12] develop methods for texture synthesis on surfaces. These methods also generalize Wei and Levoy's multiscale image texture synthesis algorithm [13]. We believe that our approach has several advantages. First, we define a fast method for neighborhood sampling that guarantees that there will not be any folds in the sampling grid, in contrast to relaxation [14] and surface marching [12]. Second, we synthesize directly to surface texture maps rather than to a densely tesselated mesh, meaning that our algorithm will be faster and require much less memory. However, we do require that the surface be covered with texture maps, and that charts can be constructed. Finally, we also generalize Ashikhmin's algorithm, which gives good results for many textures that are handled poorly by the multiscale synthesis algorithms.

3 Overview

Given a 2D example and a 3D surface, our goal is to create a texture that "looks like" it came from the same process that generated the example. We make the assumption that this is equivalent to requiring the texture on every small surface neighborhood to "look like" the texture on some neighborhood in the example.[1] The example and synthesized textures will be discretized into samples, although not necessarily with the same sample density. The texture may be from any domain: in particular, we explore image textures, transparency textures and geometric textures. For brevity, we refer to the synthesized texture as *target*.

[1] This formulation is based on the *Markov condition* assumption on the texture: we assume that the texture has local statistics (the value at a point depends only on its local neighborhood), and stationary statistics (the dependence is the same for every point). These assumptions imply that the the surface texture will "look like" the example texture if this joint density of texture values is the same for the surface as for the example.

Review of image texture synthesis: We now briefly review image texture synthesis algorithms; see the relevant papers for details. Efros and Leung [4] synthesize a texture in scan-line order. For each pixel, the already-synthesized values in neighborhood of the target pixel x are collected, and the example texture is searched to find pixel y with the neighborhood that matches the target neighborhood as closely as possible. The value of the pixel y is copied to the pixel x. Wei and Levoy [14] synthesize a Gaussian pyramid from coarse to fine. Each level of the pyramid is synthesized as in the Efros-Leung algorithm, except that samples are collected from the neighborhood at both the current scale and the coarser scale, and that TSVQ is used to accellerate the nearest-neighbors search.

Ashikhmin [2] synthesizes a texture in the same manner as Efros-Leung, except that only a restricted range of candidates for each pixel is tested. When synthesizing a value for a pixel x, we first consider each already-synthesized pixel y_i adjacent to x. Some source pixel y_i' in the example texture was previously copied to y_i. If we were to continue copying from the same patch of the example texture used for y_i', then x would get the value at location $y_i' + (x - y_i)$; this location is used as a candidate for x. Each y_i generates a candidate. We compare the neighborhood of each candidate to the neighborhood of x, and copy the color from the closest match y_i to x.

Texture representation: For simplicity, we assume that the example texture is resampled on a rectangular sampling grid, i.e. that it is an image. The target texture is represented by samples on the surface (i.e. by texture maps). We assume that there is a collection of rectangular images mapped to the surface and the texture samples will be stored in these images.

An important feature of our approach is that the synthesis process is independent of the choice of the texture-mapping parameterization: given a parameterization, our method will synthesize a texture without distortion on the surface. However, parametrization distortion may result in blurry textures in some areas.

Idea of our algorithms: For each sample of the target texture, we consider the previously-synthesized texture within a small neighborhood of the sample. Then we locate a similar neighborhood in the example and copy the value of the texture in the center of the neighborhood to the target sample.

Several issues need to be addressed to make this idea practical:

- how to pick a neighborhood of samples from the target texture,
- how to compare neighborhoods in the target and example,
- how to find similar neighborhoods in the example.

The two methods that we describe use somewhat different approaches to these problems. Each method has its strengths and weaknesses, illustrated in Section 6. Both methods use a common sampling pattern for the example and target textures to make the comparison possible, but use different approaches to resampling as discussed below.

Surface orientation: As most interesting textures are anisotropic, orientation must be established on the surface. We use a vector field to specify the correspondence between orientation on the surface and orientation in the domain of the example texture. A pair of orthogonal tangent vector fields v_1 and v_2 is used for this purpose. To compare the texture on the surface neighborhood centered at x to a neighborhood centered at y in

the example texture we establish a map between the neighborhoods, mapping x to y and v_1 and v_2 to the coordinate directions in the example.

The field v_1 is computed using the method described in [7][2]. The field v_2 is computed as the cross-product of v_1 and the oriented surface normal. The field could also be specified interactively, as in [11, 12].

Synthesis methods: Our first method is based on [13] and described in Section 4. For each generated sample x, we attempt to find the neighborhood in the entire example texture that matches the neighborhood of x as closely as possible. The sampling pattern of the example is used to resample the neighborhood of the target. As the example is a regularly sampled image, the fixed sampling pattern makes it possible to accelerate the search with standard nearest-neighbors algorithms, such as TSVQ.

The second method, based on [2] (Section 5), selects candidate values only from example neighborhoods that are spatially close in the example with some already-synthesized sample near x (*coherent synthesis*). This makes it necessary to keep track of the source of each generated sample. In this method, we use the target sampling pattern for comparison of neighborhoods, which is simpler than resampling the target. Acceleration (such as with TSVQ) is unnecessary in this method, because only a few neighborhoods will be tested.

4 Multiscale Synthesis

Our first method is based on the multiscale image synthesis procedure of Wei and Levoy [13]. In this method, we first synthesize a coarse version of the surface texture and then perform coarse-to-fine refinement. This method allows us to efficiently capture both coarse and fine-scale statistics, while performing several iterative refinements to the texture.

Our algorithm requires that the surface is covered by an atlas of sufficiently large overlapping charts. A chart is a map from a part of a surface into the plane; every point should be in the interior of a chart. Charts allow us to efficiently sample small neighborhoods of points.

Chart construction is relatively easy for a large class of texture-mapped surfaces, including polygonal meshes. If a surface is tiled with quadrilateral or any other polygonal texture maps, we can construct vertex charts from the texture maps. The texture maps themselves form a mesh on the surface. We can create a chart for each vertex of the mesh by flattening the collection of texture-mapped regions sharing the vertex to the plane, as explained below. Clearly, the resulting charts cover the whole surface, and any point on the surface is in the interior of one of the charts.

Chart construction is particularly simple for multiresolution subdivision surfaces, which we use in our implementation. There are several methods available for converting an arbitrary mesh to this representation [8, 6, 15], but we emphasize that this is not essential for our algorithm; as long as a tiling of the surface with texture maps is available, any representation can be used.

Our algorithm begins by creating a Gaussian pyramid for the example image and for each of the target texture maps. Every level of the hierarchy will be synthesized, from coarse-to-fine.

[2] Since we are optimizing a vector field, and do not desire 90^o-invariance as in [7], the optimization formula is of the form $\sum \cos((\theta_i - \varphi_{ij}) - (\theta_j - \varphi_{ji}))$ instead of $\sum \cos 4((\theta_i - \varphi_{ij}) - (\theta_j - \varphi_{ji}))$.

Our synthesis methods iterate over the surface texture maps in breadth-first order. Within each texture map, we synthesize in scan-line order, starting from a scan-line that is adjacent to an already-synthesized texture.

We use a brute-force search to synthesize the coarsest level. In order to synthesize a sample x, we sample the previously synthesized target texture in a neighborhood of x using the regular pattern of the example, and the surface marching technique described in the next section. Then we exhaustively search the example to find the nearest match under a weighted l^2-norm. We then copy the central sample from the best-matching neighborhood to x. Samples are weighted with a Gaussian kernel in image space. We sample texture values by bilinear interpolation of the four nearest neighboring values from the target texture maps. However, if some of these values have not yet been generated, then instead we use the nearest already-generated neighbor. If none of the values are available, then no sample value is generated at that location. This brute-force search is inefficient, but the coarse-level synthesis is nonetheless fast because there are very few samples to search over or to synthesize at the coarsest level of the pyramids.

We synthesize each of the remaining levels using a two-pass algorithm based on Wei and Levoy's [13] hole-filling algorithm. In the first pass, we synthesize each sample of a level of the hierarchy using a 5×5 neighborhood that contains only samples from the coarser level of the pyramid. In the second pass, we refine the texture at the current level using the composition of the 9×9 neighborhood from the current level and a 5×5 one from the coarser level. This means that all samples in each neighborhood have already been synthesized, allowing us to use TSVQ [5] or Approximate Nearest Neighbors [1] to accelerate the nearest-neighbors searches in both passes. The best match found during these searches is copied to the target sample. We also introduce some randomness into the search, with the same randomization used in [3, 13]: we locate the eight nearest matches found during the TSVQ traversal with backtracking, discard all matches that have a distance worse than the best match by a factor of 0.1, and then randomly pick one of the remaining matches by uniform random sampling.

We use two different methods for sampling neighborhoods on a surface. For the coarsest levels of the image hierarchy, we use *surface marching*, in which we traverse over smoothed geometry to locate sample points. For the remainder of the image hierarchy, we perform *chart sampling*, in which we construct sampling neighborhood patterns in a globally-defined parametric domain. In our experiments, chart sampling performs twice as fast; marching is only used for coarse levels where chart sampling cannot be used (for reasons described below).

4.1 Surface Marching

In the surface marching algorithm, we collect a grid of sample locations that corresponds to a grid of locations in the plane (Figure 1), using a tesselated mesh representation of the surface. For each sample point, we compute the angle and distance in the plane to the point. We then draw a straight path from the center surface point in the computed distance and direction (with respect to the orientation field on the mesh) to find the corresponding surface sample point. When the path intersects a mesh edge, the line is continued on the adjacent face, at the same angle to the mesh edge as on the previous face.

Note that other choices of path shape could be used to march from the center point to the target sample point. In particular, Turk [12] uses a path that corresponds to marching up or down in the texture domain, then left or right to find the sample point. In general, however, each of these paths may reach different parts of the surface. We

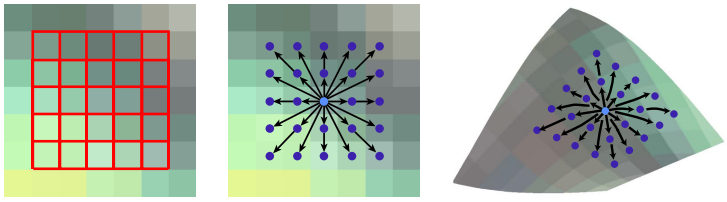


Fig. 1. Surface marching for neighborhood sampling. (a) A 5×5 rectangular surface neighborhood pattern on an image. (b) Each sample location in the neighborhood may be reached by a straight line from the center pixel. (b) A corresponding sampling pattern on a surface. From the center sample, we "march" over the surface, in directions corresponding to the straight-line directions in the image plane. This gives us a set of surface sample locations. Values at each location are determined by bilinear sampling from the texture map. The orientation of the pattern is determined by the surface vector field.

believe that the diagonal path described above is preferable, because it is the shortest such path, and thus least prone to distortion due to irrelevant features.

Regardless of which path we use, there are several problems with the surface marching approach. First, it is not guaranteed to give even sampling of a surface neighborhood in irregular geometry. Second, the sampling pattern is numerically unstable, as minute surface variations can cause substantial variations in the pattern. Finally, this method is relatively slow, because of the many geometric intersection and projection operations required.

4.2 Chart Sampling

Rather than trying to move on the surface from one face to the next, one can take advantage of a suitable surface parameterization. Recall that our goal is to be able to sample the texture at (approximately) regularly spaced locations around a sample x. Suppose a sufficiently large part of the surface around x is parameterized over a planar domain V. Then we can sample in the parametric domain V, choosing the sampling pattern in such a way that the corresponding sample points on the surface form the desired approximately regular arrangement (Figure 2). This can be achieved by using the Jacobian of the map from V to distort the sampling pattern in V.

The crucial assumption of the method is that the size of the neighborhood to be sampled is sufficiently small, so that the parameterization for the neighborhood is sufficiently close to linear and each neighborhood fits on the image of one *chart* $g(V)$. When this does not hold, as happens at the coarsest levels of synthesis, chart sampling cannot be used and we perform marching instead.

Chart sampling in detail: To explain more precisely how chart sampling is performed we need some additional notation. Let U_i be rectangular texture domains on which the surface is parameterized and f_i be the parameterizations (Figure 2). In practice, f_i are represented by piecewise-linear approximations defined by arrays of vertices, but it is convenient to consider f_i and other maps as continuous for now. Note that the images of boundaries of the rectangular domains form a mesh on the surface, with images of corners of the rectangles corresponding to vertices of the mesh.

We define overlapping charts, each corresponding a mesh vertex. The chart map g is defined implicitly by specifying maps Φ_i^v for each vertex, mapping each U_i into a

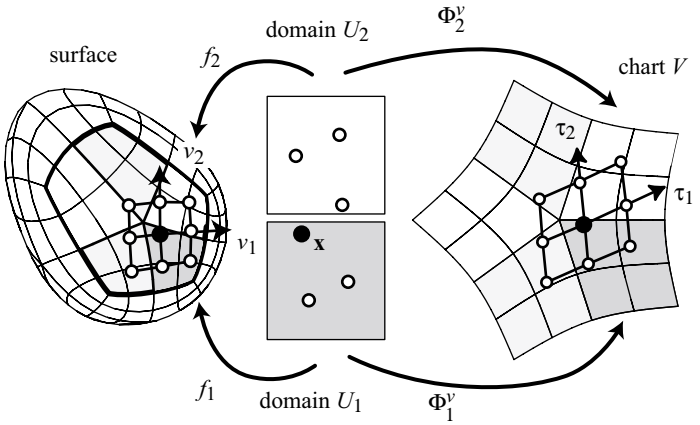


Fig. 2. Chart sampling. In order to sample the texture in the neighborhood of x, we construct its sampling pattern in the chart. The pattern is distorted in the chart such that it will be aligned with the surface orientation field (v_1 and v_2) and roughly square.

common planar domain V. The maps Φ_i^v should agree on the boundaries of the domains U_i which map to the same curves on the surface. Then g is taken to be $g = f_i \circ (\Phi_i^v)^{-1}$ on $\Phi_i^v(U_i) \subset V$.

The maps Φ_i^v should be defined using the same procedure as is used to define the parametrizations f_i, in order to ensure that the chart maps are smooth. For example, if an optimization process was used to define f_i, the same process should be used to define Φ_i^v, for some choice of the values for the images of the boundaries. We explain how we defined chart maps for subdivision surfaces at the end of this section.

In addition to parametrizations and chart maps, we assume orthogonal unit tangent vector fields v_1 and v_2 are defined on all U_i. These fields specify local orientation for synthesized anisotropic textures, and may be discontinuous.

We use the notation Dh to denote the differential of a map $h : R^s \to R^t$, which is a linear map given by the matrix $(\partial h^i / \partial x^j)_{ij}$.

Now we are ready to define the procedure to compute samples. Given a sample x in the domain U_1, we wish to compute a set of samples x_{lm} in domains U_i such that their corresponding surface points $f_i(x_{lm})$ form a pattern approximating a square grid with step δ in each direction, centered at $f_1(x)$. The samples are computed in several steps (Figure 2):

1. Map x to the chart domain V: $y = \Phi_1^v(x)$.
2. Compute chart domain vectors τ_1 and τ_2 that correspond to v_1 and v_2, respectively. For this, we require that the differential $Df_1(D\Phi_1^v)^{-1}$ maps τ_1 to v_1 and τ_2 to v_2. As v_1 and v_2 are tangent vectors, they can be written as $v_j = Df_1 w_j$ $j = 1, 2$, for some two-dimensional w_j. Then $\tau_j = D\Phi_1^v w_j$, $j = 1, 2$.
3. Compute sample locations in the chart domain: $y_{lm} = y + l\tau_1\delta + m\tau_2\delta$, for $l, m = -4..4$ (for a 9×9 sampling pattern).
4. Map the sample locations back to one of the parametric domains U_i, depending on which part $\Phi_i^v(U_i)$ of the chart domain V they are located: $x_{lm} = (\Phi_i^v)^{-1}(y_{lm})$.
5. Sample the texture values from the texture maps. For each sample location, the location x_{lm} is sampled by bilinear interpolation in texture map i, where i is determined by the parametric domain used in the previous step.

This procedure takes advantage of the assumption that the sampling neighborhood is small enough that various maps can be replaced by their linearized versions.

The maps f_i and Φ_i^v are represented as samples at vertex locations. To find a value of a map at an arbitrary point of a domain U_i, we use bilinear interpolation; to invert a map, we use point location and bilinear interpolation.

Chart sampling for subdivision surfaces: For a subdivision surface, we use one domain (and, thus, one texture map) per face of the control mesh. To compute chart maps Φ_i^v, we proceed as follows. For a vertex with valence k, we assign coordinates of the vertices of a regular planar k-gonal star as initial values of Φ_i^v to the corners of each rectangular domain U_i. We assign the coordinates of the center of the star to the corner of each U_i corresponding to v. Next, we extend the k-gonal star with an extra layer of similar quads. Finally, we apply subdivision to the two-dimensional coordinates to obtain values of Φ_i^v everywhere on U_i. This procedure is similar to computation of the *characteristic map*; in fact, values of the characteristic map also can be used as chart maps. See [15] for further details on characteristic maps and how they can be computed.

5 Coherent Synthesis

We now describe the *coherent synthesis* algorithm, based on Ashikhmin's algorithm for image texture synthesis [2]. This algorithm is based on the observation that the l^2-norm is an imperfect measure of perceptual similarity. Instead, it attempts to copy large coherent regions from the example texture, since such regions are guaranteed to have the appearance of the example texture, although there might be seams between them. This method runs much faster than the other methods and produces higher-quality results for many textures. The chart representation used in the previous section is not necessary here, nor is any multiresolution representation. However, like Ashikhmin's algorithm, this method does not work well for some smoothly varying textures.

This method runs in a single pass over all samples on the surface. For each synthesized sample, we record the floating point coordinates in the texture map that this sample was copied from. To synthesize a sample x, we find nearby samples which are already synthesized and use them to look up corresponding locations in the example texture. We displace these locations in order to obtain candidate source samples for x. These candidates are chosen so that they correspond to continuations of the samples already copied from the example. For each candidate, we collect a neighborhood of the same connectivity as the original target neighborhood, and compare it to the target neighborhood. We copy the value of the closest-matching candidate to x. Note that only a few candidates are considered, so no search acceleration is necessary. This makes it possible to resample the example texture rather than the target.

Now we describe the algorithm in greater detail. We treat the example texture as continuous, and use bilinear interpolation to evaluate between texture samples. The algorithm is illustrated in Figure 3.

1. To synthesize a sample x coming from an texture map I, we collect all synthesized samples x_i, $i = 0 \ldots m$ in the texture map I which are no more than 2 samples away from x. In the interior of the texture map, this corresponds to a 5×5 neighborhood of pixels in the texture domain. If x is less than two samples away from the texture map boundary, we also retrieve samples in the texture map sharing the boundary with I. Finally, if x is less than two samples away from a corner of an texture map, we also collect adjacent samples from all texture maps

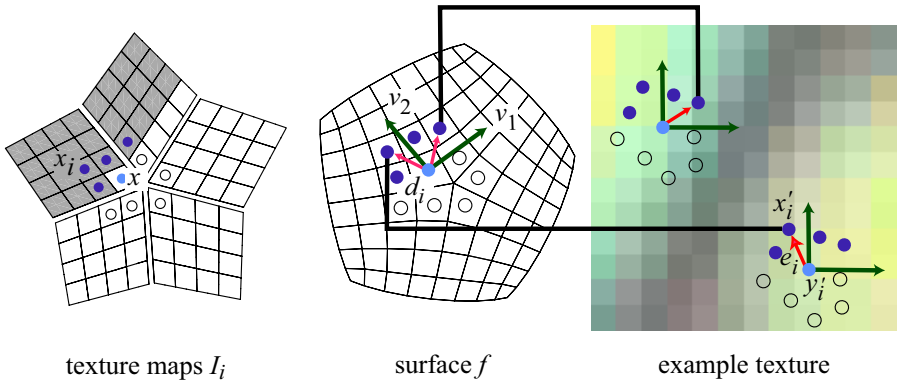


Fig. 3. Coherence synthesis. In order to synthesize a texture value for a point x, we examine each of its already-synthesized neighbors x_i (filled circles). In the figure, we show a point x that occurs at a "corner" where five texture maps meet on the surface. (The open circles indicate unsynthesized pixels, which are ignored during this step.) Each neighbor proposes a candidate location y'_i from the example, based on its own location in the example and its distance from x on the surface. Each y'_i corresponds to the texture location for x *if* we were to continue the texture patch used for x_i to x. The best candidate is computed by comparing the candidate neighborhoods with l^2.

sharing the corner. If no synthesized samples are located, a random value from the example texture is selected.

2. For each collected sample x_i, we compute the 3D displacement $d_i = f(x_i) - f(x)$ to the target surface point, and project d_i into the tangent plane at $f(x)$ to obtain tangent displacements d_i^t. The tangent displacements can be represented in the local coordinate system (v_1, v_2) by 2D vectors e_i: $d_i^t = e_i^1 v_1 + e_i^2 v_2$.
3. Our next goal is to locate candidate locations for x in the example texture. For each of the neighboring samples x_i, we look up the corresponding location x'_i in the example texture. We use these samples to generate candidate locations $y'_i = x'_i - e_i$, i.e. by looking up the value which is located in the example in the same way with respect to x'_i as x is with respect to x_i. Note that, unlike in Ashikhmin's method, the location and the displacement should be represented as floating point numbers in order to prevent errors due to round-off.
4. Now we need to choose which of the candidates is used to get the value for the target. To do this, we compare neighborhoods of y'_i with the neighborhood of x. We use the same set of displacements e_i to get samples around y'_i which are arranged in the same pattern as x_i around x, i.e. we consider neighborhoods $N(y'_i)$ consisting of samples $y'_{ij} = y'_i + e_j$. Undefined values are discarded when the distance is computed. Among these neighborhoods we choose the one for which the l^2 distance from the the neighborhood of x is minimal

6 Experiments

In Figure 4, we demonstrate coherent synthesis of a nut texture on a cow model. Note that the texture maps appear distorted, because they are synthesized to appear undistorted on the surface. With coherent synthesis, the texture has high quality, but some small discontinuities are visible. Figures 5 and 6 show the transparency and displace-

ment maps created with multiscale synthesis. In Figure 7, different scales of the same texture were synthesized with multiscale synthesis and the same vector field. Figure 8 compares the multiscale and coherent algorithms. The quality of the results is very similar to those of the 2D algorithms when applied to these textures. In our experience, one can predict the results of the surface synthesis by running the 2D algorithms. Figure 9 demonstrates synthesis of transparency maps. The results of coherent synthesis on complex models are displayed in Figure 10.

Timings for these experiments were as follows:

Mesh	Texels		Example size		Method		Time (min)	
Cow (Fig 4)	369,664		96×96		C		3	
3D Cross (Fig 5)	399,360		64×64		M		10	
Ball (Fig 6)	98,304		64×64		M		2.5	
Dog (Fig 7)	696,320	174,080	64×64	64×64	M	M	18	5
Torus (Fig 8)	114,688		64×64	64×64	M	C	1.5	0.5
			192×192	192×192	M	C	290	0.5
Ball (Fig 9)	98,304		64×64		M		2.5	
Cow (Fig 10)	289,792		128×128		C		2	
Horse (Fig 10)	275,968		128×128		C		2	

For meshes with multiple results, the positions in the table correspond to the positions in the figures. The second column shows the total number of samples that were synthesized in texture maps for each result. The fourth column shows the synthesis method used: "M" for multiscale or "C" for coherent. These timings compare favorably with those reported by Wei and Levoy [14] and Turk [12]; our multiscale synthesis appears to be about 3 times faster for generating a comparable number of samples with a given example size. Our coherent synthesis is dramatically faster than multiscale synthesis or the other methods. It appears that the multiscale synthesis does not scale well when both the example size and neighborhood size grow; the flower texture on the torus (Figure 8) took nearly five hours to compute. This time can be reduced to a few minutes by reducing the neighborhood size to 5×5 (as done by Turk [12]), by reducing the example texture size, or by using coherent synthesis.

7 Discussion and Future Work

We have presented efficient methods for synthesizing a texture onto a 3D surface from a 2D example texture. These methods produce textures with similar quality and speed to their 2D counterparts. This means that those textures that work well with Ashikhmin's algorithm work well with our coherence algorithm. Hence, there is reason to hope that these strategies may be applied to future 2D texture synthesis algorithms as well.

There are a number of directions for future work. All of our sampling operations use either point sampling or bilinear sampling. However, since texels represent area averages of a function, weighted area integrals should be used instead.

Surface reflectance functions and material properties, such as BRDFs or fur [9], can be synthesized, perhaps via straightforward extensions of the ideas presented here.

Real-time procedural shaders for example-based texture synthesis would allow complex surface textures to be generated in real time. However, this appears difficult because the samples generated by the algorithms are interdependent.

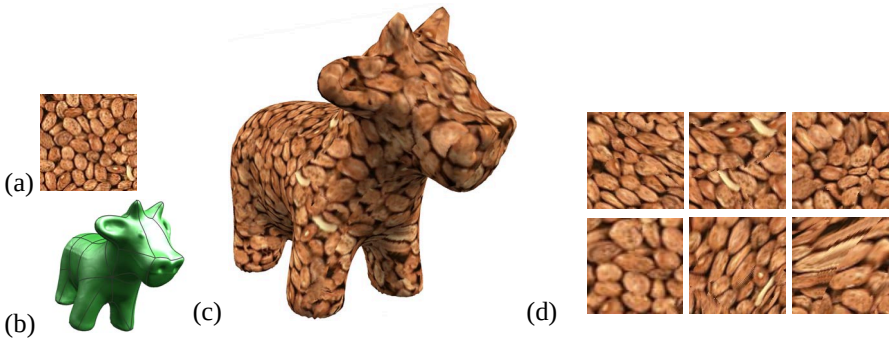
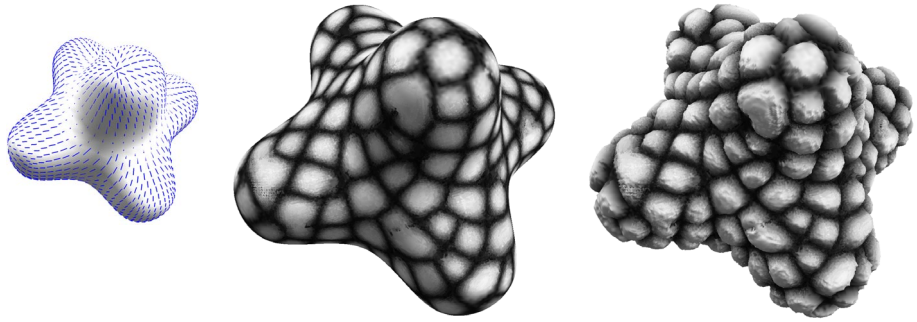


Fig. 4. Surface texture synthesis. (a) Example texture, obtained from VisTex database and downsampled to 96×96. (b) Cow model, showing edges corresponding to edges of top-level faces. (c) Synthesis result using coherent method based on Ashikhmin's algorithm [2]. (d) Representative texture maps generated during the process. Variations in surface shape appear as distortions in the texture maps. The surface is covered by a total of 50 texture maps.

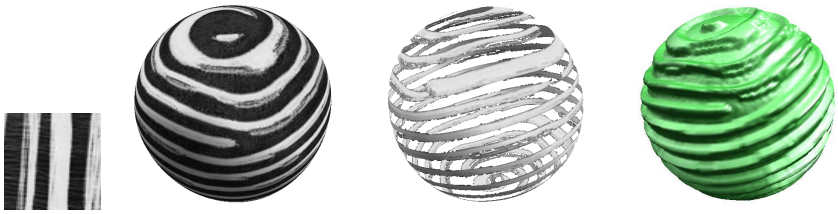
References

1. Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An Optimal Algorithm for Approximate Nearest Neighbor Searching in Fixed Dimensions. *Journal of the ACM*, 45(6):891–923, 1998.
2. Michael Ashikhmin. Synthesizing natural textures. *2001 ACM Symposium on Interactive 3D Graphics*, pages 217–226, March 2001.
3. Jeremy S. De Bonet. Multiresolution sampling procedure for analysis and synthesis of texture images. *Proceedings of SIGGRAPH 97*, pages 361–368, August 1997.
4. Alexei Efros and Thomas Leung. Texture Synthesis by Non-parametric Sampling. *7th IEEE International Conference on Computer Vision*, 1999.
5. Allen Gersho and Robert M. Gray. *Vector Quantization and Signal Compression*. Kluwer Academic Publishers, 1992.
6. Igor Guskov, Kiril Vidimče, Wim Sweldens, and Peter Schrder. Normal meshes. *Proceedings of SIGGRAPH 2000*, pages 95–102, July 2000.
7. Aaron Hertzmann and Denis Zorin. Illustrating smooth surfaces. *Proceedings of SIGGRAPH 2000*, pages 517–526, July 2000.
8. Aaron W. F. Lee, Wim Sweldens, Peter Schröder, Lawrence Cowsar, and David Dobkin. Maps: Multiresolution adaptive parameterization of surfaces. *Proceedings of SIGGRAPH 98*, pages 95–104, July 1998.
9. Jerome E. Lengyel, Emil Praun, Adam Finkelstein, and Hugues Hoppe. Real-time fur over arbitrary surfaces. *2001 ACM Symposium on Interactive 3D Graphics*, pages 227–232, March 2001.
10. Fabrice Neyret and Marie-Paule Cani. Pattern-based texturing revisited. *Proceedings of SIGGRAPH 99*, pages 235–242, August 1999.
11. Emil Praun, Adam Finkelstein, and Hugues Hoppe. Lapped textures. *Proceedings of SIGGRAPH 2000*, pages 465–470, July 2000.
12. Greg Turk. Texture Synthesis on Surfaces. *Proc. SIGGRAPH 2001*, August 2001. To appear.
13. Li-Yi Wei and Marc Levoy. Fast Texture Synthesis Using Tree-Structured Vector Quantization. *Proceedings of SIGGRAPH 2000*, pages 479–488, July 2000.
14. Li-Yi Wei and Marc Levoy. Texture Synthesis Over Arbitrary Manifold Surfaces. *Proc. SIGGRAPH 2001*, August 2001. To appear.
15. D. Zorin, P. Schröder, T. DeRose, L. Kobbelt, A. Levin, and W. Sweldens. Subdivision for modeling and animation. SIGGRAPH 2000 Course Notes.



(a) (b) (c)

Fig. 5. (a) Surface with orientation field. Note the singularity of the field at the top of the model. (b) Synthesized texture using multiscale synthesis of the first texture from Figure 8 in grayscale. The texture appears consistent at the singularity. (c) Texture mapping and displacement mapping with the same texture.



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

Fig. 6. Textured sphere generated with multiscale synthesis. (a) Example texture. (b) Synthesized texture. (c) Texture mapping plus transparency mapping (using the same texture). (d) Displacement mapping.

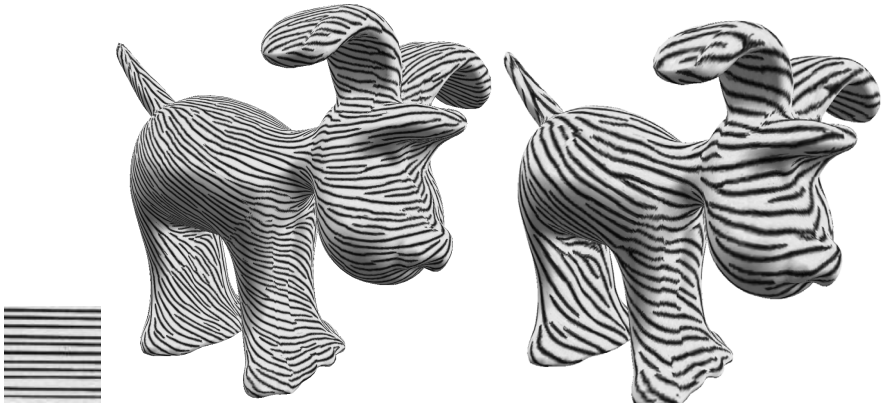


Fig. 7. Zebra dog, generated with multiscale synthesis. Varying the scale parameter creates a texture with a different size on the surface.

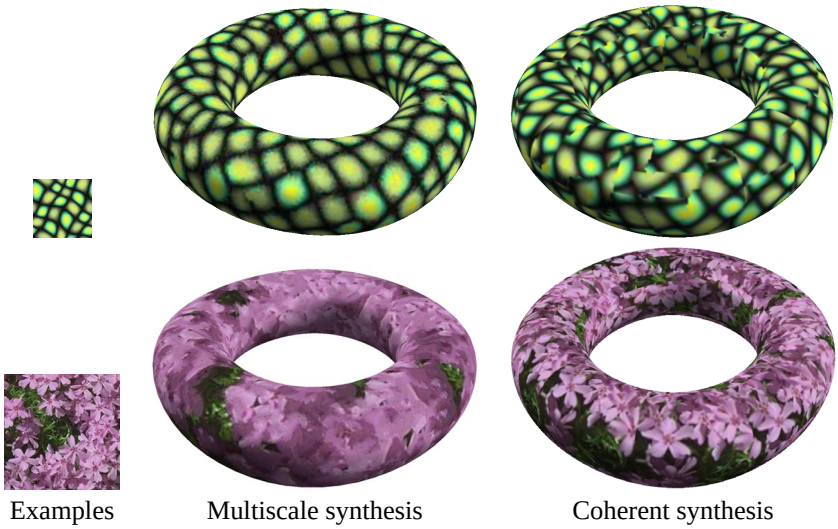


Fig. 8. Comparison of multiscale (based on Wei-Levoy [13]) and coherent algorithms (based on Ashikhmin [2]). The results are comparable to those of the image texture synthesis algorithms.

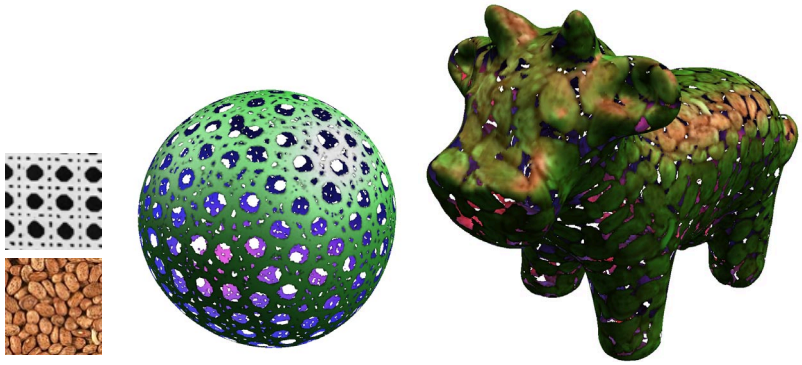


Fig. 9. Transparency mapping. *Left:* Example textures. *Middle:* Wicker ball, generated from first texture by multiscale synthesis. *Right:* Bronze cow, generated by coherent synthesis from second texture blended with a green surface color.

Fig. 10. Chia cow and sea horse, generated with coherent synthesis.