Texture Synthesis on Surfaces

Greg Turk

GVU Center, College of Computing
Georgia Institute of Technology

Abstract

Many natural and man-made surface patterns are created by interactions between texture elements and surface geometry. We believe that the best way to create such patterns is to synthesize a texture directly on the surface of the model. Given a texture sample in the form of an image, we create a similar texture over an irregular mesh hierarchy that has been placed on a given surface.

Our method draws upon texture synthesis methods that use image pyramids, and we use a mesh hierarchy to serve in place of such pyramids. First, we create a hierarchy of points from low to high density over a given surface, and we connect these points to form a hierarchy of meshes. Next, the user specifies a vector field over the surface that indicates the orientation of the texture. The mesh vertices on the surface are then sorted in such a way that visiting the points in order will follow the vector field and will sweep across the surface from one end to the other. Each point is then visited in turn to determine its color. The color of a particular point is found by examining the color of neighboring points and finding the best match to a similar pixel neighborhood in the given texture sample. The color assignment is done in a coarse-to-fine manner using the mesh hierarchy. A texture created this way fits the surface naturally and seamlessly.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—display algorithms; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—surfaces and object representations

Keywords: Texture synthesis, texture mapping.

1 Introduction

There are a wide variety of natural and artificial textures that are influenced by the surfaces on which they appear. We will use the term *surface texture* to describe such geometry-influenced textures and to distinguish them from *solid textures* [17, 15]. Natural examples of surface textures include the pattern of bark on a tree, spots and stripes on a wide variety of animals (mammals, fish, birds, etc.), the placement of hair and scales on an animal, and the pattern of flowers and trees on a hillside. Human-made textures that are tailored to the surface geometry include the fabric pattern on furniture, the stone patterns on walls and buildings, and the marks of a chisel on a sculpture. Most techniques in computer graphics for making surface textures have concentrated either on the placement of an already existing texture on a given surface or on the synthesis of texture on a regular 2D array of pixels. The texture synthesis method presented in this paper is guided by our belief that these two tasks, texture synthesis and texture placement, should be performed simultaneously to create surface textures.

An ideal texture creation system would allow a user to provide a 3D model to be textured, a sample of the texture to be placed on the model, and a guide to the orientation of the texture. To texture a leopard, for example, the user should be able to scan an image of leopard spots and then give this image (the *texture sample*) to the texturing system. The system would then make similar looking spots all over the surface of a polygonal cat model, guided by orientation hints from the user. Many methods exist that will create an arbitrary amount of additional texture from a given texture sample, and we refer to these as *texture-from-sample* methods. Unfortunately, all such methods that have been published create the new texture over a regular lattice of pixels. The user is still left with the burden of wrapping this texture over a model.

In this paper we use ideas that are adapted from texture-from-sample methods to directly synthesizing texture on a polygonal surface. We first create a hierarchy of meshes over the surface, and we use this mesh hierarchy much like an image pyramid [1]. Each point in the mesh hierarchy is eventually given a color, and the collection of colored points form the final texture. We create the new texture by performing operations on these surface points in a way that mimics image processing operations used in several texture-from-sample methods. The main challenge is adapting the 2D pixel grid operations to similar operations on a mesh hierarchy. The specific approach that we use is to color a point based on finding a close match between neighboring points in the mesh that have already been colored and similar 2D pixel neighborhoods in the given sample texture. The distance metric used for matching is the sum of the squared differences between the color components. A key aspect of performing such matches is to visit the points in such an order so that when a given point is visited, all of the points to one side of this point have already been assigned a color. We achieve this by sweeping across all of the points according to a user-defined vector field, and this vector field determines the orientation of the texture.

2 Previous Work

2.1 Texture Placement

There are many ways in which to take an existing texture from a rectangular pixel array and wrap it onto a surface. The goals of these methods are to avoid noticeable seams between texture patches and to minimize the amount of stretching and distortion of the pattern. Maillot et al. used a deformation tensor to describe an energy measure of distortion that they minimized over a surface made up of triangles [13]. They also use an atlas to piece together a final texture from several patches. A similar energy-guided approach was taken by Levy and Mallet [12]. Their energy term penalizes distortions, but also incorporates additional constraints such as user-defined curves and cuts in the surface. Related to these energy-minimizing methods is the texture pelting approach of Piponi and Borshukov [18]. This method treats the surface of a model like the skin of an animal that is opened at the belly and then stretched onto a circular rack. Pedersen described a way of allowing a user to interactively position texture patches on implicit surfaces with low distortion [16].

turk@cc.gatech.edu

ACM SIGGRAPH 2001, 12-17 August 2001, Los Angeles, CA, USA
© 2001 ACM 1-58113-374-X/01/08...$5.00

The lapped texture technique of Praun et al. takes one or more irregularly shaped texture patches and place many copies of the patches in an overlapping fashion over a surface [19]. These patches are oriented according to a user-defined vector field and they are mapped in a way that minimizes distortion. For many textures this method produces excellent results, the nature of the overlapping patches is often unnoticeable.

2.2 Procedural Texture Synthesis

We describe here a few of the many methods that have been proposed for creating textures by procedural means. Perlin and Peachey independently invented the solid texture – a function that returns a color value at any given point in 3-space [17, 15]. Solid textures are ideal for simulating surfaces that have been carved out of a block of material such as wood or marble. Perlin also introduced the 3D noise function, which can be used to create patterns such as water waves, wood grain and marble. Worley created a cellular noise function, a variant of 3D noise that has discontinuities, and this function is useful for creating patterns such as waves and stones [29]. Neyret and Cani have developed a novel method of generating a small number of triangular tiles (typically four) that can be used to texture a surface [14]. Each of the tiles is created in such a way that its edge matches the edge of other tiles so that the texture appears to be seamless when the tiles are placed adjacent to one another.

Reaction-diffusion is a chemical process that builds up patterns of spots and stripes, and this process can be simulated to create textures. Witkin and Kass demonstrated that a wide variety of patterns can be created using variations of one basic reaction-diffusion equation [27]. Turk demonstrated that a simulation of reaction-diffusion can be performed on an array of cells that have been placed over a polygonal surface [23]. Because the simulation proceeds directly on the surface, the spot and stripe patterns of this method are undistorted and without seams. Fleischer et al. demonstrated how interacting texture elements on a surface can be used to create texture geometry such as scales and thorns on a surface [5]. Walter and Fournier showed that another biological mechanism, cell cloning, can be simulated on a collection of cells to produce a variety of patterns found on animals [24].

2.3 Texture Synthesis from Samples

Many people have noted the limitations of the procedural texture synthesis approach, namely that creating a new texture requires a programmer to write and test code until the result has the right "look". A different approach to texture synthesis is to allow the user to supply a small patch of the desired texture and to create more texture that looks similar to this sample.

Simoncelli and Portilla make use of statistics that summarize relations between samples in a steerable pyramid in order to synthesize a texture [20]. Their method is a very successful example of the parametric method of analysis and synthesis, where image statistics are used to describe the texture and to create more texture. We refer the interested reader to their bibliography for many other parametric approaches. Heeger and Bergen make use of Laplacian and steerable pyramid analysis of a texture sample to create more texture [8]. They initialize a pyramid with white noise, create a pyramid from the texture sample, and then modify the noise so that its histogram matches the histograms of the color samples at each level in the sample texture's pyramid. Collapsing the pyramid gives a new image, and repeated application of this entire process creates a texture similar to the original. DeBonet also makes use of a multi-scale pyramid analysis to perform synthesis [2]. He makes use of two Laplacian pyramids (one for analysis and one for synthesis) as well as edge and line filters to analyze the texture. He visits the levels of the synthesis pyramid from top to bottom, and the "ancestor" samples of a pixel to be synthesized are matched against the analysis pyramid. The new pixel is selected randomly from among the best matches.

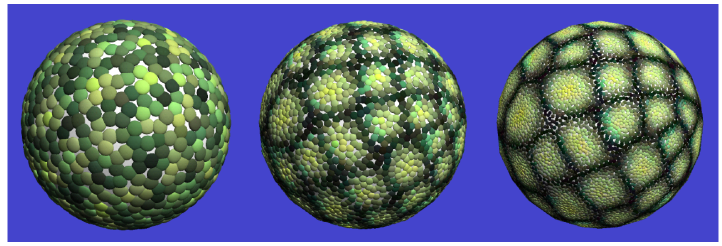


Figure 1: Vertex colors on three levels in the mesh hierarchy. The orientation field for this example flows from left to right on the sphere, and the texture used is the same as on the ray in Figure 5. Mesh vertices are rendered as flattened spheres to show their colors.

Efros and Leung make a more direct use of the texture sample [4]. They first create a tiny seed image by copying a few pixels from the texture sample, and then they visit pixels surrounding this seed in a spiral pattern. They examine the pixels in a large square patch surrounding a given pixel, and look for the best few matches to this patch in the texture sample. They randomly select from among these matches, and this is the new color for the synthesized pixel. Wei and Levoy use a similar method to synthesize texture, but they visit the pixels in a raster scan order and they also use a multi-scale framework [25]. Instead of matching neighborhood pixels from a single image, they perform the matching based on two adjacent levels in Gaussian pyramids. They use vector quantization to dramatically speed up this matching process.

The texture creation method of our paper combines the textures-from-samples method of Wei and Levoy and the surface synthesis approach exemplified by [23, 5]. We have recently learned that other researchers have also extended texture-from-sample methods to surfaces and have produced wonderful results [26, 30].

3 Creating a Mesh Hierarchy

Our own work adapts ideas from the texture-from-sample methods to the task of creating a texture that is made specifically for a given polygonal surface. Unfortunately all of these methods assume that one is working with a regular grid of pixels, and there is no way to create such a regular grid over an arbitrary surface. Instead, we create a set of points that evenly cover the surface but that are not in a strictly regular arrangement. Because hierarchical synthesis techniques produce high-quality results, we make use of a hierarchy of points. The highest level of the hierarchy is a set of points that sparsely covers the model. The next level of the hierarchy contains four times this number of points, and this pattern repeats down to the finest level of the point hierarchy. A mesh is formed for each level of the hierarchy, and each mesh specifies the connectivity between points within a given level.

Given a polygonal model of an object to be textured, why don't we just use a texture synthesis method that operates directly on the vertices of the original mesh of polygons? For the simple reason that we have no guarantees about the density of these original vertices. We do not know if there are enough of these vertices to create a detailed texture once each is assigned a color. Furthermore, the vertex density may vary a good deal from one location to the next, and these density variations will cause problems during synthesis. Given that we need to create our own mesh over the surface, there are still choices to be made. We can attempt to re-mesh the surface so that we have a semi-regular mesh structure (like the polygons created from subdivision surfaces), or we can use an irregular mesh structure. In this paper we have opted for an irregular mesh structure, although we believe that similar methods to our own can be applied to semi-regular meshes as well.

Our goal in creating a mesh hierarchy is to match the basic structure of a Gaussian pyramid [1]. For a Gaussian pyramid $G(I)$, we will call $G_1(I)$ the highest resolution level of the pyramid, and

$G_2(I)$, $G_3(I)$ and so on are successively lower resolution levels in the pyramid. Each pyramid level $G_{k+1}(I)$ has one-quarter the number of pixels than in the higher resolution level $G_k(I)$. A pixel $G_k(i,j)$ at level k can be said to be the *child* of its *parent* pixel $G_{k+1}(\lfloor i/2 \rfloor, \lfloor j/2 \rfloor)$ in the lower resolution level $k+1$, and this child pixel is one of four pixels that share the same parent. Another way to view an image pyramid is that some pixels are present in more than one level in the pyramid, and that these pixels have a color associated with each level in which they are present. From the highest resolution level of the pyramid (G_1, the original image), one out of four pixels is also retained in the next level G_2. One fourth of *these* pixels are also present in G_3, and so on. We wish to retain this same structure in our mesh hierarchy.

3.1 Point Placement and Connectivity

Assume we are building an m-level mesh hierarchy, and that at each level k of the hierarchy is a mesh $M_k = (V_k, T_k)$ described by its vertices V_k and its triangles T_k. We create such a hierarchy by placing n points on the surface, connecting these to form a mesh, placing $3n$ additional points on the surface, creating a second mesh that contains all $4n$ points, adding more points to make a total of $16n$ points, and so on. We place the original n points on the surface at random, then use repulsion between points to spread them out evenly over the surface. There are several published methods that place points on surfaces using repulsion [23, 28, 5, 11], and we use the method of Turk [23]. These first n points will become the mesh vertices V_m at the lowest resolution level of the hierarchy. After these first n points have been evenly placed, their positions are fixed, $3n$ more point are put on the surface, and these new points are repelled by one another and by the n original points. The result is two sets of points that together evenly cover the surface. The union of these two sets of points form the vertices V_{m-1} of mesh M_{m-1}. Note that all of the points in V_m are also in V_{m-1}, much like when a pixel is present in several levels of the Gaussian pyramid. The point placement process is performed m times, and the union of all points that have been placed on the surface are the vertices V_1 of the most detailed mesh M_1. Figure 1 shows the vertices from a three-level mesh hierarchy on a sphere. These vertices are rendered as flattened spheres to show their texture color, and the method of arriving at these colors will be described later.

Once all the points have been placed on the surface, the mesh connectivity must be calculated for each level of the mesh hierarchy. We connect a point by projecting nearby points to a tangent plane and performing Delaunay triangulation, and this determines which other points should be connected to the point in question. This projection method can on rare occasions cause nearby points to disagree on whether or not they should be connected, in which case we force the two points to be connected to one another.

3.2 Operations on Mesh Hierarchy

While performing texture synthesis we will make use of several quantities that are stored at the mesh vertices, including color $\mathbf{C}(v)$, a vector $\mathbf{O}(v)$ for texture orientation, and a scalar value $s(v)$ that we call the *sweep distance* from the synthesis initiation point. In order to perform texture synthesis, we make use of several operations on a mesh that act on these quantities:

- Interpolation
- Low-pass filtering
- Downsampling
- Upsampling

As it turns out, only the first two of these, interpolation and low-pass filtering, are absolutely necessary for the method of this paper. Upsampling and downsampling are useful for accelerating some portions of the method. We describe our implementation of these four operations below, and we will use color as the quantity being operated on with the understanding that operations on other quantities are similar.

Interpolation on a mesh is the process of determining the color at a given position p on the surface, where p might not be at a mesh vertex. To perform color interpolation, we use weighted averages of the colors at nearby mesh vertices. This method finds all of the mesh vertices $v_1, v_2, \ldots, v_t$ within a particular radius r of the point p. The value of the color $\mathbf{C}(p)$ is then:

$$\mathbf{C}(p) = \frac{\sum_{i=1}^{t} w(|p - v_i|/r)\mathbf{C}(v_i)}{\sum_{i=1}^{t} w(|p - v_i|/r)} \qquad (1)$$

We use $w(x) = 2f^3 - 3f^2 + 1$ for the weighting function, which has the properties $w(0) = 1$ and $w(1) = 0$. Mesh vertices near p give the largest contribution, and their contributions fall off smoothly as their distances approach r.

Another important operation that we use is to low-pass filter (blur) the colors at the mesh vertices. Here we borrow techniques from mesh smoothing [22, 3, 7]. The basic step in mesh smoothing is to move a vertex from its old position v_{old} to a new position v_{new} that is influenced by the n vertices that are directly connected to the vertex on the mesh:

$$v_{new} = v_{old} + t \sum_{i=1}^{n} w_i (v_i - v_{old}) \qquad (2)$$

In the above equation, the value of t must be fairly small (e.g. $t = 0.1$) to guarantee stability. This equation is applied repeatedly to all the vertices in order to smooth a mesh. The values w_i in the above equation weight the contribution of the vertex v_i, and we weight according to inverse edge length (normalized by the sum of all the weights), as suggested in [22] and [3]. Similarly to Equation 2, we can calculate the new color at a vertex v based on the colors of adjacent mesh vertices:

$$\mathbf{C}_{new}(v) = \mathbf{C}_{old}(v) + t \sum_{i=1}^{n} w_i (\mathbf{C}(v_i) - \mathbf{C}_{old}(v)) \qquad (3)$$

We can define values α and β_i and re-group the terms to write this in a slightly simpler form:

$$\mathbf{C}_{new}(v) = \alpha \mathbf{C}_{old}(v) + \sum_{i=1}^{n} \beta_i \mathbf{C}(v_i) \qquad (4)$$

We perform low-pass filtering of the colors on a mesh by repeated application of Equation 4.

The two other mesh operations (upsampling and downsampling) can be implemented directly from the first two. Downsampling is the operation used to create a Gaussian pyramid. It is the process of taking the colors at one pyramid level, blurring them, and then dropping every other pixel horizontally and vertically to make a four-to-one reduction in image size. These new pixels then make up the next lower resolution level in the pyramid. We can perform a similar operation on meshes. For a vertex v that appear in more than one level in a mesh hierarchy, we keep a separate color $\mathbf{C}_k(v)$ for each level k on which the vertex appears. We downsample from mesh M_k to the lower-resolution mesh M_{k+1} by blurring the colors on mesh M_k and then inheriting the color $\mathbf{C}_{k+1}(v)$ at a vertex v in mesh M_{k+1} from its blurred color value in mesh M_k.

The other between-level operation, upsampling, is also simple to implement on a mesh hierarchy. On pyramids, upsampling is the process of enlarging an image on level $k+1$ to produce a new image for level k that has four times as many pixels. We upsample from level $k+1$ to level k by taking the position of a vertex v in mesh M_k and interpolating the color at that position using weighted average mesh interpolation on the less-detailed mesh M_{k+1}.

With these four operations in hand, we have the necessary tools to accomplish our first task in texture synthesis: specifying the orientation of the texture over the surface.

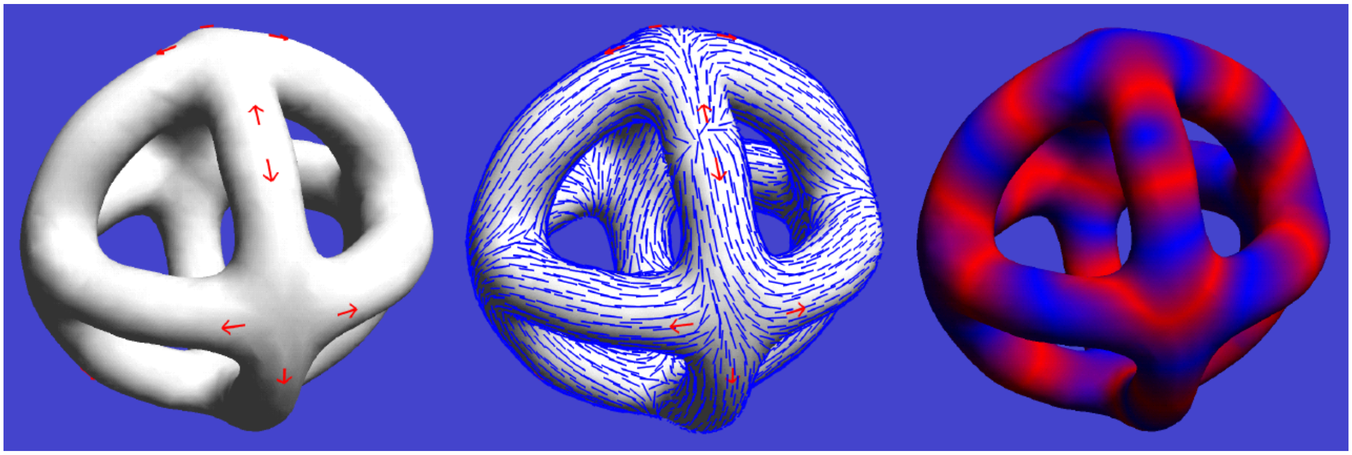


Figure 2: User's orientation input (left), the interpolated orientation field (middle), and the sweep values shown using color cycling (right).

4 Vector Fields and Surface Sweeping

Many natural and man-made textures have a strong directional component to them. In order to preserve the directional nature of textures, an orientation field must be specified over the surface to be textured. We do this by allowing a user to pick the direction of the texture at several locations and then interpolating the vectors over the remainder of the surface. We typically used roughly a dozen user-defined vectors for the examples shown in this paper. Several methods have been presented that interpolate vectors over a surface, including [19, 9]. We present a fast new method of performing vector interpolation that uses our mesh hierarchy.

4.1 Vector Field Creation

Our vector field interpolation method is inspired by the pull/push sparse interpolation method of Gortler et al. [6]. First, each vertex in the mesh M_1 is assigned a zero length vector except where the user has specified a direction. We perform a number of downsampling operations, and this "pulls" the user-defined vector values up to the coarsest mesh M_m. Many of the vertices on this mesh still have zero length vectors, so we need to perform interpolation over this mesh. We fix the non-zero vector values, and then diffuse the vector values over the rest of the surface using Equation 4 (adapted to vector values). At each diffusion step we project the vectors onto the surface's tangent plane. Once all vertices on this coarse mesh are non-zero, we then upsample the vector values to the mesh M_{m-1}, M_{m-2} and so on until we arrive at mesh M_1. We normalize the vectors after each upsampling. After the final upsampling step, all of the vertices of the finest mesh have a vector value. Figure 2 (left and middle images) demonstrates the creation of a vector field by this method. The method is fast, typically less than 30 seconds for a four-level mesh hierarchy with 256,000 vertices.

4.2 Surface Sweeping

Once we have created an orientation field, we then use it to define an ordering to the points on the surface. Our goal is to use this ordering to make a sweep across the surface that follows the vector field. Such a sweep over the surface will mimic sweeping down the scanlines of a raster image.

We begin the ordering process by randomly selecting an anchor vertex A on the surface from which we will measure distances along the vector field. We have found that the choice of anchor vertex is not important to the results of our method. Our task is then to assign a scalar value $s(v)$ that we call the *sweep distance* to each vertex v that will measure distance along the vector field from A. The further downstream a vertex is from the anchor vertex, the larger its value $s(v)$ will be, and vertices upstream from the anchor will take on negative sweep distances. If we rotate our vector field by 90 degrees, then the sweep distance resembles a *stream function* for 2D incompressible fluid flow [10].

We calculate the values $s(v)$ on a mesh using a modified diffusion process. Initially the anchor point A is given a value $s(A)$ of zero, and the values for $s(v)$ for all other vertices are derived from this. Similar to diffusion of color values (described in Section 3.2), the new value $s(v)$ for a vertex is given by a weighted sum of the values that the neighboring vertices believe that v should take on. Consider a vertex w that is adjacent to v (that is, they share an edge of the mesh). We will calculate how much further along the vector field v is than w, and we measure these distances in the direction of the local orientation of the vector field. We calculate the consensus orientation of the vector field near the two vertices by averaging their orientations $(\mathbf{O}(v)+\mathbf{O}(w))/2$, and then by projecting this vector onto the tangent plane of the surface and normalizing it. Call this consensus orientation $\mathbf{O}_{vw}$. We project the positions $\mathbf{v}$ and $\mathbf{w}$ of the two vertices onto this consensus orientation and take the difference: $\Delta_w = \mathbf{v}\cdot\mathbf{O}_{vw} - \mathbf{w}\cdot\mathbf{O}_{vw}$ (see Figure 3). This value Δ_w measures how much further downstream v is from w, so w believes that $s(v)$ should be equal to $s(w)+\Delta_w$. We calculate the new value for $s(v)$ as a weighted average of the values that its neighboring vertices believe it should have:

$$s_{new}(v) = \alpha s(v) + \sum_{i=1}^{n} \beta_i (s(v_i) + \Delta_{v_i}) \tag{5}$$

Propagating the values $s(v)$ across the mesh consists of repeated application of Equation 5 at each mesh point, with one additional modification. Each vertex of the mesh is in one of two states, *assigned* or *unassigned*, and each vertex v also has a current approximation for $s(v)$. Initially only the anchor point A is in the assigned state. When Equation 5 is used to calculate the value of $s(v)$ at a given vertex, only those vertices that have already been assigned are allowed to contribute to the weighted sum, and the values for α and β_i are adjusted accordingly. A vertex changes its state from unassigned to assigned when at least one of its neighbors contributes to the sum in Equation 5.

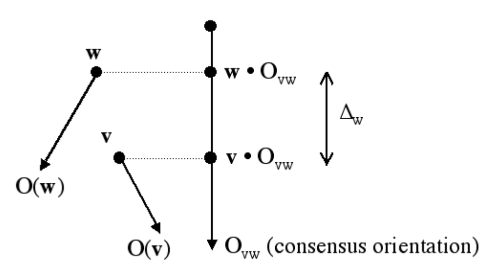
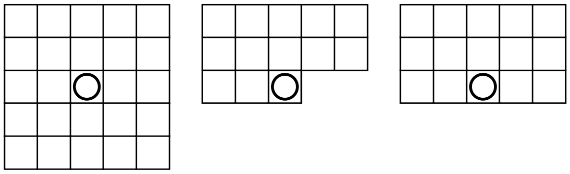


Figure 3: Relative distance along vector field of vertices v and w.



(a) Full square (b) Wei-Levoy causal (c) Half square

Figure 4: Three different pixel neighborhoods.

Just as with vector field creation, we calculate the values for $s(v)$ first on the coarsest mesh M_m, then on the next finer mesh and so on down to the finest mesh M_1. At this point, all of the vertices of this most detailed mesh have a value $s(v)$ assigned to them. The coarse-to-fine approach makes calculating the sweep distance fast. Once we know each value $s(v)$, we sort the vertices in increasing order of $s(v)$, and this defines an order in which to visit the mesh points. Figure 2 (right image) illustrates the values $s(v)$ on a topologically complex surface using color that repeatedly cycles as s increases. It requires 45 seconds to calculate the s values on our most complex mesh hierarchies with 256,000 vertices.

5 Textures from Neighborhood Colors

The heart of our texture synthesis method is determining the color of a mesh vertex based on the colors that have already been assigned to nearby mesh vertices. Our method is inspired by the texture synthesis methods of Efros and Leung [4] and Wei and Levoy [25]. In both these methods, the pixel color being chosen in the synthesized image, $S(i,j)$, is assigned the color of a best-match pixel from I. The quality of a match is determined by calculating the (possibly weighted) sum of squared differences between the already colored pixels around $S(i,j)$ and the pixels surrounding a candidate pixel $I(a,b)$. The color of the candidate pixel $I(a,b)$ whose neighborhood give the smallest value is copied to the output pixel $S(i,j)$. If the input texture is non-periodic, no comparisons are done with the neighborhoods that are at the edges of the image I. Notice that the first few pixels that are synthesized will have few or no neighboring pixels with colors that have already been assigned. These methods initialize the colors in the output image with values which act as a "seed" for the texture synthesis process.

The size and shape of a pixel's neighborhood affect the quality of the synthesis results, and larger neighborhoods usually give higher quality results. Figure 4 illustrates three pixel neighborhoods: (a) a fully populated 5×5 neighborhood, (b) Wei and Levoy's causal 5×5 neighborhood, and (c) a half-square 5×5 neighborhood. The circles in this figure mark the pixel whose color is to be determined, and this pixel's color is not used in neighborhood matching. Wei and Levoy point out that a neighborhood like that of (a) uses many pixels that have not yet been assigned a color, and this produces poor synthesis results. They use neighborhood (b) because only those pixels that have already been given a value in S are used if the pixels are visited in raster scan order. We use neighborhoods (a) and (c), and we will discuss these more later.

When a neighborhood $\mathbf{N}$ contains k pixels, we can think of the neighborhood as a flat array of $3k$ values taken from the pixel colors: $\mathbf{N}=\{r_1,g_1,b_1,\ldots r_k,g_k,b_k\}$. All of the color components are treated exactly the same, so the distinction between the colors can be dropped from the notation. The match value $D(\mathbf{M},\mathbf{N})$ between two neighborhoods $\mathbf{M}=\{m_1,\ldots m_{3k}\}$ and $\mathbf{N}=\{n_1,\ldots n_{3k}\}$ is the component-by-component sum of the squared differences.

This method of using neighborhood comparisons to determine a pixel color is based on the Markov Random Field (MRF) model for textures. This model assumes that the color of a pixel is based on a probability distribution that is given by neighboring pixel values. Clearly there is a chicken-and-egg problem: pixel A and B may be in each other's neighborhoods, so the probability distribution for A depends on the color of B and vice-versa. There are several ways to approximately satisfy the MRF model, and one way is to simply assign the color of one before the other, as we do in our method. Using a hierarchical approach (described later) gets around this mutual dependency between pixel colors to some extent. Finding the best match between neighborhoods is one way of sampling a probability distribution for the current pixel.

5.1 Synthesis on a Mesh

A non-hierarchical version of our texture synthesis method on a mesh proceeds as follows. We make a pass through all of the vertices in the mesh, ordered by their values $s(v)$, and pick a color for this vertex from the pixel that has the best match to the vertex's neighborhood.

When both the input and output images are regular grids of pixels, performing neighborhood matching is straightforward. In the case of texture synthesis on a surface, however, mesh vertices on the surface are not arrayed in nearly as regular a pattern as pixels. We need to define what it means to compare neighborhood colors on a mesh with pixel colors in an image. The ingredients we need to do this are the mesh interpolation operator from Section 3.2 and the vector field on the surface. The vector field gives us a local frame of reference to orient our movements near each vertex on the mesh. We can move over the surface either along the vector field in the direction $\mathbf{O}(v)$, or perpendicular to it along $\mathbf{P}(v)$, which we define to be $\mathbf{O}(v)$ rotated 90 degrees counter-clockwise about the surface normal. The vector fields $\mathbf{O}$ and $\mathbf{P}$ provide us with a local coordinate system, and allow us to treat the region surrounding a point as if it was a piece of the plane. We move in units of r over the surface, where r is the average distance between mesh vertices.

We will adopt the convention that pixel locations (i,j) in an image increase in i as we move right, and increase in j as we move down the image. Similarly, we will move over a surface in the direction $\mathbf{P}(v)$ when we move to the "right" of a vertex, and we will move in the $\mathbf{O}(v)$ direction when we want to move "down" on the surface. Suppose we wish to compare the neighborhood at mesh vertex v with the pixel neighborhood at pixel $I(a,b)$. As an example, let us find the corresponding mesh location for the pixel $I(a+1,b)$ that is directly to the right of pixel $I(a,b)$. Call the values $(1,0)$ the *pixel offset* for this neighboring pixel. We find the corresponding point on the mesh by starting at v and traveling a distance r in the direction $\mathbf{P}(v)$. In general, for a pixel offset of (i,j), we find the corresponding point on the surface by starting on the surface at v, traveling a distance ir in the direction $\mathbf{P}(v)$, and then a distance jr in the direction $\mathbf{O}(v)$. We use color interpolation to find the color at this point on the mesh, and this mesh color is then used in the neighborhood matching. The task of traveling in the direction of $\mathbf{O}(v)$ or $\mathbf{P}(v)$ over the surface is accomplished as is done during point repelling, by moving over a polygon until an edge is reached and then folding the path onto the next polygon.

Note that at any given vertex, we only need to calculate its neighborhood colors $\mathbf{N}(v)$ just once. These colors are then compared against the neighborhood colors of all pixels in the sample image I in order to find the best match. The vertex in question gets its new color $\mathbf{C}_{best}$ from the pixel of the sample texture I that has the closest neighborhood match. Here is pseudo-code for our texture synthesis method for an input image I:

```
For each vertex v on mesh
  C(v) = color of random pixel from I
For each vertex v on mesh (ordered by s(v))
  construct neighborhood colors N(v)
  smallest_match = BIG
  For each pixel (a,b) in I
    construct neighborhood colors M(a,b)
    new_match = D(N(v),M(a,b))
    If (new_match < smallest_match)
      smallest_match = new_match
      C_best = I(a,b)
  C(v) = C_best
```

5.2 Multi-Level Synthesis

The texture synthesis method as described above produces patterned surfaces, but the patterns are not always a good match to the input texture. In order to produce higher-quality textures we adopt a multi-level synthesis approach similar to that of Wei and Levoy [25]. Our method is a coarse-to-fine approach, and we make use of a Gaussian pyramid of the sample texture and our multi-level mesh hierarchy. The colors used for the neighborhood matching are taken from either one or two levels of these hierarchies, and we make use of several kinds of neighborhoods. Before describing the full multi-level process, some notation will be useful.

Figure 4 (a) and (c) show the two neighborhoods we use, called the *full-square* and *half-square* neighborhoods, respectively. Consider a neighborhood of a pixel $G_k(a,b)$ on level k in a Gaussian pyramid $G(I)$. The notation $F(n,0)$ refers to a full-square neighborhood of $n \times n$ pixels that uses colors from the nearby pixels to the current pixel $G_k(a,b)$. Similarly, $H(n,0)$ is a neighborhood made of nearby pixel colors in a half-square pattern, with n pixels on its longest side. The notation $F(n,1)$ refers to a square neighborhood that gets its colors from pixels at the next lower resolution pyramid level, and it is centered at pixel $G_{k+1}(\lfloor a/2 \rfloor, \lfloor b/2 \rfloor)$. Each of these neighborhoods has a similar meaning on a mesh hierarchy. Recall that many of the vertices are present in several levels of the hierarchy, and that each vertex v stores a separate color $C_k(v)$ for each level k in the hierarchy. The neighborhoods $F(n,0)$ and $H(n,0)$ at a vertex take their colors by interpolation of vertex colors at the current level k. The locations of these colors are found by moving over the mesh in steps of length $r2^{k-1}$. The $F(n,1)$ neighborhood takes its colors from mesh level $k+1$, and the locations for neighborhood colors are found by moving in steps of length $r2^k$ over the surface.

Our best texture results come from making multiple sweeps over the surface, alternating between two types of neighborhoods that can be though of as an *extrapolating* neighborhood and a *feature refinement* neighborhood. The extrapolating neighborhood we use is $F(n,1)$, and it has the effect of creating colors at level k solely based on colors from levels $k+1$. Because mesh level $k+1$ has fewer vertices than level k, this extrapolation produces a low-detail pattern relative to the mesh density on level k. Thus we use a second pass to add more details, and this is done by making use of the more detailed color information available on the current level k, as well as color samples from $k+1$. Our feature refinement neighborhood is the concatenation of the two full-square neighborhoods $F(n,0)$ and $F(\lfloor n/2 \rfloor, 1)$.

The final ingredient needed for texture synthesis on a mesh is to seed the texture creation process. We first color the vertices at level k of the mesh from randomly selected pixels on level k of the Gaussian pyramid $G_k(I)$. Then we use the neighborhood $H(n,0)$ to create an initial pattern on level k from these random colors. We use this half-square neighborhood to create the initial pattern because only those vertices to one side of the current vertex have been assigned meaningful color values, due to the way in which we sweep over the surface. Notice that this is the only synthesis step that does not make use of color values from higher up in the hierarchy. We have found that the quality of the pattern after this initial synthesis pass is key to the quality of the final texture. This is the reason we synthesize the texture in the sweep order – we have found that this gives us the best coarse initial pattern.

To create a detailed texture, we perform synthesis using several passes at three or four levels in the mesh hierarchy (see Table 1). Here are the meshes and neighborhood sizes that we use for three-level synthesis:

M_3 with $H(7,0)$	Create initial pattern
M_2 with $F(7,1)$	Extrapolate
M_2 with $F(7,0)+F(3,1)$	Refine
M_1 with $F(7,1)$	Extrapolate
M_1 with $F(7,0)+F(3,1)$	Refine

Figure 1 shows texture creation at three stages of hierarchical synthesis. The left image shows the results after the first pass on the low-resolution mesh M_3. The middle image shows how more detail is added on the intermediate resolution mesh, and the right image shows the finished texture on the high resolution mesh. For textures with especially large features we use four mesh levels, but we keep the same neighborhood sizes as given above.

6 Displaying the Texture

We are ready to display the texture on the surface once we have determined a color for every vertex on the finest mesh in the hierarchy. One possibility is to use per-vertex color and display the detailed mesh M_1. Because this mesh can be overly detailed, we choose instead to display the texture on the original user-provided mesh. We use the approach of Soucy et al. [21] to make a traditional 2D texture map T from the synthesized texture. For each triangle in the original mesh, we map it to a corresponding triangle in T. The triangles in T are colored using interpolation of the synthesized texture colors. The triangles in T that we use are 45 degree right triangles of uniform size, but it is also possible to use triangles that are better fit to the mesh triangle shape and size. The resulting texture can be rendered on the surface at interactive rates. The images in Figure 5 were all rendered in this manner using 2048×2048 textures on an SGI Onyx with InfiniteReality graphics. Our input models are composed of between 10,000 and 20,000 triangles, and these render with textures at real-time frame rates.

7 Results

Figure 5 show our synthesis results on complex models. The synthesis times varies from a few minutes for simple models to more than an hour when the texture sample is large (Table 1). We used meshes with 256,000 vertices for most models.

The stingray is textured using a wiggly checker image that has been used by a number of researchers in texture synthesis [2, 4, 25]. The vector field spreads out from a point on the ray's snout, and the texture must spread out as well to match the user's desired orientation. Nevertheless, the created pattern is comparable in quality to 2D synthesis results. The octopus model shows that the synthesis method has no trouble creating a pattern on branching surfaces (in this case, eight ways). The middle left image shows scales that we have placed on the bunny model. The sample image of the scales is non-periodic (does not tile), yet the synthesis algorithm is able to create as much texture as it needs. Notice that the rows of scales can be followed all the way from the ears down to the tail. The high degree of order of this texture is a result of visiting the mesh vertices in sweep order during the coarse-level synthesis stage.

The middle right image shows a texture made of interlocking diagonal strands of hooks that has been synthesized on the a model with complex topology, namely three blended tori. The original texture is periodic, and this periodicity is respected on the created surface pattern. The strands wrap themselves all the way around the tubes of the model, and the strands come together in a reasonable manner where four tubes join. The zebra model has been textured using a sample image of English text. This texture shows that fine features such as letter forms can be captured by our approach. The

Model	Mesh Points	Levels Used	Texture Size	Time
Ray	64,000	3	64×64	7
Octopus	256,000	3	64×64	34
Bunny	256,000	3	128×128	80
Tori	256,000	4	64×64	23
Zebra	256,000	4	256×256	108
Elephant	256,000	4	64×64	29

Table 1: Models, mesh levels used, sample textures and synthesis times (in minutes) on an SGI Octane2 with a 360 MHz R12000.

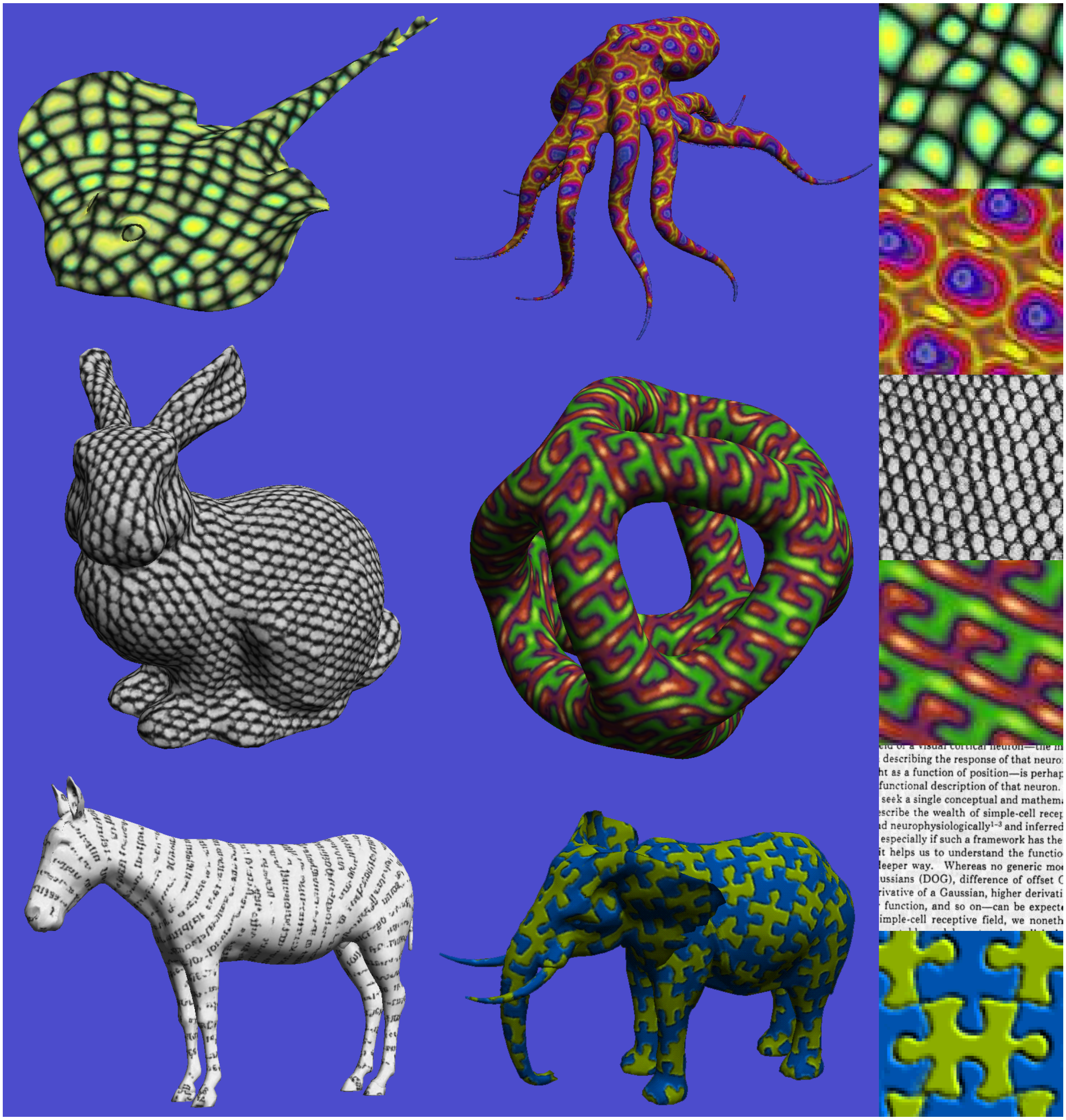


Figure 5: Results of our texture synthesis method on six models. Input textures are shown at the right.

puzzle pieces on the elephant show that the method creates plausible shapes even when the created pattern doesn't replicate the input texture exactly.

It is worth considering whether other existing methods can create similar results to those in Figure 5. The reaction-diffusion [23, 27] and the clonal mosaic methods [24] are really the only previous *synthesis* methods that tailor a texture to a given surface. Neither of these approaches can generate the kinds of textures that we show. Probably the highest-quality texture *mapping* method to date is the lapped texture work of Praun et al. [19]. Their paper shows a pattern of scales over a complex model, but close examination of the published images show artifacts where the patch borders blend together. Moreover, they note in their paper that textures that are highly structured or that have strong low-frequency components will cause patch seams to be noticeable when using their method. The textures we show on the stingray, tori and elephant have these characteristics, and thus would be poor candidates for using the lapped texture approach.

8 Conclusion and Future Work

We have presented a method of synthesizing texture that is specifically made for a given surface. The user provides a sample image and specifies the orientation of the texture over the surface, and the rest of the process is entirely automatic. The technique may be used for surfaces of any topology, and the texture that our method produces follows a surface naturally and does not suffer from distortion or seams. Key to our approach is the ability to perform image processing operations on an irregular mesh hierarchy.

There are several directions for future work. One possibility is to use the vector quantization approach of Wei and Levoy to speed up the texture creation process. Another is to adapt other image processing operations such as edge and line detection to irregular meshes, and this may lead to even better texture synthesis results. Another intriguing possibility is to use synthesis methods to produce appearance changes other than color, such as creating normal and displacement maps. Finally, the approach we have taken is to use a 2D image as the input texture, but it should be possible to extend this method to taking patterns directly from other surfaces. Imagine being able to "lift" the color, bumps and ridges from one model and place them onto another surface.

9 Acknowledgements

This work was funded in part by NSF CAREER award CCR–9703265. Much thanks is due to Ron Metoyer, Jonathan Shaw and Victor Zordan for help making the video. We also thank the reviewers for their suggestions for improvements to this paper.

References

[1] Burt, Peter J. and Edward H. Adelson, "The Laplacian Pyramid as a Compact Image Code," *IEEE Transactions on Communications*, Vol. COM-31, No. 4, April 1983, pp. 532–540.

[2] De Bonet, Jeremy S., "Multiresolution Sampling Procedure for Analysis and Synthesis of Texture Images," *Computer Graphics Proceedings*, Annual Conference Series (SIGGRAPH 97), August 1997, pp. 361–368.

[3] Desbrun, Mathieu, Mark Meyer, Peter Schroder and Alan H. Barr, "Implicit Fairing of Irregular Meshes using Diffusion and Curvature Flow," *Computer Graphics Proceedings*, Annual Conference Series (SIGGRAPH 99), August 1999, pp. 317–324.

[4] Efros, A. and T. Leung, "Texture Synthesis by Non-Parametric Sampling," *International Conference on Computer Vision*, Vol. 2, Sept. 1999, pp. 1033–1038.

[5] Fleischer, Kurt, David Laidlaw, Bena Currin and Alan Barr, "Cellular Texture Generation," *Computer Graphics Proceedings*, Annual Conference Series (SIGGRAPH 95), August 1995, pp. 239–248.

[6] Gortler, Steven J., Radek Grzeszczuk, Richard Szeliski and Michael F. Cohen, "The Lumigraph," *Computer Graphics Proceedings*, Annual Conference Series (SIGGRAPH 96), August 1996, pp. 43–54.

[7] Guskov, Igor, Wim Sweldens and Peter Schroder, "Multiresolution Signal Processing for Meshes," *Computer Graphics Proceedings*, Annual Conference Series (SIGGRAPH 99), August 1999, pp. 325–334.

[8] Heeger, David J. and James R. Bergen, "Pyramid-Based Texture Analysis/Synthesis," *Computer Graphics Proceedings*, Annual Conference Series (SIGGRAPH 95), August 1995, pp. 229–238.

[9] Hertzmann, Aaron and Denis Zorin, "Illustrating Smooth Surfaces," *Computer Graphics Proceedings*, Annual Conference Series (SIGGRAPH 2000), July 2000, pp. 517–526.

[10] Kundu, Pijushi K., *Fluid Mechanics*, Academic Press, San Diego, 1990.

[11] Lee, Aaron W., David Dobkin, Wim Sweldens and Peter Schroder, "Multiresolution Mesh Morphing," *Computer Graphics Proceedings*, Annual Conference Series (SIGGRAPH 99), August 1999, pp. 343–350.

[12] Levy, Bruno and Jean-Laurent Mallet, "Non-Distortion Texture Mapping For Sheared Triangulated Meshes," *Computer Graphics Proceedings*, Annual Conference Series (SIGGRAPH 98), July 1998, pp. 343–352.

[13] Maillot, Jerome, Hussein Yahia and Anne Verroust, "Interactive Texture Mapping," *Computer Graphics Proceedings*, Annual Conference Series (SIGGRAPH 93), August 1993, pp. 27–34.

[14] Neyret, Fabrice and Marie-Paule Cani, "Pattern-Based Texturing Revisited," *Computer Graphics Proceedings*, Annual Conference Series (SIGGRAPH 99), August 1999, pp. 235–242.

[15] Peachey, Darwyn R., "Solid Texturing of Complex Surfaces," *Computer Graphics*, Vol. 19, No. 3, (SIGGRAPH 85), July 1985, pp. 279–286.

[16] Pedersen, Hans Kohling, "Decorating Implicit Surfaces," *Computer Graphics Proceedings*, Annual Conference Series (SIGGRAPH 95), August 1995, pp. 291–300.

[17] Perlin, Ken, "An Image Synthesizer," *Computer Graphics*, Vol. 19, No. 3, (SIGGRAPH 85), July 1985, pp. 287–296.

[18] Piponi, Dan and George Borshukov, "Seamless Texture Mapping of Subdivision Surfaces by Model Pelting and Texture Blending," *Computer Graphics Proceedings*, Annual Conference Series (SIGGRAPH 2000), July 2000, pp. 471–478.

[19] Praun, Emil, Adam Finkelstein, and Hugues Hoppe, "Lapped Textures," *Computer Graphics Proceedings*, Annual Conference Series (SIGGRAPH 2000), July 2000, pp. 465–470.

[20] Simoncelli, Eero and Javier Portilla, "Texture Characterization via Joint Statistics of Wavelet Coefficient Magnitudes," *Fifth International Conference on Image Processing*, Vol. 1, Oct. 1998, pp. 62–66.

[21] Soucy, Marc, Guy Godin and Marc Rioux, "A Texture-Mapping Approach for the Compression of Colored 3D triangulations," *The Visual Computer*, Vol. 12, No. 10, 1996, pp. 503–514.

[22] Taubin, Gabriel, "A Signal Processing Approach to Fair Surface Design," *Computer Graphics Proceedings*, Annual Conference Series (SIGGRAPH 95), August 1995, pp. 351–358.

[23] Turk, Greg, "Generating Textures on Arbitrary Surfaces Using Reaction-Diffusion," *Computer Graphics*, Vol. 25, No. 4, (SIGGRAPH 91), July 1991, pp. 289–298.

[24] Walter, Marcelo and Alain Fournier, "Clonal Mosaic Model for the Synthesis of Mammalian Coat Patterns," *Proceedings of Graphics Interface*, Vancouver, BC, Canada, June 1998, pp. 82–91.

[25] Wei, Li-Yi and Marc Levoy, "Fast Texture Synthesis using Tree-structured Vector Quantization," *Computer Graphics Proceedings*, Annual Conference Series (SIGGRAPH 2000), July 2000, pp. 479–488.

[26] Wei, Li-Yi and Marc Levoy, "Texture Synthesis Over Arbitrary Manifold Surfaces," *Computer Graphics Proceedings*, Annual Conference Series (SIGGRAPH 2001), August 2001 (these proceedings).

[27] Witkin, Andrew and Michael Kass, "Reaction-Diffusion Textures," *Computer Graphics*, Vol. 25, No. 4, (SIGGRAPH 91), July 1991, pp. 299–308.

[28] Witkin, Andrew P. and Paul S. Heckbert, "Using Particles to Sample and Control Implicit Surfaces," *Computer Graphics Proceedings*, Annual Conference Series (SIGGRAPH 1994), July 1994, pp. 269–277.

[29] Worley, Steven, "A Cellular Texture Basis Function," *Computer Graphics Proceedings*, Annual Conference Series (SIGGRAPH 96), August 1996, pp. 291–294.

[30] Ying, Lexing, Aaron Hertzmann, Henning Biermann, Denis Zorin, "Texture and Shape Synthesis on Surfaces," submitted for review.