Eurographics Symposium on Rendering 2003
Per Christensen and Daniel Cohen-Or (Editors)

Fast Texture Synthesis on Arbitrary Meshes

Sebastian Magda,[1†] David Kriegman[2‡]

[1] University of Illinois at Urbana-Champaign
[2] University of California at San Diego

Abstract

While texture synthesis on surfaces has received much attention in computer graphics, the ideal solution that quickly produces high-quality textures with little user intervention has remained elusive. The algorithm presented in this paper brings us closer to that goal by generating high-quality textures on arbitrary meshes in a matter of seconds. It achieves that by separating texture preprocessing from texture synthesis and accelerating the candidate search process. The result of this is a mapping of every triangle in a mesh to the original texture sample with no need for additional texture memory. The whole process is fully automatic, yet still user controllable. It also places no special restrictions on the mesh or on the texture, and the original mesh is not modified in any way. A preprocessed texture sample can be used to synthesize a texture map on any number of meshes.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation display algorithms; I.2.10 [Artificial Intelligence]: Vision and Scene Understanding texture

1. Introduction

Texture synthesis is the process of replicating the statistical and perceptual properties of a user-specified example over a larger surface. Since the surface can have arbitrary topology and the mapping quality is judged using human perception, it is a difficult process to automate. In the next section we describe several proposed methods that attempt to solve that problem. We propose an alternative method that comes with several advantages, particularly separation of texture preprocessing from synthesis, automation, and fast synthesis of quality textures. These make it ideal for texturing a large set of objects (a forest scene containing many trees is a good example), or for interactive mesh modeling. Once a texture is preprocessed, the texture synthesis takes only seconds. A library of preprocessed textures can be stored on a disk and used when needed. Since the algorithm essentially performs texture mapping, it is also very memory efficient, and the rendering process is fast.

Figure 1: *Bunny with synthesized basket texture.*

2. Previous Work

There are many ways to generate a texture, and the field of texture synthesis is very broad. Anything from parametric

† e-mail: magda@uiuc.edu
‡ e-mail: kriegman@cs.ucsd.edu

methods such as reaction-diffusion and solid texturing, to texture-from-sample and tiling can be used depending on the application. We focus only on a small subset of the texture synthesis research, where the goal is to synthesize a texture onto an arbitrary 2-D mesh from a sample texture image.

2.1. Point-based Texturing

Point-based approaches attempt to synthesize textures by coloring a point at a time. To color the current point, the sample texture is searched for a pixel whose neighbors have similar colors to those in the current neighborhood.

Two prime examples of this approach are the works of Wei and Levoy [16] and Turk [14]. These two similar techniques synthesize the texture by vertex coloring. The appearance of the original texture sample is achieved by assigning colors to individual vertices in a densely resampled mesh. The synthesis algorithm is an adaptation of the multi-resolution algorithm for flat textures by Wei and Levoy [15]. In their algorithm, the texture is synthesized from coarse to fine scale using tree-structured vector quantization. This technique works well mostly for smooth textures with small features, and this limitation also applies to synthesis on surfaces. The surface synthesis generalization is achieved by creating a multi-resolution mesh pyramid that corresponds to the texture's image pyramid. Vertex neighborhoods are then flattened and resampled into a regular image grid for candidate search. Texture orientation is obtained from a user-defined or generated surface vector field.

The Wei and Levoy / Turk method has recently been extended to synthesize *bidirectional texture functions* (BTF) by Tong et al. [13]. A BTF is a 6-D function that represents the images of a surface under variation in viewing direction and lighting direction [2]. By capturing both the mesostructure and the reflectance variation of a surface, a BTF can be used to render surfaces with greater realism. Since the dimensionality of BTFs is much larger than 2-D textures, a more compact representation is necessary for synthesis in a reasonable time. Tong et al. achieve that by representing BTFs with *surface textons* and synthesizing the surface BTF from a texton label map. A texton is essentially a fundamental texture element, and a texture can be defined as being a pattern of a finite set of such elements. Leung and Malik [6] introduced the concept of 3-D textons - a generalization of a texton to viewing and lighting variations. As 3-D textons have a very large dimensionality, Tong et al. defined surface textons to be the elements of the inner-product space spanned by the 3-D textons and used them to speed up the neighborhood comparisons. Our algorithm also makes use of textons with the goal of accelerating the synthesis process. However, this is achieved by having a large number of texton clusters and reducing the average size of each cluster.

A slightly different approach is to synthesize the texture directly to a texture atlas, rather than coloring vertices. Consider the texturing algorithms of Ying et al. [17] and Gorla et al. [4]. Both algorithms again use a generalized version of Wei and Levoy's multi-resolution algorithm [15] (Ying also uses Ashikhmin's algorithm [1] as an alternative), but the synthesis is done in the texture atlas space. Although mesh resampling is no longer necessary, both algorithms need a large amount of texture memory space to store the synthesized texture maps.

In general, point-based methods can produce high-quality results for a subset of smooth textures with small feature sizes (with the exception of Ashikhmin's variation of Ying's algorithm that does a better job on other textures). However, these methods either require an additional mesh resampling step or a large surface texture atlas for storing the result. Running times are measured in minutes.

2.2. Patch-based Texturing

Patch-based methods attempt to synthesize larger textures by copying selected regions of the sample texture and, in some way, obscuring region boundaries. By introducing some randomness into the process of patch selection, a high quality texture can be obtained without obvious repetitiveness. Our algorithm fits in this category.

One way to achieve a seamless texture with no obvious periodicity is to use tiling. Neyret and Cani [10] proposed a technique that uses a small set of triangular tiles to texture a mesh. The triangles in the tile set must tile seamlessly with any other triangle for any orientation. This restricts the possible textures to isotropic patterns. The tile set must be either created by the user, or possibly generated using procedural methods. The tiles are randomly assigned to triangles during the texturing process resulting in a seamless texture. In order to minimize texture distortion, the mesh is resampled so that triangles became more uniform.

A solution to patch synthesis of anisotropic textures was proposed by Praun et al. [11]. His method works by randomly pasting and overlapping large texture patches onto the mesh and aligning them with the local surface tangential vector field and the scale. The user-created patches (usually just one) have irregular shape and should produce no visible seams when overlapped. Patch edges are alpha-blended to hide any seams. The resulting texturing can either be synthesized into a texture atlas, or rendered at run time. Unfortunately since texture patches are not matched on their boundary, this method does not do well for more structured textures with sharp discontinuities or textures with low-frequency components.

Soler et al. [12] have recently proposed a multi-scale hierarchical algorithm that texture-maps triangle patches on the sample texture. This method can be viewed as a generalization of the texture quilting algorithm by Efros and Freeman [3]. The algorithm works by stitching small texture patches, while trying to match them on the boundaries. Soler et al. applied this generic algorithm to surfaces. The algorithm

starts by first constructing a hierarchy of face clusters for the mesh. The texturing process begins with the top level patches and recurses into the mesh hierarchy when it fails to find a good fit for the current patch. For each patch, the algorithm searches the sample texture for the best fit with the neighboring patches in a small region along the patch edges. The search process is accelerated by converting to the Fourier space. This reduces the complexity from $O(N^2)$ to $O(NlogN)$, where N is the number of pixels in the texture. The search must be repeated for different patch orientations, and in practice it is done for a fixed number of angles. Although the resulting texture often contains some seams on patch edges, seams can be further reduced by locally readjusting texture coordinates on patch boundaries. This method can produce very good results for isotropic and many anisotropic textures. Very regular textures with sharp edges (brick wall, for example) tend to expose the underlying patch structure, as the texture orientation is essentially fixed over the entire patch (even though patch boundary vertices are adjusted through relaxation). In addition, the user has little control over the texture flow on the surface (other than the choice of the original patch). The running time is closely related to the number of resulting patches. That depends both on the mesh geometry and the structure of the texture sample, and synthesis may require anywhere from a few minutes to a few tens of minutes.

In general, patch-based methods are much more memory efficient, since they do not need a texture atlas. They also tend to do a better job with more complicated textures, although Neyret's and Praun's algorithms pose some additional restrictions. Our work is most similar to Soler's method, as we also use an extension of the texture quilting algorithm.

3. The Algorithm

The basic idea behind the algorithm is to separate texture generation into two independent phases: texture preprocessing and texture synthesis. The concept of dividing the texture generation process into preprocessing and synthesis phases has been successfully applied before to 2-D textures, but not to patch-based texture synthesis [18]. While texture preprocessing is relatively slow, it only has to be done once per texture. At the same time, texture preprocessing makes the actual texture synthesis process fast.

3.1. Texture Preprocessing

The goal of texture preprocessing is to identify sets of pixels in the sample texture whose neighborhoods have similar appearance. One way to accomplish this is to characterize the neighborhood of a pixel as a 2-D texton. In the work of Leung and Malik, the texture is convolved with a stack of N_f carefully chosen filters, resulting in a N_f-dimensional feature vector at each pixel. K-means clustering is performed on the N_f-D feature vectors, and the cluster centers are taken as the set of 2-D textons. Pixels, that end up in the same cluster have similar appearance and are assigned the same texton label.

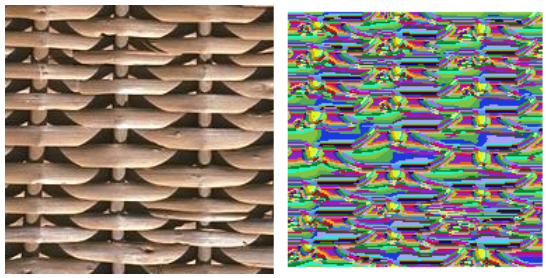

Figure 2: *Texture used in Fig. 1 and the corresponding texton texture* τ. *In the texton texture, the pseudo-colors represent texton labels, and therefore pixels with the same color (texton label) have similar neighborhoods in the original sample. The texture was clustered using 160 textons, and each texton was* 11×11 *pixels.*

Although the filter-stack approach can be used, a much simpler method that uses just a single filter is also sufficient. At each pixel the feature vector is computed by multiplying the local $n \times n$ neighborhood with a single centered Gaussian kernel resulting in a n^2-dimensional feature vector. 2-D textons are the cluster centers after k-means clustering, just as before. Although the resulting feature vectors are larger (for n=11 the vectors are 121-Dimensional versus 48-D in Leung and Malik (For color textures numbers are per color channel.)), the longer computation time is still acceptable. Clustering is only done once per texture during the preprocessing step, and the results can be saved for later use.

The result of texture preprocessing is a set of textons and a 2-D *texton texture* τ. Each position in τ contains an index (texton label) of the texton that best characterizes the pixel's local neighborhood (See Fig. 2).

The *texton texture* is actually stored in a more convenient representation as a *texton bucket array* B. Each texton bucket $B[t]$ stores offsets to all pixels whose neighborhoods are characterized by the same texton t. $B[t][n]$ refers to the n-th pixel offset in $B[t]$. This representation significantly reduces search time during synthesis, as shown in the next section. Additionally, we precompute distances between every pair of textons (using a square difference measure), and store the results in a lookup table.

3.2. Texture Synthesis

The process of synthesis is essentially an extension of the generic texture quilting algorithm [7, 3]. The basic idea is to 'quilt' fragments of the original texture to cover a larger surface in a way that avoids any obvious seams on the quilt

boundaries. All existing 2-D texture quilting methods use square quilting patches for their simplicity, unfortunately this will no longer apply to arbitrary 2-D manifolds in a 3-D space. Since the basic 3-D primitive in computer graphics is a triangle, it makes sense to use triangles as our quilting blocks. The difficulty comes from the fact that, unlike fixed size square patches in 2-D texture quilting, triangles in a typical mesh have different shapes and sizes. The texture orientation will also be different from one triangle to another. Therefore, we need to extend the basic quilting approach to handle arbitrarily shaped patches.

The texture synthesis process takes as input a preprocessed texture sample and a triangular mesh. For each triangle in the mesh, the output is a set of corresponding texture coordinates for the three vertices. This amounts to finding a similarity transformation (rotation, translation, and scale) between each triangle in the mesh and the texture coordinates. For this similarity transformation, the scale is a user input, and in the current implementation it is a constant over all patches. The rotation part of the transformation is given by defining a surface vector field over the mesh, as discussed in Section 3.2.1. The final part of the rigid transformation is the 2-D translation; it is determined by searching for a translation such that the boundaries of the corresponding texture patch have similar appearance to the textured boundaries of the neighboring triangles. This later search is the crux of our synthesis process; it uses the preprocessed texton texture and is described in Section 3.2.2.

3.2.1. Surface to Texture Mapping

As mentioned previously, the similarity transformation between a mesh face and the 2-D texture map includes a rotation, and the angle of rotation is given by a surface *tangential vector field* over the mesh. The vector field represents the local texture orientation. In our implementation we create it by initially propagating seed vertex directions and then locally smoothing the surface vector field, but any of the existing surface vector field synthesis methods [14, 16, 11] can be used as well.

In addition to the surface vector field orientation (corresponding to the local texture orientation), we define the scale at which the texture will map onto the surface. We have chosen to define it in terms of the average triangle (patch) size Z_{patch}, where the triangle size is defined as the length of its longest side. The texture scale will be the ratio of Z_{patch} to the texture size $Z_{texture}$ (length of its shortest side, since the texture can be rectangular):

$$\tau_{scale} = \frac{Z_{patch}}{Z_{texture}} \qquad (1)$$

Since the synthesis algorithm finds a mapping for each triangular face, we compute the surface vector field for each face by averaging the orientation vectors of its vertices. The whole surface vector field computation can be fully automated and is quite fast, taking at most a few seconds for a large mesh with 100,000 vertices.

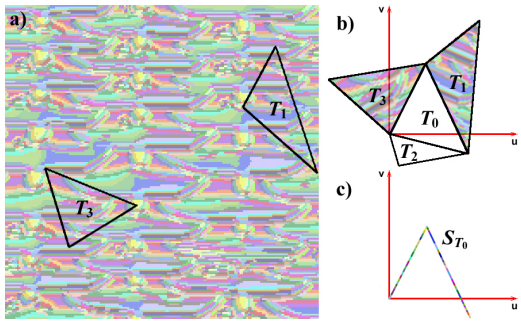

Figure 3: *Texton string generation: Textured neighbors T_1 and T_3 are first transformed from their local texture coordinates (a) to T_0's frame of reference (b). The texton string S_{T_0} is a list of pairs consisting of texton labels lying on the textured edges and their corresponding offsets from T_0's origin vertex (c).*

3.2.2. Finding the Best Patch

The synthesis process starts from a randomly selected triangle, and the texture is grown on the surface from that initial face. At each step, we try to select the triangular patch from our sample texture that best matches along edges with all textured neighbors in the mesh. We continue until the whole surface is covered.

Finding a good patch normally involves comparing some boundary region in the candidate patch to the boundary region of the neighbors already textured. This is generally a time-consuming process. Fortunately it can be made much faster, thanks to the preprocessing step. Since textons already contain the neighborhood information, we no longer need to directly compare boundary regions. In fact, we only need to compare textons lying directly on the edge. The width of the boundary region is already encapsulated in the size of the pixel neighborhoods used to compute textons. By using textons, we automatically reduce the problem from a 2-D region comparison to a 1-D string comparison.

The first step in the texture patch search process is the construction of a *texton string* S along the edges bordering textured triangles. We define S as a list of pairs consisting of texton labels t and texture offsets o: $S_{\{t,o\}}$. For each adjacent textured triangle T_n (where n is 1,2,or 3), we use a line-drawing algorithm to walk along the edge of our to-be-textured triangle T_0 in its texture space (see Fig. 3). We randomly set one of T_0's vertices as the origin. For each point on the edge, we add to S_t the nearest texton label in the *texton texture* τ found by transforming the point to the texture coordinates of T_n. Essentially this process encapsulates a 2-D transformation (rotation plus translation) of an edge from

the local texture coordinates of T_0 to T_n's frame of reference. The corresponding texton offsets (in the T_0's texture space) are stored in the offset list S_o. The process continues until we end up with a *texton string* $S_{\{t,o\}}$ containing texton indices and texture offsets for all edges bordering textured faces.

You may notice that we do not rotate textons around their center pixels back to T_0's texture space. All textons are precomputed for just one texture orientation. This introduces additional errors when matching edges of two faces with different texture orientations. A possible solution would be to either compute rotated textons during synthesis (very slow) or precompute textons for some discrete set of orientations (would require either an enormous texton distance lookup table or a slower distance computation during synthesis). Luckily, the texture orientation for adjacent faces usually does not vary by much. The only exception are faces near surface vector field singularities. These, however, are usually few and, in general, are difficult to texture, even if we try to do it by hand. In addition, since all textons were weighted during preprocessing with a symmetric Gaussian function, the effects of rotation are further diminished.

At this point we could try to search for the best matching patch to cover T_0. A naive way to accomplish that would be to do 1-D *texton string* comparisons in the 2-D texton texture τ. The best-match texture offset x for S would be the one with the smallest error:

$$E[x] = \sum_{k=1}^{length(S)} dist(\tau[x + S_o[k]], S_t[k]) \qquad (2)$$

Notice that we already have precomputed distances between all texton pairs, so $dist(t_1, t_2)$ is just a simple lookup.

There is a way to speed up the search process even more by taking advantage of our *texton bucket array* B, defined earlier. Each bucket $B[t]$ contains offsets to pixels with similarly appearing neighborhoods (same texton label t). Essentially, each bucket $B[t_k]$ contains best candidates for a texton $t_k = S_t[k]$. We can use it to quickly identify a set of good candidate texture offsets for the texton string S.

Let $N[x]$ be the number of texton matches between the string S and the texton texture τ, when S's origin is placed at the texture offset position x (Fig. 4):

```
Let N = {0}
For each element k in S
    For each texture offset n in B[S_t[k]]
        Increment N[B[S_t[k]][n] − S_o[k]]
```

At the end, the best candidate texture offsets for S will be at locations for which the values in N are the greatest. This faster search method is very crude by nature, as it only considers exact texton matches. We still need to use Eq. 2 to find the best choice among the top m candidates from N by computing $E[x]$ for each of the top locations. Again, we use the precomputed texton distance lookup table.

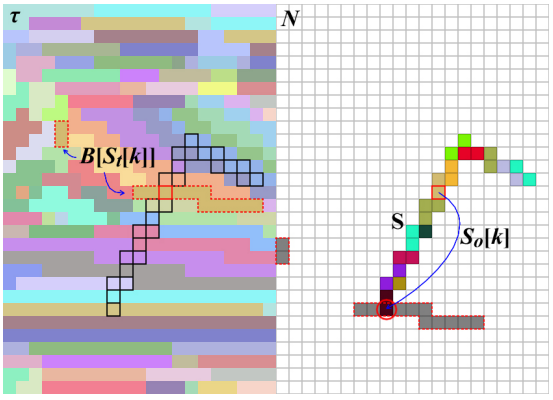

Figure 4: *Searching for the best offset location for the texton string S (on the right) in the texton texture τ (magnified τ fragment shown on the left with the corresponding fragment of the counter array N on the right): For each position k in S, the algorithm iterates through the list of matching texton offsets $B[S_t[k]]$ and increments N at the string's origin. (The increment location $N[B[S_t[k]][n] - S_o[k]]$ for this example is marked with a red circle.)*

Once the top candidate is selected, we set the texture coordinates for each vertex of T_0. Since the texture is mapped directly onto each triangle using the same texture scale τ_{scale}, the process will not introduce any distortion to the texture. The only artifacts present will be seams along edges where the algorithm failed to find a good match.

3.3. Rendering

Displaying the textured mesh is really no different than any other texture-mapped object. All we need to store in texture memory is simply the original texture sample. We have assigned texture coordinates to vertices of every triangle during the texture synthesis process and they map directly into the sample texture. If the displayed triangles are relatively small on the screen, either due to a dense mesh, or because the object is placed far from the viewer in the 3-D view, any existing edge discontinuities will be hardly noticeable. The seams will also be less obvious for isotropic textures and those with less structure. For other cases, where better image quality is desirable, edge discontinuities can be hidden by *edge blending*. Figure 5 shows an extreme closeup of a textured object demonstrating the effects of edge blending.

3.3.1. Edge Blending

The idea behind edge blending is to smoothly blend textures over edges to the adjacent neighboring triangles. Since each

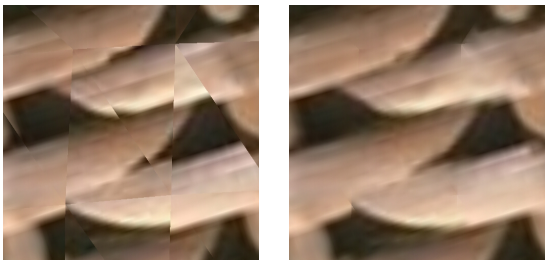

Figure 5: *Extreme close-up of Fig.1 showing the effects of edge blending: textured mesh triangles without blending (left) and with blending (right).*

triangle in a complete mesh has three neighbors, this will require blending its texture with the three neighboring textures. Blending is accomplished by multiplying the neighboring textures with a smooth blending function that decreases toward the corner opposite to the edge (Fig. 6). During the texture synthesis process, we store an additional set of texture coordinates at each vertex for blending with the texture across the edge opposite to the vertex. We end up storing two sets of texture coordinates per vertex for each triangle.

The blending process is very straightforward, requiring several texture blending operations and can be easily accomplished through multi-texturing. Current off-the-shelf consumer graphics boards still require several rendering passes to accomplish this (currently between 2-4 passes), but as the next generation hardware will support more texturing units, it will become feasible to do it in just one pass. Since runtime blending currently carries a performance penalty, an alternative solution would be to blend the textures off-line and store them in a texture atlas. This will of course carry a storage penalty instead.

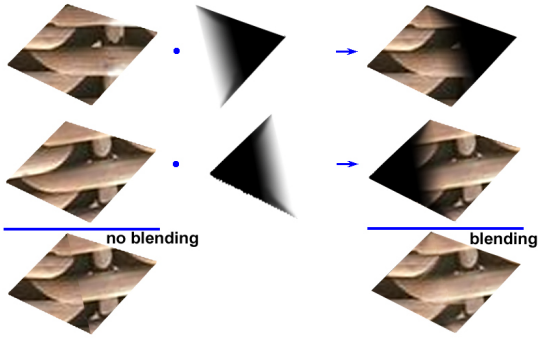

Figure 6: *The blending process across an edge: The texture from a neighboring triangle is extended over the edge and modulated with a blending function. The result is then combined with the underlying texture.*

3.3.2. Blending Metric

In order to reduce the cost of real-time blending, we can specify a quality metric to decide whether to blend an edge or not. The idea is to combine the edge's visibility measure with the edge's texture continuity measure. The edge's visibility is simply a measure of the projected edge length in the screen space. The longer the projected edge, the more likely it will be noticed by the viewer. While the correct thing to do would be to perform a projective transformation of an edge to the screen, this approach would require dealing with projection singularities and is rather computationally expensive. In our case it is sufficient to simply use the Euclidean distance from the edge's midpoint m to the viewer v to scale the edge length d projected to camera coordinates:

$$d_{screen}(d,m,v) = \lambda \frac{d}{\|v-m\|} \tag{3}$$

where $\lambda = \frac{w}{\varphi}$ is ratio of w pixels across the field of view φ. d_{screen} will give the approximate length of an edge in pixels. This screen-space metric is commonly used in other applications, such as 3D terrain visualization [8].

The texture continuity error E_t across an edge can be expressed as a Euclidean distance of color texture values along the edge for the two adjacent textures. Notice that we do not want to make neighborhood comparisons here, as we are only interested in differences directly on the edge. The error can be precomputed and stored at each edge.

The resulting blending metric can be expressed as the texture continuity error scaled by the projected edge length:

$$\Phi(d,m,v) = \lambda E_t \frac{d}{\|v-m\|} \tag{4}$$

Edge blending will be applied when Φ exceeds a user-specified threshold for quality.

4. Results

We have tested the algorithm on a variety of textures and models. Figure 8 shows several examples for different combinations of meshes and textures. Each mesh contained between 5000 and 8200 faces. Texture sizes varied between 64x64 and 256x256. Tables 1 and 2 show sample running times for preprocessing and for texture map synthesis.

The number of textons used for a particular texture influences the synthesis quality. Using too few textons does not capture enough texture variations, while having too many causes some region similarities to be missed. We have determined that the number of textons necessary for a particular texture can be simply related to the texture size. Setting this number to be about the square-root of the number of pixels

Table 1: *Texture preprocessing timings recorded on a 1.3GHz Athlon for some examples shown in Fig. 8. For all cases, clustering was stopped when 99% of pixels did not change cluster during the last iteration.*

Texture size	Textons used	Texton neighborhood size	Preprocessing time
64x64	60	11	7 sec
128x128	100	11	38 sec
200x200	160	11	7 min, 12 sec
256x256	200	11	16 min, 37 sec

Table 2: *Texture map synthesis timings recorded on a 1.3GHz Athlon for the bunny model shown in Fig. 8. The model contains 8192 faces and the running time is linear in the number of faces.*

Texture size	Textons used	Synthesis time
64x64	60	$<$1 sec
128x128	100	2 sec
200x200	160	4 sec
256x256	200	7 sec

(see Tab. 2) produces good results for the vast majority of textures.

Synthesis quality is quite similar to the quality of the generic 2-D quilting algorithm [7, 3]. In general, it is better for larger texture samples, since the quilting process is able to find better candidates in a larger sample. Since the algorithm cannot introduce any texture distortion, it is also more effective for textures that are less regular. It is simply impossible to maintain the structure of a very regular texture (a checkerboard, for example) without introducing distortions. Therefore there are some noticeable edge discontinuities for the bunny with the brick texture (Fig. 8). Since the algorithm is essentially an extension of quilting, it is similarly sensitive to the average feature size present in the texture. As in the quilting algorithm, it is desirable for the quilting block size (average triangle size in our case) to be about the same as the average feature size. (Although we do not disscuss it here, it's quite reasonable to extend the algorithm to quilt patches of triangles in order to avoid this problem). Despite these, the results are quite pleasing for the vast majority of common textures, even in texture areas near singularities (see Fig. 7).

Figure 7: *An example of texture synthesis around a surface vector field singularity point.*

5. Discussion and Future Work

This paper introduced a fast, patch-based method for automatically texturing triangular meshes. The method is fast because texture samples are preprocessed to characterize neighborhoods of the sample, to represent a neighborhood in terms of a finite texton vocabulary, and to compute a similarity measure between all pairs of textons. Thanks to the texton representation we are able to reduce the candidate patch search to a simple 1D string comparison. At this point in time, synthesis requires a few seconds for moderately sized meshes, and so real-time synthesis, perhaps on meshes with deforming geometry and topology, should soon be achievable in either software or using the forthcoming generation of graphics boards. As seen by way of example, meshes are textured with few noticeable seams or artifacts. Typically these arise when texturing a triangle for which its three neighbors have already been textured, and no triangular patch within the texture sample matches well along the three boundaries. This is a consequence of sacrificing some quality in favor of a much faster synthesis time, by performing a greedy search for patches. Alternative search mechanisms which exploit the preprocessed representations and the surface geometry will possibly improve the synthesis quality.

Finally, we are extending the underlying synthesis method to other classes of texture data besides color texture images, particularly polynomial texture maps and bidirectional texture functions. This will allow the surfaces to be rendered with textures whose appearance varies with changes of viewpoint and lighting (e.g., moss, grass, sponge, pebbles, etc.). With suitable preprocessing to define a texton vocabulary and an appropriate means to render the specific class of tex-

ture data, the method of Section 3.2.2 can be used to determine a mapping between each mesh triangle and the texture data.

Acknowledgements

Special thanks to the attendees of the UIUC Graphics Seminar as well as P. Belhumeur and M. Koudelka for their comments and suggestions.

References

1. M. Ashikhmin. Synthesizing natural textures. *Proceedings of the 2001 symposium on Interactive 3D graphics*, pp. 217–226, 2001.
2. K.J. Dana, B. van Ginneken, S.K. Nayar, and J.J. Koenderink. Reflectance and Texture of Real-World Surfaces. *ACM Trans. on Graphics*, **18**:1–34, 1999.
3. A.A. Efros and W.T. Freeman. Image quilting for texture synthesis and transfer. *SIGGRAPH 2001*,pp. 341–346, 2001.
4. G. Gorla, V. Interrante, and G. Sapiro. Growing fitted textures. *SIGGRAPH 2001 Sketches and Applications*, pp. 191, 2001.
5. D. Hochbaum and D. Shmoys. A best possible heuristic for the k-center problem. *Mathematics of Operations Research*, **18**(2):180–184, 1985.
6. T. Leung and J. Malik. Representing and recognizing the visual appearance of materials using 3d textons. *International Journal of Computer Vision*, **43**(1):29–44, 2001.
7. L. Liang, C. Liu, Y. Xu, B. Guo, and H. Shum. Real-time texture synthesis by patch-based sampling. *ACM Transactions on Graphics (TOG)*, **20**(3):127–150, 2001.
8. P. Lindstrom and V. Pascucci. Terrain Simplification Simplified: A General Framework for View-Dependent Out-of-Core Visualization. *IEEE Transactions on Visualization and Computer Graphics*, **8**(3):239-254, 2002.
9. X. Liu, Y. Yu, and H. Shum. Synthesizing bidirectional texture functions for real-world surfaces. *SIGGRAPH 2001*, pp. 97–106, 2001.
10. F. Neyret and M. Cani. Pattern-based texturing revisited. *SIGGRAPH 1999*, pp. 235–242, 1999.
11. E. Praun, A. Finkelstein, and H. Hoppe. Lapped textures. *SIGGRAPH 2000*, pp. 465–470, 2000.
12. C. Soler, M. Cani, and A. Angelidis. Hierarchical pattern mapping. *SIGGRAPH 2002*, pp. 673–680, 2002.
13. X. Tong, J. Zhang, L. Liu, X. Wang, B. Guo, and H. Shum. Synthesis of bidirectional texture functions on arbitrary surfaces. *SIGGRAPH 2002*, pp. 665–672, 2002.
14. G. Turk. Texture synthesis on surfaces. *SIGGRAPH 2001*, pp. 347–354, 2001.
15. L. Wei and M. Levoy. Fast texture synthesis using tree-structured vector quantization. *SIGGRAPH 2000*, pp. 479–488, 2000.
16. L. Wei and M. Levoy. Texture synthesis over arbitrary manifold surfaces. *SIGGRAPH 2001*, pp. 355–360, 2001.
17. L. Ying, A. Hertzmann, H. Biermann, and D. Zorin. Texture and Shape Synthesis on Surfaces. *Proceedings of the Eurographics Symposium on Rendering 2001*, pp. 301–312, 2001.
18. S. Zelinka and M. Garland. Towards real-time texture synthesis with the jump map. *Proceedings of the Eurographics Symposium on Rendering 2002*, pp. 99-104, 2002.

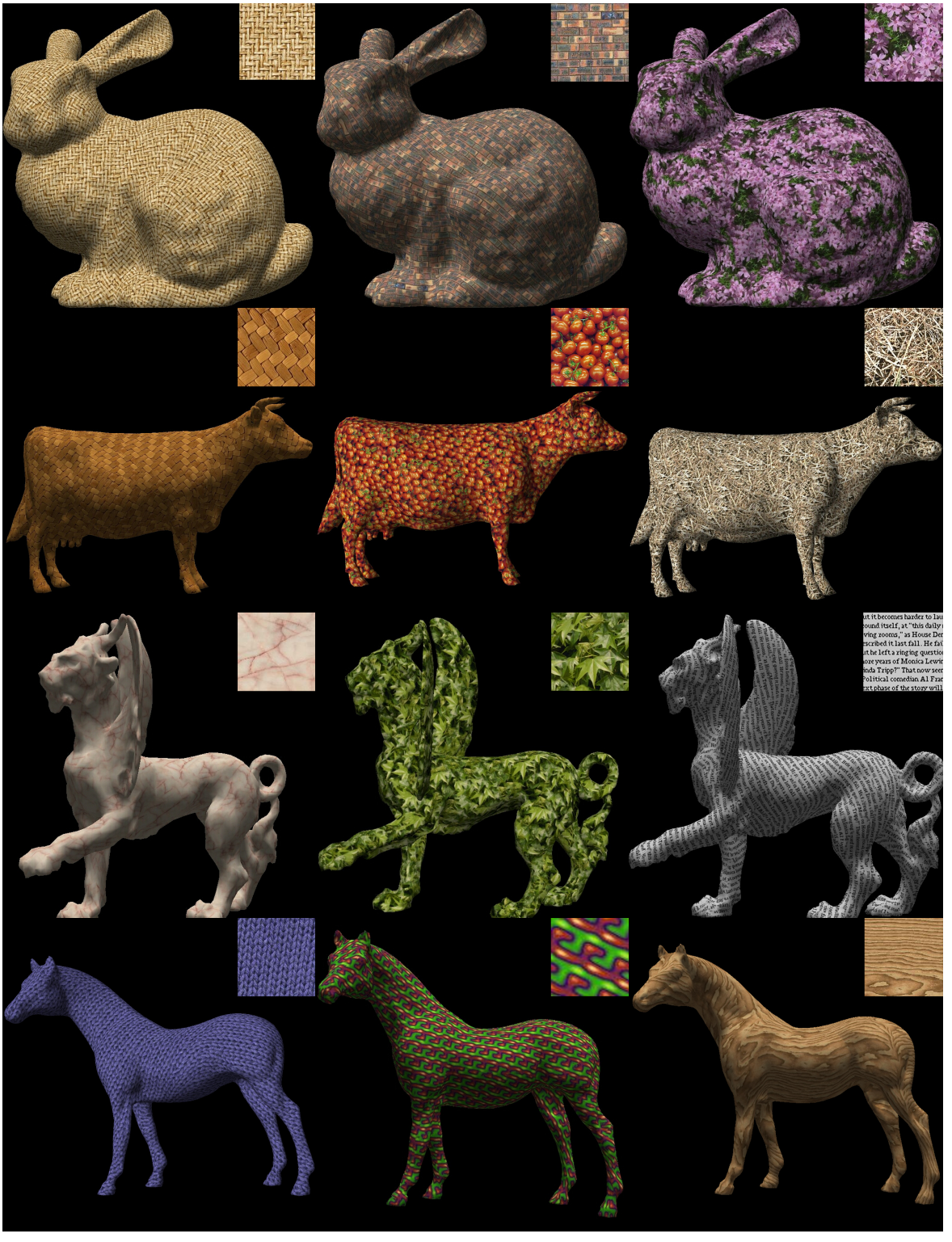

Figure 8: *Examples of texture synthesis results for various meshes and texture samples. Texture samples shown had different sizes and were rescaled for the figure.*