Decorating Surfaces with Bidirectional Texture Functions

Kun Zhou, Peng Du, Lifeng Wang, Yasuyuki Matsushita, Jiaoying Shi, Baining Guo, and Heung-Yeung Shum

***Abstract*— We present a system for decorating arbitrary surfaces with bidirectional texture functions (BTF). Our system generates BTFs in two steps. First, we automatically synthesize a BTF over the target surface from a given BTF sample. Then we let the user interactively paint BTF patches onto the surface, such that the painted patches seamlessly integrate with the background patterns. Our system is based on a patch-based texture synthesis approach known as quilting. We present a graphcut algorithm for BTF synthesis on surfaces, and the algorithm works well for a wide variety of BTF samples, including those which present problems for existing algorithms. We also describe a graphcut texture painting algorithm for creating new surface imperfections (e.g., dirt, cracks, scratches) from existing imperfections found in input BTF samples. Using these algorithms we can decorate surfaces with real-world textures that have spatially-variant reflectance, fine-scale geometry details, and surfaces imperfections. A particularly attractive feature of BTF painting is that it allows us to capture imperfections of real materials and paint them onto geometry models. We demonstrate the effectiveness of our system with examples.**

***Index Terms*— bidirectional texture function, texture synthesis, interactive surface painting.**

I. Introduction

Texture mapping was introduced in [5] as a way to add surface detail without adding geometry. Texture-mapped polygons have since become the basic primitives of the standard graphics pipeline. Unfortunately, texture-mapped surfaces do have a distinctive look that sets them apart from reality: they cannot accurately respond to changes in illumination and viewpoint. Real-world surfaces often are not smooth but covered with textures that arise from both spatially-variant reflectance and fine-scale geometry details known as *mesostructures* [16]. Real surfaces also exhibit *imperfections*, e.g., dirt, cracks, and scratches, which usually result from rather complicated physical processes. Capturing these surface characteristics is a challenging goal for computer graphics.

To bring us closer to that goal, we develop a system for decorating arbitrary surfaces with BTFs [6]. Our system supports two high-level texturing operations: tiling and painting. Given a BTF sample, the tiling operation automatically synthesizes a BTF that fits the target surface naturally and seamlessly. The BTF can model spatially-variant reflectance and mesostructures. Moreover, the BTF can be measured from real materials. Thus the tiling operation provides a convenient way to cover a surface with fairly realistic textures.

Manuscript received January 20, 2002; revised August 13, 2002.

K.Zhou, L.Wang, Y.Matsushita, B.Guo and H.-Y. Shum are with Microsoft Research Asia, 3F, Beijing Sigma Center, No. 49, Zhichun Road, Haidian District, Beijing 100080, PRC. E-mail: {kunzhou, lfwang, yasumat, bainguo, hshum}@microsoft.com.

P.Du and J.Shi are with Zhejiang University, Hangzhou 310027, PRC. This work was done while P.Du was intern at Microsoft Research Asia. E-mail: {dupeng, jyshi}@cad.zju.edu.cn.

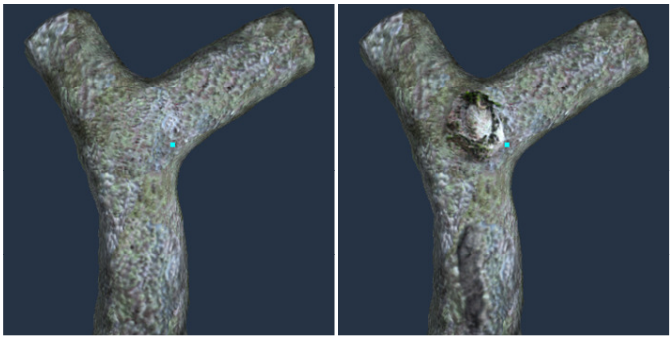

Fig. 1. Decorating arbitrary surfaces using BTFs. Left: Result of BTF synthesis. Right: Result after introducing surface imperfections into the homogeneous BTF shown on the left.

The painting operation further enhances realism by introducing imperfections and other irregular features as shown in Fig. 1 and Fig. 7. This operation is valuable because BTFs generated by our synthesis algorithm, as well as by most other synthesis algorithms, are homogeneous across the whole surface. With the painting operation, we can break this global homogeneity by adding irregular local features. In particular, we can capture imperfections of real materials and paint them onto the surface, such that the painted imperfections fit in seamlessly with the background patterns. 3D painting is a well-established technique for creating patterns on surfaces [13]. BTF painting extends traditional techniques in two ways. First, it provides a way to achieve superior realism with BTFs measured from real materials. Second, BTF painting reduces the demand on artistic talents and the tedium of creating realistic imperfections.

There are two main challenges in developing our system. First, BTF synthesis on surfaces remains hard for many BTF samples. The most difficult problem is maintaining mesostructures of the input BTF sample. An existing algorithm [26] addresses this problem with partial success. However, synthesizing a BTF pixel-by-pixel as in [26] leads to fundamental problems (e.g., L2-norm being a poor measure for perceptual similarity [1]) in maintaining mesostructures. An alternative is to synthesize BTFs by copying patches of the input sample (i.e., quilting [8]). Since mesostructures are copied along with the patches, this approach is particularly effective for maintaining mesostructures. Unfortunately, patch seams still present a problem for BTFs. Although techniques exist for hiding seams in surface textures, these techniques do not generalize well to BTFs. For example, [21] used blending to hide patch seams, but blending will create inconsistent mesostructures in the blended areas (e.g., see [26]).

We present an algorithm for quilting BTF patches by using graphcut [3], [17] to minimize the discontinuity across

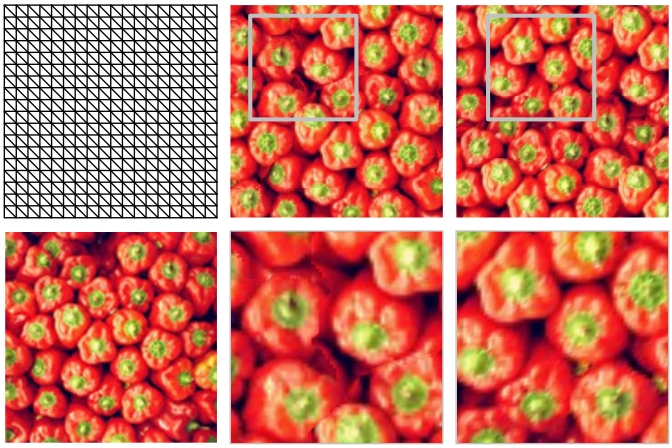

Fig. 2. A straightforward extension of graphcut to [25] leads to suboptimal quality when the synthesis results are viewed from a close distance, as the results in the middle column demonstrate. These results are obtained by incorporating graphcut into [25], which works directly on the original input mesh (left column, top). Notice the patch seams. The right column shows results produced by the GIM-based sampling approach. The input texture sample is shown in the left column (bottom).

patch seams on arbitrary surface meshes. A straightforward extension of graphcut to *hierarchical pattern mapping* [25] can be used to generate texture coordinates for each triangle of the mesh. However this would lead to textures that reveal patch seams when viewed close up. This could be a potential problem for any attempt to apply graphcut on surfaces. Our algorithm solves this problem by densely re-sampling the surfaces using geometry images [12]. We call this approach GIM-based sampling. Specifically, given an input mesh we create a dense mesh by densely sampling the input mesh using multi-chart geometry images (MCGIM) [24]. The texture synthesis is accomplished by using graphcut to generate texture coordinates for each vertex of the dense mesh. Because a texture value is computed for each vertex of the dense mesh, the synthesized textures can be viewed from any distance just like those obtained with pixel-based algorithms (e.g., [26]). Fig. 2 compares the results of the GIM-based sampling approach with a straightforward extension of graphcut to [25].

The second challenge we face is finding a user-friendly way to introduce irregular features into a background pattern. Graphcut techniques suggest a straightforward approach to merging a foreground feature with the background pattern, i.e., we simply constrain the feature to a desired location and use graphcut to find the merging seam. However, this approach only supports verbatim copying of *existing features*. To allow the user to generate *new features*, we formulate the constrained graphcut problem for texture and BTF painting. For texture synthesis, only smoothness energy is considered for finding seams that minimize discontinuity [17]. For texture painting, both smoothness and constraint energies are used so that the user's specification of the new feature is incorporated into the graphcut problem as constraints. Generating a new feature with the graphcut painting algorithm is easy. The user only needs to specify the rough shape and location of the desired new feature; our system will synthesize the actual feature and have it merged seamlessly with the background pattern.

We demonstrate the effectiveness of our system with a variety of examples. Note that our techniques work for ordinary color textures, which may be regarded as BTFs with a single viewing direction and a single lighting direction.

II. Related Work

Texture Synthesis: Algorithms for synthesizing textures on surfaces can be divided into two categories. The first category [11], [27], [29], [32], [26], [33], [34] is based on per-pixel non-parametric sampling [9], [28]. Per-pixel sampling is susceptible to the problems caused by the fact that the commonly used L2-norm is a poor measure of perceptual similarity. For this reason algorithms in this category have difficulty maintaining texture patterns with certain types for textures [1], [14], [34]. Currently there is no general solution, but remedies exist for various specific scenarios [1], [34].

Algorithms in the second category synthesize textures by copying patches of the input texture sample. Since texture patterns are directly copied onto the target surface, these algorithms are not seriously affected by the issue with the L2-norm. Earlier algorithms randomly paste patches and use alpha-blending to hide patch seams [23], [31]. Quilting [8], [19], [25], [21], [17] generates significantly better results by carefully placing patches to minimize the discontinuity across patch seams. After placing patches, [19], [21] simply use alpha-blending to hide patch seams, while [8], [17] further enhance the smoothness across the seams by searching for the "min-cut" seams.

For image textures, [17] recently demonstrated that quilting with graphcut produces arguably the best results on the largest variety of textures. In this paper we show how to perform graphcut on surfaces.

Decorating Surfaces: For decorating implicit surfaces, [22] proposed a set of texturing operations which include tiling and positioning of small images onto surfaces. [15], [10], [4] presented several 3D surface painting systems. For decorating surfaces with imperfections, existing techniques (e.g., [2], [30], [7] mostly focus on generating synthetic surface imperfections. Complementary to their approaches, our techniques synthesize imperfections from real-world and synthetic samples.

III. BTF Synthesis

Given a mesh M and input BTF sample, we first build a dense mesh M_d and then synthesize a BTF value for each vertex of M_d by quilting patches of the input sample on M_d.

A. GIM-based Sampling

As mentioned, it is possible to incorporate graphcut techniques into a synthesis algorithm that works directly on the original input mesh. One such algorithm is pattern mapping [25], which generates texture coordinates for each triangle of the input mesh. Unfortunately, the synthesis results are not ideal when viewed from a close distance, as Fig. 2 demonstrates. With the pattern mapping approach, two adjacent triangles on the mesh may be far apart in the texture space. During texture synthesis, textures on two such triangles are matched at a fixed sampling resolution (this resolution is a

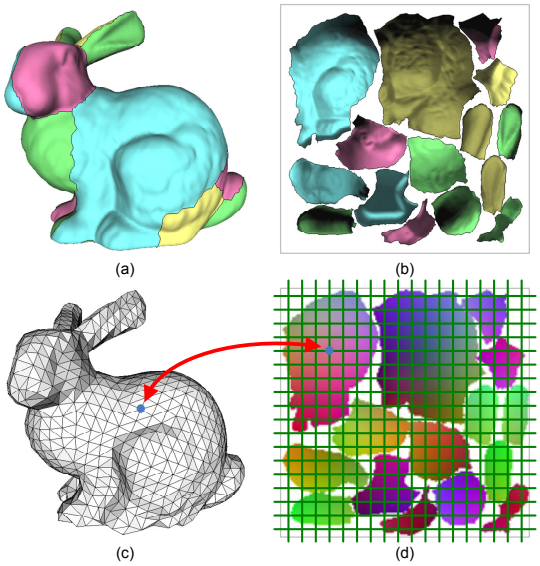

Fig. 3. Construction of a dense mesh for BTF synthesis. (a) and (b) Texture atlas. (c) and (d) Multi-chart geometry image mesh (MCGIM), which samples a surface using a regular grid. Each mesh vertex is a grid point.

While there are still un-synthesized vertices in M_d **do**
  pick the most constrained un-synthesized vertex v
  build a work patch $P(v)$ centered at v
  found_work_patch = FALSE
  While *found_work_patch* == FALSE **do**
    If $P(v)$ lies inside a single chart
      found_work_patch = TRUE
      obtain the work image $I(v)$ from the chart
    Else
      parameterize $P(v)$ using LSCM
      If *parameterization_distortion* $\leq$ threshold T_d
        found_work_patch == TRUE
        create the work image $I(v)$ by resampling $P(v)$
      Else
        decrease the size of $P(v)$
  found_cut = FALSE
  While *found_cut* == FALSE **do**
    patch matching in $I(v)$
    patch fitting in $P(v)$ using graphcut
    If the optimal seam cost is below threshold T_c
      found_cut = TRUE
    Else
      decrease the size of $I(v)$ and $P(v)$
  get BTF values for vertices of $P(v)$

Fig. 4. Pseudo-code for texture quilting on the dense mesh.

parameter of the synthesis algorithm). When synthesis results are viewed at a higher resolution, the seam between the two triangles becomes noticeable.

Our GIM-based sampling approach provides an effective and general solution for applying graphcut to surfaces. From the input mesh M, we construct the dense mesh M_d as an MCGIM [24]. The MCGIM uses a texture atlas to resample M and zippers chart boundaries to obtain a "watertight" mesh M_d. We create the texture atlas using the method proposed in [35], which partitions a mesh into fairly large charts and parameterizes the charts with minimized stretch. Fig. 3 provides an example of MCGIM. The sampling rate of the MCGIM is a user-specified parameter. For examples reported in this paper, the sampling rate is either 512×512 or 1024×1024.

The main advantage of the dense mesh M_d is the one-to-one correspondence between vertices of M_d and pixels in the texture atlas. Because of the correspondence, synthesizing BTF values for vertices on M_d is the same as that for pixels of the charts of the texture atlas. The pixel-vertex correspondence makes it easy to flatten a large patch of M_d without introducing noticeable texture distortion. In fact, for a pixel in the interior of a chart, the pixel's 2D neighborhood directly provides neighboring samples and a local flattening. We will use the pixel-vertex correspondence to simultaneously work on a surface patch of M_d and its corresponding image and switch freely between the two.

B. Quilting on the Dense Mesh

From the input of an MCGIM M_d and a BTF sample S, the pseudo-code shown in Fig. 4 produces BTF values of all vertices of M_d.

The algorithm synthesizes the surface BTF by copying patches of the sample BTF. Each time, the next BTF patch P_b to be generated is around an un-synthesized vertex v which is the most constrained, i.e., having the largest number of immediate vertex neighbors that are synthesized.

Work Patch/Image: To compute the next BTF patch P_b, we start by building a work patch $P(v)$ centered at v. We find $P(v)$ by a breadth-first traversal starting at v. The number of levels in the breadth-first traversal is defined as a user-supplied parameter called the work patch size r_w. For a 512×512 MCGIM M_d, a typical working patch size is $r_w = 32$. Intuitively, we can think of the work patch $P(v)$ as a disk of radius r_w.

From the work patch $P(v)$ we derive the work image $I(v)$ using a continuous parameterization of $P(v)$ by considering two cases. The first case is simple. If $P(v)$ lies completely inside a chart of the MCGIM M_d, then $I(v)$ is simply the corresponding sub-image of the chart image and no parameterization or resampling is necessary. The second case is more complex. If $P(v)$ crosses chart boundaries, we parameterize $P(v)$ using the least squares conformal mapping (LSCM) [18] and resample the result to obtain the work image $I(v)$. To minimize the texture distortion caused by LSCM we monitor the area distortion by a threshold T_d, which is set to 4 in our current implementation. For resampling, we set the sampling step to the average edge length of the work patch. If a sampling point p is located in a triangle with three synthesized vertices, the BTF value at p is interpolated from these vertices and the sampling point is marked as synthesized. Otherwise p is marked as un-synthesized. This way we obtain a partially synthesized work image $I(v)$.

Patch Matching: Having built the work patch and image, we can now calculate the next BTF patch P_b using graphcut. For quilting with graphcut, the main tasks are patch matching and patch fitting [17]. Patch matching places a candidate patch (the input sample) over a target area by comparing the candidate patch and the synthesized pixels in the target area. Patch fitting applies graphcut to select a piece of the

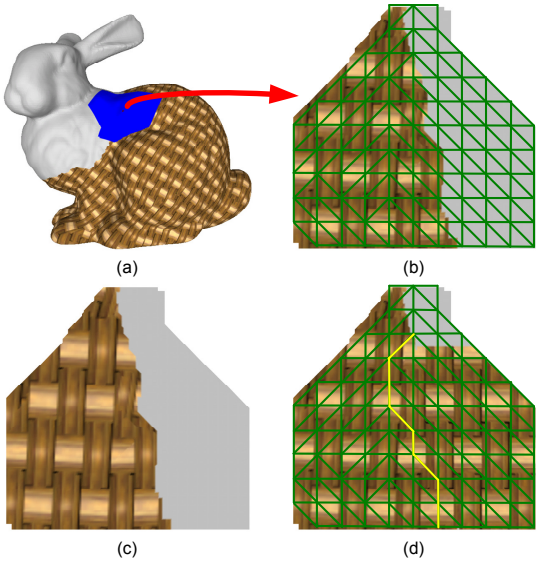

Fig. 5. BTF synthesis using a work patch and a work image. (a) Finding a work patch on the surface. (b) The initial work patch. (c) Work image with synthesized and un-synthesized pixels. (d) Work patch after fitting a new texture patch, with the graphcut seam shown in yellow.

candidate patch to be the next synthesized BTF patch P_b. We perform patch matching in the work image $I(v)$ using the entire patch matching method proposed in [17]. The sum-of-squared-differences (SSD) cost for the BTF is normalized with the area of the overlap region between the input sample and the work image, and the patch placement with the minimal cost is selected for patch fitting. The SSD-based search is accelerated using FFT as in [17].

A technical difference between our patch matching and that of [17] is that we use both translations and rotations, whereas they only use translations. The reason for this difference is that surface quilting involves a vector field. The vector field is important for orienting anisotropic textures on surfaces and for BTF rendering, which requires a local frame at each surface point [26]. We designed a user interface to let the user specify vectors at a few key faces and then interpolate these vectors to other faces using radial basis functions [23].

During patch matching, we calculate an average orientation for the work image using the vector field on the work patch. The average orientation is then used to align the input BTF sample S for patch matching. In our system we pre-compute 72 rotated versions of S and select the best for patch matching.

Patch Fitting: After the input sample S is placed over the work patch $I(v)$ by patch matching, we can easily map S onto the work patch $P(v)$ using the correspondence between $I(v)$ and $P(v)$. Now every vertex of $P(v)$ covered by the mapped and rotated input sample S gets a new BTF value, and every synthesized vertex of $P(v)$ also has an old BTF value. Based on these old and new values, we apply graphcut to select a piece of S by regarding vertices of $P(v)$ as graph nodes and the edges of $P(v)$ as graph edges. The BTF values of the selected piece are copied to the vertices of $P(v)$ if the seam cost is below a prescribed threshold T_c. Note that old seams from previous graphcut steps can be accounted for the same way as in [17].

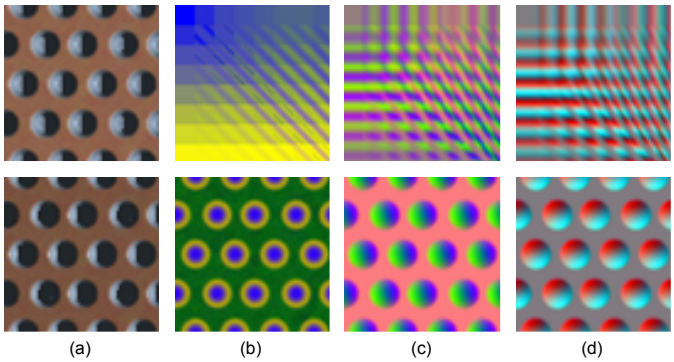

Fig. 6. Singular value decomposition of a BTF. (a) Two images of the BTF with different viewing and lighting directions. (b) - (d) The three most significant eigen pixel appearance functions (PAF; top row) and geometry maps (bottom row). A 4D PAF is packed into a 2D image, in which each row corresponds to the varying light direction and each column corresponds to the varying view direction.

Hierarchical Search: For high-quality results we only accept a piece selected by graphcut if the seam cost is below the threshold T_c. If the seam cost exceeds T_c we decrease the work patch size r_w by a constant factor (which is set to 0.8 for examples reported in this paper) and repeat the search. We decrease r_w as many times as needed until r_w reaches the minimum size of 4. We employ the same hierarchical search strategy to control the area distortion of the LSCM parameterization when we build the work patch. Other researchers (e.g. [25]) used similar ideas for texture synthesis.

Handling BTF Values: The BTF is a 6D function $f(\mathbf{x}, \mathbf{v}, \mathbf{l})$, where $\mathbf{x} = (x, y)$ is the texture coordinate. $\mathbf{v} = (\theta_v, \phi_v)$ and $\mathbf{l} = (\theta_l, \phi_l)$ are the lighting and viewing directions in spherical coordinates. For each texel $\mathbf{x} = (x, y)$, the BTF value is a 4D function discretized into a high-dimensional vector. This high dimension requires careful treatment when storing BTFs and calculating SSD costs for patch matching and fitting. For efficient storage of various BTFs including rotated copies of the input sample and the synthesized surface BTFs, we only store the texture coordinates as in [26]. When necessary the actual BTF value is retrieved from the input sample using texture coordinates.

For efficient SSD cost calculation, we factorize the BTF as a sum of products of 2D and 4D functions using singular value decomposition (SVD) as in [20].

$$f(\mathbf{x}, \mathbf{v}, \mathbf{l}) \approx \sum_{i=1}^{n} g_i(\mathbf{x}) p_i(\mathbf{v}, \mathbf{l}),$$

where $g_i(\mathbf{x})$ is called a geometry map and $p_i(\mathbf{v}, \mathbf{l})$ is called an eigen point appearance function (PAF). Fig. 6 shows a SVD factorization of a 128×128 BTF sample (the viewing and lighting resolution is $12 \times 8 \times 12 \times 8$). The geometry maps depend on texture coordinates only, whereas the PAF is a function of the viewing and lighting directions. [20] developed an algorithm which synthesizes a surface BTF using the corresponding geometry maps. We adopt this algorithm for surface quilting with $n = 40$ and calculate the SSD costs with low-dimensional vectors.

For ordinary color textures, we store vertex colors directly into M_d and thus create a texture map (usually of size $512 \times$

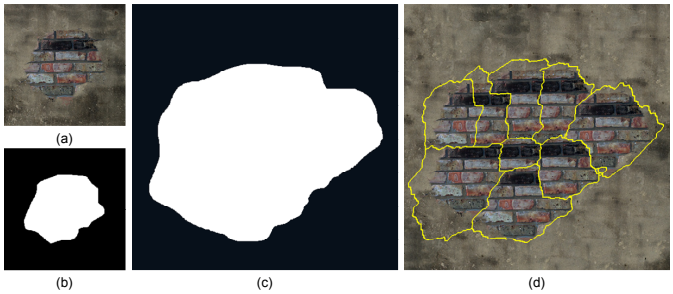

Fig. 7. Constructing a constrained quilt. (a) Input texture sample. (b) Specifying a foreground feature (shown in white) in the input sample. (c) User-specified foreground region F (shown in white) in the output texture. (d) A constrained quilt with patch boundaries (cut seams) shown in yellow.

512 or 1024×1024). With this texture map, the synthesized surface texture can be rendered using the standard texture mapping pipeline.

IV. BTF Painting

Suppose that we are given a sample texture S with a foreground feature over a background pattern. Suppose that the background pattern has been synthesized onto the target mesh M. Our BTF painting system lets the user interactively synthesize new foreground features on M by roughly specifying their shapes and locations. Prior to this synthesis, we assume that the user has roughly specified the foreground feature in S.

The key ingredient of the painting system is our graphcut texture painting algorithm, which minimizes both smoothness and constraint energy to build a quilt of patches.

A. Graphcut Texture Painting

Fig. 7 shows the construction of a constrained quilt Q. We assume that the sample texture S has been partitioned into two parts: the background pattern S_b and foreground feature S_f. We also assume that the user has specified a foreground region F of the target mesh M. The constrained quilt $Q(F)$ is a quilt consisting of patches of M such that $Q(F)$ completely covers F and each patch of $Q(F)$ is directly texture-mapped from a patch in the input sample S. The constraint on $Q(F)$ is that F should be textured only by the foreground feature S_f.

To construct the constrained quilt $Q(F)$ using graphcut, we encode the user-specified constraints as *constraint functions*. We define a function m_0 on the target mesh M such that $m_0(p) = 1$ if point p belongs to F and $m_0(p) = 0$ otherwise. Similarly, for every patch P of the constraint quilt $Q(F)$ we define a function m_P such that $m_P(p) = 1$ if point p belongs to the foreground feature and $m_P(p) = 0$ otherwise. With these functions defined, we introduce constrained graphcut, which is the core part of graphcut texture painting.

Constrained Graphcut: For simplicity we describe constrained graphcut for image textures. Consider two overlapping constrained patches A and B as shown in Fig. 8 (a). Each pixel p of patch A has a texture value $f_A(p)$ and a constraint value $m_A(p)$. Similarly, each pixel p of patch B has a texture value $f_B(p)$ and a constraint value $m_B(p)$. Finally, for every pixel p of the overlapping region, we have a user-specified constraint value $m_0(p)$.

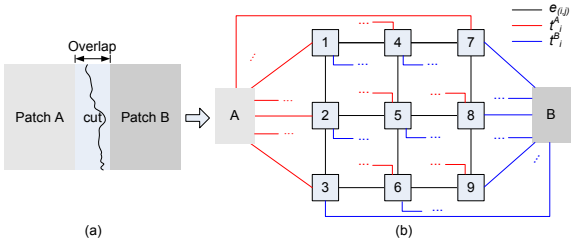

Fig. 8. Graph formulation for the seam finding problem in graphcut texture painting.

Our goal is to assign a patch label σ_p to every pixel p in the overlapping region ($\sigma_p = A$ or B), so that the region is divided into two parts by a seam of minimum cost (energy). The energy we minimize is defined as

$$E(\sigma) = E_{data}(\sigma) + E_{smooth}(\sigma).$$

In general graphcut problems, the smoothness energy E_{smooth} measures the extent to which σ is not smooth, while the data energy E_{data} measures the difference between σ and some known data [3]. For graphcut texture painting, E_{smooth} measures how well the patches fit together along their seams, whereas $E_{data} = E_{constraint}$ measures how well the synthesized texture satisfies the user specified constraint m_0.

Fitting patches together while minimizing E_{smooth} along the seams is done the same way as graphcut texture synthesis [17]. The energy E_{smooth} is defined as in [3]

$$E_{smooth}(\sigma) = \sum_{p,q} V_{(p,q)}(\sigma_p, \sigma_q),$$

where $\sum_{p,q}$ is the sum over all pairs of adjacent pixels in the overlapping region. The smoothness function $V_{(p,q)}(\sigma_p, \sigma_q)$ is defined as

$$V_{(p,q)}(\sigma_p, \sigma_q) = ||f_{\sigma_p}(p) - f_{\sigma_q}(p)|| + ||f_{\sigma_p}(q) - f_{\sigma_q}(q)||,$$

where $f_A(p)$ and $f_B(p)$ are pixel p's texture values for patches A and B respectively.

For graphcut texture painting, we need to further satisfy the user-specified constraint m_0. This is where graphcut texture painting differs from graphcut texture synthesis, which uses E_{smooth} only. We incorporate the user-specified constraint into quilting by making use of the energy E_{data} defined as

$$E_{data}(\sigma) = E_{constraint}(\sigma) = \sum_{p} D_p(\sigma_p),$$

where $\sum_p$ is the sum over all pixels in the overlapping region. The function $D_p(\sigma_p)$ is defined as

$$D_p(\sigma_p) = ||m_{\sigma_p}(p) - m_0(p)||$$

where $m_{\sigma_p}(p)$ is the constraint value of patch σ_p at pixel p, while $m_0(p)$ is the user-specified constraint value at pixel p.

We use the optimal swap move approach [3] to minimize the energy $E(\sigma)$. Fig. 8 (b) illustrates the graph construction. The nodes of the graph consist of all pixels in the overlapping region and two terminals A and B. Each pixel p in the

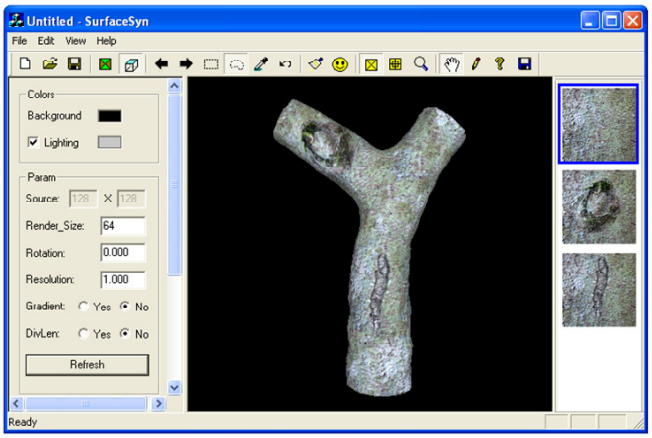

Fig. 9. Our painting user interface. Input samples with surface imperfections are shown on the right-hand side panel.

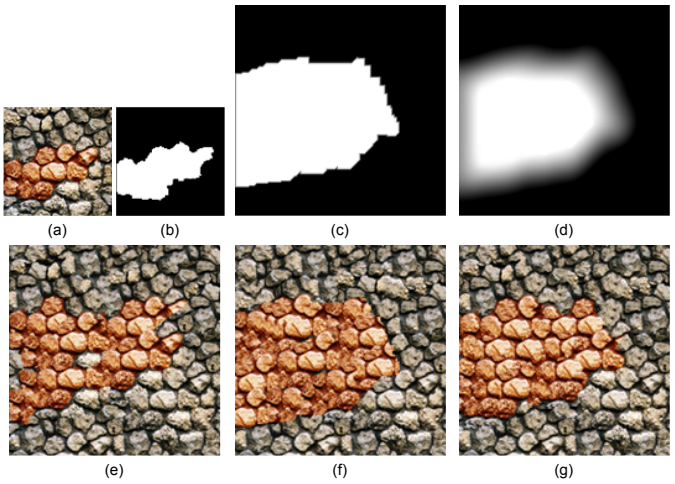

Fig. 10. Computing the constraint function m_0 in our painting system. (a) Input sample. (b) Specifying a foreground feature in the input sample. (c) User-specified m_0. (d) m_0 computed by our painting system. (e) Painting results using a naive modification of m_0. (f) Painting results using m_0 shown in (c). Notice the broken texture patterns in the orange area. (f) Painting results using m_0 shown in (d).

overlapping region is connected to the terminals A and B by edges t_p^A and t_p^B, respectively. These edges are referred to as t-links (terminal links). Each pair of the adjacent pixels (p, q) in the overlapping region is connected by an edge $e_{(p,q)}$ called an n-link (neighbor link). The weights of the edges are

$$weight(t_p^A) = D_p(A), weight(t_p^B) = D_p(B)$$

$$weight(e_{(p,q)}) = V_{(p,q)}(A, B)$$

Applying the min-cut algorithm to the constructed graph produces the minimum cost cut that separates node A from node B. As explained in [3], any cut in the graph must include exactly one t-link for any pixel p. Thus, any cut leaves each pixel p in the overlapping region with exactly one t-link, which defines a natural label σ_p according to the minimum cost cut C,

$$\sigma_p = \begin{cases} A & t_p^A \in C \\ B & t_p^B \in C \end{cases}$$

The approach for handling old seams in graphcut texture synthesis also works with graphcut texture painting.

Quilting on Surfaces: The surface quilting algorithm described in Section III can be adopted for graphcut texture painting on surfaces with modifications of the patch matching and fitting steps. For patch matching, we use both the BTF values and the constraint values (m_s and m_0 values) to perform a joint search for optimal patch placement. For patch fitting, we use constrained graphcut.

B. Painting System

Fig. 9 exhibits a screenshot of our painting system. For user interactivity we do not render surfaces with the BTF. Instead we display an ordinary color texture that provides a quick preview of the actual BTF. The color texture is obtained as one of the BTF images (usually the one with the front parallel view and head-on lighting).

Our painting system modifies the user-specified constraint function m_0 to improve the quality of the constrained quilt. As mentioned, the user specifies the foreground region F on the target surface and thus defines the constraint m_0 such that $m_0 = 1$ over F and $m_0 = 0$ elsewhere. Unfortunately, m_0 defined this way has an abrupt change along the boundary of F and this abrupt change often leads to destroyed texture patterns in nearby areas, as illustrated in Fig. 10 (f). A naive solution to this problem is to weaken m_0 as $m_0 = \lambda$ over F and $m_0 = 0$ elsewhere for some small λ. From the definition of the data energy E_{data}, if m_0 is set to a small value λ, E_{data} will play a less important role in the total objective energy. However, when λ is small, m_0 ceases to be effective and background elements start to appear in F as shown in Fig. 10 (e). Our solution is to expand a transition zone from the boundary of F and interpolate the values of m_0 in the transition zone using linear interpolation as shown in Fig. 10 (d). The width of the transition zone is a user-adjustable parameter. Fig. 10 (g) shows a synthesis result with the transition zone.

In addition to synthesizing new foreground features, our system also supports verbatim copying of the foreground feature from the input sample onto the surface. This is a straightforward extension of the verbatim copying technique for image textures ([17] called this interactive blending and merging). A useful feature of our system is that it previews a verbatim copying operation by directly projecting the foreground feature onto the surface. Although the projected foreground feature is not seamlessly merged with the background, the preview still provides valuable visual feedback. With our system the user can slide the foreground feature on the surface to find a desired pasting location since the foreground feature can be mapped onto the surface extremely quickly using the work patch/image at the target location.

V. Results

BTF Synthesis: Our synthesis algorithm generates good results on a wide variety of BTFs, including those that cannot be handled well by existing pixel-based BTF synthesis algorithms. Fig. 11 provides examples of our synthesis results. Fig. 12 (a) shows an example of a BTF that cannot be handled

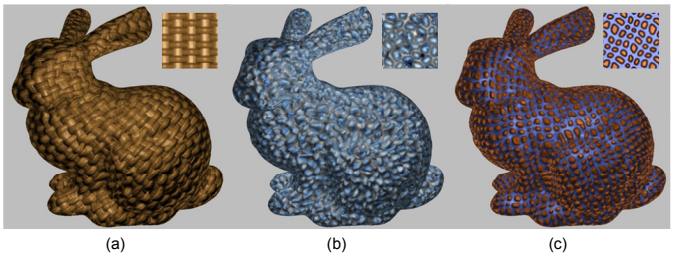

Fig. 11. BTF synthesis results for three synthetic BTF samples.

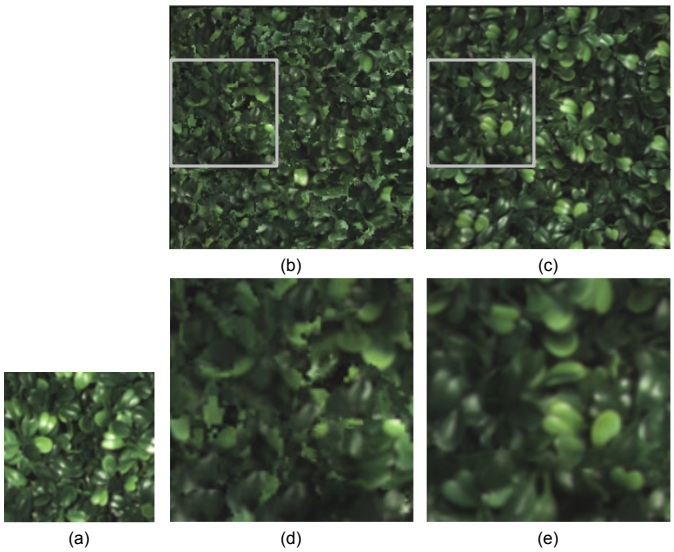

Fig. 12. Comparison of pixel-based BTF synthesis with our synthesis algorithm. (a) Input sample. (b) and (d) Pixel-based systhesis result. Notice that mesostructure of the leaves are not well maintained. (c) and (e) Our result.

well with existing techniques. This BTF is measured from real plastic leaves. Fig. 12 (b) and (c) exhibit the synthesis results of the pixel-based algorithm proposed by [26] and our algorithm respectively. Our result faithfully captures the mesostructures of the original BTF, while the pixel-based algorithm does not.

BTF synthesis is an off-line process that usually takes about 10 - 20 minutes for a 128×128 input sample (the lighting and viewing resolution is $12 \times 8 \times 12 \times 8$). The timing depends on the number of charts in the MCGIM dense mesh M_d and the size of M_d (512×512 or 1024×1024). The timings are measured on a PC with a Xeon 3.0 GHz processor.

BTF Painting: BTF painting (verbatim copying) is an interactive process. With BTF painting we can capture imperfections of real materials and paint them onto geometry models. Fig. 14 (a) shows a BTF captured from a real knitwear with a small imperfection. Fig. 14 (b) and (c) exhibit results generated by BTF synthesis followed by interactive painting. With this approach, we can generate real-world imperfections that are difficult to obtain by physically-based simulation techniques. Note that the foreground synthesis is an off-line process just like the background BTF synthesis. It depends on the sizes of the foreground areas specified by users.

Fig. 13 and 15 provide more examples of surface decoration with BTFs and imperfections. The BTF samples used in these examples are modelled by an artist and rendered using a ray tracer. Acquiring these BTF samples is difficult and time consuming. With our system, the user only has to generate small BTF samples and can decorate arbitrarily large objects using automatic BTF synthesis followed by interactive BTF painting.

Fig. 13. Tree bark generated with BTF synthesis followed by BTF painting. The input BTF sample is shown in the lower left.

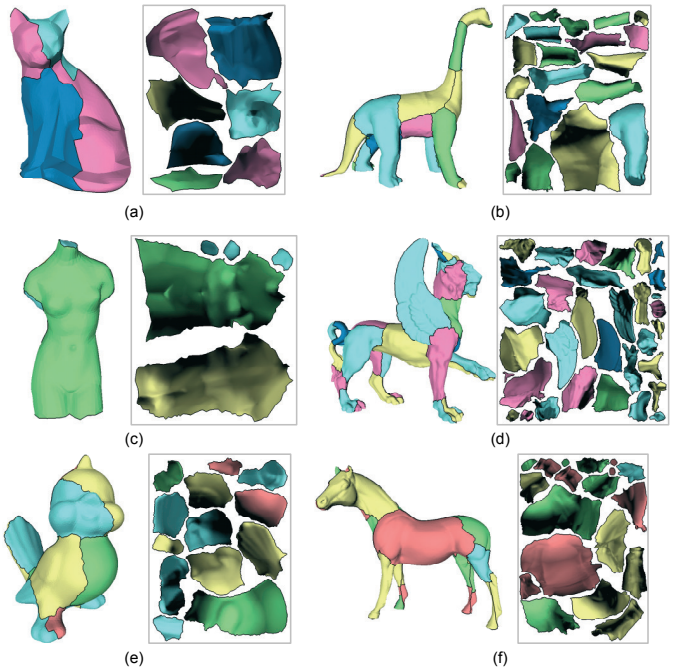

Fig. 17. Texture atlases for models used in Fig.16.

Texture synthesis: Note that our surface quilting algorithm also works well for ordinary color textures. Fig. 16 provides synthesis results for some texture samples with highly structured patterns, using both a pixel-based algorithm [27] and our algorithm. For these texture samples, the pixel-based algorithm always causes the texture patterns to break apart while our surface quilting algorithm can preserve the integrity of texture patterns very well. Fig. 17 shows the texture atlases for models used in Fig. 16.

VI. Conclusion

We presented a system for decorating surfaces with BTFs. This system is based on a graphcut algorithm for BTF synthesis on surfaces and a graphcut texture painting algorithm. Our work on BTF synthesis demonstrates that quilting BTF

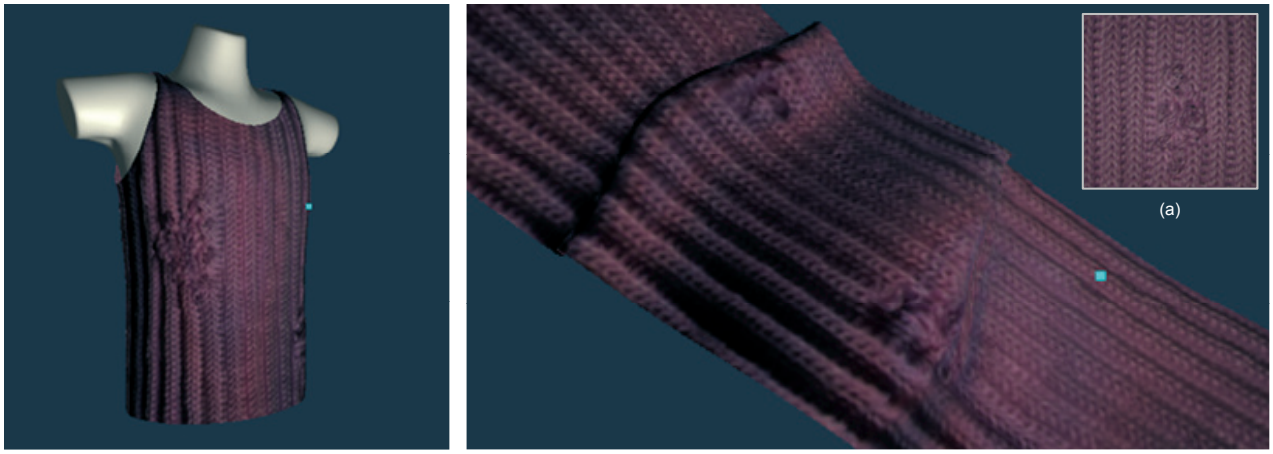

Fig. 14. Decorating surfaces with BTFs measured from real materials. (a) Input BTF sample with imperfections. (b) and (c) Surfaces decorated with BTF synthesis followed by BTF painting.

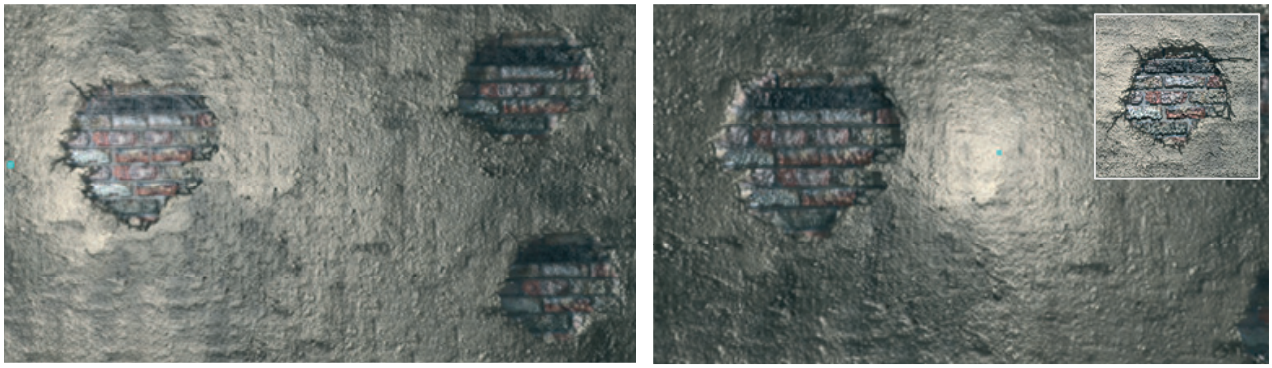

Fig. 15. Surfaces decorated with BTF synthesis followed by BTF painting. The input BTF sample is shown in the upper right.

patches with graphcut provides an effective way to maintain mesostructures, which is not possible with existing techniques. Our graphcut texture painting algorithm allows us to interactively paint with BTF patches. With BTF painting we can measure surface imperfections from a real material and paint them onto geometry models, making graphics models more interesting and better resemble real-world objects. In future work, we are interested in improving the speed of our synthesis algorithm.

ACKNOWLEDGEMENTS

We thank Steve Lin for his help in video production and proofreading of this paper. Thanks to anonymous reviewers for their constructive critiques on our ill-fated Siggraph 2004 submission. Yi Wang wrote the BTF rendering code and generated the BTF rendering results shown here. Jiaoying Shi was supported by NSFC (No. 60033010).

REFERENCES

[1] Michael Ashikhmin. Synthesizing natural textures. *2001 ACM Symposium on Interactive 3D Graphics*, pages 217–226, March 2001.

[2] Norman I. Badler and Welton Becket. Imperfection for realistic image synthesis. *Journal of Visualization and Computer Animation*, 1(1):26–32, August 1990.

[3] Yuri Boykov, Olga Veksler, and Ramin Zabih. Fast approximate energy minimization via graph cuts. *IEEE Trans on Pattern Analysis and Machine Intelligence*, 23(11):1–18, November 2001.

[4] Nathan A. Carr and John C. Hart. Painting detail. *ACM Transactions on Graphics*, 23(3):842–849, August 2004.

[5] Edwin Catmull. A subdivision algorithm for computer display of curved surfaces. 1974.

[6] Kristin J. Dana, Bram van Ginneken, Shree K. Nayar, and Jan J. Koenderink. Reflectance and texture of real-world surfaces. *ACM Transactions on Graphics*, 18(1):1–34, January 1999.

[7] Julie Dorsey, Alan Edelman, Justin Legakis, Henrik Wann Jensen, and Hans Køhling Pedersen. Modeling and rendering of weathered stone. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 225–234, August 1999.

[8] Alexei A. Efros and William T. Freeman. Image quilting for texture synthesis and transfer. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 341–346, August 2001.

[9] Alexei A. Efros and Thomas K. Leung. Texture synthesis by non-parametric sampling. In *Proceedings of International Conference on Computer Vision*, September 1999.

[10] Mark Foskey, Miguel A. Otaduy, and Ming C. Lin. Artnova: Touch-enabled 3d model design. In *IEEE Virtual Reality Conference*, pages 119–126, March 2002.

[11] Gabriele Gorla, Victoria Interrante, and Guillermo Sapiro. Growing fitted textures. *SIGGRAPH 2001 Sketches and Applications*, page 191, August 2001.

[12] Xianfeng Gu, Steven J. Gortler, and Hugues Hoppe. Geometry images. *ACM Transactions on Graphics*, 21(3):355–361, July 2002.

[13] Pat Hanrahan and Paul E. Haeberli. Direct wysiwyg painting and texturing on 3d shapes. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, volume 24, pages 215–223, August 1990.

[14] Aaron Hertzmann, Charles E. Jacobs, Nuria Oliver, Brian Curless, and David H. Salesin. Image analogies. *Proceedings of SIGGRAPH 2001*, pages 327–340, August 2001.

[15] Takeo Igarashi and Dennis Cosgrove. Adaptive unwrapping for interac-

Fig. 16. Texture synthesis results. (a) and (d) are the input texture samples. (b) and (e) show the synthesis results using a pixel-based algorithm [27]. (c) and (f) show the results using our algorithm.

tive texture painting. In *ACM Symposium on Interactive 3D Graphics*, pages 209–216, March 2001.

[16] Jan J. Koenderink and Andrea J. Van Doorn. Illuminance texture due to surface mesostructure. *Journal of the Optical Society of America*, 13(3):452–463, 1996.

[17] Vivek Kwatra, Arno Schödl, Irfan Essa, Greg Turk, and Aaron Bobick. Graphcut textures: Image and video synthesis using graph cuts. *ACM Transactions on Graphics*, 22(3):277–286, July 2003.

[18] B. Lévy, S. Petitjean, N. Ray, and J.-L. Mallet. Least squares conformal maps for automatic texture atlas generation. In *Proceedings of SIGGRAPH 2002*, pages 362–371, 2002.

[19] Lin Liang, Ce Liu, Yingqing Xu, Baining Guo, and Heung-Yeung Shum. Real-time texture synthesis using patch-based sampling. *ACM Transactions on Graphics*, 20(3), July 2001.

[20] Xinguo Liu, Yaohua Hu, Jingdan Zhang, Xin Tong, Baining Guo, and Heung-Yeung Shum. Synthesis and rendering of bidirectional texture functions on arbitrary surfaces. *IEEE Trans on Visualization and Computer Graphics*, 10(3):278–289, May 2004.

[21] Sebastian Magda and David Kriegman. Fast texture synthesis on arbitrary meshes. In *Eurographics Symposium on Rendering*, June 2003.

[22] Hans Køhling Pedersen. Decorating implicit surfaces. In *Proceedings of SIGGRAPH 95*, pages 291–300, August 1995.

[23] Emil Praun, Adam Finkelstein, and Hugues Hoppe. Lapped textures. *Proceedings of SIGGRAPH 2000*, pages 465–470, July 2000.

[24] P.V. Sander, Z. Wood, S.J. Gortler, J. Synder, and H. Hoppe. Multi-chart geometry images. In *Symposium on Geometry Processing 2003*, pages 146–155, 2003.

[25] Cyril Soler, Marie-Paule Cani, and Alexis Angelidis. Hierarchical pattern mapping. *ACM Transactions on Graphics*, 21(3):673–680, July 2002. (Proceedings of ACM SIGGRAPH 2002).

[26] Xin Tong, Jingdan Zhang, Ligang Liu, Xi Wang, Baining Guo, and Heung-Yeung Shum. Synthesis of bidirectional texture functions on arbitrary surfaces. *ACM Transactions on Graphics*, 21(3):665–672, July 2002.

[27] Greg Turk. Texture synthesis on surfaces. *Proceedings of SIGGRAPH 2001*, pages 347–354, August 2001.

[28] Li-Yi Wei and Marc Levoy. Fast texture synthesis using tree-structured vector quantization. *Proceedings of SIGGRAPH 2000*, pages 479–488, July 2000.

[29] Li-Yi Wei and Marc Levoy. Texture synthesis over arbitrary manifold surfaces. *Proceedings of SIGGRAPH 2001*, pages 355–360, August 2001.

[30] Tien-Tsin Wong, Wai-Yin Ng, and Pheng-Ann Heng. A geometry dependent texture generation framework for simulating surface imperfections. In *Eurographics Rendering Workshop 1997*, pages 139–150, June 1997.

[31] Y. Q. Xu, B. Guo, and H. Y. Shum. Chaos Mosaic: Fast and Memory Efficient Texture Synthesis. In *Microsoft Research Technical Report MSR-TR-2000-32*, April 2000.

[32] Lexing Ying, Aaron Hertzmann, Henning Biermann, and Denis Zorin. Texture and shape synthesis on surfaces. *Proceedings of 12th Eurographics Workshop on Rendering*, pages 301–312, June 2001.

[33] Steve Zelinka and Michael Garland. Interactive texture synthesis on surfaces using jump maps. In *Eurographics Symposium on Rendering: 14th Eurographics Workshop on Rendering*, pages 90–96, June 2003.

[34] Jingdan Zhang, Kun Zhou, Luiz Velho, Baining Guo, and Heung-Yeung Shum. Synthesis of progressively variant textures on arbitrary surfaces. *ACM Transactions on Graphics*, 22(3):295–302, July 2003.

[35] Kun Zhou, John Snyder, Baining Guo, and Heung-Yeung Shum. Isocharts: Stretch-driven mesh parameterization using spectral analysis. In *Symposium on Geometry Processing 2004*, pages 47–56, 2004.