Synthesis and Rendering of Bidirectional Texture Functions on Arbitrary Surfaces

Xinguo Liu, Yaohua Hu, Jingdan Zhang, Xin Tong,
Baining Guo, and Heung-Yeung Shum, *Senior Member*, *IEEE*

Abstract—The bidirectional texture function (BTF) is a 6D function that describes the appearance of a real-world surface as a function of lighting and viewing directions. The BTF can model the fine-scale shadows, occlusions, and specularities caused by surface mesostructures. In this paper, we present algorithms for efficient synthesis of BTFs on arbitrary surfaces and for hardware-accelerated rendering. For both synthesis and rendering, a main challenge is handling the large amount of data in a BTF sample. To addresses this challenge, we approximate the BTF sample by a small number of 4D point appearance functions (PAFs) multiplied by 2D geometry maps. The geometry maps and PAFs lead to efficient synthesis and fast rendering of BTFs on arbitrary surfaces. For synthesis, a surface BTF can be generated by applying a texton-based sysnthesis algorithm to a small set of 2D geometry maps while leaving the companion 4D PAFs untouched. As for rendering, a surface BTF synthesized using geometry maps is well-suited for leveraging the programmable vertex and pixel shaders on the graphics hardware. We present a real-time BTF rendering algorithm that runs at the speed of about 30 frames/second on a mid-level PC with an ATI Radeon 8500 graphics card. We demonstrate the effectiveness of our synthesis and rendering algorithms using both real and synthetic BTF samples.

Index Terms—Bidirectional texture function, reflectance and shading models, texture synthesis, mesh parameterization, texture mapping, surfaces.

1 Introduction

Textures in the traditional graphics sense represent color or albedo variations on smooth surfaces. Real-world textures, on the other hand, arise from both spatially-varying surface reflectance and mesostructures, i.e., small but visible local geometric details [1]. Mesostructures are responsible for the fine-scale shadows, occlusions, and specularities that are integral parts of real-world surface appearance [2], [3], [4], [5]. The BTF introduced by Dana et al. [2], [6] is a representation of real-world textures that can model surface mesostructures and reflectance variations. The BTF is a 6D function whose variables are the 2D position and the viewing and lighting directions. The BTF is also a generalization of the BRDF to include surface position variation. The BTF can be either measured from real-world materials [2] or generated synthetically [5]. Synthetic BTFs provide an efficient way to render surfaces with complex (synthetic) appearance models and geometry details.

In this paper, we address two key issues for BTFs on arbitrary surfaces. The first is the efficient synthesis of BTFs. Given a BTF sample and an arbitrary surface mesh, we wish to synthesize a BTF on the mesh such that: 1) the surface BTF is perceptually similar to the given BTF sample in all viewing/lighting conditions and 2) the surface BTF exhibits a consistent mesostructure when viewing and lighting directions change. The requirement of a consistent mesostructure is where surface BTF synthesis differs fundamentally from surface texture synthesis [7], [8], [9] since conventional textures ignore mesostructures completely.

A BTF can be mapped onto surfaces using texture mapping techniques. However, BTF mapping on arbitrary surfaces can introduce inconsistent mesostructures. The usual technique for texture mapping arbitrary surfaces is to use a collection of overlapping patches [10], [11] and textures in the overlapping regions are blended to hide seams (e.g., see [11]). This technique works well for many textures, but, for the BTF, blending can introduce inconsistent mesostructures [5]. Of course, BTF mapping on arbitrary surfaces also suffers from the usual problems of texture mapping. These problems include texture distortion, seams between texture patches, and considerable user intervention needed for creating good-quality texture maps [7], [8], [9].

A possible way to achieve a consistent mesostructure on a surface is to apply surface texture synthesis techniques to directly surface BTF synthesis. The sample BTF may be regarded as a 2D texture map in which the BTF value at a pixel is a 4D function of the viewing and lighting directions and this 4D function can be discretized into a vector for texture synthesis. Unfortunately, this approach incurs a huge computational cost because of the large amount of data in a BTF sample. At the resolution of $12 \times 5 \times 12 \times 5$, the BTF value at a pixel is a 10,800-dimensional vector, as opposed to the usual 3D RGB vectors [7], [8], [9]. Since texture synthesis time grows linearly with the vector dimension, a surface BTF can take days [7] or even months [8] to compute.

The second issue we address is hardware-accelerated BTF rendering. The simple operations needed for BTF rendering [5] are amenable to hardware implementation, yet there is no existing system for fast BTF rendering

- *X. Liu, Y. Hu, X. Tong, 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: {xgliu, yaohu, xtong, bainguo, hshum}@microsoft.com.
- *J. Zhang is with the Department of Computer Science, University of North Carolina, Chapel Hill, NC.*

Manuscript received 26 Nov. 2002; revised 28 May 2003; accepted 26 Aug. 2003.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCG-0019-1102.

using off-the-shelf graphics cards. For a given viewing/lighting combination, a simple method to render a BTF on a parametric surface is to generate a 2D texture and map it onto the surface using hardware texture mapping (see Section 4.1). Unfortunately, this method requires loading all BTF images into memory and thus incurs a large space overhead. In addition, the rendering is relatively slow ($1 \sim 2$ frames per second) because a time-consuming routine for evaluating the BTF needs to be called millions of times during rendering.

We have developed techniques for efficient synthesis of BTFs on arbitrary surfaces and hardware-accelerated BTF rendering. The basis of our techniques is an approximation scheme for BTFs. From our earlier discussion, we can see that, for both BTF synthesis and hardware-accelerated rendering, the main challenge is handling the large amount of data in a BTF sample. For example, roughly 700 MB of storage are needed to store a BTF with a spatial resolution of 256×256 and a sampling rate of $6 \times 10 \times 6 \times 10$ for the lighting and viewing directions. To address this challenge, we approximate the BTF sample by a small number of 4D point appearance functions (PAFs) multiplied by 2D geometry maps. This approximation can reduce a 700 MB BTF sample to just a few MB with less than 2 percent approximation error.

The combination of geometry maps and PAFs leads to efficient synthesis and fast rendering of BTFs on arbitrary surfaces. For synthesis, we showed [5] that surface BTFs can be efficiently synthesized using surface textons derived from 3D textons. With geometry maps and PAFs, we can further improve the synthesis efficiency significantly. Specifically, a surface BTF can be synthesized by applying the algorithm of [5] to a small set of 2D geometry maps while leaving the companion 4D PAFs untouched. We will also show that a surface BTF synthesized using geometry maps is well-suited for hardware-accelerated rendering. We present a rendering algorithm that leverages the capabilities of the latest graphics hardware to achieve real-time performance. On a mid-level PC with an ATI Radeon 8500 graphics card, our system can achieve a speed of 30 frames/second.

The approximation of the BTF by low-dimensional texture functions is the key to efficient BTF synthesis and rendering. A good approximation should achieve high accuracy with a small number of terms and the resulting low-dimensional texture functions must facilitate surface BTF synthesis and hardware-accelerated rendering. Fast rendering algorithms for surface light fields [12] and the bidirectional reflectance distribution functions (BRDFs) [13], [14] decompose the 4D surface light fields and BRDFs into sums of products of 2D functions. BRDFs are parameterized the same way as PAFs, even if PAFs don't necessarily share all the properties of BRDFs (positivity, symmetry, etc.). Decomposition of the 6D BTF is more difficult because of the extra dimensions involved. One possibility is to decompose the BTF into a sum of products of three 2D functions. We have attempted this approach but found that the approximation errors were large and a large number of rendering passes were needed. BTF approximation by 4D PAFs multiplied by 2D geometry maps gives good results with a small number of terms. Although the PAFs are 4D, we have developed hardware-accelerated techniques for rendering them by leveraging the multitexturing and volume texturing capabilities of the latest graphics hardware. In particular, we demonstrate that calculations of 4D PAFs for arbitrary lighting and viewing directions can be done by hardware-supported 3D interpolation.

An important task in hardware-accelerated rendering of BTFs on arbitrary surfaces is the conversion of surface geometry maps to 2D geometry maps on the unit square $[0,1] \times [0,1]$. To accomplish this task, we establish a texture atlas for the underlying surface mesh and then use a splatting algorithm to "render" the vertex data of the surface geometry maps into the unit square $[0,1] \times [0,1]$.

We will demonstrate the effectiveness of our synthesis and rendering algorithms using both real and synthetic BTF samples. With the increasing availability of BTF samples measured from real-world textures [2], surface BTFs provide a way to decorate real-world geometry with real-world textures.

The rest of this paper is organized as follows: After summarizing related work in Section 2, we describe surface textons, geometry maps, and PAFs in Section 3 and present our algorithm for surface BTF synthesis. Our hardware-accelerated rendering algorithm is detailed in Section 4. Section 5 reports synthesis and rendering results. We conclude in Section 6 with discussion of future research topics.

2 Related Work

2.1 BTF

Dana et al. introduced the BTF and built the CUReT database, which consists of BTF samples measured from real-world materials [2]. The CUReT database has been widely used for statistical surface appearance analysis and recognition [2], [3], [15]. Dana et al. also demonstrated some BTF rendering results in [2]. However, the images in the CUReT database are not enough to synthesize novel images for arbitrary viewing and lighting directions. To address this problem, Liu et al. introduced a method to synthesize arbitrary viewing/lighting BTF samples from a sparse set of samples [4]. More recently, Tong et al. proposed a method for synthesizing a BTF on arbitrary surfaces using surface textons [5]. In [5], the issue of hardware-accelerated rendering is not addressed. We address that issue in this paper. In addition, we propose a significantly more efficient algorithm for synthesizing BTFs on arbitrary surfaces.

Several rendering techniques are closely related to the BTF despite the fact that they do not explicitly use the BTF as defined by Dana et al. [2]. Daubert et al. proposed an efficient technique for modeling and rendering cloth with replicated weaving or knitting patterns [16]. They modeled fine geometry details of a small piece of cloth and then sampled its appearance under varying viewing and light directions. Their representation is very close to a BTF, but they fit the data with a variant of the Lafortune model. The polynomial texture map (PTM) proposed by Malzbender et al. can be regarded as a BTF with the viewing direction fixed [17]. Dischler proposed a 4D texture representation to describe and render macro geometry on surfaces [18]. However, his algorithm takes several minutes to render a frame.

2.2 Decomposition and Compression

Most image-based rendering methods need to address the issue of compression. For surface light fields, Wood et al.

[19] used a lumisphere to assemble all visible rays from a surface point and compress the data by principal function analysis. Nishino et al. [20] proposed an eigentexture method to compress a sequence of images under different viewing and illumination conditions. In their recent work on light field mapping [12], Chen et al. approximated the discrete surface light field as a sum of a small number of products of 2D maps for data compression.

2.3 Texture Synthesis

Generating textures on arbitrary surfaces has been an active area of research [7], [8], [9], [10], [11], [21]. One approach maps texture patches onto the target surface [10]. A good representative work following that approach is the lapped texture technique by Praun et al. [11]. They randomly paste texture patches onto the surface following orientation hints provided by the user. To hide the mismatched features across patch boundaries, textures in the overlapping regions are blended. This technique works well for a large class of textures, but, for highly structured textures and textures with strong low-frequency components, the seams along patch boundaries are still evident [11].

A number of algorithms have been proposed for directly synthesizing textures on arbitrary surfaces. Turk's algorithm [8], Wei and Levoy's algorithm [7], and the multiresolution synthesis algorithm by Ying et al. [9] are general-purpose algorithms based on the search strategy proposed by Wei and Levoy [22]. The algorithm by Gorla et al. [21] is also a general-purpose algorithm, based on the search strategy proposed by Efros and Leung [23]. These algorithms tend to be slow, but they can be accelerated by using either tree-structured vector quantization [7], [22] or a kd-tree [9].

2.4 Hardware-Accelerated Rendering

The arrival of programmable GPUs makes it possible to achieve a variety of realistic rendering effects in real-time [13], [14], [24]. The work on rendering arbitrary BRDFs [13], [14] is particularly relevant to BTF rendering, although the BTF is a higher-dimensional function. Kautz and McCool [13] separated the BRDF into a combination of several 2D textures products by the SVD, which minimizes RMS error. Texture mapping and compositing operations were then invoked to reconstruct samples of the BRDF at every pixel. McCool et al. [14] proposed a homomorphic factorization for approximating the BRDF by products of three 2D textures. This method avoids negative numbers, minimizes relative error, and is well-suited for high-performance rendering.

3 BTF Synthesis

A BTF can be regarded as a mapping from the 4D space of viewing and lighting directions to the space of all 2D images [2], [4]: $V \times L \rightarrow I$, where V and L are viewing and lighting directions and I is the space of all 2D images. In other words, we can view a BTF as a collection of images indexed by viewing and lighting directions. A BTF is a 6D reflectance field:

$$f(\mathbf{x}, \mathbf{v}, \mathbf{l}), \tag{1}$$

where $\mathbf{x} = (x, y)$ is the spatial coordinate, $\mathbf{v} = (\theta_v, \phi_v)$ is the reflectance/view direction, and $\mathbf{l} = (\theta_l, \phi_l)$ is the incident/lighting direction. $f(\mathbf{x}, \mathbf{v}, \mathbf{l})$ provides the connection between reflected flux in a direction $\mathbf{v}$ and incident flux in another direction $\mathbf{l}$ at the same point $\mathbf{x}$ on a material sample. The parameter space for the spatial variable $\mathbf{x}$ is called the geometry plane of a BTF or, simply, the BTF plane.[1] For each point $\mathbf{x}$, we define $f(\mathbf{v}, \mathbf{l}) = f(\mathbf{x}, \mathbf{v}, \mathbf{l})$ as the *point appearance function* (PAF) at $\mathbf{x}$. A BTF can be mapped onto an arbitrary surface through a mapping that provides a correspondence between each surface point and a point in the BTF plane.

In this section, we first introduce two BTF representations that we need for surface BTF synthesis and hardware-accelerated rendering. Using these representations, we then describe a new algorithm for surface BTF synthesis. This algorithm is significantly more efficient than the algorithm proposed in [5] and the synthesized surface BTF supports hardware-accelerated rendering.

3.1 Texton Maps

The surface texton map, or *texton map* for short, is a BTF representation for efficient synthesis of BTFs on arbitrary surfaces. Surface textons were proposed ealier [5] based on 3D textons [3]. The main idea behind 3D textons is that, at the local scale, there are only a small number of perceptually distinguishable mesostructures and reflectance variations on the surface. The surface textons use this idea to extract a compact set of data that suffices for surface BTF synthesis.

Surface textons are extracted from a sample BTF f as follows: 1) build a vocabulary of 3D textons from the sample BTF f, 2) assign texton labels to the pixels in the geometry plane of f to get a texton map t_{in}, and 3) construct the surface texton space by calculating the dot-product matrix M and discarding the appearance vectors of 3D textons.

3.1.1 3D Texton Vocabulary

The construction of 3D textons is mostly based on the original 3D texton paper by Leung and Malik [3]. As in [3], we construct 3D textons from a BTF using K-means clustering. To capture the appearance of mesostructures at different viewing/lighting conditions, we treat the BTF sample f as a stack of n images and filter each image with a filter bank of $n_b = 48$ Gaussian derivative filters. For each pixel of f, the filter responses of n_s selected images are concatenated into an $n_s n_b$-dimensional vector, called the appearance vector. All the appearance vectors are then clustered using the K-means algorithm. The resulting K-means centers $\{\mathbf{t}_1, \ldots, \mathbf{t}_{n_t}\}$ are the 3D textons. See [5] for more details on the construction of texton vocabulary.

3.1.2 Texton Map

Once we have the texton vocabulary $\{\mathbf{t}_1, \ldots, \mathbf{t}_{n_t}\}$, we can easily assign a texton label to each pixel in the geometry plane of f. Let $\{\mathbf{v}_1, \ldots, \mathbf{v}_{n_t}\}$ be the texton vocabulary's appearance vectors. For each pixel p, the texton label is obtained by $t_{in}(p) = \arg\min_{j=1}^{n_t} \|\mathbf{v}(p) - \mathbf{v}_j\|^2$, where $\mathbf{v}(p)$ is p's appearance vector. The resulting t_{in} is called a texton map.

3.1.3 Texton Space

The texton space $\mathbf{S}$ is a linear space spanned by $\{\mathbf{t}_1, ..., \mathbf{t}_{n_t}\}$:

1. Here, $\mathbf{x}$ corresponds to (u, v) or (s, t) in conventional texture mapping.

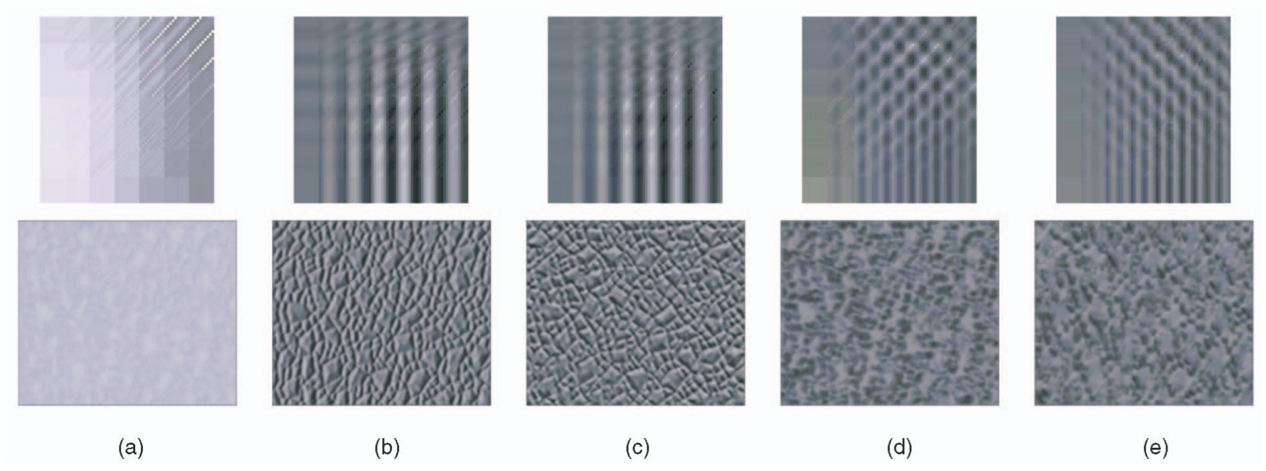


Fig. 1. The five most significant PAFs (top row) and the corresponding geometry maps (bottom row) of the BTF data “plaster04.” As Table 1 shows, the singular value decreases from (a) to (e). A 4D PAF is packed into a 2D image consisting of a 2D array of subimages in which each row corresponds to a change in the lighting direction and each column corresponds to a change in the viewing direction. The azimuth angle is the faster changing parameter in each row and column.

$$\mathbf{S} = \left\{ s | s = \sum_{i=1}^{n_t} a_i \mathbf{t}_i, a_i \in R \right\}.$$

Each element in $\mathbf{S}$ is called a surface texton or, simply, a texton. The inner product in $\mathbf{S}$ can be defined using a dot-product matrix M:

$$M = (a_{ij})_{n_t \times n_t}, \tag{2}$$

where $a_{ij} =< v_i, v_j >$, the inner product of the appearance vectors v_i and v_j associated with textons $\mathbf{t}_i$ and $\mathbf{t}_j$.

With M in hand, we can easily calculate the inner product of any pair of vectors in $\mathbf{S}$. Let $a = \sum_{i=1}^{n_t} a_i \mathbf{t}_i$ and $b = \sum_{i=1}^{n_t} b_i \mathbf{t}_i$ be two surface textons in $\mathbf{S}$. Their inner product is:

$$< a, b >= (a_0, a_1, \ldots, a_{n_t}) M (b_0, b_1, \ldots, b_{n_t})^T. \tag{3}$$

As shown in [5], the main advantage of the surface texton is that we can generate the BTF on an arbitrary surface by synthesizing a surface texton map. The synthesis of the surface texton map does not require appearance vectors. Surface textons play the same role in BTF synthesis [5] as pixels in color texture synthesis [7], [8], [9]. The pre-computed dot-product matrix in (3) leads to an efficient way for computing the distance between two surface textons and thus significantly reduces the CPU and memory costs for BTF synthesis. See [5] for further discussions.

3.2 Geometry Maps and PAFs

The second representation that we will need is a factorization of the BTF as a sum of products of 2D and 4D functions. By truncating this sum to a small number of the most significant terms, we can obtain a BTF approximation that is easy to render using graphics hardware. A BTF $f(\mathbf{x}, \mathbf{v}, \mathbf{l})$ may be regarded as a six-dimensional color array, which we can factorize using singular value decomposition (SVD). We form this color array as

$$A = \begin{pmatrix} f(\mathbf{x}_1, \mathbf{v}_1, \mathbf{l}_1) & \cdots & f(\mathbf{x}_1, \mathbf{v}_m, \mathbf{l}_m) \\ \vdots & \ddots & \vdots \\ f(\mathbf{x}_n, \mathbf{v}_1, \mathbf{l}_1) & \cdots & f(\mathbf{x}_n, \mathbf{v}_m, \mathbf{l}_m) \end{pmatrix}, \tag{5}$$

where n is the number of pixels of each BTF image, whereas m is the number of images of the BTF ($n \gg m$). Each row of A corresponds to a PAF at a point on the BTF plane. Each column of A corresponds to a 2D image of the BTF sample with fixed viewing and lighting directions.

Applying the SVD to the matrix A, we get $A = \mathbf{G \Lambda P}$, where $\mathbf{G}$ and $\mathbf{P}$ are, respectively, the left and right eigen matrix and $\mathbf{\Lambda}$ is a diagonal matrix with singular values λ_1 through λ_m in decreasing order. If we merge Λ into $\mathbf{G}$, then the BTF becomes a sum of products

$$f(\mathbf{x}, \mathbf{v}, \mathbf{l}) = \sum_i g_i(\mathbf{x}) p_i(\mathbf{v}, \mathbf{l}). \tag{5}$$

We call each $g_i(\mathbf{x})$ a geometry map and each $p_i(\mathbf{v}, \mathbf{l})$ an eigen PAF. This decomposition is performed for each color channel.

Fig. 1 exhibits the five most significant geometry maps and eigen PAFs as images by adjusting their range (which may include both positive and negative values) to $[0, 255]$ and visualizing each four-dimensional PAFs as an array of 2D images. The geometry map is a texture function that is dependent only on spatial coordinates. The PAF is a bidirectional function of the viewing and illumination directions. The patterns in the geometry maps have a very close relationship with the geometry details in the original BTF data (see Fig. 2) and it is for this reason we call each $g_i(x, y)$ a geometry map. Here, “geometry” refers to the meso-geometry (i.e., geometry detail) instead of the gross shape of the surface.

Table 1 shows the 10 most significant singular values of the BTF data “plaster04.” The rapid decrease of the singular values and the errors indicates that truncating (5) to a few terms introduces only small approximation errors. If we keep the most significant k terms in (5), the resulting summed squared residuals is of the order of $\sum_{i=k+1}^{m} \lambda_i^2$.

For the BTF data “plaster04,” we can approximate the BTF using five terms with less than 2 percent error. The quality of the 5-term approximation is illustrated in Fig. 2, where we render the original BTF data and the 5-term approximation on the same quadric surface patch using the software rendering algorithm described in Section 4.1. The renderings of the two

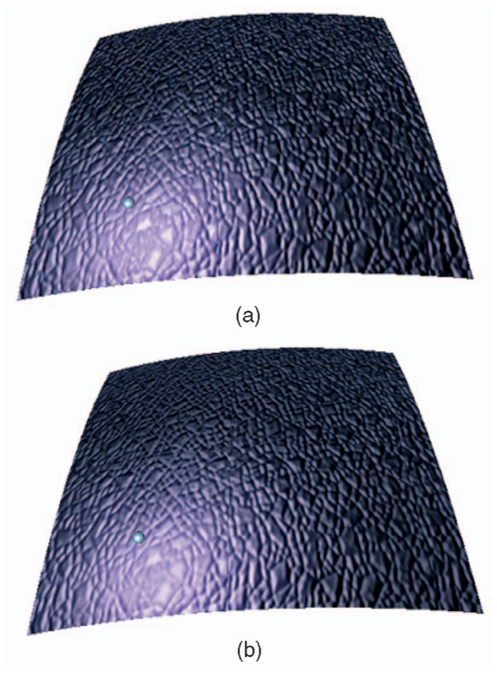


Fig. 2. Quality comparison for the BTF data "plaster04." (a) The original BTF. (b) The 5-term approximation. The two images are very much the same except specular highlights are slightly blurred.

data sets are similar, although the size of the 5-term approximation is only 2.9 MB. This is a dramatic reduction from the 617 MB of the original BTF data.

3.2.1 Discussions

Note that $\mathbf{x} = (x, y)$, $\mathbf{v} = (\theta_v, \phi_v)$, and $\mathbf{l} = (\theta_l, \phi_l)$. In the above decomposition scheme, we divide the BTF parameters $x, y, \theta_v, \phi_v, \theta_l, \phi_l$ into two groups, $\{x, y\}$ and $\{\theta_v, \phi_v, \theta_l, \phi_l\}$, when forming the matrix A in (4). Other parameters groupings are possible. For example, we can use $\{x, y, \theta_v\}$ and $\{\phi_v, \theta_l, \phi_l\}$. We choose $\{x, y\}$ and $\{\theta_v, \phi_v, \theta_l, \phi_l\}$ because we empirically found out that the singular values resulting from this grouping decrease the fastest.

We can further decompose the PAFs into a combination of 2D functions by using the SVD. For separable PAFs, the BTF can be decomposed into a sum of products, each consisting of three 2D functions. Although 2D functions are easy to render as texture maps, this decomposition overall does not make hardware rendering any easier. If each separation step uses a 5-term approximation, then the final approximation will consist of 25 terms. As a result, many rendering passes are needed for hardware rendering (see Section 4.3). Of course, there are also problems with error accumulation in texture multiplication and issues with nonseparable PAFs.

3.3 Surface BTF Synthesis

Using surface textons and geometry maps, we will now describe a new algorithm for surface BTF synthesis. This algorithm is significantly more efficient than the algorithm proposed in [5]. The quality of the synthesized surface BTFs is very good, although not as good as that of [5]. More importantly, the surface BTF synthesized using the new algorithm supports hardware-accelerated rendering.

TABLE 1
Ten Most Significant Values of the BTF Data "Plaster04"

k	1	2	3	4	5	6	7	8	9	10
λ	33.7	4.9	4.0	2.1	1.3	1.0	0.96	0.94	0.7	0.6
λ^2	1134	24	16	4.4	1.8	1.0	0.94	0.9	0.49	0.47
e%	6	4	2.6	2.2	2	1.98	1.9	1.8	1.79	1.75

$e = (\sum_{i=k+1}^{m} \lambda_i^2 / \sum_0^m \lambda_i^2) \cdot 100.$

The idea of the new algorithm is as follows: Suppose that we have an N-term approximation of the sample BTF and we rewrite the approximation in matrix form,

$$f(\mathbf{x}, \mathbf{v}, \mathbf{l}) = G(\mathbf{x})^T P(\mathbf{v}, \mathbf{l}), \tag{6}$$

where

$$\begin{aligned} G(\mathbf{x}) &= (g_0(\mathbf{x}), g_1(\mathbf{x}), \dots, g_N(\mathbf{x})) \\ P(\mathbf{v}, \mathbf{l}) &= (p_0(\mathbf{v}, \mathbf{l}), p_1(\mathbf{v}, \mathbf{l}), \dots, p_N(\mathbf{v}, \mathbf{l})). \end{aligned}$$

The key observation of the new algorithm is that the surface BTF $f'(\mathbf{x}, \mathbf{v}, \mathbf{l})$ can be synthesized by generating a set of new geometry maps

$$G'(\mathbf{x}) = (g'_0(\mathbf{x}), g'_1(\mathbf{x}), \dots, g'_N(\mathbf{x})) \tag{7}$$

without touching the PAFs in $P(\mathbf{v}, \mathbf{l})$, i.e.,

$$f'(\mathbf{x}, \mathbf{v}, \mathbf{l}) = G'(\mathbf{x})^T P(\mathbf{v}, \mathbf{l}),$$

where $\mathbf{x}$ is now a point on the target surface. Because N is usually quite small, synthesizing $G'(\mathbf{x})$ requires much less work than directly synthesizing $f'(\mathbf{x}, \mathbf{v}, \mathbf{l})$. For all examples in this paper, $N \leq 40$. Typically, $N = 12$.

To synthesize $G'(\mathbf{x})$, we treat $G(\mathbf{x})$ as a stack of images and apply the three steps described in Section 3.1 for surface texton extraction. Once we have the surface textons, we can synthesize a surface texton map exactly as in [5] and the surface texton map now defines the surface geometry map $G'(\mathbf{x})$. In practice, the target surface is first densely retiled using the retiling algorithm in [25] and $G'(\mathbf{x})$ is a vector signal defined for each vertex on the retiled mesh of target surface.

4 BTF Rendering

In this section, we start by describing a simple algorithm for rendering the BTF using the texture mapping hardware that is standard for most graphics cards. By describing this algorithm, we also provide the necessary background information on BTF rendering. The algorithm is quite efficient for moderate-sized objects and for BTFs that can be completely loaded in memory, although the rendering speed is far from real time even for very simple scenes. For real-time rendering using the latest graphics hardware, we first reparameterize the BTF for better resampling of the viewing and lighting directions. Then, we describe hardware-accelerated rendering algorithms for surfaces, including surfaces that can be parameterized on rectangular regions and arbitrary surfaces.

4.1 Software Rendering

Given a model mapped with a BTF $f(\mathbf{x}, \mathbf{v}, \mathbf{l})$ on its surface, we want to render it under arbitrary viewing and lighting settings. To take advantage of the texture mapping functionality available on most graphics cards, we first

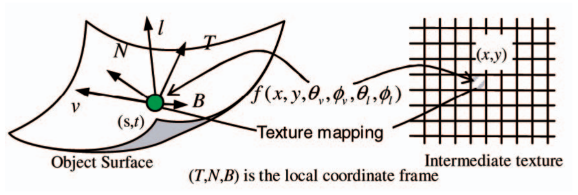

l
N
T
v
B
(s,t)
$f(x,y,\theta_v,\phi_v,\theta_l,\phi_l)$
Texture mapping
(x,y)
Object Surface
Intermediate texture
(T,N,B) is the local coordinate frame

Fig. 3. Mapping the BTF from the parameter space to the object surface.

build an intermediate image under the current viewing and lighting setting and then map it onto the surface as texture. The intermediate image size is the BTF plane size multiplied by the replication number on the surface. The intermediate image's pixel value is computed as follows: As shown in Fig. 3, each pixel $\mathbf{x}$ in the intermediate texture image corresponds to a point $\mathbf{q}(s,t)$ on the surface $\mathbf{S}(s,t)$ by the texture mapping transformation. The local coordinate frame $\{T,N,B\}$ of point $\mathbf{q}$ can be obtained and used to compute the local viewing direction $\mathbf{v}_q$ and lighting directions $\mathbf{l}_q$. Then, the value of pixel $\mathbf{x}$ is given by BTF value $f(\mathbf{x},\mathbf{v}_q,\mathbf{l}_q)$.

Fig. 4 shows an example of software BTF rendering with the BTF data "tube" mapped onto a quadric surface patch using the method described above. For each pixel (x,y) in the intermediate texture, the corresponding points $\mathbf{q}$ and the local coordinates frame $\{T,N,B\}$ can be computed in a preprocessing step and stored for reuse. To save computation for the intermediate texture, we can omit back-facing pixels in the texture. For the example in Fig. 4, we obtained $1 \sim 2$ frames per second. As expected, many desired visual effects, such as specularities, self-shadows, and self-occlusion, are present.

4.2 Reparametrization of BTF

In practice, the BTF is represented by discrete sample values, as is the case with the CUReT database [26]. For rendering efficiency, it is important to find a good parameterization for the viewing and lighting directions so that we can achieve good resampling and interpolation of the BTF on a regular grid in the viewing and lighting hemispheres. With a good parameterization, the viewing and lighting directions corresponding to a uniform sampling grid of the parameter space are evenly distributed on the hemisphere. Clearly, uniform sampling of the angular values produces unevenly distributed directions on the hemisphere, as Fig. 5b illustrates (oversampling near the pole and undersampling near the equator). The elevated concentric map gives a parameterization with improved distribution of the viewing and lighting directions on the hemisphere.

The elevated concentric map Ω is a map from the unit square to the unit hemisphere with $z > 0$. Ω is represented as a concatenation of the concentric map Φ and the elevation map Ψ [27],

$$\Omega(s,t) = (\Psi \circ \Phi)(s,t). \tag{8}$$

The concentric map Φ is a function from the unit square to the unit disk. $\Phi(s,t) = (\rho(s,t), \phi(s,t))$ with $(s,t) \in [0,1] \times [0,1]$ and $(\rho,\phi) \in [0,1] \times [0,2\pi]$. For a point (s,t) in a unit square with $|s| > |t|$ and $s > 0$, the concentric map is defined as

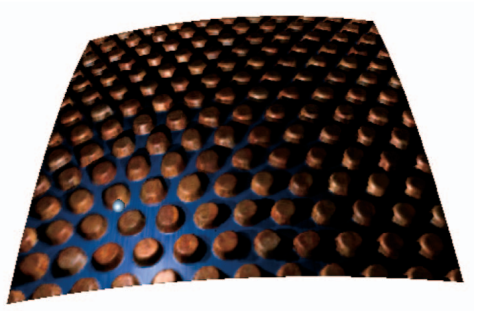


Fig. 4. Software rendering of the BTF data "tube." The small light blue point near the bottom left corner indicates the light source position.

$$\rho(s,t) = s, \qquad \phi(s,t) = \frac{\pi}{4}\frac{t}{s}.$$

The definition of $\Phi(s,t)$ on other three symmetric parts of the unit square is similar.

The elevation map $\Psi(\rho,\phi) : [0,1] \times [0,2\pi] \longmapsto \mathcal{R}^3$ is a mapping from the unit disk to the unit hemisphere, defined as: $x = \rho\sqrt{2-\rho^2}\cos(\phi)$, $y = \rho\sqrt{2-\rho^2}\sin(\phi)$, and $z = 1 - \rho^2$.

It is shown in [27] that the concentric map Φ maps evenly distributed samples in the unit square to evenly distributed samples in the unit disk (as illustrated in Fig. 5c) and the elevation map Ψ maps uniformly distributed points in the unit disk to uniformly distributed points on a unit hemisphere. Therefore, the elevated concentric map (as a concatenation Φ and Ψ) will map evenly distributed samples in the unit square to evenly distributed samples on the hemisphere. Fig. 5d shows the improved sample distribution as compared to the traditional spherical parameterization in Fig. 5b.

The elevated concentric map $\Omega(s,t)$ is invertible with

$$\begin{aligned}(s,t) = \Omega^{-1}(x,y,z) &= \Omega^{-1}(x,y,\sqrt{1-x^2-y^2}) \\ &\equiv \Omega^{-1}(x,y).\end{aligned} \tag{9}$$

Thus, we can reparameterize the BTF as follows:

$$f(\mathbf{x},\mathbf{v},\mathbf{l}) = f(\mathbf{x}, s_v, t_v, s_l, t_l),$$

where $(s_v,t_v) = \Omega^{-1}(x_v,y_v)$, $(s_l,t_l) = \Omega^{-1}(x_l,y_l)$, and (x_v,y_v), (x_l,y_l) are the x, y-components of the viewing and lighting directions. From now on, we will use these parameters for hardware-accelerated BTF rendering. Note that the above x, y-components can be easily generated from dot products of the relevant normalized vectors. It is expensive to compute the inverse map Ω^{-1} in the pixel shader.

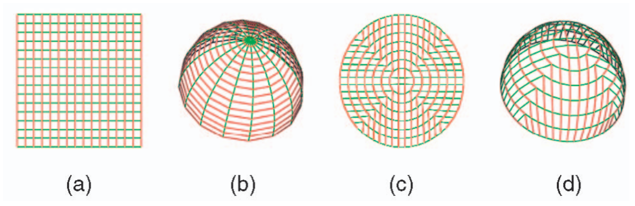


Fig. 5. Maps defined on the unit square. The green lines and red lines in (a) are mapped to the lines of the same color in (b), (c), and (d). (a) A 16×16 grid in a unit square. (b) The hemisphere of a polar map. (c) The unit disk of a concentric map. (d) The hemisphere of a concentric sphere map.

Fortunately, we can look it up by a 2D dependent texture. See Section 4.3 for details.

4.3 Hardware-Accelerated Rendering

Now, we consider hardware-accelerated rendering of BTFs on surfaces that can be parameterized on rectangular domains. Let us first examine the basic operations for rendering a BTF on a surface. As Fig. 3 illustrates, each pixel p in the rendered image is calculated through the following steps:

- Determine texture coordinate (x, y) in the BTF plane for the pixel p.
- Calculate the viewing and lighting parameters (s_v, t_v, s_l, t_l) in the local coordinate frame $\{T, N, B\}$ of the surface point $\mathbf{q}$ that corresponds to p.
- Evaluate $f(x, y, s_v, t_v, s_l, t_l)$ according to (6).

For surfaces that can be parameterized on rectangular domains, the first step is the same as in traditional texture mapping. The last two steps are very slow for a software implementation. However, these two steps are well-suited for a hardware implementation on graphics cards with vertex and pixel shaders. To carry out these two steps in hardware, we send texture coordinates, normal and tangent vectors into graphics cards as vertex data and load the geometry maps and PAFs into graphics cards as textures.

In the vertex shader, the viewing and lighting directions are transformed into each vertex's local coordinate frame using the vertex normal and tangent vectors. The x and y-components of the local lighting and viewing direction, (x_v, y_v) and (x_l, y_l), are scan converted to generate the local lighting and viewing directions for each pixel. Thus, in the pixel shader, each pixel has its own texture coordinate (x, y), local viewing direction (x_v, y_v), and local lighting direction (x_l, y_l).

In the pixel shader, we first calculate the viewing parameters (s_v, t_v) based on (x_v, y_v) and the lighting parameters (s_l, t_l) based on (x_l, y_l). Then, we evaluate the BTF at $(x, y, s_v, t_v, s_l, t_l)$ according to (6). In (6), the BTF is approximated by a sum of a few products of geometry maps and PAFs. (If the summed terms' number exceeds the graphics capability to handle in a single pass, multiple rendering passes can be used.) It is straightforward to get the value of a geometry map using the spatial coordinate (x, y) since all required geometry maps have been loaded as textures. Obtaining the values of the PAFs is harder because a PAF is four-dimensional while current graphics hardware typically does not support 4D texturing.

We calculate the PAF using the volume texturing capability available on current graphics hardware. For a given 4D PAF $p(s_v, t_l, s_l, t_l)$ with its parameter t_l discretized as $\{t_{l_0}, \ldots, t_{l_{n(t_l)-1}}\}$, we create a sequence of volume textures $\{S_0, \ldots, S_{n(t_l)-1}\}$, where $S_i(s_v, t_v, s_l) = p(s_v, t_v, s_l, t_{l_i})$. S_i is thus the 3D slice of the 4D PAF at t_{l_i}. From these volume textures we can evaluate the PAF as follows:

$$p(s_v, t_v, s_l, t_l) = \begin{cases} S_0(s_v, t_v, s_l) & \text{if } t_l < t_{l_0} \\ S_{n(t_l)-1}(s_v, t_v, s_l) & \text{if } t_l \geq t_{l_{n(t_l)-1}} \\ (1-w)S_i(s_v, t_v, s_l) + & \\ wS_{i+1}(s_v, t_v, s_l) & \text{if } t_{l_i} \leq t_l < t_{l_{i+1}}, \end{cases}$$

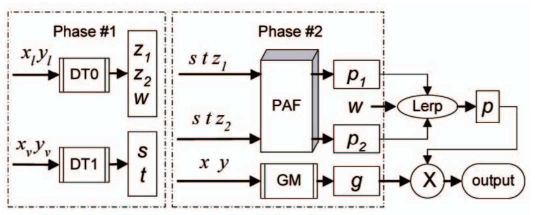


Fig. 6. Flow chart of the pixel shader to compute the multiplication of geometry maps and PAFs. Lerp denotes linear interpolation between p_1 and p_2 by w. $\otimes$ denotes multiplication. The pixel shader takes as input the BTF's spatial coordinates (x, y), the local viewing and lighting directions (x_v, y_v, x_l, y_l). DT0 is the dependent texture to store the global texture coordinates z_1, z_2 and the interpolation weight w. DT1 is another dependent texture to transform (x_v, y_v) to BTF's parameter (s_v, t_v).

where $w = (t_l - t_{l_i})/(t_{l_{i+1}} - t_{l_i})$. In practice, the volume texture sequence $\{S_0, \ldots, S_{n(t_l)-1}\}$ is combined into a single volume texture and loaded into the graphics hardware. This volume texture is indexed by the texture coordinate

$$\left(s_v, t_v, z_1 = t_{l_i} + \frac{s_l}{n(t_l)}\right).$$

Fig. 6 illustrates the design of the pixel shader. In Fig. 6, z_2 is either the same as z_1 or $t_{l_{i+1}} + s_l/n(t_l)$. $p_1 = S_i(s_v, t_v, s_l)$, whereas p_2 is either the same as p_1 or $S_{i+1}(s_v, t_v, s_l)$. The cost of a PAF evaluation is two volume texture accesses and one linear interpolation.

4.3.1 *Implementation Details*

To facilitate hardware implementation, we uniformly resample the geometry maps and PAFs so that each dimension has 2^m samples for some integer m.

We also need to carefully handle the negative values resulting from SVD decomposition. Some existing graphics cards (Nvidia GeForce3, ATI Radeon8500) allow signed texture input, but clamp negative output to 0. We resolve this problem using two rendering passes. The first pass adds $(g_i \cdot p_i)_+$ and the second pass subtracts $(-g_i \cdot p_i)_+$, where $(\cdot)_+$ denotes the operation of clamping negative values to 0.

Another issue is dealing with the floating-point representation of geometry maps and PAFs on graphics hardware that has only limited precision for textures. We need to quantize the geometry maps and PAFs. To minimize quantization errors, we scale each pair of geometry map $g_i(x, y)$ and PAF $p_i(s_v, t_v, s_l, t_l)$ by factors of s and $1/s$, respectively. This scaling does not change the product of $g_i(x, y)$ and $p_i(s_v, t_v, s_l, t_l)$. We choose s such that g_i and p_i have the same mean-squared norms, i.e.,

$$\frac{\sum_{j=0}^{n_g-1}(g_i(x_j, y_j)s)^2}{n_g} = \frac{\sum_{k=0}^{n_p-1}(p_i(s_v, t_v; s_l, t_l)/s)^2}{n_p}.$$

The results are then scaled by another factor $\sqrt{t_i}$ to the range of $(-128, 128]$ and truncated into integers. During rendering, the product $g_i \cdot p_i$ is first divided by t and then added into the frame buffer. The division operation in the pixel shader is done by shifting. It is obvious that the factor t_i should be chosen as large as possible, but without overflowing the scaled results.

Finally, the change of variables for reparamerization is not a trivial matter on current graphics hardware. Pixel shaders do not have enough power to calculate the viewing parameters (s_v, t_v) from (θ_v, ϕ_v) according to (9). Our solution is to densely sample the reparameterization function in (9) into a 2D texture and build a dependent texture in advance to store the values of (s_v, t_v). In the pixel shader, we only need to fetch these values from the dependent texture. In addition to (s_v, t_v), z_1, z_2, and w are also precomputed and loaded as a dependent texture. Thus, texture fetching in the pixel shader has two phases, as shown in Fig. 6. In phase #1, the texture coordinates of volume textures and the interpolation weight w are fetched from the two dependent textures. In phase #2, the two PAF values p_1 and p_2 are fetched from the PAF volume texture.

4.4 BTF on Arbitrary Surfaces

Our final topic of this section is rendering the BTF on an arbitrary surface mesh S. We will focus on surface BTFs synthesized using the algorithm described in Section 3.3. As mentioned, the synthesized BTF $f'(\mathbf{x}, \mathbf{v}, \mathbf{l}) = G'(\mathbf{x})^T P(\mathbf{v}, \mathbf{l})$, where $G'(\mathbf{x})$ is a set of geometry maps defined by (7). Note that $G'(\mathbf{x})$ is a vector signal defined for each vertex on the retiled mesh of target surface. In order to apply the hardware-accelerated algorithm described in Section 4.3 to the surface BTF $f'(\mathbf{x}, \mathbf{v}, \mathbf{l})$, all we need to do is to convert $G'(\mathbf{x})$ from a set of surface geometry maps defined on the vertices of the retiled mesh to a set of 2D geometry maps defined on the unit square $[0, 1] \times [0, 1]$.

The conversion of geometry maps takes two steps. First, we find a parameterization for the original surface mesh S over the parameter space $[0, 1] \times [0, 1]$. Then, we use a novel splatting algorithm to generate a set of 2D geometry maps on $[0, 1] \times [0, 1]$ using the surface geometry maps of $G'(\mathbf{x})$. It is possible to directly parameterize the densely retiled mesh S' and thus avoid the splatting step. However, the retiled mesh, typically having over 250k vertices, is too dense to allow efficient parameterization without distortion.

4.4.1 *Mesh Parameterization*

Parameterization of arbitrary surface meshes is an active research area in computer graphics (e.g., [28], [29], [30]). The source of difficulties for parameterization are twofold. First, the geometric complexity of surfaces leads to distortion of the parameterization. Second, the topology complexity of surfaces makes cutting inevitable [28]. As long as the mesh is not homeomorphic to a disk, we must cut the mesh to make it possible to map it to a rectangular parameter space. In our case, we want a parameterization that has low distortion and we find that a texture atlas consisting of a set of charts [30] works well for our purpose. Unlike [30], we build a UI that allows the user to semi-automatically cut open a mesh. A main benefit of this semi-automatic scheme is its flexibility: The user can cut the charts on an "as needed basis" to reduce distortion. Although any texture atlas scheme would work here, our experience indicates that an atlas created by an experienced user often has much less distortion than charts generated by fully automatic methods. The main drawback of our scheme is that it requires manual work. However, this is not really a bottleneck in our system because creating an atlas is much faster than surface BTF synthesis. Fig. 7a shows a texture atlas of the Stanford

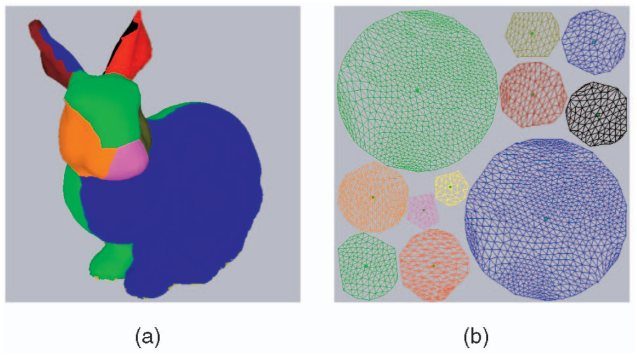


Fig. 7. Example of (a) mesh partition and (b) packing.

bunny created using our system. The individual charts are color-coded.

After the mesh is partitioned into charts, each chart can be parameterized on a polygonal region of the parameter space $[0, 1] \times [0, 1]$ by a variety of methods. We use the parameterization in [31]. The size of a polygonal region is chosen according to the size of the corresponding chart.

The last step of mesh parameterization is to pack all charts into the parameter space $[0, 1] \times [0, 1]$. An optimal packing is known to be an NP-hard problem [30]. For simplicity, we let the user interactively generate a packing through a UI. The interactions include moving, scaling, and rotating charts. Fig. 7b shows a packing obtained this way for the Stanford bunny.

4.4.2 *Splatting*

To generate a set of 2D geometry maps on the parameter space $[0, 1] \times [0, 1]$, we splat the vector value $G'(\mathbf{x}_i)$ of each vertex $\mathbf{x}_i$ of the retiled mesh S' into the parameter space. This technique was inspired by surface splatting [32]. The conventional surface splatting projects 3D points into the 2D screen space; our splatting method maps vertex values of the geometry maps into the parameter space $[0, 1] \times [0, 1]$, as follows:

```
**For each** vertex $\mathbf{x}_i$ on the retiled mesh $S'$
    find the nearest triangle $T_i$;
    compute the homography $H$ of $T_i$'s parameterization;
    calculate the Jacobian $J$ of the homography $H$;
    find the texture coordinate $(x_i, y_i)$ of $\mathbf{x}_i$
    render the vertex data as splats centered at $(x_i, y_i)$
```

The nearest triangle can be found by computing the distances between $\mathbf{x}_i$ and all the triangles in the original surface mesh S and choosing the triangle with the smallest distance. This brute-force search is very slow for large models; we speed it up by first forming a small but conservative set of candidate triangles.

For the nearest triangle T_i with vertices $\mathbf{q}_0$, $\mathbf{q}_1$, and $\mathbf{q}_2$, the homography H is obtained by (s_{q_k}, t_{q_k}), the texture coordinate of $\mathbf{q}_k$, and $(\hat{x}_{q_k}, \hat{y}_{q_k})$, the coordinate of $\mathbf{q}_k$ in the plane of the triangle T_i as follows:

$$\begin{pmatrix} s_{q_k} \\ t_{q_k} \\ 1 \end{pmatrix} = H \cdot \begin{pmatrix} \hat{x}_{q_k} \\ \hat{y}_{q_k} \\ 1 \end{pmatrix}, H = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix},$$

TABLE 2
Some of the BTF Data Used in Our Experiments

BTF data	x	y	s_v	t_v	s_l	t_l	storage
tube	128	128	8	8	8	8	192MB
hole	128	128	8	8	8	8	192MB
bean	128	128	8	8	8	8	192MB
weave	128	128	8	8	8	8	192MB
			θ_v	ϕ_v	θ_l	ϕ_l	
plaster04	235	187	7	10	7	10	617MB

The viewing and lighting directions of the BTF data "plaster04" is parameterized by the tilt and azimuth angles following [4].

where $k = 0, 1, 2$. We splat the vertex data into the parameter space by computing their contribution to each pixel in the parameter space using the algorithm proposed in [33]. The Jacobian matrix is

$$J = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}.$$

The algorithm in [33] is similar to EWA splatting in [32], except replace the Gaussian kernel [32] by the sinc function to achieve better results.

The splatting leaves some holes since the parameter space $[0, 1] \times [0, 1]$ is not fully covered by the polygonal regions resulted from parameterization, as is shown in Fig. 7b. These holes will interfere with the mipmapping of the splatting process. We avoid this problem by flood-filling the holes.

5 Results

We have implemented our surface BTF synthesis and rendering algorithms on a Pentium III 866 MHz PC with 256 MB RAM and an ATI Radeon 8500 graphics card. Some of the BTF data used in our experiments are summarized in Table 2. The BTF data of "tube" (Fig. 4), "hole" (Fig. 11), and "bean" (Fig. 12) are synthetic data generated by ray-tracing height fields. The BTF data of "weave" is generated by ray tracing a meso-structure consisting of some interleaved, deformed cylinders. The BTF data of "plaster04" is based on measurements of real-world surfaces from the CUReT database [26] (the novel views are generated using the method in [4]). Table 2 gives their sampling rates in all six dimensions.

An important issue for both BTF synthesis and rendering is the accuracy of the approximation using geometry maps and PAFs. Fig. 8 shows the power spectrum of the BTF data reported in this paper. For a given error bound, N varies for an acceptable approximation. This is not surprising as the number of terms needed depends on the complexity of mesostructures and radiance variations of the underlying BTF. Fig. 9 exhibits the quality of N-term approximations for the BTF data "hole" and $N = 6, 12, 14$. When the BTF approximation is viewed in the BTF plane, the approximation is not good enough for $N = 12$ and we need $N = 40$ to get a good approximation. On the other hand, when the BTF is synthesized on a surface, an N-term approximation with $N = 12$ looks quite good, as Fig. 10 demonstrates.

Fig. 11 and Fig. 12 show the synthesized BTFs "hole" and "bean" on the Stanford bunny using the algorithm described in Section 3.3. With the geometry maps, the synthesis efficiency improves greatly and synthesis quality

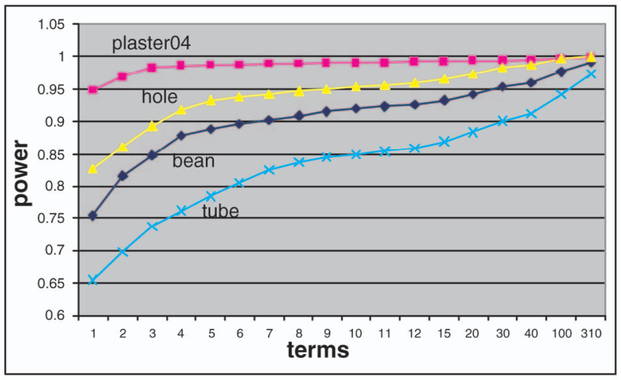


Fig. 8. Power spectrum of the BTF data "plaster04," "hole," "bean," and "tube."

is comparable to that of [5]. With the algorithm in [5], it typically takes about $60 \sim 70$ minutes to extract surface textons using K-means clustering. With the geometry maps, this process takes only $4 \sim 5$ minutes on the same machine.

Fig. 13 exhibits rendering results of N-term approximations for different N. In Fig. 13a, $N = 1$ and the rendering result resembles that of traditional texture mapping. As more terms are added, we start to see specularities, self-shadowing, and self-occlusions caused by surface mesostructures. This example shows that our algorithm can render the images in a progressive way. In Fig. 13d, $N = 10$ and the rendering of the mesostructures is very convincing. Table 3 summarizes the rendering performance for different

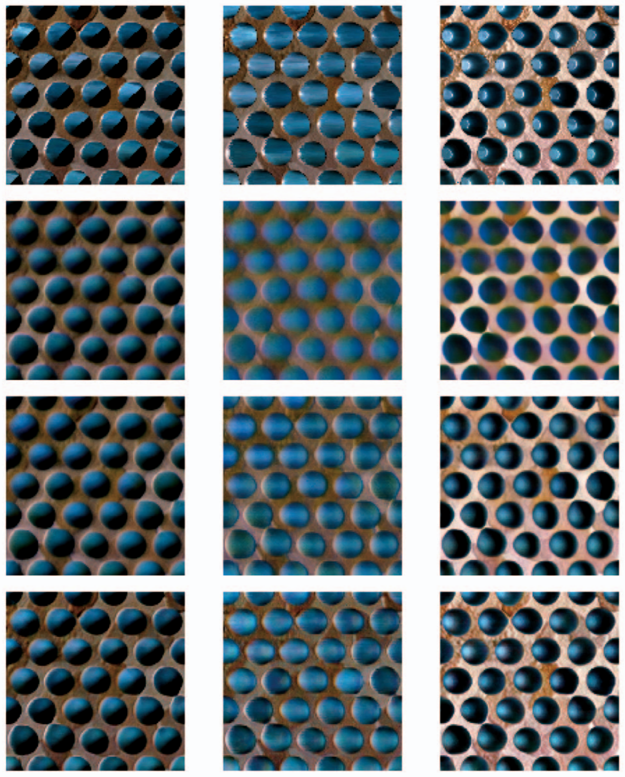


Fig. 9. Quality of N-term approximations for the BTF data "hole." The top row shows the original BTF. The second, third, and fourth rows from top to bottom are for N-term approximations with $N = 6, 12, 40$, respectively.

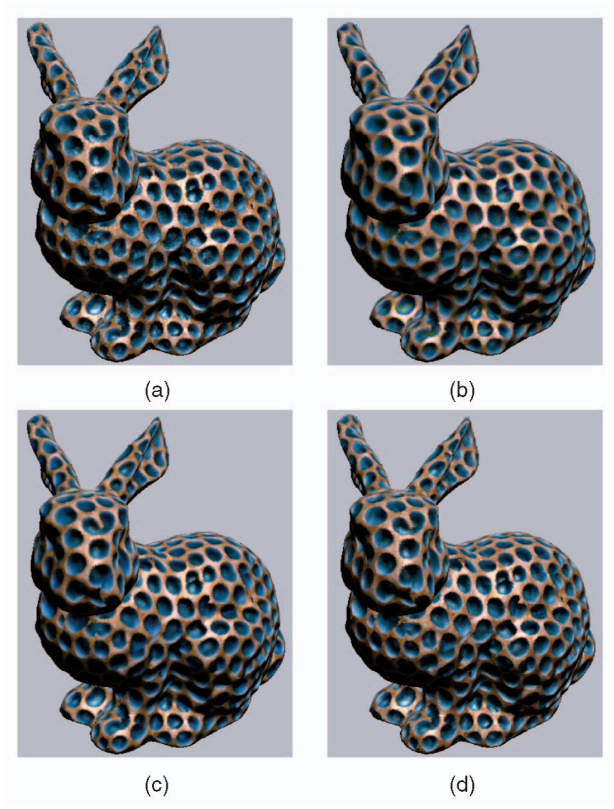


Fig. 10. Quality of surface BTF synthesized with *N*-term approximations for the BTF data "hole." (a) uses the original BTF data. (b), (c), (d) use 6, 12, 40-terms approximations, respectively. The synthesized results are visualized with the same view and light settings.

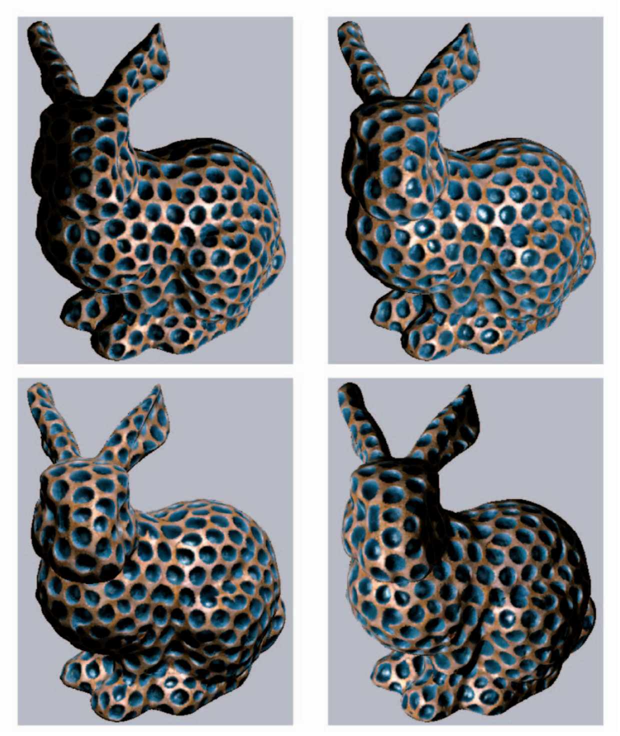


Fig. 11. Surface BTF synthesis result with the BTF data "hole" using the algorithm described in Section 3.3. Some images were rendered with the same view point but different lighting.

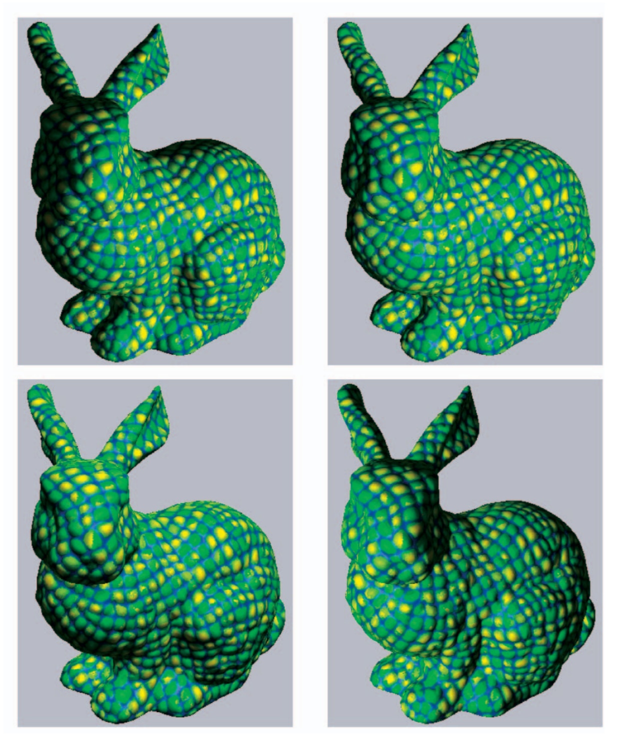
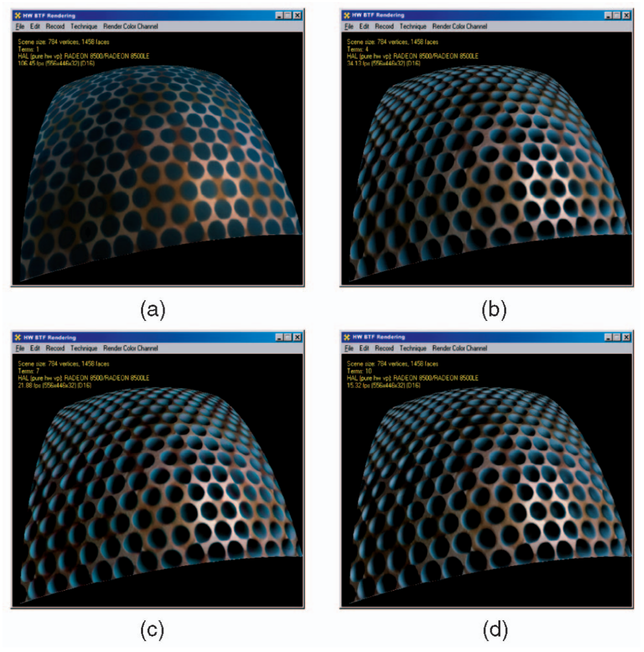


Fig. 12. Surface BTF synthesis result with the BTF data "bean" using the algorithm described in Section 3.3. Some images were rendered with the same view point but different lighting.



```
Scene size: 784 vertices, 1458 faces
Terms: 1
HAL (pure hw vp): RADEON 8500/RADEON 8500LE
106.45 fps (556x446x32) (D16)
```

```
Scene size: 784 vertices, 1458 faces
Terms: 4
HAL (pure hw vp): RADEON 8500/RADEON 8500LE
34.13 fps (556x446x32) (D16)
```

```
Scene size: 784 vertices, 1458 faces
Terms: 7
HAL (pure hw vp): RADEON 8500/RADEON 8500LE
21.88 fps (556x446x32) (D16)
```

```
Scene size: 784 vertices, 1458 faces
Terms: 10
HAL (pure hw vp): RADEON 8500/RADEON 8500LE
15.32 fps (556x446x32) (D16)
```

Fig. 13. Hardware-accelerated rendering of BTF on a parametric surface. The image size is about 550×450. The model consists of 784 vertices and 1,458 triangles.

geometry complexities with a 5-term approximation of the BTF data "plaster04."

We also implemented our hardware accelerated BTF rendering algorithm on the ATI Radeon 9700 graphics card. Some of the latest rendering results are shown in Fig. 14, Fig. 15, Fig. 16, and Fig. 17. In these figures, the BTFs are first synthesized onto the target model using the synthesis

TABLE 3
Hardware-Accelerated BTF Rendering Performance on a Pentium III 866 Mhz PC with 256 MB of RAM and an ATI Radeon 8500 Graphics Card

vertex num	tri num	fps	vertex num	tri num	fps
1100	2000	42.85	4700	7800	28.35
6600	11000	21.17	17000	30000	10.63
47890	60000	9.45			

A 5-terms approximation is used for the BTF data.

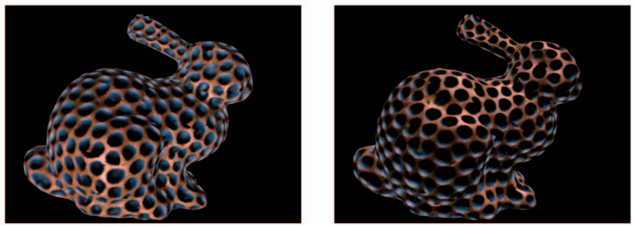


Fig. 14. Hardware-accelerated rendering of BTF "hole" with different lighting directions.

algorithm in Section 3, then synthesized result are converted from vertex signals to 2D textures as described in Section 4.4. The BTF data are all 12-term approximations. Both the Bunny model and the Tweety model consist of about 10,240 vertices and 20,476 triangles. The image size are all 800×600. We achieved about 17 fps in these experiments.

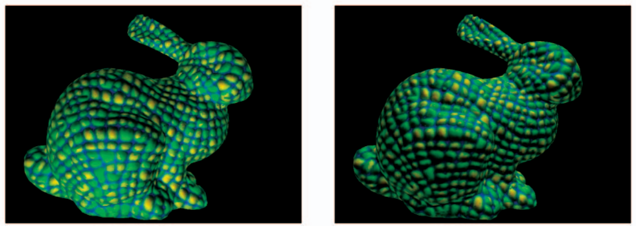


Fig. 15. Hardware-accelerated rendering of BTF "bean" with different lighting directions.

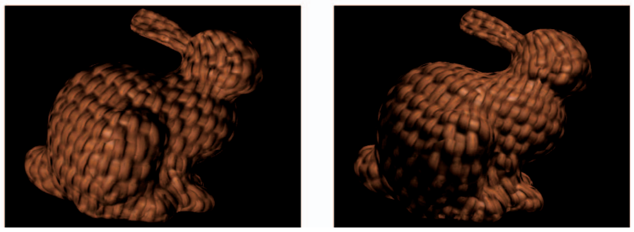


Fig. 16. Hardware-accelerated rendering of BTF "weave" with different lighting directions.

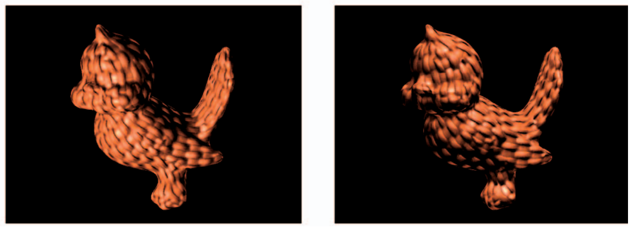


Fig. 17. Hardware-accelerated rendering of BTF "weave" with different lighting directions. These images are rendered using the same BTF data as those in Fig. 16, but the intensity of the light source is somewhat higher.

6 Conclusion and Discussion

We presented algorithms for efficient synthesis and hardware-accelerated rendering of BTFs on arbitrary surfaces. Our algorithms are based on an approximation of the BTF by a number of PAFs multiplied by geometry maps. We showed that our approximation can achieve high accuracy with a small number of geometry maps and PAFs. We also demonstrated that the PAFs and geometry maps lead to significantly more efficient surface BTF synthesis when compared to the algorithm in [5] and that the PAFs and geometry maps can be quickly rendered by leveraging the multitexturing and volume texturing capabilities of the latest graphics hardware. Our experiments indicate that our rendering algorithm can render surface BTFs in real time on a mid-level PC with the latest graphics cards.

We plan to explore a number of topics in our future work. One issue is that, for BTF decomposition, the number of terms needed to approximate the original BTF depends on the complexity of the geometry details and reflectance properties of the BTF. It will be desirable to establish a more explicit measure of this complexity. It will be more significant to develop a more effective method to decompose and compress the BTF data. Our rendering algorithm is limited to point light source. We will also investigate the new algorithms to efficiently render BTF with other types of light sources.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments. Jingdan Zhang was with Tsinghua University when he participated in this work during his internship at Microsoft Research Asia.

References

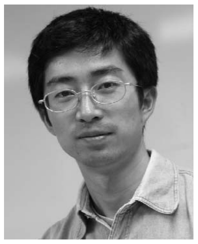
[1] J.J. Koenderink and A.J.V. Doorn, "Illuminance Texture Due to Surface Mesostructure," *J. Optical Soc. Am.*, vol. 13, no. 3, pp. 452-463, 1996.
[2] K.J. Dana, B. van Ginneken, S.K. Nayar, and J.J. Koenderink, "Reflectance and Texture of Real-World Surfaces," *ACM Trans. Graphics*, vol. 18, no. 1, pp. 1-34, Jan. 1999.
[3] T. Leung and J. Malik, "Representing and Recognizing the Visual Appearance of Materials Using 3D Textons," *Int'l J. Computer Vision*, vol. 43, no. 1, pp. 29-44, June 2001.
[4] X. Liu, Y. Yu, and H.-Y. Shum, "Synthesizing Bidirectional Texture Functions for Real-World Surfaces," *Proc. SIGGRAPH 2001*, pp. 97-106, Aug. 2001.
[5] X. Tong, J. Zhang, L. Liu, X. Wang, B. Guo, and H.-Y. Shum, "Synthesis of Bidirectional Texture Functions on Arbitrary Surfaces," *ACM Trans. Graphics*, vol. 21, no. 3, pp. 665-672, July 2002.
[6] K.J. Dana, B. van Ginneken, S.K. Nayar, and J.J. Koenderink, "Reflectance and Texture of Real-World Surfaces," *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, pp. 151-157, 1997.
[7] L.-Y. Wei and M. Levoy, "Texture Synthesis over Arbitrary Manifold Surfaces," *Proc. SIGGRAPH 2001*, pp. 355-360, Aug. 2001.
[8] G. Turk, "Texture Synthesis on Surfaces," *Proc. SIGGRAPH 2001*, pp. 347-354, Aug. 2001.

[9] L. Ying, A. Hertzmann, H. Biermann, and D. Zorin, "Texture and Shape Synthesis on Surfaces," *Proc. Eurographics Rendering Workshop 2001,* pp. 301-312, June 2001.
[10] J. Maillot, H. Yahia, and A. Verroust, "Interactive Texture Mapping," *Proc. SIGGRAPH '93,* pp. 27-34, Aug. 1993.
[11] E. Praun, A. Finkelstein, and H. Hoppe, "Lapped Textures," *Proc. SIGGRAPH 2000,* pp. 465-470, July 2000.
[12] W.-C. Chen, J.-Y. Bouguet, M.H. Chu, and R. Grzeszczuk, "Light Field Mapping: Efficient Representation and Hardware Rendering of Surface Light Fields," *ACM Trans. Graphics,* vol. 21, no. 3, pp. 447-456, July 2002.
[13] J. Kautz and M.D. McCool, "Interactive Rendering with Arbitrary BRDFs Using Separable Approximations," *Proc. Eurographics Rendering Workshop,* pp. 28-292, 1999.
[14] M.D. McCool, J. Ang, and A. Ahmad, "Homomorphic Factorization of BRDFs for High-Performance Rendering," *Proc. Siggraph 2001,* pp. 171-178, Aug. 2001.
[15] K.J. Dana and S.K. Nayar, "Histogram Model for 3D Textures," *Proc. IEEE Conf. Computer Vision and Pattern Recognition,* pp. 618-624, 1998.
[16] K. Daubert, H.P.A. Lensch, W. Heidrich, and H.-P. Seidel, "Efficient Cloth Modeling and Rendering," *Proc. Eurographics Rendering Workshop,* pp. 63-70, June 2001.
[17] T. Malzbender, D. Gelb, and H. Wolters, "Polynomial Texture Maps," *Proc. SIGGRAPH 2001,* pp. 519-528, Aug. 2001.
[18] J.M. Dischler, "Efficiently Rendering Macro Geometric Surface Structures Using Bi-Directional Texture Functions," *Proc. Eurographics Rendering Workshop,* pp. 16-180, 1998.
[19] D.N. Wood, D.I. Azuma, K. Aldinger, B. Curless, T. Duchamp, D.H. Salesin, and W. Stuetzle, "Surface Light Fields for 3D Photography," *Proc. SIGGRAPH 2000,* pp. 287-296, Aug. 2000.
[20] K. Nishino, Y. Sato, and K. Ikeuchi, "Eigen-Texture Method: Appearance Compression Based on 3D Model," *Proc. IEEE Conf. Computer Vision and Pattern Recognition,* vol. 1, pp. 618-624, June 1999.
[21] G. Gorla, V. Interrante, and G. Sapiro, "Growing Fitted Textures," *SIGGRAPH 2001 Sketches and Applications,* p. 191, Aug. 2001.
[22] L.-Y. Wei and M. Levoy, "Fast Texture Synthesis Using Tree-Structured Vector Quantization," *Proc. SIGGRAPH 2000,* pp. 479-488, 2000.
[23] A.A. Efros and T.K. Leung, "Texture Synthesis by Non-Parametric Sampling," *Proc. IEEE Int'l Conf. Computer Vision,* pp. 1033-1038, Sept. 1999.
[24] W. Heidrich and H.-P. Seidel, "Realistic, Hardware-Accelerated Shading andLighting," *Proc. SIGGRAPH '99,* pp. 171-178, 1999.
[25] G. Turk, "Re-Tiling Polygonal Surfaces," *Proc. SIGGRAPH '92,* pp. 55-64, July 1992.
[26] CUReT, http://www.cs.columbia.edu/cave/curet, year?
[27] P. Shirley and K. Chiu, "A Low Distortion Map between Disk and Square," *J. Graphics Tools,* vol. 2, no. 3, pp. 45-52, 1997.
[28] X. Gu, S.J. Gortler, and H. Hoppe, "Geometry Images," *ACM Trans. Graphics,* vol. 21, no. 3, pp. 355-361, July 2002.
[29] P. Alliez, M. Meyer, and M. Desbrun, "Interactive Geometry Remeshing," *ACM Trans. Graphics,* vol. 21, no. 3, pp. 347-354, July 2002.
[30] B. Lévy, S. Petitjean, N. Ray, and J. Maillot, "Least Squares Conformal Maps for Automatic Texture Atlas Generation," *ACM Trans. Graphics,* vol. 21, no. 3, pp. 362-371, July 2002.
[31] P.V. Sander, J. Snyder, S.J. Gortler, and H. Hoppe, "Texture Mapping Progressive Meshes," *Proc. SIGGRAPH 2001,* pp. 409-416, Aug. 2001.
[32] M. Zwicker, H. Pfister, J. van Baar, and M. Gross, "Surface Splatting," *Proc. SIGGRAPH 2001,* pp. 371-378, Aug. 2001.
[33] K. Deng, J. Zhang, L. Wang, and B. Guo, "Texture Mapping with a Jacobian-Based Spatially-Variant Filter," *Proc. IEEE Pacific Graphics,* pp. 460-461, Oct. 2002.

Xinguo Liu received the PhD degree and BS degree from the Department of Mathematics at Zhejiang University. He is an associate researcher in the Visual Computing Group at Microsoft Research Asia. Before joining Microsoft, he was with the State Key Lab. of *CAD&CG* at Zhejiang University. His main research interests are in geometry processing, image-based appearance modeling, real-time rendering, and character animation.

Yaohua Hu received the BS degree from the School of Electronic Information and Control Engineering, Beijing University of Technology. He is an assistant researcher in the Internet Graphics Group at Microsoft Research Asia. His main interest is the impact of science on human life, such as games and graphics.

Jingdan Zhang received the BE degree and MS degrees from the Computer Science and Technology Department of Tsinghua University in 2000 and 2003. He is currently a graduate student at the University of North Carolina, Chapel Hill. His main research interests are computer graphics and related applications.



Xin Tong received the PhD degree from Tsinghua University and the MS and BS degrees from ZheJiang University. He is a researcher in the Internet Graphics Group at Microsoft Research Asia. His research interests are in appearance modeling and rendering, texture synthesis, image-based modeling and rendering, and mesh compression.

Baining Guo received the PhD and MS degrees from Cornell University and the BS degree from Beijing University. He is the research manager of the Internet Graphics Group at Microsoft Research Asia. Before joining Microsoft, he was with the Microcomputer Research Labs at Intel Corporation in Santa Clara, California. His main research interests are in modeling and rendering. His current projects include networked games, appearance modeling, texture synthesis, natural phenomena, and image-based rendering.

Heung-Yeung Shum joined Microsoft Research after receiving the PhD degree in robotics from the School of Computer Science, Carnegie Mellon University in 1996. He is a senior researcher and the Managing Director of Microsoft Research Asia (MSRA). He has authored or coauthored more than 100 papers in computer vision, computer graphics, and robotics and holds more than 20 US patents. He is on the editorial boards for the *IEEE Transactions on Circit System Video Technology*, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, and *Graphical Models*. His research interests are computer vision, computer graphics, video representation, learning, and visual recognition. He is a senior member of the IEEE.