Construction and Rendering of Trimmed Blending Surfaces with Sharp Features on a GPU

Dae-Hyun Ko, Jieun Lee, Seong-Jae Lim, and Seung-Hyun Yoon

We construct surfaces with darts, creases, and corners by blending different types of local geometries. We also render these surfaces efficiently using programmable graphics hardware. Points on the blending surface are evaluated using simplified computation which can easily be performed on a graphics processing unit. Results show an eighteen-fold to twenty-fold increase in rendering speed over a CPU version. We also demonstrate how these surfaces can be trimmed using textures.

Keywords: Blending surface, GPU-based rendering, vertex program, fragment program, sharp features.

Manuscript received Mar. 13, 2010; revised Oct. 6, 2010; accepted Oct. 25, 2010.

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (No. 2010-0015293), and supported by the MKE (Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program supervised by the NIPA (National IT Industry Promotion Agency) (NIPA-2010-C1090-1021-0013), and also supported by the Strategic Technology Development Program of MCST/MKE/KEIT [KI001798, Development of Full 3D Reconstruction Technology for Broadcasting Communication Fusion].

Dae-Hyun Ko (Phone: +82 10 2240 3786, email: daehyun.ko@gmail.com) is with the DMC R&D Center, Samsung Electronics, Suwon, Rep. of Korea.

Jieun Lee (email: JieunJadeLee@gmail.com) is with the Department of Computer Engineering, Chosun University, Gwangju, Rep. of Korea.

Seong-Jae Lim (email: sjlim@etri.re.kr) is with the Contents Research Division, ETRI, Daejeon, Rep. of Korea.

Seung-Hyun Yoon (corresponding author, email: shyun@dongguk.edu) is with the Department of Multimedia Engineering, Dongguk University, Seoul, Rep. of Korea.

doi:10.4218/etrij.11.1510.0091

I. Introduction

Modern graphics processing units (GPUs) have a programmable graphics pipeline which allows various rendering tasks in computer graphics and animation to be addressed at a higher level than hitherto. One such task is the direct rendering of application-oriented shape representations, such as non-uniform rational B-spline (NURBS) surfaces, implicit surfaces, and subdivision surfaces. These formulations are compact and allow differential properties to be computed with precision. However, these advantages may be outweighed by the need to tessellate such surfaces for rendering. To avoid this drawback, many GPU-based techniques [1]-[4] for rendering analytic surfaces directly have been proposed.

In this context, our current concern is the manifold-based blending surface construction scheme recently proposed by Ying and Zorin [5]. They construct a C^{∞}-continuous surface by blending local geometries on overlapping charts. This new technique provides many advantages, such as analytical formulae for surface properties, simplicity, and local control. However, the evaluation of points on one of these blends requires rather involved computations of evaluation parameters, local geometries, and blending functions. However, we note that most of these operations can easily be performed by a GPU.

We extend the scheme of Ying and Zorin [5] to propose a blending surface that can represent sharp features, such as darts, corners, and creases. In subdivision surfaces, features of this sort can be realized by applying different subdivision rules [6]-[8]. However, very little effort has been applied to the expression of sharp features in blending surfaces [9]. We address this problem with a technique in which features of this sort are initially created as sharp edges on a control mesh.

Depending on the topology of these edges, different types of local geometries, such as points, curves, or surface patches, are created. These are then blended together to generate the required features.

We go on to present an efficient algorithm for rendering and trimming our extended blending surfaces using a GPU. Points on the surface are evaluated by a repetitive sequence of matrix and vector computations which can easily be implemented in a GPU. Experimental results show an eighteen-fold to twenty-fold increase in rendering speed, compared with a CPU-based implementation. We also demonstrate how these new surfaces can be trimmed in the GPU using textures.

The rest of this paper is organized as follows. In section II, we review related recent work. In section III, we briefly summarize the blending scheme proposed by Ying and Zorin [5]. Our extension of their scheme to sharp features is explained in section IV. We then propose an efficient algorithm for rendering and trimming a blended surface in section V and compare the performance of GPU-based and CPU-based implementations in section VI. Finally, we conclude this paper in section VII.

II. Related Work

We look first at blends based on manifold theory and then briefly review GPU-based techniques for rendering related surface representations.

1. Blending Surfaces

Grim and Hughes [10] used a technique based on the concept of the manifolds to construct surfaces of arbitrary topology. This technique has subsequently been applied to the parameterization of surfaces [11] and to the fitting of surfaces to point cloud data [12]. Cotrina and Pla [13] described a similar algorithm for constructing C^k-continuous surfaces with boundary curves of B-spline form. This formulation may be seen as a generalized B-spline surface. An even more general version of this approach was subsequently proposed by Cotrina and others [14], which can produce three different types of surfaces. However, these techniques require complicated transition functions.

Ying and Zorin [5] have introduced a technique for constructing a smooth surface of arbitrary topology by constructing charts in the complex plane and combining them with transition functions with simple forms. This approach provides both C^∞ continuity and local control of the surface; moreover, the resulting surfaces are visually satisfactory. This is the jumping-off point for our technique. Gu and others [15] have put forward a theoretical and computational framework for manifold splines defined on a control mesh of arbitrary topology. They also provided a practical algorithm for creating a triangular B-spline surface on such a control mesh.

A new approach to the construction of rational manifold surfaces has been proposed by Vecchia and others [16]. Their surfaces consist of rational tensor-product and triangular surface patches, which are combined using transition functions obtained by sub-chart parameterization. Recently, Vecchia and Jüttler [9] further extended this technique to include different types of sharp features. We also aim to generate sharp features on a blending surface, but our technique is different, and we also consider the computational aspects in more detail.

2. Rendering Using a GPU

The capability of modern GPUs to perform general computations has been widely used, and GPU-based rendering techniques have recently been proposed for several shape representations. Guthe and others [1] put forward a GPU-based technique for rendering and trimming spline surfaces approximated by bicubic Bézier surfaces. Points on the latter are computed in a vertex program, and the surface is then trimmed using a texture in a fragment program. However, their technique cannot render the spline surface exactly since it has to be approximated. Krishnamurthy and others [2] computed points on a trimmed NURBS surface directly using a fragment program. They store the control points, the knot vectors, and pre-computed values of the basis functions for sampled parameters in texture memory and compute points on the surface at the fragment programming stage. Although their technique shows a fifty-fold increase in rendering speed over a CPU version, the values of the basis functions must be pre-computed and maintained during rendering. This contrasts with our blending approach, in which local geometries are evaluated efficiently in the GPU. A GPU-based technique for rendering of spline surfaces by ray-casting was proposed by Pabst and others [17]. They render the convex hull of a spline surface, and only the fragments in which a ray intersects the surface are subsequently shaded in a fragment program.

Turning to implicit surfaces, Sigg and others [3] presented a GPU-based technique for ray-casting quadrics, and Kanai and others [18] have suggested a method for ray-casting sparse low-degree implicit surfaces. In the latter, intersections of rays with the surface are computed in a fragment program. A GPU-based method for rendering iso-surfaces extracted from tetrahedral grids has been proposed by Reck and others [19]. Tetrahedra are streamed into a vertex program which extracts the iso-surface for a given value and renders it immediately.

Yasui and others [4] introduced GPU-based rendering for the reflection lines of a subdivision surface, and Bolz and others

[20] have put forward a technique for computing points on a Catmull-Clark subdivision surface using a GPU. In the latter technique, tessellations of the basic functions are precomputed and stored in texture memory. At run-time, a fragment program receives the control mesh and evaluates the corresponding subdivision surface.

III. Manifold-Based Blending

Now we will summarize the approach of Ying and Zorin [5] to constructing a blending surface.

1. Charts

For each vertex $\mathbf{v}_i$ of a control mesh, a chart U_i is defined as a subset of the complex plane whose shape depends on the valence k of the vertex. Figure 1 shows the construction of a chart for a vertex of valence 6. The chart is formed by transforming a unit square [0, 1]×[0, 1] in the complex plane using the conformal mapping $z \to z^{4/k}$ (Fig. 1(b)) and then rotating them through an angle $\phi = 2\pi h / k$ as follows (Fig. 1(c)):

$$f(z) = (r(\cos\theta + i\sin\theta))^{\frac{4}{k}}(\cos\phi + i\sin\phi)$$
$$= r^{\frac{4}{k}}(\cos((4\theta + 2\pi h)/k) + i\sin((4\theta + 2\pi h)/k)),$$

where $z = r(\cos\theta + i\sin\theta) \in [0,1]\times[0,1]$, and h is a counter-clockwise index which enumerates the edges connected to the vertex $\mathbf{v}_i$. The final chart U_i is the union of these transformed squares for $h = 0, 1, \ldots, k–1$ (Fig. 1(d)).

If the control mesh has quadrilateral faces, then each face has four corresponding charts. To compute the local geometry and blending function on each chart, a sampling parameter z in a unit square is transformed to an evaluation parameter z_i on each chart U_i, $i = 1, 2, 3, 4$ (see section V.2 for more details).

2. Local Geometries

A local geometry $P_i(z)$: $U_i \to \mathbf{R}^3$ is defined on each chart to represent the shape of a blending surface around a vertex $\mathbf{v}_i$ as follows:

$$P_i(z) = [p_{i,x}(u,v) \quad p_{i,y}(u,v) \quad p_{i,z}(u,v)]^T,$$

where the parameters u and v are the real and imaginary parts of $z \in U_i$, respectively. Ying and Zorin [5] use polynomials of high degree to represent local geometries that approximate the subdivision surface of a control mesh, whereas we employ different types of local geometry on each chart (see section IV.1 for more details).

3. Blending Functions

On each chart U_i, we define a smooth blending function $w_i(z)$: $U_i \to \mathbf{R}$, such that $w_i(z)=1$ at $z=0+i0$ and $w_i(z)=0$ at $z \in \partial U_i$, where ∂U_i is the boundary of chart U_i. We formulate a univariate scalar function $\eta(t)$ and then construct a blending function over the unit square [0, 1]×[0, 1] as a product of $\eta(u)\eta(v)$. The final blending function $w_i(z)$ is then constructed by mapping $\eta(u)\eta(v)$ to U_i; the form of this mapping depends on the valence of the vertex $\mathbf{v}_i$. Figure 2 shows the construction of the blending function for a vertex of valence 6. Ying and Zorin [5] use a function $\eta(t)$ which is infinitely differentiable at t=0 and t=1 to construct a C^∞-continuous blending surface, whereas we employ a Hermite interpolant to facilitate a GPU implementation.

The final surface is constructed by blending all the local geometries together. For a given parameter value z in the unit square [0, 1]×[0, 1] that corresponds to a face, a point $S(z)$ on the surface is computed as

$$S(z) = \sum_{i=1}^{4} w_i(z_i) P_i(z_i), \qquad (1)$$

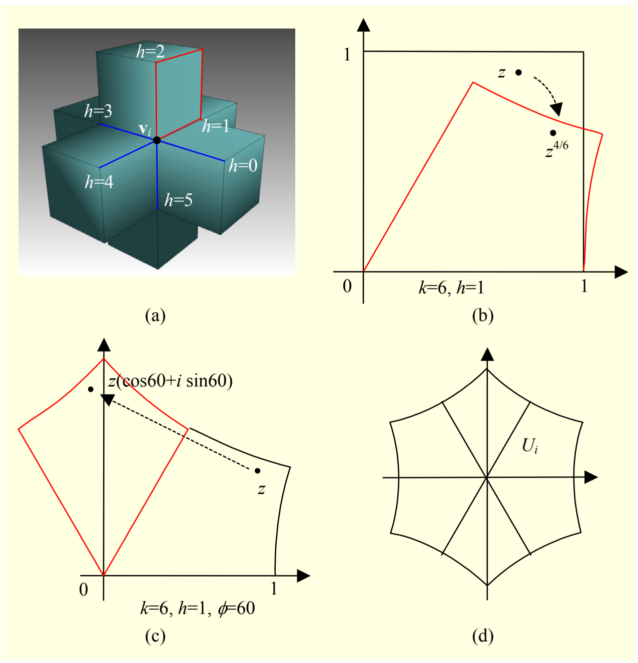

Fig. 1. Constructing the chart for a vertex of valence 6.

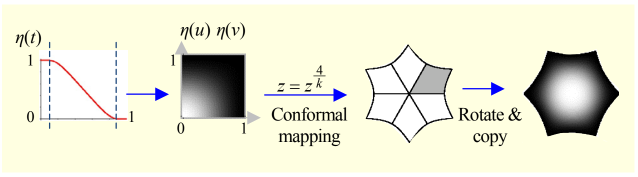

Fig. 2. Construction of a blending function $w_i(z)$: tone shows value, from black (0) to white (1).

where z_i is the corresponding parameter on each of the charts U_i, i=1, 2, 3, 4, that share the face (see Fig. 9), and $P_i(z_i)$ are the local geometries.

IV. Extending Blending Surface to Incorporate Sharp Features

We now extend the formulation of the blending surface described above to model sharp features.

1. Local Geometries

We take a new approach to the definition of local geometries. Edges on a control mesh can be labeled as sharp by the user, and then a different type of local geometry is created on each chart, which depends on the number and configuration of the sharp edges that meet at a vertex. Figures 3(a), (b), (c), and (d) show how sharp edges influence the local geometry.

Let n be the number of sharp edges that meet at a vertex $\mathbf{v}_i$. For n=0, we use a surface patch $P_i(z)$: $U_i \rightarrow \mathbf{R}^3$ as the local geometry on the corresponding chart U_i. This patch has the following bi-quadratic polynomial representation, which approximately interpolates the positions of the vertices neighboring $\mathbf{v}_i$:

$$P_i(z) = \begin{bmatrix} 1 & u & u^2 \end{bmatrix} \begin{bmatrix} \mathbf{c}^i_{1,1} & \mathbf{c}^i_{1,2} & \mathbf{c}^i_{1,3} \\ \mathbf{c}^i_{2,1} & \mathbf{c}^i_{2,2} & \mathbf{c}^i_{2,3} \\ \mathbf{c}^i_{3,1} & \mathbf{c}^i_{3,2} & \mathbf{c}^i_{3,3} \end{bmatrix} \begin{bmatrix} 1 \\ v \\ v^2 \end{bmatrix}, \qquad (2)$$

where $\mathbf{c}^i_{m,n} = \begin{bmatrix} x^i_{m,n} & y^i_{m,n} & z^i_{m,n} \end{bmatrix}^T$, and the parameters u and v are the real and imaginary parts of $z \in U_i$, respectively.

The coefficient vectors $\mathbf{c}^i_{m,n}$ in (2) can be determined by minimizing the following functional:

$$L(\mathbf{c}^i_{1,1}, \mathbf{c}^i_{1,2}, \ldots, \mathbf{c}^i_{3,3}) = \sum_{l=1}^{k} \left\| v^l_i - P_i(z_l) \right\|^2,$$

subject to $P_i(0+i0)=\mathbf{v}_i$, where k is the valence of a vertex $\mathbf{v}_i$, and $\mathbf{v}^l_i$ is the l-th neighboring vertex of $\mathbf{v}_i$. The constraint $P_i(0+i0)=\mathbf{v}_i$ forces the coefficient $\mathbf{c}^i_{1,1}$ to be $\mathbf{v}_i$, so that the surface goes through the vertex $\mathbf{v}_i$, which is the origin of each chart. Figure 4(a) shows the vertices of a control mesh around the vertex $\mathbf{v}_i$, and Fig. 4(b) shows the corresponding surface patch, which passes through the central vertex $\mathbf{v}_i$.

For n=1 or $n \geq 3$, we simply take the vertex $\mathbf{v}_i$ itself as the local geometry:

$$P_i(z) = \mathbf{v}_i, \quad \text{for all } z \in U_i.$$

In the special case in which all local geometries are vertices of a control mesh, the blending surface will be the original control mesh itself.

When n=2, we take a quadratic Bézier curve $P_i(z)$ as the local geometry:

$$P_i(z) = \sum_{j=0}^{2} \mathbf{q}^i_j B^2_j(\sigma_i(z)), \qquad (3)$$

where $\mathbf{q}^i_j$ are control points, and $B^2_j()$ are quadratic Bernstein basis functions. If e_1=($\mathbf{v}_j$, $\mathbf{v}_i$) and e_2=($\mathbf{v}_i$, $\mathbf{v}_k$) are the first and second sharp edges of $\mathbf{v}_i$, respectively, then the control points are determined as

$$\mathbf{q}^i_0 = \mathbf{v}_j, \ \mathbf{q}^i_1 = \mathbf{v}_i, \text{ and } \mathbf{q}^i_2 = \mathbf{v}_k.$$

The curve parameter for $z \in U_i$ is computed by a function

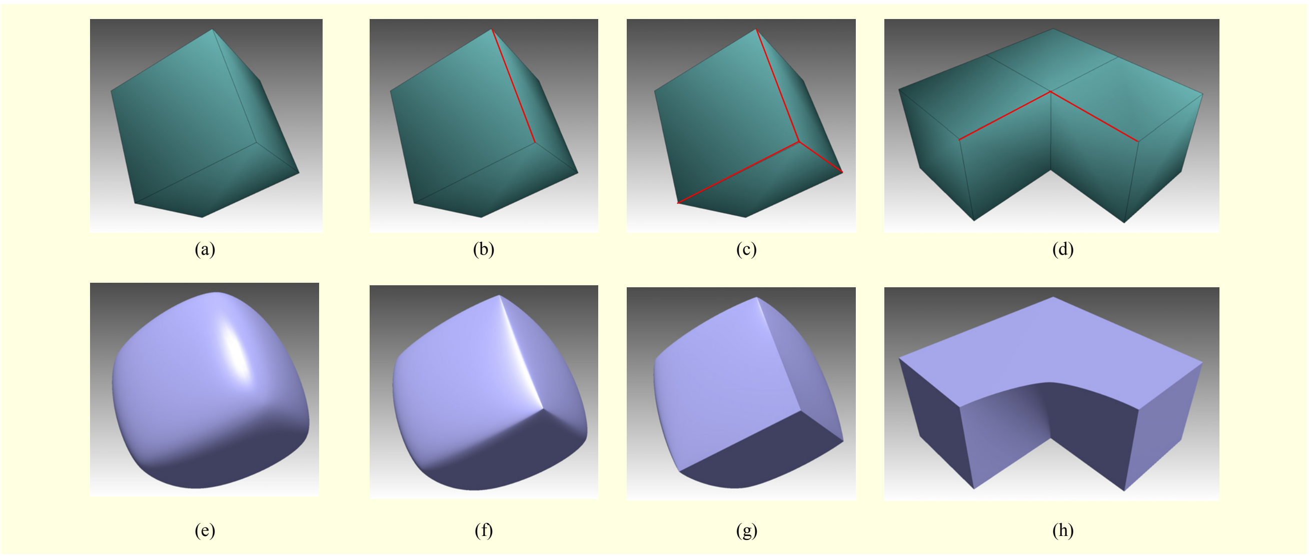

Fig. 3. Creating different types of sharp feature by blending different types of local geometry. Sharp edges are shown in red.

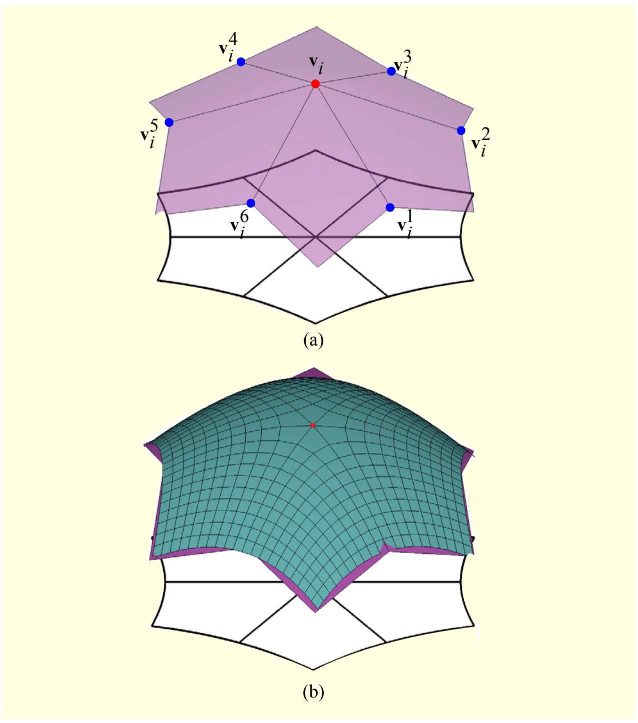

Fig. 4. Local geometry: (a) vertex $\mathbf{v}_i$ and its neighboring vertices and (b) constructed surface patch $P_i(z)$.

Table 1. Extended local geometries.

Numbers of sharp edges	Sharp features	Local geometry
0	Smooth	Surface patch
1	Dart	Point
2	Crease	Curve
≥ 3	Corner	Point

$\sigma_i(z): U_i \to [0,1]$ (see section V.3).

Table 1 lists the possible local geometries and the corresponding sharp features that result. A smooth region is created when four surface patches are blended (Fig. 3(e)), and a dart is created between two vertices which share a sharp edge (Fig. 3(f)). A crease is created when a quadratic curve is blended with other types of local geometries (Fig. 3(h)), and a corner is created at the vertex when $n\geq 3$ (Fig. 3(g)).

2. Blending Functions

We take a similar approach to the construction of a blending function to Ying and Zorin [5] (see section III.3). Our only deviation from their scheme is to use the Hermite interpolant $\eta_1(t)=2t^3-3t^2+1$, as shown in Fig. 5. To create a C^k-continuous blending surface, $\eta_1(t)$ and $\eta_2(t)$ $(=\eta_1(1-t))$ need to be C^k-continuous at t=0 and t=1, respectively. Since $\eta'_1(0)=\eta'_2(0)$ and $\eta'_1(1)=\eta'_2(1)$, the Hermite interpolant guarantees that the resulting surface is C^1-continuous, except at the sharp features defined by the user. We use Hermite functions because they can be evaluated by modern GPUs, which allows our blending scheme to be implemented efficiently as a vertex program. Figures 6(a) and (b) show the blending functions on charts with different valences using the Hermite interpolant $\eta_1(t)$.

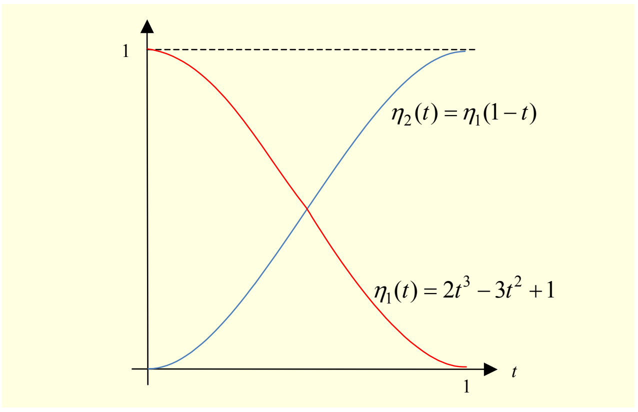

Fig. 5. Hermite interpolants with $\eta'_1(0)=\eta'_2(0)$ and $\eta'_1(1)=\eta'_2(1)$ producing a C^1-continuous blending surface.

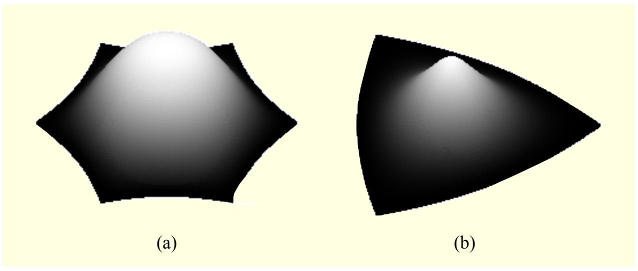

Fig. 6. Blending functions using Hermite interpolant on charts of valence: (a) 6 and (b) 3.

V. GPU-Based Evaluation and Rendering

We now show how to evaluate points on the blending surface using programmable graphics hardware. The overall flow of the algorithm is shown in Fig. 7.

1. Preprocessing

We start by computing the coefficients of the local geometries for all the vertices of a control mesh and then create a texture image from their coefficients (in the RGB channels). This is done in the CPU. Figure 8 shows the coefficients of different types of local geometry stored in a texture image with 9 rows and w columns, where w is the number of vertices of the control mesh. The texture images are then transmitted to the texture memory of the GPU.

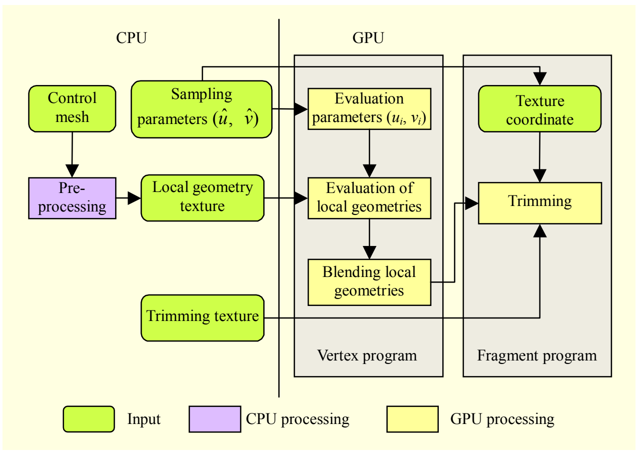

Fig. 7. Algorithm for evaluating and rendering trimmed blending surface.

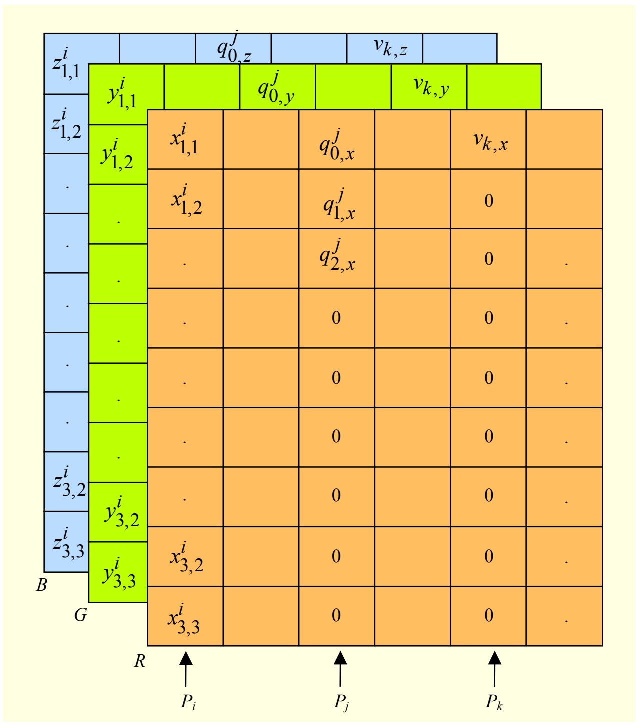

Fig. 8. Coefficients of local geometries for charts U_i, U_j, and U_k stored in a texture image. P_i is a surface patch, P_j is a curve, and P_k is the vertex $\mathbf{v}_k$.

2. Computing Evaluation Parameters

As shown in Fig. 7, the evaluation process starts with the transmission of sampling parameters to a vertex program in the graphics pipeline. We assume that the parametric domain of each face is a unit square [0, 1]×[0, 1] on which the parameter $\hat{z} = \hat{u} + i\hat{v}$ is uniformly sampled as shown in Fig. 9. The values of the sampling parameter $\hat{z} = \hat{u} + i\hat{v}$ are then transformed to the coordinates $z_i = u_i + iv_i$ on each chart U_i, i = 1, 2, 3, 4, that corresponds to the face. For this, four polar

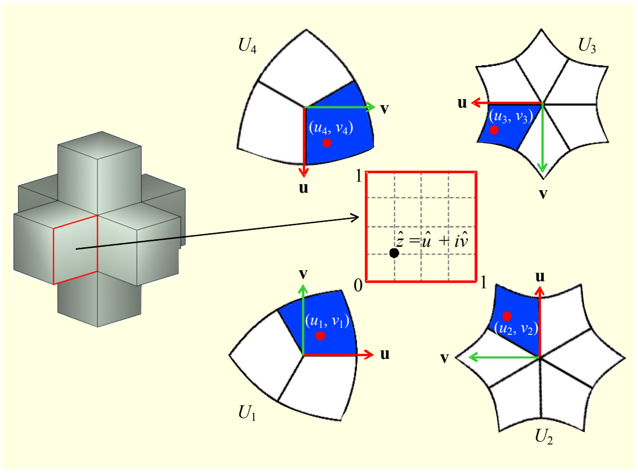

Fig. 9. Sampling parameter $\hat{z} = \hat{u} + i\hat{v}$ on unit square transformed to evaluation parameter z_i on each chart U_i, i = 1, 2, 3, 4.

coordinates $\hat{z}_i = (\hat{r}_i, \hat{\theta}_i)$, i = 1, 2, 3, 4, expressed in the coordinate system of each chart, are computed as

$$(\hat{r}_1, \hat{\theta}_1) = (\sqrt{\hat{u}^2 + \hat{v}^2}, \text{atan2}(\hat{v}/\hat{u})),$$
$$(\hat{r}_2, \hat{\theta}_2) = (\sqrt{\hat{v}^2 + (1-\hat{u})^2}, \text{atan2}((1-\hat{u})/\hat{v})),$$
$$(\hat{r}_3, \hat{\theta}_3) = (\sqrt{(1-\hat{u})^2 + (1-\hat{v})^2}, \text{atan2}((1-\hat{v})/(1-\hat{u}))),$$
$$(\hat{r}_4, \hat{\theta}_4) = (\sqrt{(1-\hat{v})^2 + \hat{u}^2}, \text{atan2}(\hat{u}/(1-\hat{v}))).$$

The evaluation parameters on a chart of valence k_i are computed in polar form z_i=(r_i, θ_i) on each chart U_i, i = 1, 2, 3, 4, as

$$z_i = (\hat{z}_i)^{4/k_i} = (\hat{r}_i(\cos\hat{\theta}_i + \sin\hat{\theta}_i))^{4/k_i} = (\hat{r}_i^{4/k_i}, 4\hat{\theta}_i/k_i) = (r_i, \theta_i).$$

To compute points on the local geometries, the evaluation parameters need to be represented in the form z_i=u_i+iv_i on the corresponding chart U_i. To achieve this, we form a 4×4 diagonal matrix R with diagonal elements r_i and a 4×2 matrix Θ with sine and cosine of θ_i. The final evaluation parameter z_i=u_i+iv_i on each chart U_i, i = 1, 2, 3, 4, is computed as

$$\begin{bmatrix} u_1 & v_1 \\ u_2 & v_2 \\ u_3 & v_3 \\ u_4 & v_4 \end{bmatrix} = R\Theta = \begin{bmatrix} r_1 & 0 & 0 & 0 \\ 0 & r_2 & 0 & 0 \\ 0 & 0 & r_3 & 0 \\ 0 & 0 & 0 & r_4 \end{bmatrix} \begin{bmatrix} \cos\theta_1 & \sin\theta_1 \\ \cos\theta_2 & \sin\theta_2 \\ \cos\theta_3 & \sin\theta_3 \\ \cos\theta_4 & \sin\theta_4 \end{bmatrix}.$$

Since GPUs provide an efficient mechanism for evaluating sin, cos, and atan2 functions, together with multiplication of vectors and matrices which are built-in data types, the evaluation parameters z_i=u_i+iv_i can be computed very efficiently. Once the four evaluation parameters have been determined, we can evaluate points on the local geometries and the blending functions.

3. Evaluation of Local Geometries

It would be inefficient to employ different procedures for evaluating each of the three types of local geometry. We will now show how a unified evaluation procedure can be used for all three types of local geometry without reducing the efficiency of the GPU. This evaluation procedure consists of three steps:

Step 1. Fetch the texture memory and recover the coefficient matrix used in (2).

Step 2. Form the two vectors in (2) to match the specific type of local geometry.

Step 3. Evaluate the local geometry by performing the vector and matrix multiplications specified in (2).

When n=1 or n≥3, the local geometry $P_i(z)$ is the vertex $\mathbf{v}_i$ itself. Since $\mathbf{v}_i$ is stored in the top row of the texture image, the local geometry $\mathbf{v}_i$ $(=\mathbf{c}^i_{1,1})$ can be computed using (2) by setting the evaluation parameter z_i to 0 + i0.

When n=0, we form the two vectors of the monomial basis functions in (2) from the evaluation parameter z_i=u_i+iv_i in step 2 and then compute a point on the surface patch $P_i(z_i)$ in step 3. Two matrix-vector multiplications are required for each coordinate, making a total of six operations.

When n=2, the curve parameters $\sigma_i(z)$ in (3) have to be determined to evaluate a point on the Bézier curve $P_i(z)$, which is the local geometry of the chart U_i. The curve parameter $\sigma_i(z)$ on each chart U_i can be computed directly using sampling parameters. If $\hat{z}$ is a sampling parameter of a face f, then the curve parameter $\sigma_i(z)(=\sigma_i(\hat{z}))$ is determined as

$$\sigma_i(\hat{z})=\begin{cases}0.5-|\hat{z}|\cos\theta, & \text{if} \quad e_1=(\mathbf{q}^i_0,\mathbf{q}^i_1)\in f,\\ 0.5+|\hat{z}|\cos\theta, & \text{if} \quad e_2=(\mathbf{q}^i_1,\mathbf{q}^i_2)\in f,\\ 0.5, & \text{if} \quad e_1\notin f \text{ and } e_2\notin f,\end{cases} \tag{4}$$

where e_1 and e_2 are the edges connected to the vertex $\mathbf{v}_i(=\mathbf{q}^i_1)$ which are labeled as sharp, and θ is the angle between $\hat{z}$ and the sharp edge on the face f. Figure 10 shows how to determine the curve parameters $\sigma_i(\hat{z})$ of $\hat{z}\in f_0$, which can be computed from the expression $0.5-|\hat{z}|\cos\theta$ since $e_1 \in f_0$. Similarly, the curve parameters of $\hat{z}\in f_1$ can be computed from $0.5+|\hat{z}|\cos\theta$ since $e_2 \in f_1$. However, the curve parameters of $\hat{z}\in f_2$ are 0.5 since $e_1\notin f_2$ and $e_2\notin f_2$.

The control points for a Bézier curve $P_i(z)$ are stored in the first three rows of the texture image. Therefore, a point on the curve can be evaluated by replacing the two vectors of the monomial basis functions with $[1 \ \ 0 \ \ 0]$ and $\left[B^2_0(\sigma_i) \ \ B^2_1(\sigma_i) \ \ B^2_2(\sigma_i)\right]^T$, respectively, in step 2. Table 2 lists the different pairs of vectors that must be formed in step 2 to evaluate different types of local geometry.

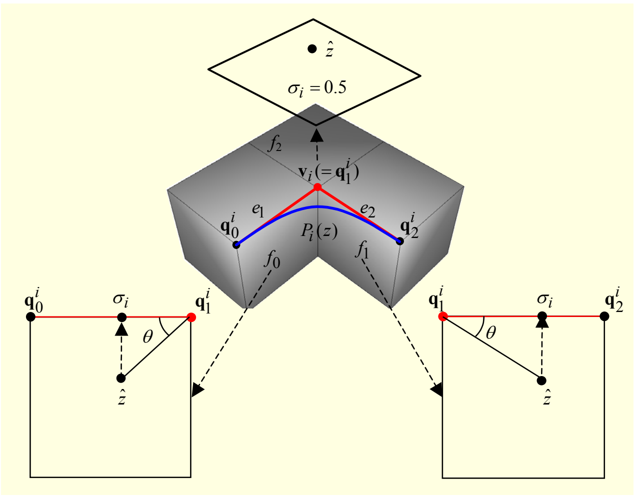

Fig. 10. Evaluation parameter $\sigma_i(z)$ computed from the sampling parameters $\hat{z}=\hat{u}+i\hat{v}$ of a face when a quadratic Bézier curve $P_i(z)$ is the local geometry of a vertex v_i.

Table 2. Pairs of vectors used in step 2 for different configuration of edges.

Number of sharp edges	1st vector	2nd vector
0	$[1 \ \ u_i \ \ u_i^2]$	$[1 \ \ v_i \ \ v_i^2]^T$
1 or ≥ 3	$[1 \ \ 0 \ \ 0]$	$[1 \ \ 0 \ \ 0]^T$
2	$[1 \ \ 0 \ \ 0]$	$[(1-\sigma_i^2) \ \ 2(1-\sigma_i)\sigma_i \ \ \sigma_i^2]^T$

4. Blending Local Geometries

The final blending surface is then constructed by blending four points from the local geometries. This requires computing the blending functions $w_i(z_i)$ for these points. Since we use a Hermite interpolant, we can use the Hermite interpolation function provided by the shading language of the GPU. First we form a 3×4 matrix P from the points on the local geometries and a 4D vector $\mathbf{w}$ from the blending functions w_i, which we have already computed. The final blended point $\mathbf{p}$ can then be computed in a single matrix and vector multiplication as follows:

$$\mathbf{p}=P\mathbf{w}=\begin{bmatrix}p_{1,x} & p_{2,x} & p_{3,x} & p_{4,x}\\ p_{1,y} & p_{2,y} & p_{3,y} & p_{4,y}\\ p_{1,z} & p_{2,z} & p_{3,z} & p_{4,z}\end{bmatrix}\begin{bmatrix}w_1\\ w_2\\ w_3\\ w_4\end{bmatrix}. \tag{5}$$

5. Trimming

Since the sampling parameters are sampled in a unit square

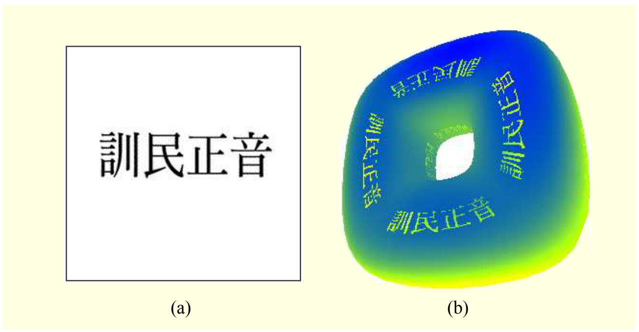

Fig. 11. (a) Trimming texture and (b) trimmed blending surface.

[0, 1]×[0, 1] for each face, we can easily trim the surface using a trimming texture in a fragment program. As shown in Fig. 11(a), a trimming texture is a binary image in which the region to be trimmed is colored black. The vertex program returns the sampling parameters $\hat{z} = \hat{u} + i\hat{v}$ to the graphics pipeline, and these are interpreted as texture coordinates $(\hat{u}, \hat{v})$ of the blending surface. The linearly interpolated texture coordinates are then used in a fragment program to create fragments that will not be rendered. A trimming texture and a trimmed blending surface are shown in Figs. 11(a) and (b), respectively.

VI. Experimental Results

We implemented the part of our rendering algorithm that runs on the CPU in C++, and the part that runs on the GPU was implemented in the Cg shading language [21]. The CPU that we used was a Pentium-IV running at 3.2 GHz with a 1 Gb main memory, and the GPU was an NVIDAI GeForce 6800 Ultra.

Figures 12(a), (c), and (e) show some example control meshes, and Figs. 12(b), (d), and (f) show the resulting blending surface generated by our GPU-based implementation. The performance of our rendering technique is not influenced by the types of local geometry to be blended because the computations are the same for all the geometries, which is because we use the unified evaluation procedure described in section V.3. We measured the rendering speed for blending surfaces without sharp features.

Table 3 gives the geometric characteristics of each control mesh and the rendering performance of the CPU and GPU implementations. There are 30×30 sampling parameters on each face in every case.

The performance of our rendering technique is linearly dependent on the number of faces in the control mesh. It is also affected by the number of sampling parameters on each face, and this relationship is shown in Fig. 13. Our experimental results suggest that our GPU-based implementation runs about

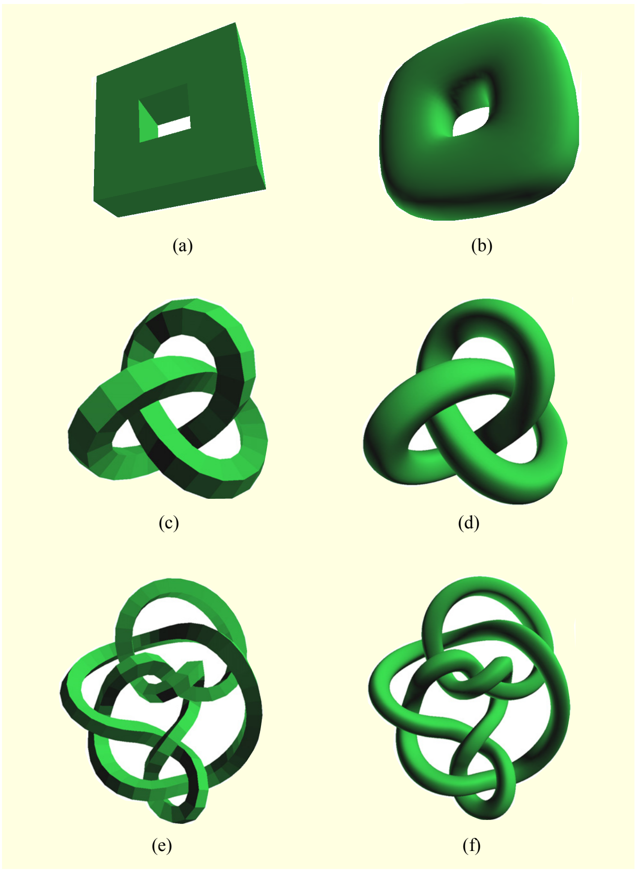

Fig. 12. Control meshes and corresponding blend surfaces.

Table 3. Geometric information and rendering performance.

Mesh	Number of charts	Number of faces	GPU fps	CPU fps	GPU/ CPU
(a)	16	32	438.2	25.40	17.25
(c)	176	179	73.0	3.84	19.01
(e)	442	442	30.1	1.48	20.33

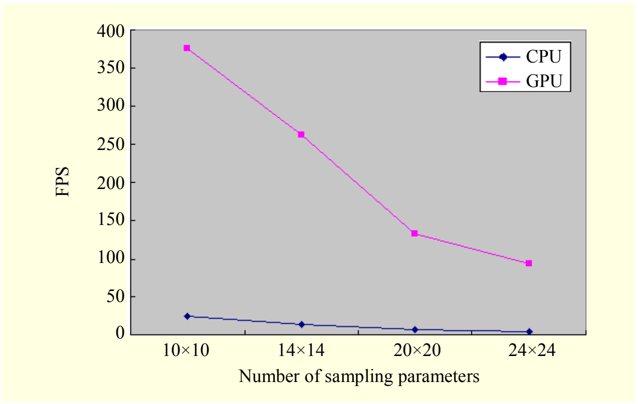

Fig. 13. Comparative performance of CPU and GPU implementations using different numbers of sampling parameters on each face of example in Fig. 12(c).

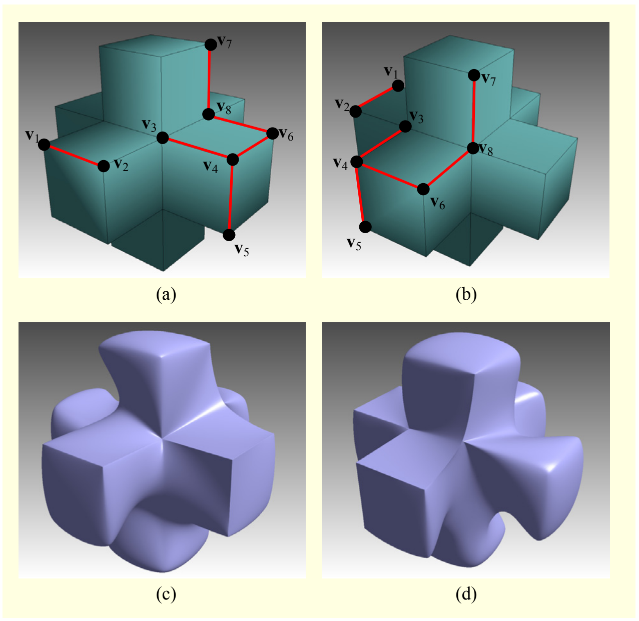

Fig. 14. Blending surface with various sharp features. Marked edges are shown in red.

Table 4. Local geometries for vertices in example of Fig. 14(a).

Local geometry	Vertices
Surface patch	All vertices not connected by sharp edges
Vertex	$\mathbf{v}_1$, $\mathbf{v}_2$, $\mathbf{v}_3$, $\mathbf{v}_4$, $\mathbf{v}_5$, $\mathbf{v}_6$, $\mathbf{v}_7$,
Bézier curve	$\mathbf{v}_8$

18 to 20 times faster than its CPU-based counterpart.

Figure 14 shows an example of a complicated blending surface which exhibits all the types of sharp features that we allow: the dart, crease, and corner. Figures 14(a) and (b) show the sharp edges (in red) selected by the user in two different views. Table 4 gives the local geometries of the connected vertices. Although the edge joining $\mathbf{v}_2$ and $\mathbf{v}_3$ is not shown to be selected in Fig. 14(a), a dart is generated in the blending surface as shown in Fig. 14(c). This is because the local geometries of $\mathbf{v}_2$ and $\mathbf{v}_3$ are defined to be vertices. Figure 14(d) shows a crease across the surface which is generated by simultaneously blending a Bézier curve, vertex, and local patch.

VII. Conclusion

We have extended the blending scheme of Ying and Zorin [5] to include sharp features. Depending on the number of sharp edges that meet at a vertex, sharp features in the form of a dart, crease, or corner are generated by blending the local geometry appropriately.

We have also presented an efficient technique for rendering a blended surface using a vertex program on a GPU, yielding an eighteen-fold to twenty-fold increase in rendering speed compared to a CPU-based implementation. Since each face of the control mesh is rendered as a separate entity, the surface can easily be trimmed in a fragment program using a texture.

Our current CPU implementation does not use acceleration techniques, such as multiple cores and multi-threading, which would considerably increase the rendering speed. We plan to apply these techniques to the CPU implementation. We also intend to utilize the geometric programming facilities recently introduced in Shader 4.0 to improve the performance of our algorithm on the latest GPUs.

References

[1] M. Guthe, A. Balas, and R. Klein, "GPU-Based Trimming and Tessellation of NURBS and T-Spline Surface," *ACM Trans. Graphics*, vol. 24, no. 3, 2005, pp. 1016-1023.

[2] A. Krishnamurthy, R. Khardekar, and S. McMains. "Direct Evaluation of NURBS Curves and Surfaces on the GPU," *Proc. ACM Symp. Solid Physical Modeling*, 2007, pp. 329-334.

[3] C. Sigg et al., "GPU-Based Ray-Casting of Quadratic Surfaces," *Proc. Eurographics Symp. Point-Based Graphics*, 2006, pp. 59-65.

[4] Y. Yasui and T. Kania, "Surface Quality Assessment of Subdivision Surfaces on Programmable Graphics Hardware," *Proc. Int. Conf. Shape Modeling Appl.*, 2004, pp. 129-136.

[5] L. Ying and D. Zorin, "A Simple Manifold-Based Construction of Surfaces of Arbitrary Smoothness," *ACM Trans. Graphics*, vol. 23, no. 3, 2004, pp. 271-275.

[6] H. Biermann, A. Levin, and D. Zorin. "Piecewise Smooth Subdivision Surface with Normal Control," *Proc. ACM Siggraph*, 2000, pp. 113-120.

[7] T. DeRose, M. Kass, and T. Truong, "Subdivision Surfaces in Character Animation," *Proc. ACM Siggraph*, 1998, pp. 85-94.

[8] H. Hoppe et al., "Piecewise Smooth Surface Reconstruction," *Proc. ACM Siggraph*, 1994, pp. 295-302.

[9] G-D. Vecchia and B. Jüttler, "Piecewise Rational Manifold Surfaces with Sharp Features," *Proc. 13th IMA Int. Conf. Math. Surfaces XIII*, 2009, pp. 90-105.

[10] C. Grim and J. Hughes, "Modeling Surfaces of Arbitrary Topology Using Manifolds," *Proc. ACM Siggraph*, 1995, pp. 359-368.

[11] C. Grim, "Simple Manifolds for Surface Modeling and Parameterization," *Proc. Shape Modeling Int.*, 2002, p. 237.

[12] C. Grim, J. Crisco, and D. Laidlaw, "Fitting Manifold Surfaces to 3D Point Clouds." *J. Biomech. Eng.*, vol. 124, no. 1, 2002, pp. 136-140.

[13] J. Cotrina and N. Pla, "Modeling Surfaces from Meshes of

Arbitrary Topology," *Computer Aided Geometric Design*, vol. 17, no. 7, 2000, pp. 643-671.

[14] J. Cotrina, N. Pla, and M. Vingo, "A Generic Approach to Free Form Surface Generation," *Proc. ACM Symp. Solid Modeling Appl.*, 2002, pp. 35-44.

[15] X. Gu, Y. He, and H. Qin, "Manifold Spline," *Proc. ACM Symp. Solid Physical Modeling*, 2005, pp. 27-38.

[16] G-D. Vecchia, B. Jüttler, and M.-S. Kim, "A Construction of Rational Manifold Surfaces of Arbitrary Topology and Smoothness from Triangular Meshes," *Computer Aided Geometric Design*, vol. 25, no. 9, 2008, pp. 801-815.

[17] H.-F. Pabst et al., "Ray Casting of Trimmed NURBS Surfaces on the GPU," *Proc. IEEE Symp. Interactive Ray-Tracing*, 2006, pp. 151-160.

[18] T. Kanai et al., "GPU-Based Rendering of Sparse Low-Degree Implicit Surfaces," *Proc. 4th Int. Conf. Computer Graphics Interactive Techniques in Australasia and Southeast Asia*, 2006, pp. 165-171.

[19] F. Reck et al., "Real-Time Isosurface Extraction with Graphics Hardware," *Proc. Eurographics*, 2004, pp. 33-36.

[20] J. Bolz and P. Schröder, "Evaluation of Subdivision Surfaces on Programmable Graphics Hardware," 2003, submitted for publication.

[21] NVIDIA, Inc. NVIDIA, Cg Toolkit User's Manual, 2006.

Dae-Hyun Ko received his BS in computer engineering from Seoul National University in 1999 and the MS and PhD in computer science and engineering from Seoul National University in 2001 and 2007, respectively. He is currently a senior engineer at Digital Media and Communications R&D Center, Samsung Electronics. His research interests are in computer graphics and geometric modeling.

Jieun Lee received the BS in computer science and engineering from Ewha Womans University in 1997, the MS from POSTECH in 1999, and the PhD from Seoul National University in 2007. From 1999 to 2002, she worked at LG Electronics Institute of Technology as a research engineer and mainly participated in the MPEG-7 standardization activity. She is currently an assistant professor at the School of Computer Engineering, Chosun University, Korea. Her research interests are in geometric modeling, computer graphics, and multimedia information processing.

Seong-Jae Lim received the BS in computer engineering from Chonnam National University, Korea, in 1999, and the PhD in information and communication engineering from Gwangju Institute of Science and Technology, Korea, in 2006. From 2004 to 2005, he was a visiting scholar at the Medical Image Processing Laboratory, University of Pennsylvania. He is currently a senior member of the engineering staff of the Computer Graphics Research Team at ETRI. His research interests are in computer vision, computer graphics, and medical imaging.

Seung-Hyun Yoon received the BS in mathematics from Hanyang University in 2001 and the PhD in computer science and engineering from Seoul National University in 2007. He is currently an assistant professor of the Department of Multimedia Engineering, Dongguk University. His research interests are in computer graphics and geometric modeling.