Fragment-based Evaluation of Non-Uniform B-spline Surfaces on GPUs

Takashi Kanai

The University of Tokyo

ABSTRACT

In this paper, we propose a fragment-based evaluation method for non-uniform B-spline surfaces using recent programmable graphics hardware (GPU). A position on a non-uniform B-spline surface is evaluated by the linear combination of both control points and B-spline basis functions. Hence the computational costs can be reduced by pre-computing a knot interval of a parameter from a knot vector. We show that efficient computation of positions and normal vectors for B-spline surfaces can be done on GPUs by applying these ideas. Our algorithm computes an exact position, a derivative and a curvature per fragment. We demonstrate that this achieves high-quality surface rendering. We also discuss about the extension to NURBS and trimmed surfaces.

Keywords: Non-uniform B-spline surface, B-spline basis function, Knot interval, GPU, Fragment program, Floating point texture.

1. INTRODUCTION

Parametric free-form surface representations such as Bezier or B-spline surfaces are nowadays used in many CAD systems. Solid models in such CAD systems are often defined by the boundary representation (B-rep) whose faces consist of a set of parametric surfaces. In some cases trimmed surfaces are generated by geometric operations such as set operations. Among these parametric forms of surfaces, non-uniform B-spline surface has many well-known advantages such as local support of basis functions, arbitrary number of control points, freely-defined knots, etc.. NURBS is a rational version of such a non-uniform B-spline surface. Natural quadric surfaces can be defined by NURBS with appropriately set weights.

Especially in the area of digital engineering, visual evaluation tools [1],[2] are utilized to evaluate the geometrical properties (smoothness, variation of curvatures, roughness, etc.) of parametric surfaces. For these applications, precise evaluation of such properties is required since they are much sensitive to geometric quantities especially for 2nd order derivatives or normal vectors.

In most cases, tessellated polygons are used to render the surfaces. Lighting and shading are done by using a normal vector of each vertex and by a linear interpolation of vertex colors in each triangle. A color for a point on the surface is calculated without using a position or a normal vector on this point. However, the position and the normal vector in this case are not exact at all. To calculate the correct color for an arbitrary point on the surface, direct rendering approaches such as ray tracing or ray casting [3],[4] can be used. Unfortunately, these approaches take much time to compute.

In this paper, a method for the high-quality rendering of non-uniform B-spline surfaces using recent programmable graphics hardware (GPU) is presented. The characteristic of our approach is that we calculate an exact position and a normal vector at the pixel level in the fragment program of GPU. We can then establish a high-quality scale-independent rendering method for objects composed of B-spline surfaces. NURBS rendering can be also achieved by a simple extension of our approach.

Our contribution offers a naive implementation method to compute B-spine basis functions. By selecting an appropriate knot interval in a knot vector from an input parameter, the computational cost for basis functions can be reduced. This enables real-time rendering including the evaluation of non-uniform B-spline surfaces.

2. RELATED WORK

The most relevant approach to ours was published by Guthe et al. [5]. They propose a method for real-time rendering of trimmed NURBS on GPUs. Trimming curves on a surface are sampled in the first pass and are stored as a texture. In the second pass, a point on a surface is evaluated for each vertex of a uniformly sampled grid polygon on the vertex program, and trimming is done with a stored trim texture on the fragment program. One issue in this approach is that all of NURBS patches whose degrees are greater than three are approximated to a set

of bi-cubic rational Bezier surfaces. Geometric features may then be lost when approximating the surfaces. In [6] they later achieved an appearance-preserving extension by improving the error function for the approximation. However, the approximation itself has not been yet resolved. Another issue is that the rendering process is performed for each patch. In this case, the computational cost is higher as the number of patches is larger. In contrast, our approach can evaluate exact surfaces of arbitrary degrees. Moreover, surfaces are evaluated for each fragment rasterized from polygons; their computational costs do not depend largely on the number of patches.

There are some approaches for the rendering of free-form surfaces. Bischoff et al. [7] proposed a hardware implementation for the rendering of Loop subdivision surfaces by using the forward difference technique. Bolz and Schröder [8] pointed out that position computation of each subdivided point on a Cutmull-Clark surface can be done by the linear combination of basis functions and control points and indicated an approach to assist these computations by using GPUs. Shiue et al. [9] performed the subdivision operations on GPU by a special data structure called fragment mesh to effectively access a mesh. Boubekeur and Schlick [10] proposed a fast rendering method for the parametric form of surfaces such as N-patches by using VBOs (*Vertex Buffer Objects*) and programmable shaders. These approaches, however, render (subdivided) polygons after all. We can say that this does not address the strictness issue described in Section 1.

On the other hand, fragment-based evaluation methods for subdivision surfaces were proposed [11], [12]. Our scheme in this paper is based on their approaches and extends to non-uniform B-spline surfaces. In [11], they utilize the fact that Catmull-Clark surface is represented as an extension to bi-cubic uniform B-spline surface. Coefficients of bi-cubic uniform B-spline basis functions are uniquely defined and are then easily computed on programmable shaders. In our case, the problem is rather complicated since we need non-uniform basis functions for arbitrary degree (See Section 3.1).

For the rendering of other types of geometric representations, Loop and Blinn [13] presented a resolution-independent approach to render parametric curves on GPUs. Hable et al. [14] proposed to display triangular meshes suffered in Boolean operations by using GPUs.

3. EVALUATING NON-UNIFORM B-SPLINE SURFACES ON GPUS

In this section, we explain how to implement our fragment-based evaluation method on GPUs for non-uniform B-spline surfaces of arbitrary degree. We first describe the implementation of B-spline basis functions as well as B-spline curves. The basic idea for curves is the same as in the case for surfaces. A curve only requires a set of B-spline basis functions for a direction, whereas for surfaces an individual set of basis functions for each of two directions is required.

A non-uniform B-spline curve of degree p is represented as follows (notation is followed by [15]):

$$\mathbf{C}_p(u) = \sum_{i=0}^{n} N_{i,p}(u)\mathbf{P}_i, \quad a \le u \le b, \tag{1}$$

where $\mathbf{P}_{i,p}(u)$ denotes a control point and $N_{i,p}(u)$ denotes an i th element of B-spline basis functions of degree p defined by a non-periodic and a non-uniform knot vector U composed of m+1 knots (a, b: p+1 knots):

$$U = \{u_0, \ldots, u_m\} = \{a, \ldots, a, u_{p+1}, \ldots, u_{m-p-1}, b, \ldots, b\}.$$

One issue in implementing such basis functions on the current GPUs is that loop operations for *arbitrary* times cannot be executed: That is, let the number of repetitions in the function “for” on the shader program be m. m must be defined as a *fixed* number throughout the program. Acquiring m as external inputs or arguments of the function is not permitted. In the case of calculating basis functions, the number of control points n+1 in Eqn. (1) must be fixed during the computation on GPU. Eqn. (1) cannot be then computed as it is.

Instead, we use an implementation for B-spline basis functions in [15] based on finding knot intervals. This utilizes the fact that the number of non-zero basis function values of degree p is always p+1.

3.1 Computing B-spline Basis Functions, Curves and Surfaces

A B-spline basis function $N_{i,p}(u)$ is defined as follows:

$$N_{i,0}(u) = \begin{cases} 1 & \text{if } u_i \le u < u_{i+1}, \\ 0 & \text{otherwise}, \end{cases}$$
$$N_{i,p}(u) = \frac{u - u_i}{u_{i+p} - u_i} N_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(u). \tag{2}$$

One of the properties that should be noticed for implementing Eqn. (2) is that only p+1 function values $N_{i-p,p}, \ldots, N_{i,p}$ need to be computed when a knot interval i satisfying $u_i \le u < u_{i+1}$ is found. An algorithm for computing Eqn. (2) is as follows [15]:

1. Find a knot interval i which satisfies $u_i \le u < u_{i+1}$ in a knot vector U.
2. Compute p+1 B-spline basis function values $N_{i-p,p}, \ldots, N_{i,p}$.

We now recall the issue described above that loop operations with a variable number of repetitions cannot be handled on GPUs. This implies that the degree p is fixed in the shader program. Our solution here is to prepare a function to compute basis function values for each degree. According to the degree as an input, we calculate such function values by separate functions.

Determining a knot interval. Determining a knot interval i corresponds to the operation which extracts an element from a sorted one-dimensional array of a knot vector. This can be computed by a linear or binary search. We use here a simplified version of a binary search algorithm proposed in Purcell et al. [16] for this computation. For an input parameter u, we traverse a one-dimensional array of a knot vector U by using a binary search, and narrow the range including a solution. The number of repetition depends on the number of knot vectors; for example, we can handle 2^5=32 knots for five times repetitions and 2^6=64 knots for six times repetitions. The computational cost of this algorithm is $O(\log n)$. $u = u_m$ is a special case which cannot be calculated in the algorithm. A knot interval for this case is uniquely determined as m-p-1.

A knot vector U is stored as a one-dimensional texture and appropriate knots are acquired according to an input parameter in the shader program. We later describe how to store a knot vector to a texture in Section 3.2.

Computing B-spline basis functions. After determining a knot interval i for a parameter u, B-spline basis function values for u can be computed by the *inverse triangle method* described in [15]:

$$\begin{array}{cccc} & & & N_{i-3,3} \\ & N_{i-1,1} & N_{i-2,2} & N_{i-2,3} \\ N_{i,0} & N_{i,1} & N_{i-1,2} & N_{i-1,3} \\ & & N_{i,2} & N_{i,3} \end{array} \tag{3}$$

We start to calculate a basis function at degree 0, and compute consequent basis function values with raising degrees by using Eqn. (2). For degree p, the number of basis functions to be computed is $\sum_{i=0}^{p} i$ and the number of knots is $2p$ $(u_{i-p+1}, u_{i-p+2}, \ldots u_{i+p})$.

Evaluating B-spline curves. As in the case of basis functions, the number control points to be acquired is p+1 $(\mathbf{P}_{i-p}, \mathbf{P}_{i-p+1}, \ldots, \mathbf{P}_i)$. All control points are also stored in a texture and some of these are acquired in the fragment program according to a knot interval i. In this case, a formula to compute a position on a B-spline curve can be reduced to the following linear combination between basis function N and control point $\mathbf{P}$:

$$\mathbf{C}_p(u) = N_{i-p,p}(u)\mathbf{P}_{i-p} + N_{i-p+1,p}(u)\mathbf{P}_{i-p+1} + \ldots + N_{i,p}(u)\mathbf{P}_i. \tag{4}$$

A function to compute such positions for each degree is also separately prepared.

Computing derivatives. The derivative of a B-spline curve $\mathbf{C}'_p(u)$ is as follows:

$$\mathbf{C}'_p(u) = N'_{i-p,p}(u)\mathbf{P}_{i-p} + N'_{i-p+1,p}(u)\mathbf{P}_{i-p+1} + \ldots + N'_{i,p}(u)\mathbf{P}_i \,, \tag{5}$$

where $N'_{i,p}(u)$ denotes the first-order derivative of a B-spline basis function represented as follows (the derivation is followed by [15]):

$$N'_{i,p}(u) = \frac{p}{u_{i+p} - u_i} N_{i,p-1}(u) + \frac{p}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(u) \,. \tag{6}$$

As seen from Eqn. (6), most of the intermediate results needed to compute the above derivative can be shared with those of the position in Eqn. (2). That is, $N_{i,p-1}(u)$ and $N_{i+1,p-1}(u)$ in Eqn. (6) are the same as those Eqn. (2), and these two formulae use an identical knot vector and a set of control points. This dramatically reduces the number of instructions in the shader program.

Evaluating B-spline surfaces. The non-uniform B-spline surface of degree *p, q* with knot intervals *i, j* determined in both *u* and *v* directions is represented by:

$$\mathbf{S}_{p,q}(u,v) = \sum_{k=i-p}^{i} \sum_{l=j-q}^{j} N_{k,p}(u) N_{l,q}(v) \mathbf{P}_{k,l} \,. \tag{7}$$

The evaluation of B-spline surfaces is not so different with the case of the B-spline curve. First, we compute B-spline basis functions of both *u, v* directions, and we then apply a linear combination between those basis functions and $(p+1)\times(q+1)$ control points. For normal vectors, we compute a derivative of a B-spline surface in each of the two directions separately.

There are other efficient computational methods for B-spline surfaces such as *de Boor algorithm*. The de Boor algorithm provides a direct evaluation of a position on the surface. If we only evaluate a position, using the de Boor algorithm also achieves low computational costs. However, the advantage of our approach described above is to reuse most of the B-spline basis functions for the computation of both a position and a normal vector.

We later describe how to store control points to a texture in Section 3.2.

3.2 Texture Preparation

In this section, we describe a method to create floating point textures of knot vectors and control points needed for the computation of non-uniform B-spline surfaces.

Fig. 1 left shows the storage of knot vectors to a texture. This texture is used for most of the functions to compute B-spline surfaces described in the previous section. A floating point texture is created in each direction, then two textures are required in the *u,v* directions. We store an array of knot vectors of a patch in each row. In the column we store different knot vector arrays of patches. A column id corresponds to a patch id. In a knot vector texture, only a floating point is stored for each pixel. The number of knot vectors is stored at the top pixel in each row. This number is used for the binary search algorithm to determine the knot interval described in the previous section. The size of a knot vector texture in each direction is then (the maximum number of knot vectors + 1) × (the number of patches).

Fig. 1 right illustrates the storage of control points to a texture. This texture is used only for computing positions and derivatives of B-spline surfaces, and only a texture is prepared for all patches. We store an array of $m \times n$ control points of a patch in each row. In the column we store different control point arrays of patches. A column id corresponds to a patch id as a knot vector texture does. In each pixel, four floating points can be stored. Three of four are *x, y, z* coordinates of a control point, and the remaining element is a weight *w* for NURBS. The number of control points and degrees in both the *u, v* directions of a patch are stored at the top pixel in each row. A degree is used for various functions including the main function to determine the functions to be used. The number of control points is used only for the computation of positions and derivatives. The size of a control point texture is then (the maximum number of control points + 1) × (the number of patches).

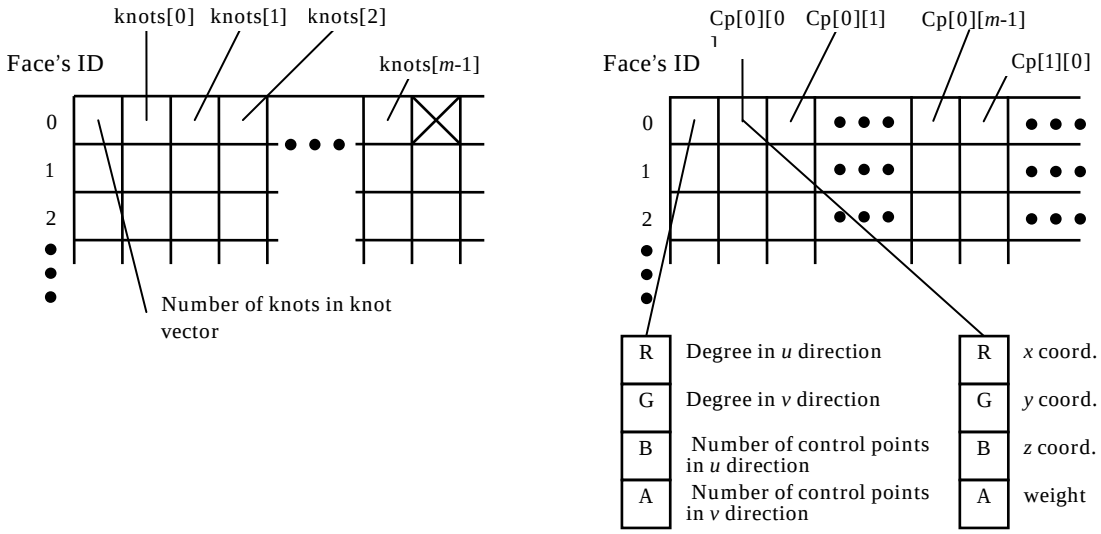

Fig. 1: 2D textures for computing non-uniform B-spline surfaces. Left: 2D texture of knot vectors in each direction. Right: 2D texture of control points.

3.3 Rendering Algorithm

In this section, we describe a rendering algorithm of non-uniform B-spline surfaces.

We prepare a tessellated polygon of surfaces in addition to textures of surface elements described in the previous section. Each face of a polygon is decomposed into a set of fragments to compute positions and normal vectors of surfaces in the rendering process. In the case of trimmed surfaces, we can perform the same process as untrimmed surfaces if we prepare a polygon of such trimmed surfaces.

Fig. 2 shows an overview of our rendering algorithm. A prepared polygon is drawn with user-specified viewing parameters in the usual rendering process. It is effective to input a two-dimensional parameter *u*,*v* and a face id of a corresponding surface as a three-dimensional texture coordinate for each vertex of a triangle. In this case, however, the maximum number of a texture parameter is limited to 3.0 due to the specification of graphics API (e.g. OpenGL). We first divide these parameters by a large number (e.g. 10,000) before putting them into a texture coordinate and multiply the same number in the shader program. Each triangle of a polygon is rasterized into a set fragments, and a linearly interpolated parameter *u*,*v* and a face id are set to a fragment program.

In the fragment program, we compute a position and a normal vector by using a two-dimensional surface parameter, a face id, and three textures (two knot vector textures and a control point texture). We first determine a knot interval from one of the surface parameters and a knot vector texture. The number of degrees is next acquired from a face id and a control point texture. According to the degree, we compute a position and a normal vector using separate functions.

	#patches	#control points	#knot vectors	#polygons	fps
surface	1	6 × 14	8 × 16	5,000	212
teapot	32	4 × 4	6 × 6	6,400	201
spray	266	18 × 18	20 × 20	53,200	126

Tab. 1: The number of patches, the maximum number of control points for a patch, the maximum number of knot vectors for a patch, the number of polygons used for rendering, and the computation time of all examples shown in this paper. The times are measured by rendering models with 512 × 512 screen size.

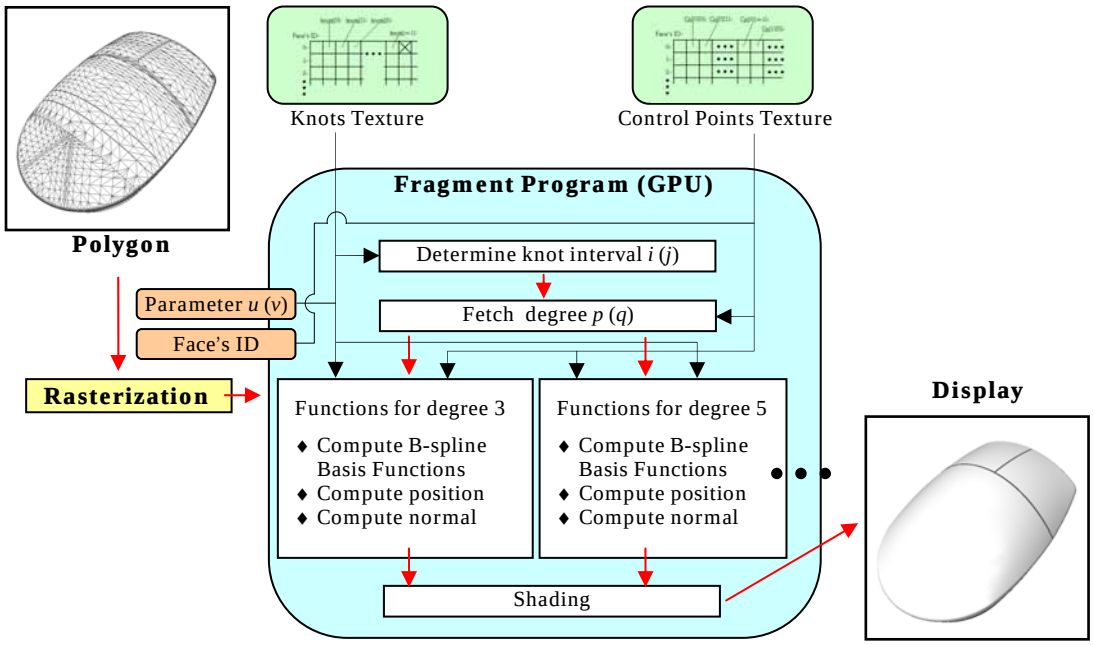

Fig. 2: Overview of our fragment-based rendering algorithm on GPUs.

In our algorithm, we have to prepare functions for each degree to compute B-spline basis functions, points and normal vectors. To reduce the functions to be prepared, we implement the algorithm as follows: We prepare functions for several degrees (e.g. 3, 5, 7 ...), not for all degrees. We apply degree-raising to surfaces of other degrees as a pre-process. Moreover, we raise one of two degrees in the *u*,*v* directions to the other (higher) if two degrees are different. The computational cost of computing such functions for larger degree is much higher. The above strategy satisfies both the reduction of functions and the suppression of computational costs.

After computing a position and a normal vector for each degree, we apply shading operations to them and finally compute a color for a fragment.

4. RESULTS AND DISCUSSION

In this section, we discuss the results of our fragment-based evaluation method for only bi-cubic non-uniform B-spline surfaces. Fig. 3 shows the rendering results of isophotes [2] to verify the effectiveness of our approach. In the left, a tessellated polygon of the B-spline surfaces is applied. Each vertex has a normal vector as well as a vertex coordinate. For each fragment, a linearly interpolated normal vector of three vertices of a triangle is used to compute isophotes. In the right, the rendering result of isophotes by our approach is shown. Artifacts appear due to linearly interpolated normal vectors in the left figure. In contrast, our approach in the right figure achieves visually smooth isophotes, because accurate normal vectors calculated at the pixel level are used.

Fig. 4 shows the rendering results for several example objects. Tab. 1 demonstrates the statistical summary of all examples. In our experiments we used an Athlon 64 X2 4800+ CPU and GeForce 7900 GTX 512MB GPU. We measure fps by the average of rotating an object.

For these surface models, we achieve fast frame rates for practical use. It can be seen from Tab. 1 that frame rates do not depend on the number of faces. Since the most time-consuming part of the rendering time is the computation of positions and normal vectors in the fragment program, the number of filled pixels provides considerably large effects.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have proposed a fragment-based evaluation method for non-uniform B-spline surfaces and the method to render such surfaces on GPUs. We have shown that we achieve a high-quality rendering even for rough tessellated polygons which has an advantage for the application of visual evaluation tool for such surfaces.

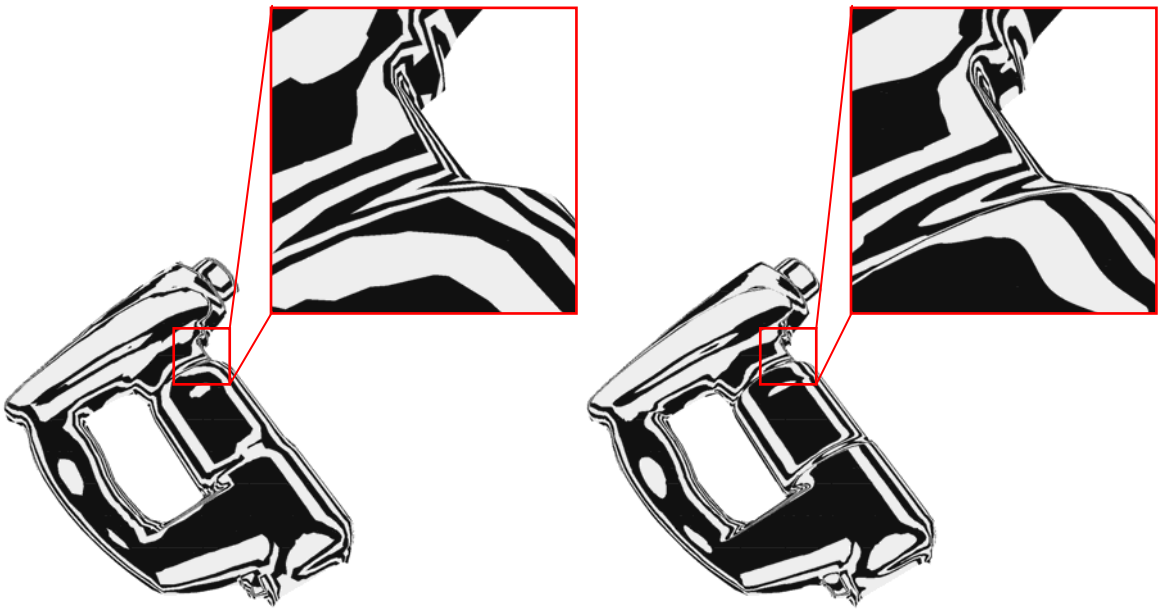

Fig. 3: Left: Display result of isophotes by using polygons. Right: Display result of isophotes by using our approach.

REFERENCES

[1] Hagen, H.; Hahmann, S.; Schreiber, T.; Nakajima, Y.; Wordenweber, B.; Hollemann-Grundstedt, P.: Surface interrogation algorithms, IEEE Computer Graphics and Applications, 12 (5), 1992, 53–60.

[2] Hahmann, S.: Visualization techniques for surface analysis, C. Bajaj (Ed.): Advanced Visualization Techniques, John Wiley, 1999.

[3] Nishita, T.; Sederberg, T. W.; Kakimoto, M.: Ray tracing trimmed rational surface patches, Computer Graphics (Proc. ACM SIGGRAPH 90), ACM Press, New York, 1990, 337–345.

[4] Martin, W.; Cohen, E.; Fish, R.; Shirley, P.: Practical ray tracing of trimmed NURBS surfaces, ACM Journal of Graphics Tools, 5 (1), 2000, 27–52.

[5] Guthe, M.; Balázs, A.; Klein, R.: GPU-based trimming and tessellation of NURBS and T-spline surfaces, ACM Transactions on Graphics (Proc. SIGGRAPH 2005), 24 (3), 2005, 1016–1023.

[6] Guthe, M.; Balázs, A.; Klein, R.: GPU-based appearance preserving trimmed NURBS rendering, Journal of WSCG, 14 (1), 2006.

[7] Bischoff, S.; Kobbelt, L. P.; Seidel, H.-P.: Towards hardware implementation of loop subdivision, Proc. ACM SIGGRAPH / EUROGRAPHICS Workshop on Graphics Hardware, ACM Press, New York, 2000, 41–50.

[8] Bolz, J.; Schröder, P.: Evaluation of subdivision surfaces on programmable graphics hardware, Tech. rep., California Institute of Technology, Computer Science Department, 2003.

[9] Shiue, L.-J.; Jones, I.; Peters, J.: A real-time GPU subdivision kernel, ACM Transactions on Graphics (Proc. SIGGRAPH 2005), 24 (3), 2005, 1010–1015.

[10] Boubekeur, T.; Schlick, C.: Generic mesh refinement on GPU, Proc. ACM SIGGRAPH/Eurographics Conference on Graphics Hardware, Eurographics Association, Aire-la-Ville, Switzerland, 2005, 99–104.

[11] Yasui, Y.; Kanai, T.: Surface quality assessment of subdivision surfaces on programmable graphics hardware, Proc. International Conference on Shape Modeling and Applications, IEEE CS Press, Los Alamitos, CA, 2004, 129–136.

[12] Kanai, T.; Yasui, Y.: Per-pixel evaluation of parametric surfaces on GPU, ACM Workshop on General Purpose Computing Using Graphics Processors, 2004, C22.

[13] Loop, C.; Blinn, J.: Resolution independent curve rendering using programmable graphics hardware, ACM Transactions on Graphics (Proc. SIGGRAPH 2005), 24 (3), 2005, 1000–1009.

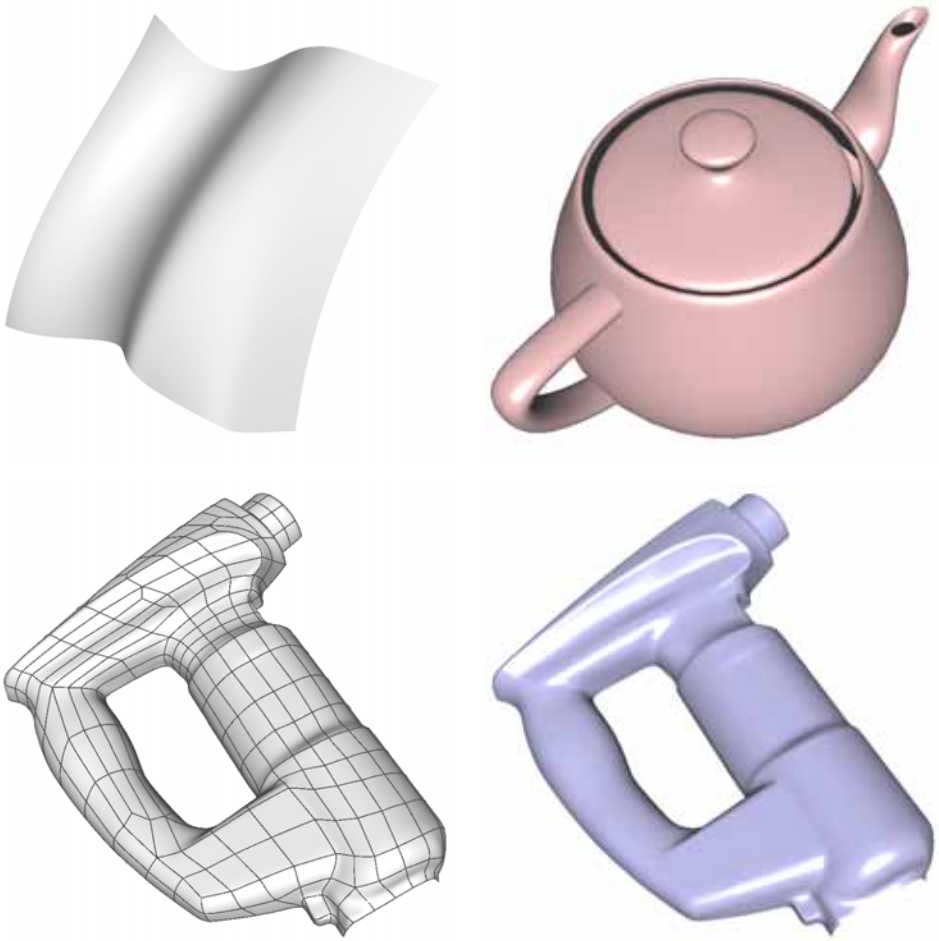

Fig. 4: Rendering results of models by our approach. Top Left: A B-spline surface patch. Top Right: “teapot” model represented by B-spline surfaces. Bottom: “spray” model. Surface boundaries are shown in the left figure.

[14] Hable, J.; Rossignac, J.: Blister: GPU-based rendering of Boolean combinations of freeform triangulated shapes, ACM Transactions on Graphics (Proc. SIGGRAPH 2005), 24 (3), 2005, 1024–1031.
[15] L. Piegl, W. Tiller, The NURBS Book, 2nd Edition, Springer-Verlag, Berlin, 1997.
[16] Purcell, T. J.; Donner, C.; Cammarano, M.; Jensen, H. W.; Hanrahan, P.: Photon mapping on programmable graphics hardware, Proc. ACM SIGGRAPH / EUROGRAPHICS Conference on Graphics Hardware, Eurographics Association, Aire-la- Ville, Switzerland, 2003, 41–50.