Surface Quality Assessment of Subdivision Surfaces on Programmable Graphics Hardware

Yusuke Yasui Takashi Kanai
Keio University SFC
Faculty of Environmental Information
5322 Endo, Fujisawa, Kanagawa, 252-8520, JAPAN.
{t00950yy/kanai}@sfc.keio.ac.jp

Abstract

In this paper, we propose a method of subdivision surface quality assessment by reflection lines on programmable graphics hardware (GPU). Using reflection lines is effective for surface quality assessment because the shapes of these lines are changed according to a slight variance of surface shapes. This fact also implies that reflection lines should be calculated precisely. We introduce an intuitive, fast and robust approach for calculating reflection lines by using a plane light source texture based on a fragment program of recently introduced GPU. In addition, we describe a method of calculating position and normal of subdivision surfaces for each fragment on GPU. As our framework for calculating reflection lines does not depend on the level of subdivision, a precise assessment can be established even for low levels of subdivision polygons.

Keywords: reflection line, programmable graphics hardware, fragment program, subdivision surface

1. Introduction

Surface quality assessment of aesthetic shape becomes evermore important in the applications of industrial design and engineering, due to rapid and widespread propagation of three-dimensional (3D) CAD/CAM systems. Especially in the design of car bodies, designers check the quality of surfaces by observing reflected images of parallel fluorescent lamps on the clay models. They check the appearance of the shape with reflection of incoming light, detect the distortion of these images, and investigate how to modify the corresponding regions.

To artificially simulate such optical phenomena on a computer, we calculate a curve of reflection image on a surface numerically from the surface geometry, the eye position and the position of a light line as inputs. Such a curve is called reflection line [11, 10](Figure 1). In earlier CAD systems, a pseudo-highlight line, a simpler simulation model, was used because of the limitation of computational resources. The recent computer hardware has enough power to calculate a set of reflection lines interactively.

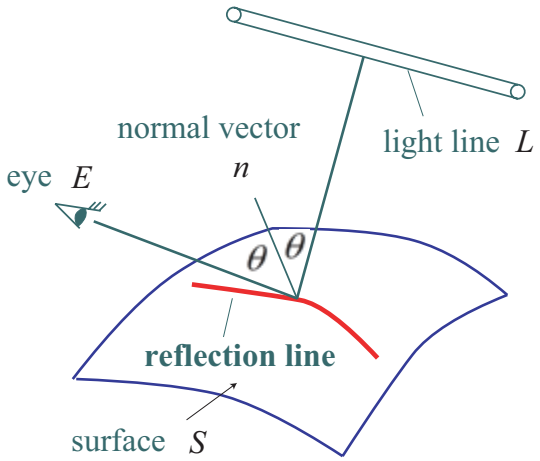


Figure 1. Reflection line.

In this paper, we propose a method of calculating and displaying reflection lines on subdivision surfaces by using programmable graphics hardware. The recent development of GPU is more rapid than that of CPU. Thanks to a programmable shader, not only the rendering speed of polygons is faster, but also a huge variety of other uses of GPU can be thought of.

The main point of this paper is that we use floating-point operations of pixels. This is one of the new functionalities of the so-called "second generation" programmable GPU such as nVIDIA GeForce FX and ATI RADEON 9700/9800. We propose an algorithm of calculating a set of reflection lines only in GPU. We also propose fragment-

based computation of positions and partial derivatives on a subdivision surface. This allows the computation of high quality reflection lines even for low levels of subdivision polygons.

2. Related Work

2.1. Reflection Lines

A Reflection line [11, 10] is a curve on a surface seen by observers, when a light line is reflected (Figure 1). The equation of a reflection line is as follows:

$$\frac{S-E}{|S-E|} - 2\left(\frac{S-E}{|S-E|}\cdot n\right)n = \frac{L-S}{|L-S|} \qquad (1)$$

where S, n, L, E denote a point on the surface, a normal vector of S, a point on a light line, and the eye position, respectively. The more simplified *highlight line* [1] is a set of points on the surface for which the distance between a straight line extending the surface normal and a light line is zero. Totally these lines are called *characteristic lines*[8]. The difference is that a reflection line is view-dependent; it depends on the surface geometry, the position of the light line and the eye position, while a highlight line depends only on the first two terms. It can be said that the former is a more physically correct simulation model than the latter. These lines are used not only for checking the quality of surfaces, but also as a guide for modifying surfaces [5, 16, 13].

It is in general impossible to analytically calculate a reflection line on surface, so the numerical calculation, for example, a method based on PDE (*partial differential equation*) [11, 10, 8] or a method based on the replacement by the intersection between a surface and a plane [1], is carried out. In the PDE-based approach, first one searches an initial point on a surface satisfying the Equation (1). It is used as a start point for finding a curve. In each step, the method finds a neighbor point on a curve by using a numerical integration method such as Runge-Kutta [14]. When a point reaches a surface boundary, the method restarts to find a point in the reverse direction. Finally a reflection line is calculated in the form of a poly-line. This is basically a one-by-one computation: if one calculates ten reflection lines for ten surfaces, a hundred computations for searching initial points and for finding reflection lines would be needed.

One problem of such numerical methods is that they cannot be calculated robustly. In case that a line passes through a region with higher curvature, the step size of numerical integration should be set to a smaller value. Especially on a concave region the image reflected by a light line is not always a line (a circle, a band, a knot, etc.). In such a case a control of the step size is needed which could make the computation unstable.

On the other hand, *zebra mapping*, a similar approach to our method, is used for easily evaluating the quality of surfaces. A texture with striped pattern is mapped on a surface, then distortions of the mapped patterns are detected. Such mapped stripe patterns, however, are not physically correct at all. Another approach called *cubic environment mapping* [7] is also a texture-based approach such as zebra mapping. There is little difference between cubic environment mapping and the basic idea of our approach. In cubic environment mapping, a reflection vector is calculated only at each vertex of a polygon and a vector for a point inside a polygon is calculated by linear interpolation. Thus the result for a point inside a polygon is not precise at all. On the other hand, our approach calculates a position and a normal vector of a subdivision surface for each fragment, establishing a more precise computation of reflected images.

2.2. Subdivision Surfaces

In this paper, we mainly focus on Catmull-Clark subdivision surfaces [4]. A Catmull-Clark surface is generated by recursively subdividing a quadrilateral control polygon. The limit surface is an uniform bi-cubic B-spline surface with C^2 continuity except at extraordinary points. Our algorithm, however, can also be applied for other types of subdivision surfaces such as Loop's subdivision surface [12].

Bolz and Schröder have proposed an algorithm for generating subdivision surfaces on the strength of hardware [2, 3]. They note that the computation of a position on a subdivision surface can be reduced to a linear combination of basis functions and control points, and then fast computation can be established by using GPU. The outputs of this approach, however, are only subdivided polygons, and are not what we are looking for. In the rendering process, a polygon is rasterized into several fragments. A position and a normal vector of each fragment is generated by a linear interpolation of vertices of a polygon. Strictly speaking, they are not exact ones of the limit surface at all. Our algorithm needs an exact position and a normal vector for each fragment.

On the other hand, Stam has shown that the exact evaluation of a subdivision surface for arbitrary parameter value is possible [15]. We implement this algorithm described in [15] on GPU. Zorin and Kristjansson have proposed an extended algorithm for piecewise smooth subdivision surfaces [17].

3. Reflection Lines Using GPU

In this section, we describe the basic idea and the algorithm of our novel approach for calculating reflection lines

on programmable graphics hardware.

3.1. Basic Idea

Reflection lines are calculated by using Equation (1). The meaning of Equation (1) can be described as follows:

> A reflection line is a set of points on a surface satisfying the condition that the angle between the vector from a point f on the surface to the eye position E and a surface normal n equals that between a vector from f to a point L on a light line and n.

The above definition can be rewritten as follows:

> For a vector from the eye position E to a point f on the surface, a reflection vector r is calculated by a surface normal n. If an intersection point x' between a light line and the line extending r is found, the set of points f on S, each of which is corresponded to x', is a reflection line.

Our idea is based on the latter definition. Figure 2 illustrates the idea. Instead of a set of light lines, an *area light source* is put directly above a surface (Figure 2(a)). It can be regarded as a set of light lines which are arranged without any spacing.

For a vector from eye position E to a point f on a surface S, a reflection vector r is calculated by a surface normal n. If a line which extends a vector r from f intersects the plane of an area light source, f should be a point of reflection. Let x be such an intersection point. The above calculation is applied for all points on a surface, and then a mapping $f \mapsto x$ from a point f on a surface to an intersection point x with an area light source can be constructed. Note that this mapping does not always have a one-to-one correspondence.

After constructing this mapping, we define a line on an area light source (Figure 2(b)). If x is on such a line, it can be defined as x' and a corresponding f is on a reflection line.

3.2. Algorithm

In this subsection we describe an algorithm for calculating a reflection line which realizes our basic idea. This algorithm is executed in the rendering process.

We prepare the eye position, the size and the height of a plane light source and a texture which is regarded as a set of light lines as inputs. Each fragment has its position and a normal vector. By using them and the eye position, we calculate the reflection vector on a fragment, and then calculate the intersection point between the reflection vector and the area light source. A texture corresponding to a set

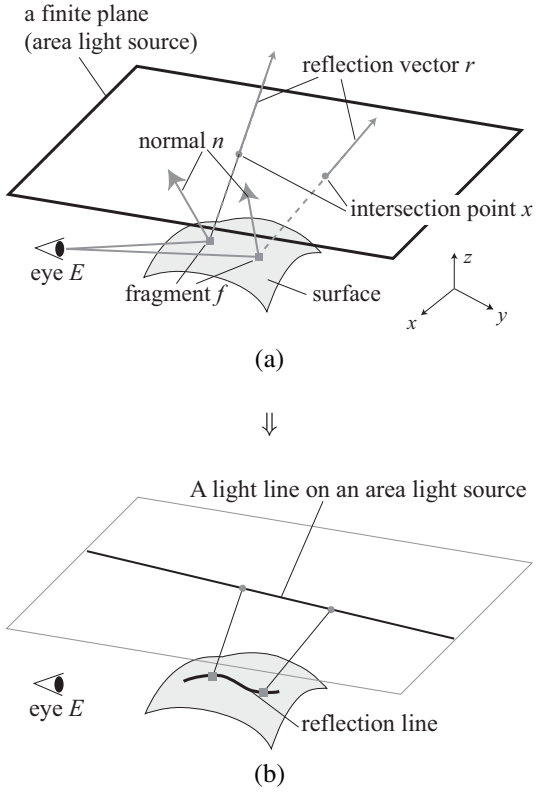


Figure 2. (a) The reflection vector $\vec{r}$ is calculated from the vector from eye position to surface ($\vec{f} - \vec{E}$) and the normal vector on the surface. An intersection point (x) is calculated from a reflection vector and an area light source. (b) The surface is colored by texture at a set of intersection points.

of light lines is mapped to such an area light source. So we fetch the color from the texture at the intersection point and assign it to a fragment. If the intersection point is on one of the light lines, a fragment is put on a reflection line.

Reflection lines can be constructed by applying this algorithm to all of the fragments on the surfaces.

4. Fragment-based Evaluation of Subdivision Surfaces Using GPU

In the rendering process, rendering primitives such as triangles are decomposed into a set of fragments. We apply the algorithm described above for each fragment.

Here we consider applying our algorithm to subdivision surfaces. In the rendering of subdivision surfaces, in general, polygons subdivided several times are used to display the surface, and we use a limit point for each vertex if needed. On the other hand, in the rasterization part dur-

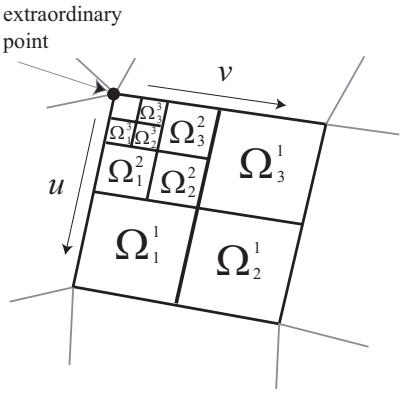


Figure 3. Positions and their partial derivatives are evaluated on each Ω_k^n plane.

ing the rendering process, a position of each fragment is calculated by a linear interpolation of vertex positions in a primitive. A calculated position does not have a totally exact value. This can be a problem in the case when polygons are coarse. Furthermore the shape of reflection lines is very sensitive to the change of surface geometry. Thus we need the exact position and normal vector of the limit surface for each fragment to calculate reflection lines precisely.

To solve this issue, we utilize an algorithm proposed by Stam [15]. Stam has shown that exact evaluation of subdivision surfaces at arbitrary parameter values is possible. In this section, we describe how to implement this algorithm on programmable graphics hardware.

4.1. Exact Evaluation of Subdivision Surfaces at Arbitrary Parameter Values

In this subsection, we review the algorithm of Stam. He has shown that the position of Catmull-Clark surface and its derivatives can be evaluated directly at a parameter value only by using its control polygon.

We assume that all the control polygons are quadrilateral. First, we subdivide an original control polygon once by using the recursive subdivision rule. By this operation, each subdivided polygon has at most one extraordinary point. The limit surface of a rectangular polygon which has no extraordinary point (called *regular* rectangle) equals uniform bi-cubic B-spline patch and it can be easily parameterized. As illustrated in Figure 3, a patch including an extraordinary point is recursively subdivided into four sub-patches. The three sub-patches which do not include the extraordinary point can also be represented by uniform bi-cubic B-spline patches.

The position on a subdivision surface for a point in two-dimensional parameter space (u, v) is represented as follows [15]:

$$s(u,v) = \hat{C}_0^T \Lambda^{n-1} X_k b(t_{k,n}(u,v)). \tag{2}$$

Each position is evaluated on Ω_k^n, where n and k are determined by the two parameters u and v. $\hat{C}_0$ in Equation (2) is represented as follows:

$$\hat{C}_0 = V^{-1} C_0, \tag{3}$$

where C_0 is a (2N+8)-dimensional column vector representing the set of control points around a rectangle. N denotes the valence of an extraordinary point. V is an invertible matrix whose columns are the eigenvectors of subdivision matrix. It is a square matrix which has (2N+8) $\times$ (2N+8) elements, and then $\hat{C}_0$ is also a (2N+8)-dimensional column vector.

In Equation (2), Λ is a diagonal matrix containing the eigenvalues of the subdivision matrix. The i-th diagonal element of Λ is the eigenvalue of the eigenvector corresponding to the i-th column of the matrix V. X_k is called the coefficient of the eigenbasis function. It has $(2N+8) \times 16$ elements which depend on the parameter k. $b(u,v)$ is the vector of the 16 cubic B-spline basis functions at the two-dimensional parameters (u, v). $t_{k,n}(u,v)$ is a function which transforms (u, v) to parameters on Ω_k^n. It can be described as follows:

$$\begin{aligned} t_{1,n}(u,v) &= (2^n u - 1, 2^n v), \\ t_{2,n}(u,v) &= (2^n u - 1, 2^n v - 1), \\ t_{3,n}(u,v) &= (2^n u, 2^n v - 1). \end{aligned} \tag{4}$$

Now we rewrite Equation (2) as follows:

$$s(u,v) = \sum_{i=1}^{2N+8} \mathbf{p}_i \, (\lambda_i)^{n-1} \sum_{j=1}^{16} x_{ijk} \, b_j(t_{k,n}(u,v)), \tag{5}$$

where $\mathbf{p}_i$ denotes the i-th element of the vector $\hat{C}_0$, λ_i denotes the i-th diagonal element of Λ, x_{ijk} denotes the i-th row and j-th column element of X_k, and b_j denotes the j-th element of the cubic B-spline basis functions.

The partial derivatives with respect to both u and v can be calculated by partial differentiation of a cubic B-spline basis function as described in Equation (5). The normal vector is then calculated from these derivatives.

We define the function ψ_i as follows:

$$\psi_i(u,v) = (\lambda_i)^{n-1} \sum_{j=1}^{16} x_{ijk} \, b_j(t_{k,n}(u,v)). \tag{6}$$

Using ψ_i, Equation (5) can be rewritten as follows:

$$s(u,v) = \sum_{i=1}^{2N+8} \psi_i(u,v) \, \mathbf{p}_i. \tag{7}$$

$\psi_i(u, v)$ is called an eigenbasis function. As a result, the evaluation of a subdivision surface at arbitrary parameter value can be treated as a linear combination of $\mathbf{p}_i$ and $\psi_i(u, v)$.

4.2. Inputs for the Algorithm

Our algorithm is executed in a fragment program. The pre-computed data are stored in the texture as inputs. Moreover, a face id and two parameters u and v are prepared for each fragment. The reason why a face id is necessary is that we need to store a face information where a fragment is rasterized from. This information is used for the texture lookup in the fragment program.

To calculate a face id and parameters u and v for each fragment, we prepare in advance a polygon which is an approximation of the subdivision surface. This polygon can be created by subdividing the original control polygon several times. In the subdivision process, face id and parameters of a subdivided polygon can be easily inherited from those of its original control polygon. Each face of a subdivided polygon is rasterized in the rendering process, and the parameters of each fragment are also created by a linear interpolation. The face id for each fragment is inherited from its parent subdivided polygon.

It is desirable that the subdivided polygon is a good approximation to (the limit surface of) the subdivision surface. If the polygon is coarse, the geometric error between the rasterized position of the polygon and the point on subdivision surface calculated by using the rasterized parameters could be large. This causes an ill effect for the final rendering result. In particular, gaps could be generated on the mesh because the resulting data computed in each fragment are rendered as points. In our experiments, however, an original control polygon subdivided only a few times can generate a good approximation even if creases or corners are included.

4.3. Texture Preparation

In this subsection, we will describe how to store the data to the texture which are needed to compute in the fragment program. We can say that using Equation (2) directly is too costly even on the current graphics hardware. This is because there have too much instruction sets. Consequently, we calculate $\hat{C}_0^T \Lambda^{n-1} X_k$ in advance and store it to the texture instead of storing each $\hat{C}_0^T$, Λ^{n-1} and X_k, respectively. This equation includes the coefficients n and k which depend on parameters u and v. Fortunately, the range of values which is assigned to these two coefficients are limited because n is defined as follows:

$$n = \lfloor min(-\log_2(u), -\log_2(v)) \rfloor + 1$$

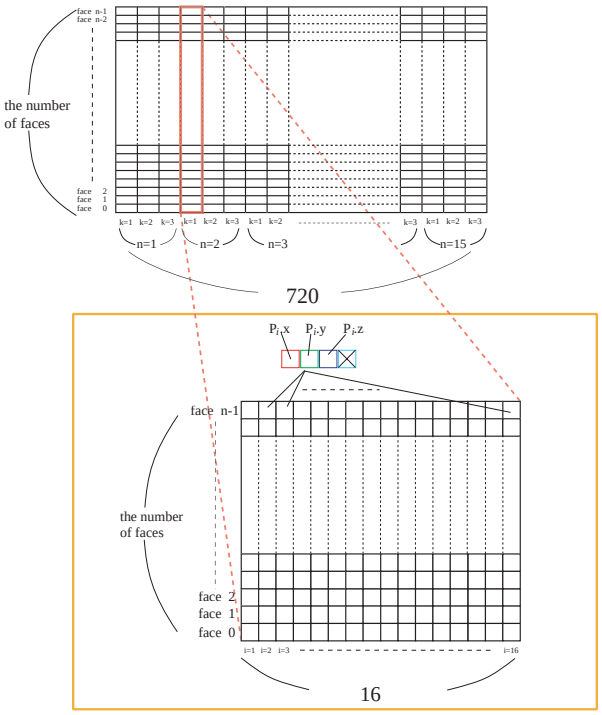


Figure 4. Procedure of storing data to texture in the form of two-dimensional tables.

where $\lfloor x \rfloor$ denotes the largest integer less than x.

To determine the maximum value of n is equivalent to how small parameters u and v are accepted. If the maximum value of n is 10, for example, parameters larger than $\frac{1}{2^{-10}} = \frac{1}{1024}$ can be evaluated precisely. Here we set the maximum value of n as 15. In this setting, parameters larger than $\frac{1}{2^{-15}} = \frac{1}{32768} < 10^{-4}$ can be evaluated. It is enough in our experiments. The value of k is only 1, 2 and 3.

Stam proposed in [15] that the evaluation of subdivision surfaces at arbitrary parameter value is regarded as the linear combination of eigenbasis functions and p_i, as shown in Equation (7). We replace this equation to the linear combination of bi-cubic B-spline basis functions and 16 points selected from $\hat{C}_0$, which are represented as P_i. P_i is defined as follows:

$$P_i = \sum_{j=1}^{2N+8} p_j(\lambda_j)^{n-1} x_{jik} \tag{8}$$

$s(u, v)$ is therefore evaluated as follow:

$$s(u, v) = \sum_{i=1}^{16} P_i b_i(t_{k,n}(u, v)) \tag{9}$$

We store $P_i (1 \leq i \leq 16)$ of each face to the texture as inputs. Since P_i depends on parameters n and k, all the

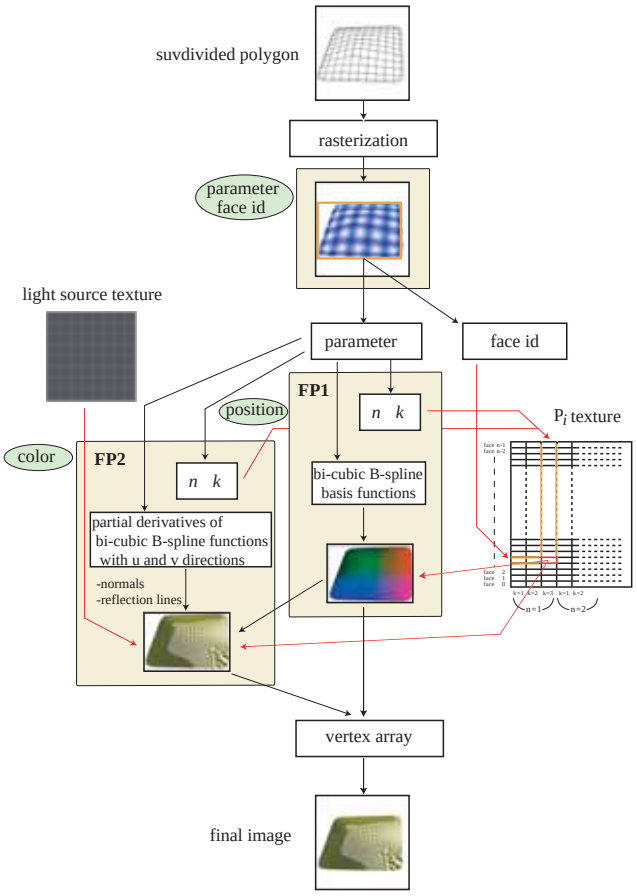


Figure 5. Overview of our algorithm on GPU.FP1 and FP2 denote the fragment programs.

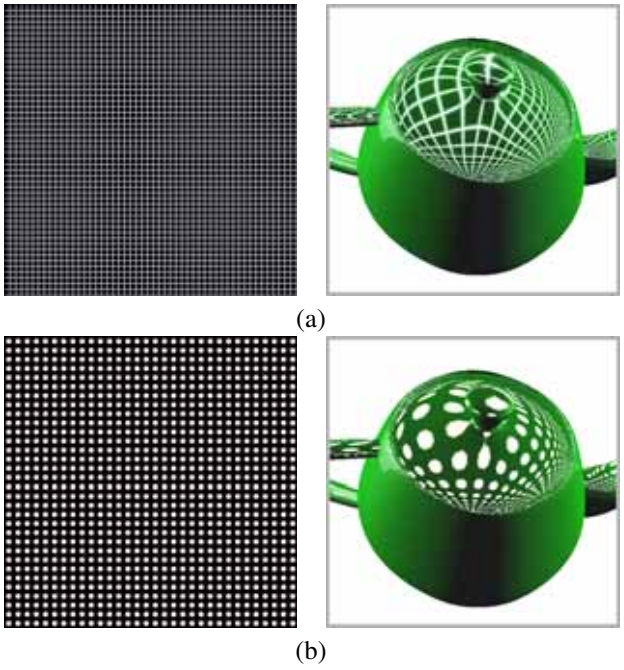


Figure 6. Light source textures and their reflection lines. (a) lattice pattern (b) regularly arranged circle pattern.

possible case of P_i have to be stored. k has 3 patterns and n has 15 patterns, therefore 45 patterns of P_i are required for each face. Figure 4 illustrates the concrete example for storing these patterns. The column corresponds to the number of faces. The row width is $16 \times 45 = 720$. The length of the column is limited by the capability of graphics hardware: If the number of faces is larger than the limit number, we use multiple blocks: We store the data in the first block, and then the consequent data exceeded to the limit are stored in the neighbor block. Multiple textures can be also used to store the data exceeded to the limit.

4.4. Algorithm

Figure 5 provides an overview of our algorithm. All the computations in our algorithm are executed in the fragment program. The input for our algorithm is described in the previous subsection.

We prepare a subdivided polygon as an input. Each face of the input mesh is decomposed into the set of fragments in the rasterization process. Each fragment has an inherited face id and linearly interpolated parameters u and v from the input mesh. These are once stored in the texture, and are used in the fragment program to compute positions and colors. When stored in the texture, we store not the whole viewport, but the smallest rectangle enclosing the polygon. This reduces the computational cost of the fragment program. Positions and colors are computed by drawing a rectangle polygon on pbuffer. The viewport size of pbuffer is the same as the transferred texture.

Positions are computed in the fragment program as follows. Firstly, we fetch the parameters and face id from the transferred texture at the current fragment. By using them, we select the 16 points from the texture where P_i is stored. The offset to access these points is computed with n and k in the fragment program. Also, we compute the bi-cubic B-spline basis functions with these parameters. Then, the position is computed by linear combination of the basis function with P_i. The result is not only copied to a vertex array but transferred as the texture to the fragment program which computes colors.

Colors are computed in another fragment program as follows. Firstly, we compute the tangent vectors of both directions u and v by the same way as computing positions. Note

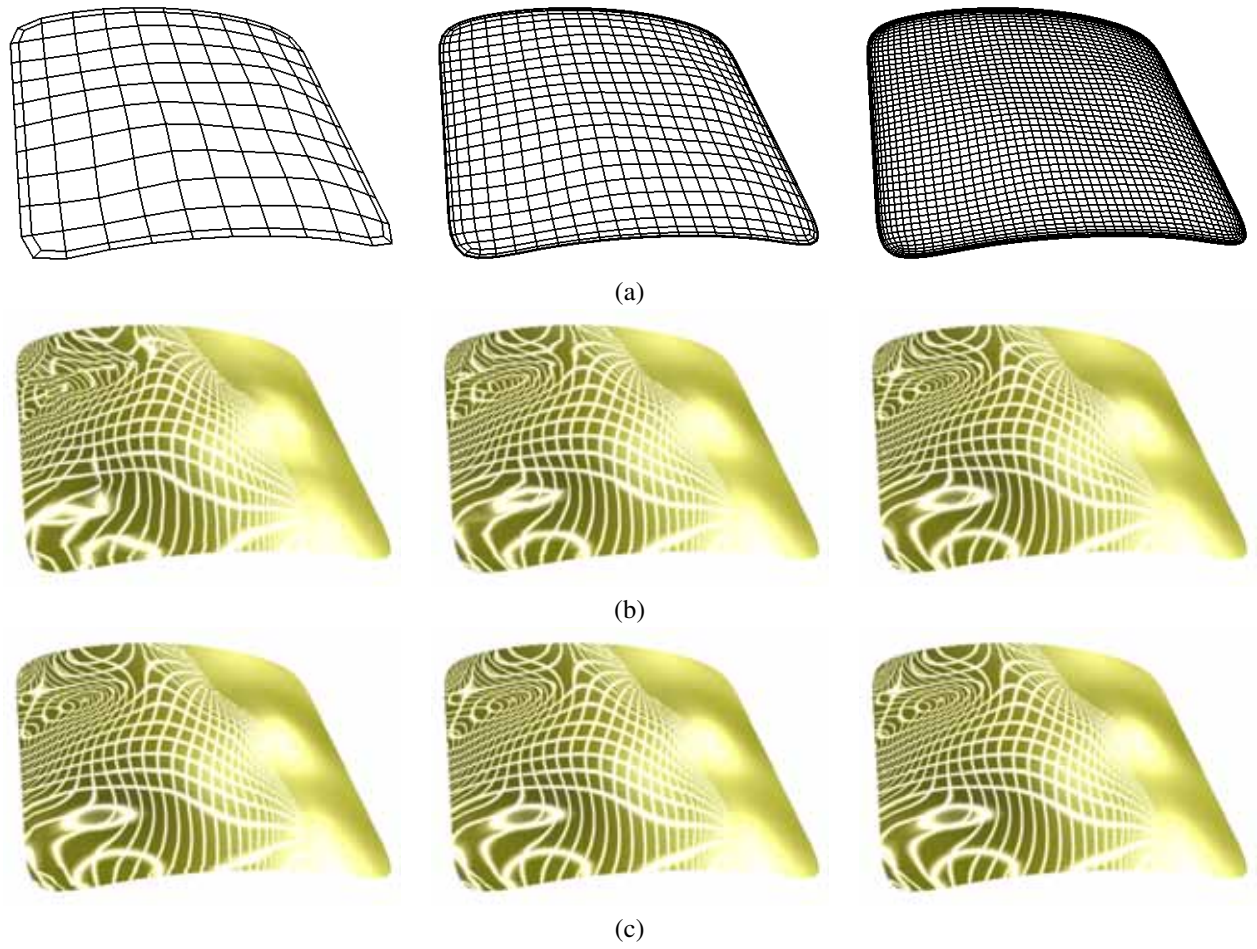


Figure 7. Reflection lines with subdivided polygons. From left to right: once, twice and three times subdivided polygons. (a) Wire-frame display. (b) Calculated by using polygons directly. (c) Calculated by our algorithm.

that the basis function we use in this fragment program is differentiated with respect to each direction. Next, we compute a normal vector by using these tangent vectors. Finally the intersection point is calculated by the method described in Section 3, and the color is computed. The eye position and positions of a surface are provided as inputs.

The data in each fragment corresponds to the position or the color of one vertex. We can obtain the final image by copying the results of the fragment program to a vertex array and render this array as points.

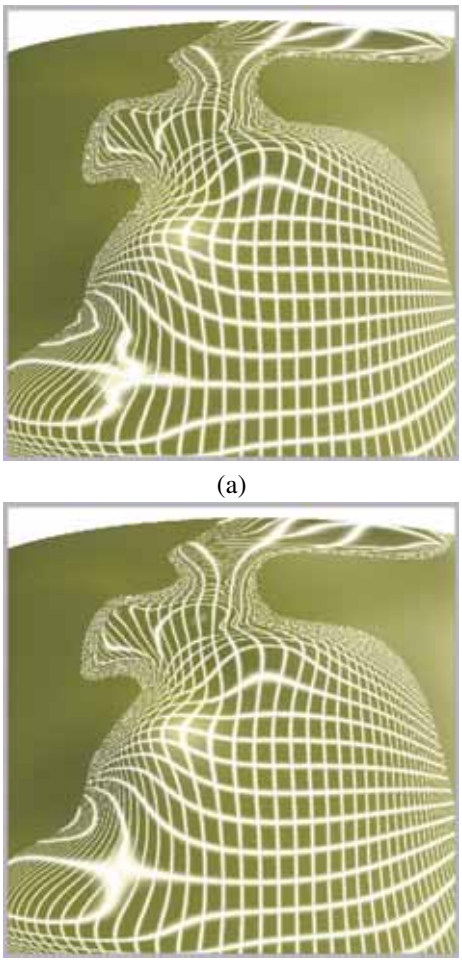
5. Results and Discussion

We prepare an image with a finite plane as a light source. Figure 6 shows two "light" images and the resulting reflection lines using these images.

Figure 6(a) shows an image of a lattice pattern. The result using this image is called *latticed reflection lines* [9]. Figure 6(b) shows an image of regularly arranged circles. This can be regarded approximately as the reflection of sphere lights. It has the interesting property that the reflected image of a circle in a convex region is an elliptical shape that stretches towards the principal direction of curvature [6].

It is desirable that the size of the finite plane in object space is larger than that of the surface. This avoids the case in which no intersection point can be found.

A calculated reflection line on a surface always has a width. This is because the width of a light line on an image is not less than the size of a pixel. The width of a reflection line depends on the resolution of the image and on the size of the finite plane in object space. The lower the resolution of an image is, or the smaller the size of the finite plane is, the thicker a reflection line is.



(a)

(b)

Figure 8. Close-up view of reflection lines. We use three times subdivided polygons in this case. (a) polygon based calculation (b) fragment based calculation.

The length of a fragment program for calculating reflection lines is very short. (roughly 30 instruction sets or less). This means that it has little effect for the whole calculation time.

Figure 7 shows several results to demonstrate the correctness of our algorithm. Figure 7(a) shows the wire-frame display results of three polygons; from left to right, once, twice and three times subdivided polygons from an original polygon, respectively. Figure 7 (b) shows the results using these subdivided polygons directly. That is, each vertex has not only a limit position but also a limit normal vector, and a position and a normal vector for each fragment is interpolated through the rasterization process. Using these positions and normal vectors, reflection lines are calculated by a

#faces	*300*	*1,200*	*4,800*
get parameter (sec.)	0.0008	0.003	0.012

Table 1. The relation between the number of faces and the time for getting parameters and face id per fragment.

#fragments	*300×300*	*500×500*	*700×700*
posistion (sec.)	0.03	0.07	0.13
color (sec.)	0.10	0.27	0.52

Table 2. The relation between the number of processed fragments and the computation time. Note that "#fragments" does not mean the window size.

fragment program. It can be seen from the results of Figure 7(b) that the correctness largely depends on the subdivision level. A three times subdivided polygon is better for the result than the one subdivided once. Figure 7 (c) shows the results calculated by our algorithm. Each figure represents the limit surface and reflection lines on them. Their results are independent of the levels of subdivision.

In once or twice subdivided polygons, it is not enough to approximate the limit surface. If the approximation is not enough, the problem can occur that the positions computed in the fragment program are different from those of pixels. This problem yields some gaps on the surface. It is basically caused by the lack of the data. One solution to address this problem is that the larger viewport than the actual one is used. However, using the larger viewport increases the number of processed fragments. Therefore it is desirable that the input mesh is fully approximated of the limit surface. Actually in the left of Figure 7(c), we use the 1.7^2 times larger viewport of actual one. In the middle of Figure 7(c), we use 1.3^2 times larger viewport. In the case of the right figure, we can obtain no gapped result by using the same size of the actual viewport.

Our method is more effective when a viewpoint comes close to the object. Figure 8 shows the example. Our method can represent the precise reflection lines, on the other hand, polygon-based one is not. Especially, it is quite noticeable on a higher curvature. Our method ensures the correctness of positions and colors for all of the fragments because it is per-pixel based evaluation. Figure 9 shows the results for more complicated polygon with 1776 faces.

We execute our algorithm on Athlon XP 2600+ GeForce FX 5900 Ultra. Our evaluation consists of four stages. The first stage is getting parameters and face id per fragment. The next stage is the calculation of positions and the third

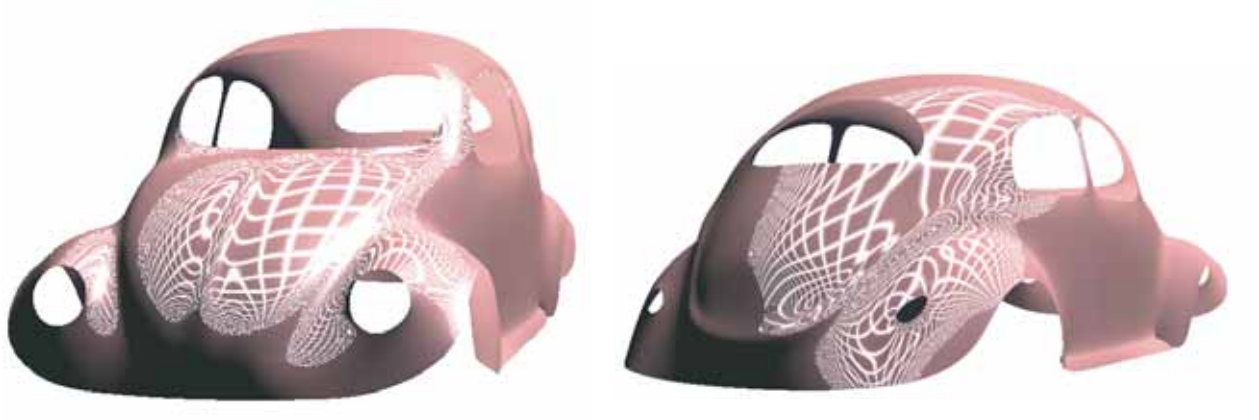


Figure 9. The result for a large polygon model ("Volkswagen", courtesy of Leif P. Kobbelt) with 1,776 faces.

stage calculates colors. The final stage is rendering. When considered about the computation time, only the first stage depends on the number of faces. The rest of three stages depend on the number of fragments, i.e. depend on the viewport on which the polygon is displayed. Table 1 shows the computation time of getting parameters and face id. Table 2 shows the time for computing positions and colors. The fragment program which calculates positions consists of 181 instructions. The one which calculates colors consists of 280 instructions. The time of renderin points is too fast to ignore it.

We can also calculate reflection lines for other representations of parametric surfaces (Beźier, B-spline, Coons, and so on). In these representations the number of instruction sets of fragment programs for calculating positions and partial derivatives is much lower than those for subdivision surfaces.

6. Conclusion and Future Work

In this paper, we propose a method for surface quality assessment of subdivision surface using reflection lines on programmable graphics hardware. We have shown that our method can calculate reflection lines precisely even for low levels of subdivision polygons. Though the computation of reflection lines is fast, the time for calculating positions and color is much slower. The whole process of our algorithm depends on current graphics hardware. We hope that it will be improved significantly in the near future.

References

[1] K.-P. Beier and Y. Chen. Highlight line algorithm for real-time surface quality assessment. *Computer Aided Design*, 26(4):268–277, 1994.

[2] J. Bolz and P. Schröder. Rapid evaluation of Catmull-Clark subdivision surfaces. In *Proc. the Seventh International Conference on 3D Web Technology (Web3D Symposium)*, pages 11–17. ACM Press, New York, 2002.

[3] J. Bolz and P. Schröder. Evaluation of subdivision surfaces on programmable graphics hardware. submitted for publication, 2003.

[4] E. Catmull and J. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer Aided Design*, 10(6):350–355, 1978.

[5] Y. Chen, K.-P. Beier, and D. Papageorgiou. Direct highlight line modification on NURBS surfaces. *Computer Aided Geometric Design*, 14(1):583–601, 1997.

[6] G. Farin. *Curves and Surfaces for CAGD*. Morgan-Kaufmann Publishers, 5th edition, 2001.

[7] N. Greene. Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications*, 6(11):21–29, 1986.

[8] M. Higashi, T. Kushimoto, and M. Hosaka. On formulation and display for visualizing features and evaluating quality of free-form surfaces. In *Proc. Eurographics '90*, pages 299–309. Elsevier, Amsterdam, 1990.

[9] M. Higashi, T. Saitoh, Y. Watanabe, and Y. Watanabe. Analysis of aesthetic free-form surfaces by surface edges. In *Proc. Pacific Graphics '95*, pages 294–305. World Scientific, Singapore, 1995.

[10] E. Kaufmann and R. Klass. Smoothing surfaces using reflection lines for families of splines. *Computer Aided Design*, 20(2):73–78, 1988.

[11] R. Klass. Correction of local irregularities using reflection lines. *Computer Aided Design*, 12(7):411–420, 1980.

[12] C. Loop. Smooth subdivision surfaces based on triangles. Master's thesis, University of Utah, Department of Mathematics, 1987.

[13] J. Loos, G. Greiner, and H.-P. Seidel. Modeling of surfaces with fair reflection line pattern. In *Proc. Shape Modeling In-*

ternational '99, pages 106–115. IEEE CS Press, Los Alamitos CA, 1999.

[14] W. H. Press, W. T. Vetterling, S. A. Teukolsky, and B. P. Flannery. *Numerical Recipes in C++: the Art of Scientific Computing*. Cambridge University Press, 2002.

[15] J. Stam. Exact evaluation of Catmull-Clark subdivision surfaces at arbitrary parameter values. In *Computer Graphics (Proc. SIGGRAPH 98)*, pages 395–404. ACM Press, New York, 1998.

[16] C. Zhang and F.-F. Cheng. Removing local irregularities of NURBS surfaces by modifying highlight lines. *Computer Aided Design*, 30:923–930, Aug. 1998.

[17] D. Zorin and D. Kristjansson. Evaluation of piecewise smooth subdivision surfaces. *The Visual Computer*, 18(5–6):299–315, 2002.