MITSUBISHI ELECTRIC RESEARCH LABORATORIES
http://www.merl.com

Hardware Support for Non-photorealistic Rendering

TR2001-23 May 2001

Abstract

Special features such as ridges, valleys and silhouettes, of a polygonal scene are usually displayed by explicitly identifying and then rendering édgesfor the corresponding geometry. The candidate edges are identified using the connectivity information, which requires preprocessing of the data. We present a non-obvious but surprisingly simple to implement technique to render such features without connectivity information or preprocessing. At the hardware level, based only on the vertices of a given flat polygon, we introduce new polygons, with appropriate color, shape and orientation, so that they eventually appear as special features.

Siggraph/Eurographics Graphics Hardware Workshop, Los Angeles, August 2001

This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of Mitsubishi Electric Research Laboratories, Inc.; an acknowledgment of the authors and individual contributions to the work; and all applicable portions of the copyright notice. Copying, reproduction, or republishing for any other purpose shall require a license with payment of fee to Mitsubishi Electric Research Laboratories, Inc. All rights reserved.

Copyright © Mitsubishi Electric Research Laboratories, Inc., 2001
201 Broadway, Cambridge, Massachusetts 02139

Submitted Jan 2001, revised and released June 2001.

Hardware Support for Non-photorealistic Rendering

Ramesh Raskar
MERL, Mitsubishi Electric Research Labs

Abstract

Special features such as ridges, valleys and silhouettes, of a polygonal scene are usually displayed by explicitly identifying and then rendering 'edges' for the corresponding geometry. The candidate edges are identified using the connectivity information, which requires preprocessing of the data. We present a non-obvious but surprisingly simple to implement technique to render such features without connectivity information or preprocessing. At the hardware level, based only on the vertices of a given flat polygon, we introduce new polygons, with appropriate color, shape and orientation, so that they eventually appear as special features.

1 INTRODUCTION

Sharp features convey a great deal of information with very few strokes. Technical illustrations, engineering CAD diagrams as well as non-photo-realistic rendering techniques exploit these features to enhance the appearance of the underlying graphics models. The most commonly used features are silhouettes, creases and intersections. For polygonal meshes, the *silhouette* edges consists of visible segments of all edges that connect back-facing polygons to front-facing polygons. A crease edge is a *ridge* if the dihedral angle between adjacent polygons is less than a threshold. A crease edge is a *valley* if the angle is greater than a threshold (usually a different, and larger one). An *intersection* edge is the segment common to the interior of two intersecting polygons.

Many techniques have been explored, most of them in 3D software applications. For polygonal scenes, silhouettes can be rendered at interactive rates by techniques described in [Rossignac92] [Markosian97] [Raskar99] [Gooch99] and [Hertzmann99]. Similarly, identifying sharp crease edges is relatively simple when the adjacency information for polygons is available. The traditional approach is to traverse through the scene polygon graph and for each edge find the relationship between adjacent polygons to determine whether it is a silhouette or a sharp crease edge. If it is, one can render the edge with the rest of the geometric primitives using a visibility algorithm. However, explicitly identifying such edges is a cumbersome process, and

raskar@merl.com

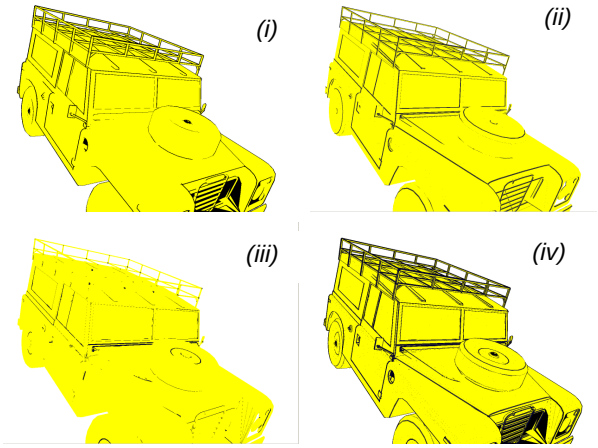

Figure 1: (i) Silhouettes, (ii) ridges, (iii) valleys and (iv) their combination for a polygonal 3D model rendered by processing one polygon at a time.

usually not supported by rendering APIs or hardware. The rendering hardware is typically more suited to working on a small amount of data at a time. For example, most pipelines accept just a sequence of triangles (or triangle soups) and all the information necessary for rendering is contained in the individual triangles. Therefore, in the absence of special data structures, such edge operations require random traversals of the scene graph to find adjacent polygons. Maintaining (or creating) a large adjacency matrix also increases the memory requirements. Hence, such operations are left to the 3D software application. In addition, for dynamic scenes or triangle soups, identifying ridges and valleys is a cumbersome task. The extra effort required on the part of the programmer is probably the primary reason why we do not see such important features being used by many during visualization.

1.1 Procedural Geometry Generation

We focus on an approach that renders special features at uniform width directly into the framebuffer using a *primitive shader* stage in the graphics pipeline. This allows for the processing of a single polygon at a time, as in traditional rendering, and still display features that would otherwise require connectivity information. We achieve this by introducing additional geometric primitives in the scene with no or only small changes in rest of the rendering pipeline, and without using additional rendering operations such as lighting or texture mapping. This type of geometric processing follows the trend of current generation of hardware supporting programmable vertex and fragment operations [DirectX][Nvidia01][Proudfoot01]. The additional polygons are to be generated on-chip, without any software assistance. Specifically, the polygon generation block is inserted between the vertex-and-primitive-assembly stage and the vertex-shading stage of the graphics pipeline (Figure 2). No data other than the vertex positions and normal of the polygon and current viewing

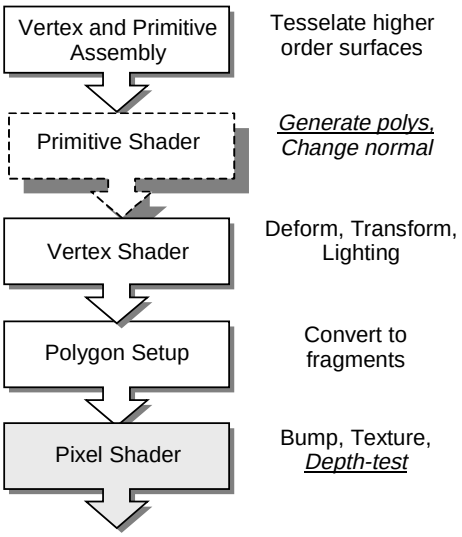

Figure 2: The new *primitive* shader in the programmable graphics pipeline. The pixel shader is also modified.

parameters is used. The high level pseudo-code for the primitive shading block is shown below.

```
For each polygon
  If back-facing
    Modify polygon geometry to display silhouettes
  Else /* if front-facing */
    Attach new polygons to display ridges or valleys
```

1.2 Motivation

Consider the psuedo-code shown in Figure 3 (using as an example the OpenGL API). Can we design hardware to support such API functionality? Can we render special features in a scene specified by a sequence of individually defined polygons? Rendering methods that can pipeline and process a single primitive at a time appear to win over even efficient methods that require simultaneous operations on multiple polygons. A common example is the ubiquitous z-buffer for visibility computation. Methods that require pre-sorting of polygons with respect to the camera, although elegant, are less popular. Rendering APIs or hardware also cannot support operations that require conditional search in datasets of unknown size. Our method thus wins over the traditional edge rendering approach by allowing for the rendering of a single polygon at a time. A secondary, but well known, problem with explicit edge rendering is that the typical polygon scan conversion pipelines are less efficient when 2D or 3D 'line' segments are displayed. The lines are filled at uniform width in screen space, and hence need to be treated separate from the polygon setup and fill.

```
glEnable (GL_SIL_RIDGE_VALLEY);
glBegin (GL_TRIANGLES):
  glVertex (…);
  glVertex (…);
  …
  …
glEnd ();
glDisable (GL_SIL_RIDGE_VALLEY);
```

Figure 3 : A desirable simple API calling sequence

The silhouette edges are view-dependent and hence, clearly, do need to be computed per frame. However, one may wonder why ridges and valleys, which are view independent (and hence fixed under rigid transformation), should be identified per frame. There are several reasons. For dynamic scenes with non-rigid deformations, the relationship between neighboring polygons can change. The vertex shaders also allow procedural deformation of vertices that can add interesting features. For example, a waving flag or chest deformation for a breathing character can be achieved using vertex-level programming, without complete feedback to the software application [Nvidia01]. 3D applications as well as vertex shaders allow key-frame animation interpolation and (vertex) morphing, which affect the dihedral angles at run-time. In some rare cases, the mesh topology itself can change over time, affecting the connectivity information. This includes intentional separation or merging of meshes, LODs, and subdivision surfaces. When the models are authored so that they are specified using higher order surfaces, it is often difficult to algebraically indicate high curvature regions, which become ridges and valleys after (possibly dynamic) tessellation.

Our goal is to directly display special features using only local calculations. Just as there are no (global) constraints about what type of polygonal objects can be scan converted, our method does not depend on the object being closed or open, concave or convex, having cusps or on the genus of the object.

Contribution

Our main contribution is a set of techniques to render special features, by using the available information contained in a single polygon and without connectivity information, allowing pipelined hardware implementation.

2 THE RENDERING PROCESS

The rendering method assumes that the scene consists of oriented convex polygons. This allows us to distinguish between front and back-facing polygons for silhouette calculations, and ensure correct notion of dihedral angle between adjacent polygons. The procedures for the four types of special features are completely independent. Note that, we perform the same four procedures on *all* the polygons in the scene, without knowing a-priori which type of special feature would be visible at that polygon. Depending on the viewing conditions and the relationship between adjacent polygons, the appropriate special feature, if enabled, emerges after rasterization.

For silhouettes and ridges, the additional polygons are procedurally introduced during the primitive shader stage. The pixel shader stage is not modified.

For valleys and intersections, in addition to the new geometry, a second depth buffer and a new type of depth test is required. Thus, both stages, the primitive shader and the pixel shader, are modified. In the appendix, we describe a less efficient method using currently available rendering resources.

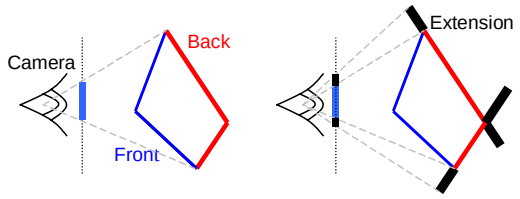

Figure 4: Silhouettes. Back-facing polygons in red before (left) and after enlargement (right).

2.1 Silhouettes

The basic idea in our approach is to enlarge each back-facing polygon so that the projection of the additional part appears around the projection of the adjacent front-facing polygon, if any (see Figure 4, shown in flatland). If there is no adjacent front-facing polygon, the enlarged part of the back-facing polygon remains hidden behind existing front-facing polygons. The normal of the back-facing polygon is flipped to ensure that it is not culled during back-face culling. To achieve a given width in the image space, the degree of enlargement for each back-facing polygon is controlled, depending on its orientation and distance with respect to the camera. This method is influenced by [Rossignac92] and [Raskar99], but there are two differences. They use two passes of rendering and consider enlargement using only orthographic (or weak perspective) projection.

```
For each polygon
  If front-facing
    Color polygon white
  Else /* if back-facing */
    Enlarge according to view and color black
    Flip normal
```

Consider the situation shown in Figure 5 for computing the required enlargement (figure in flatland). When we also consider the angle between the view vector V i.e. direction from viewpoint to the vertex, and camera axis vector C, i.e. the vector perpendicular to the image plane, the enlargement is proportional to $z(V.C)/(V.N)$, where z is depth of the vertex with respect to the camera. Both dot products can be easily calculated per polygon vertex. Note that, the angle $(V.N)$ is also commonly used during

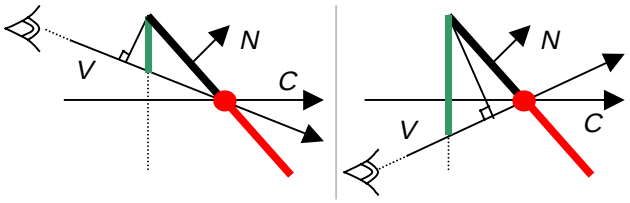

Figure 5: The effective projection (green) in the image plane of the polygon enlargement (black). The thickness of the rendered feature is controlled by changing the width of the enlargement.

the lighting calculations for highlights produced by specular reflectance for a local viewpoint. The dot product $(V.C)$ is in fact constant for a given location in image space and radially symmetric around the principal point. Thus it can be pre-calculated.

The required enlargement is also dependent on the orientation of the edge of the polygon. If E is the edge vector, so that $\cos(\alpha) = V \cdot E$, then the required enlargement is $z \sin(\alpha)\ (V \cdot C)/(V \cdot N)$ in the direction $E \times N$. The shift in edges converts an n-sided face, with vertices P_i, i=0,1..n-1, edges E_{ij}, j=(i+1)%n, into $2n$-sided planar polygon. The $2n$ vertices are given by

$$P_k + w_{sil} \left[E_{ij} \times N \right] \left[z \sin(\alpha) V \cdot C)/(V \cdot N) \right]$$

where, k=i,j and $\alpha = \cos^{-1}(V \cdot E_{i,j})$

The uniform-width silhouette edges can be rendered at different pixel widths using a parameter, w_{sil}, to control the enlargement of back-facing polygons. If desired, interesting patterns of silhouette edges can be rendered by texture mapping the enlarged part.

2.2 Ridges

For rendering silhouettes, we modify each back-facing polygon. For ridges and valleys, we modify instead each front-facing polygon. This idea, while simple, surprisingly has never been mentioned or explored in the graphics or geometry literature. Say, we want to display in black, the visible part of each edge for which the dihedral angle between adjacent polygons is less than or equal to a user selectable global threshold θ, superimposed on the original model if desired. (Strictly speaking, dihedral angle is the angle between the normals of two oriented polygons. For the sake of simplicity of description, as seen in Figure 6(i), we will use the term dihedral angle to mean angle between the 'front' faces of the corresponding planes i.e. (180^0 – angle between normals)). Typical range of the threshold θ is ($0,180^0$), but this technique can support the complete range of angles ($0,360^0$). The threshold value used for rendering in Figure 1(ii) is 120^0.

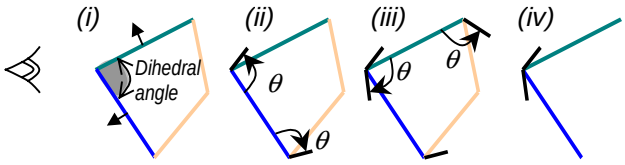

Figure 6: Ridges. (i) Front-facing polygons, (ii) and (iii) black quads at threshold angle θ are added to each edge of each front-facing polygon, (iv) at a sharp ridge, the black quads remain visible.

We add black colored quadrilaterals (or *quads* for short) to each edge of each front-facing polygon. The quads are oriented at angle θ with respect to the polygon as seen in Figure 6(ii) and 6(iii) in flatland. The visibility of the original and the new polygons is performed using the traditional depth buffer. As shown in Figure 6(iv), at a sharp ridge, the appropriate 'edge' is highlighted. When the dihedral angle is greater than θ, the added quadrilaterals are hidden by the neighboring front-facing polygons. Figure 7(i) and (ii) show new quadrilaterals that remain hidden after visibility computation in 6(iii).

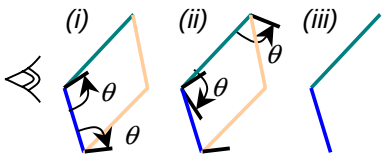

Figure 7: Ridge without sharp angle. (i) and (ii) Black quads are added, (iii) the quads remain invisible after rasterization

```
For each polygon
  If front-facing
    Color polygon white
    For each edge of the polygon
        Attach new black quad at angle θ
```

The width of the new quadrilateral, measured perpendicular to the edge, is determined using the same enlargement technique (Figure 5) used in the previous subsection for displaying silhouettes. The quadrilateral for each edge E_{ij}, i=0,1..n, j=(i+1) mod n, for n-sided polygon is defined by the two vertices P_i and P_j forming the edge E_{ij} and the two new vertices

$$P_k + w_{ridge} Q_{ij} \left[z \sin(\alpha) (V \cdot C)/(V \cdot R_{ij}) \right]$$

where k=i,j, $\alpha = \cos^{-1}(V \cdot E_{ij})$, and
normal for the new quad is, $R_{ij} = -N \cos(\theta) + (E_{ij} \times N) \sin(\theta)$
and vector perpendicular to E_{ij}, $Q_{ij} = -N \sin(\theta) - (E_{ij} \times N) \cos(\theta)$

The thickness of the rendered 'edge' can be controlled with the parameter w_{ridge}. The display of sharp ridges is, thus, possible in a single pass without connectivity information, simply by processing one polygon at a time. It is also possible to render silhouette and sharp edges together in the same rendering pass as described in the pseudo-code below. Note, again, that for silhouettes and ridges only the primitive shader stage is modified.

```
For each polygon
  If front-facing
    Color polygon white
    For each edge of the polygon
        Attach new black quad at angle θ
  If back-facing
    Enlarge and color black
```

2.3 Valleys

Our method for rendering valleys is very similar to that of rendering ridges, because both are types of sharp edges defined by dihedral angle threshold. Given a user-selectable global threshold ϕ for dihedral angle, we would like to display, say in black, visible part of each edge for which the dihedral angle is greater than ϕ, superimposed on the original model if desired. Typical range of the threshold ϕ is $(180,360^0)$. For example, in Figure 1, the threshold is 240^0. Similar to Figure 6, we add black quadrilaterals at angle ϕ to each edge of each front-facing polygon as shown in Figure 8. When the dihedral angle is greater than ϕ (Figure 8(i) and (ii)), new quadrilaterals appear behind the nearest front-facing polygons. On the other hand, when the valley is not sharp (Figure 8(iii) and (iv)), the new quadrilaterals appear in

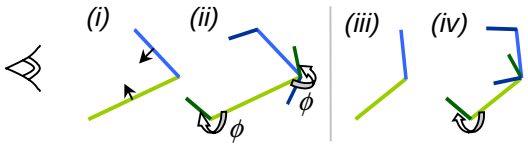

Figure 8: Valleys. (i) Front-facing polygons (ii) black quads at angle ϕ are added to each edge of each face. When the valley is sharp, the quads remain hidden. (iii) and (iv) When the valley is not sharp, the quads become visible.

front of the neighboring polygons. This is the exactly reverse of the situation for ridges, and hence leads to a more complex algorithm. How can we 'show' quads that remain occluded and 'hide' quads that appear in front of nearest front-facing polygons?

Our solution involves using two depth buffers for visibility. The idea of using multiple depth and framebuffers during scan conversion has been explored by [Rossignac95] and usually involves many passes of rendering and mixing line and polygon rendering. Here we present a new technique to trap the appropriate added quadrilaterals, as shown in Figure 9. The two depth buffers, z1 and z2, are identical except that depth values in z2 are slightly larger than corresponding values in z1. The front envelope of cyan colored region shows values in z1 and the far envelope shows values in z2. In addition to the traditional greater-than-test and less-than-test, we also need a new Boolean depth test, called *between-test*. It returns true when the incoming fragment has depth value between the existing values in z1 and z2. The simplified pseudo-code is shown below. The new quadrilaterals are to be added in the primitive shader stage in Figure 2 but the update of first and second depth buffers and depth tests are to be performed as part of the pixel shader stage.

```
For each front-facing polygon
  Render the polygon
    If (less-than-test(z1)) Update color,z1 and z2 buffer
  For each edge of the polygon
    Render new black quad at angle ϕ
      If (between-test(z1,z2)) Update color buffer
```

Thus, a new quad never updates the depth buffer. It is displayed (i.e. color buffer is updated) only if it lies between the depth interval trap (shown in cyan in Figure 9(ii)) created by two copies of the same neighboring front-facing polygon. The polygons in the scene can be processed in any order.

Consider the example in Figure 10 in flatland. In (i), first the primitive shading stage adds quads *a* and *a'* to polygon *A* at angle ϕ. During pixel shading stage, polygon *A* updates the color buffer (shown on the image plane), and both depth buffers. The quads *a*

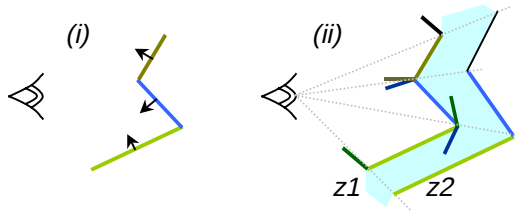

Figure 9: Using depth interval buffer to trap the added quadrilaterals

and *a'* do not affect the depth buffer. Both *a* and *a'* are checked with between-test to see if they are between previously stored values in z1 and z2. In this case, they are not and hence they do not affect the color buffer. In (ii), the polygon *B* add new quads, *b* and *b'*. During pixel shading, *B* updates the color, z1 and z2 buffer. The between-test fails for quad b but returns true for quad *b'*. Hence black colored quad *b'* updates the color buffer, emerging as a special feature. The depth buffer values remain unaffected (in this case, correspond to polygon *A*). Hence, if some polygon later occludes *A*, the color (as well as depth buffer values) will be appropriately overwritten. Finally in (iii), the polygon *C* adds news quads *c* and *c'*. However, the between-test fails for both quads and no new special features emerge.

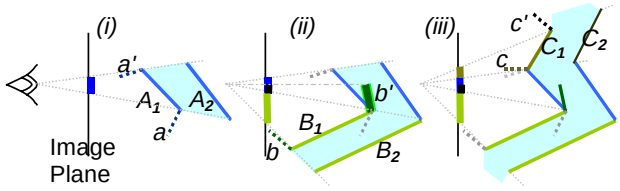

Figure 10: Steps in rendering valleys.

2.4 Intersections

Although intersections are not encountered in quality graphics models, imprecise conversions or processing can result in intersection of polygons. The technique to detect intersection between two visible front-facing polygons for a given viewpoint is very similar to the procedure for rendering sharp valleys.

```
For each front-facing polygon
  Render the polygon
    If (less-than-test(z1)) Update color,z1 and z2 buffer
    ElseIf (between-test(z1,z2)) Update color(=black) buffer
```

Each fragment during scan conversion of front-facing polygon is checked to see if it is in the in between z1 and z2 depth range (Figure 11). For a model with intersections or coplanar polygons, such fragments will be trapped and rendered if they belong to the visible front-facing polygons.

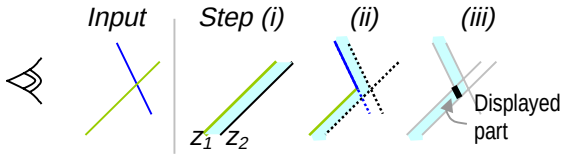

Figure 11: Two intersecting polygon interiors (left) are displayed by finding parts trapped between the two depth buffers

3 ISSUES

The techniques work wonderfully well for different relationships among viewpoint, polygons and their neighbors in the mesh. However, in some cases minor artifacts are observed. For ridges and valleys, if the dihedral angle between any two adjacent visible front-facing polygons is very close to the threshold angle, the so-called 'z-fighting' due to near-coplanarity of the polygons and the new quads is noticeable.

When the special features are rendered very wide (usually more than 20 pixels), cracks or small gaps can develop (Figure 12. Zoom in with PDF viewer to see them more clearly.) The cracks are rarely visible when the features are not rendered very wide.

The four types of geometric features are independent except when a ridge edge is also a silhouette edge. When the threshold ridge angle $\theta < 180^0$, which is usually the case, the new quads corresponding to the silhouette features will occlude the quads corresponding to ridge feature. When threshold $\theta \geq 180^0$, the rendered feature depends on the dihedral angle between the adjacent front and back-facing polygon. If silhouettes and ridges are rendered in the same color, say black, the artifacts can be avoided.

4 PERFORMANCE

As shown in Figure 2, the new quadrilaterals are added in the pipeline before vertex shading. For silhouette and ridge rendering, the output format of the primitive shader is same for the original and the new polygons. Hence, the impact on the design of the rest of the pipeline is minimal. Rendering valley and intersection, however, involves tagging new quadrilaterals so that they are processed differently when they reach the pixel shading stage. The pixel shading stage requires an additional depth buffer and comparison functionality.

For each edge of a front-facing n-sided polygon, we add two quads, one each for a ridge and a valley, i.e. 4n new vertices. For each back-facing polygon, the enlargement leads to n new vertices. To evaluate the impact of additional geometry, we need to look at the bandwidth, transform (part of vertex shader stage) and fill rate (part of pixel shader stage) performance [Seidel00]. The fill rate is not a significant issue because the screen area in pixels is only marginally higher when the new quads are also rendered. Nevertheless, the number of vertex transformations increases by a factor of 2 to 5. The overall rendering performance in many applications is, however, limited by the bandwidth needed to feed vertices to the graphics processor rather than the graphics processor cycles available for the geometry [Vlachos01][Nvidia01][Olano98]. Since the new vertices are to be generated on the chip, the bandwidth requirements do not change. This idea is also exploited, for example, in tessellating higher-order primitives in GeForce3 [Nvidia01] or curved PN triangles in ATI chips [Vlachos01]. Generation of new vertices, however, affects the performances of vertex buffers and caches.

When a 3D software application identifies special features, only a

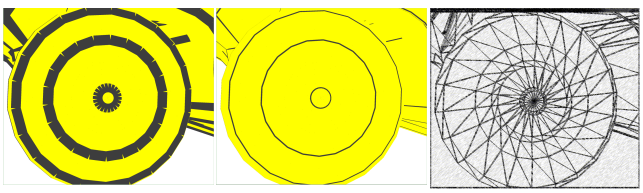

Figure 12: Cracks when rendering wide ridges, cracks disappear for thinner rendering, underlying wire frame model.

subset of the scene edges need to be rendered, but the application must maintain an edge-based data structure for adjacency computation, and thus also requires significant memory. In certain cases, it is practically impossible to detect the special features using software application e.g. during on-chip geometry modification for key frame animation interpolation, morphing and deformation using programmable vertex shading stage.

5 CONCLUSION

Although at first it appears rather unusual and then in retrospect rather simplistic, the idea of introducing new geometric primitives leads to a robust and practical set of techniques for rendering special features. To a naïve student, it may seem impossible to display edge-based features without processing edge primitives or rendering lines.

The main features of our rendering method are summarized below.

- Single polygon at a time processing
- No preprocessing or connectivity information allowing the display of dynamic scenes with non-rigid deformations and key-frame animation interpolations
- No explicit identification of special edges
- A single traversal of the scene graph (or polygon soup)
- Adding geometry procedurally using only local information
- Adding purely geometric features without lighting or texture mapping operations, allowing the original and the new primitives to be treated very similarly
- Minimal or no change to authoring tools

Many extensions of the basic functionality of the primitive shader are possible. The new quads can be rendered with appropriate lighting and texturing for additional shape cues or for non-photorealistic rendering. Partially transparent front-facing polygons will display hidden creases in shades of gray. The silhouette can be made to appear smooth by rendering all back-facing polygons with curved PN triangles [Vlachos01]. Currently, vertex and pixel shading programmability is available in graphics pipeline, but we can soon expect to see on-chip programmable shaders that manipulate one complete primitive and sometimes a collection of primitives. The primitive shader stage will allow advanced geometry modification, constrained dynamic tessellation (curved PN triangles with more boundary conditions) and smooth transition between LODs. The superior polygonal approximation of the underlying surfaces will lead to higher quality rendering of special features using our technique. Finally, when hardware support is lacking, our method can be implemented as a part of a software library to simplify the programming task of rendering special features.

ACKNOWLEDGEMENTS

We would like to thank Michael F Cohen for initial discussions on this project. We thank Fredric Vernier, Rebecca Xiong and Ajith Mascarenhas for their useful comments.

6 REFERENCES

[DirectX] Microsoft Direct X 8.0, 2001.

[Gooch99] Bruce Gooch, Peter-Pike J. Sloan, Amy Gooch, Peter Shirley and Richard Riesenfeld. Interactive Technical Illustration. Symp on Interactive 3D Graphics, 1999

[Hertzmann99] A. Hertzmann. Introduction to 3D Non-Photorealistic Rendering: Silhouettes and Outlines. SIGGRAPH Course Notes. 1999.

[Proudfoot01] Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan A Real-Time Procedural Shading System for Programmable Graphics Hardware. SIGGRAPH 2001.

[Markosian97] Lee Markosian, Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein, and John F. Hughes. Real-Time Nonphotorealistic Rendering, Computer Graphics (Proceedings of SIGGRAPH '97), August, 1997.

[Nvidia01] Nvidia Corp. User Programmable Vertex Engine. 2001.

[Olano98] Olano, Marc and Anselmo Lastra. A Shading Language on Graphics Hardware: The PixelFlow Shading System, Computer Graphics (Proceedings of SIGGRAPH '98), July, 1998.

[Raskar99] Ramesh Raskar, Michael Cohen. Image Precision Silhouette Edges. Interactive 3D Graphics Symposium, 1999.

[Rossignac92] Jarek Rossignac, Maarten van Emmerik. Hidden Contours on a Framebuffer. Proceedings of the 7th Workshop on Computer Graphics Hardware, Eurographics Sept. 1992.

[Rossignac95] Jarek Rossignac. Depth Interval Buffer. http://www.gvu.gatech.edu/~jarek/papers/

[Seidel00] Hans-Peter Seidel and Wolfgang Heidrich, Hardware Shading: State-of-the-Art and Future Challenges. Graphics Hardware Workshop. Sept, 2000.

[Vlachos01] A. Vlachos, J Peters, Chas Boyd, Jason Mitchell, Curved PN Triangles. Interactive 3D Graphics Symposium, 2001.

7 APPENDIX

When a dual depth buffer is lacking, the valleys and intersections can be rendered using one depth and one stencil buffer at the cost of multiple (upto 3) passes of rendering. The processing does not require adjacency information or preprocessing.

The details and corresponding code is available on the project website http://www.cs.unc.edu/~raskar/NPR/.