Hardware-Determined Feature Edges

Morgan McGuire and John F. Hughes*
Brown University

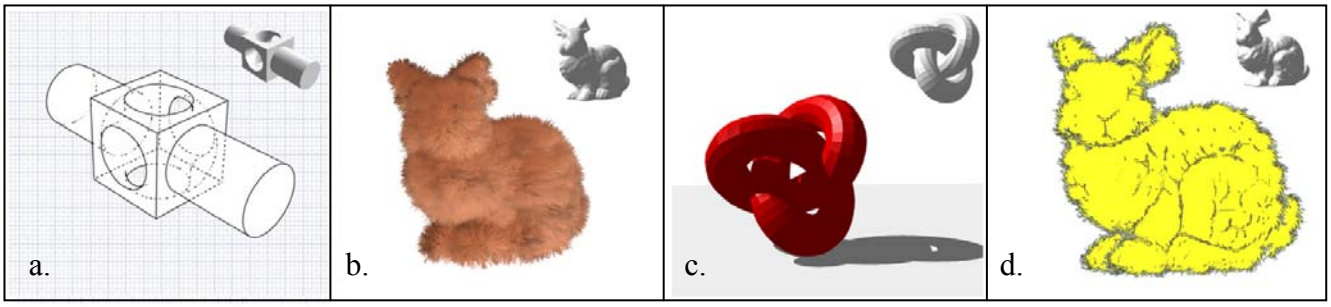

Figure 1: Hidden line, realistic fur, shadow volume, and cartoon fur styles implemented using only the GPU.

Abstract

Algorithms that detect silhouettes, creases, and other edge based features often perform per-edge and per-face mesh computations using global adjacency information. These are unsuitable for hardware-pipeline implementation, where programmability is at the vertex and pixel level and only local information is available. Card and Mitchell and Gooch have suggested that adjacency information could be packed into a vertex data structure; we describe the details of converting global/per-edge computations into local/per-vertex computations on a related 'edge mesh.' Using this trick, we describe a feature-edge detection algorithm that runs entirely in hardware, and show how to use it to create thick screen-space contours with end-caps that join adjacent thick line segments. The end-cap technique favors speed over quality and produces artifacts for some meshes.

We present two parameterizations for mapping stroke textures onto these thick lines—a tessellation-independent screen space method that is better suited to still images, and an object space method better suited to animation. As additional applications, we show how to create fins for fur rendering and how to extrude contours in world-space to create the sides of a shadow volume directly on the GPU.

The edge mesh is about nine times larger than the original mesh when stored at 16-bit precision and is constructed through a linear time pre-processing step. As long as topology remains fixed, the edge mesh can be animated as if it were a vertex mesh.

Keywords: NPR, GPU, contour, shadow volume, silhouette

1 Introduction

Many non-photorealistic rendering algorithms draw strokes to mark the edge-features of a model; these strokes are often textured, and may not follow the geometry of the model-edge exactly. Current algorithms can be coarsely divided into geometric and image-based methods. Geometric methods detect features and then convert the resulting poly-lines into strokes to be rendered. Image-based methods examine the color and depth components of the frame buffer to discern edges. Computing the edge list on the CPU and transmitting it to the GPU is a bottleneck for the first class of algorithms; reading back the frame-buffer and pixel-processing are bottlenecks in the second class.

Card and Mitchell [Card02] and later Gooch [Gooch03] described a method of packing information about adjacent faces into a "vertex" data structure that actually represents a single edge of a mesh, and from which it is easy to determine whether that edge represents a visible feature. We rediscovered this idea extended it in several ways; the present paper therefore contributes (a) a detailed explanation of the underlying method, and (b) algorithms for using the detected edges in several new ways. In particular, we present an algorithm for directly computing and rendering the visible feature edges[1] of a model with thick, textured lines that executes entirely on the GPU after an initial preprocessing step. We show how to make this algorithm work for key-frame and skin-and-bones (matrix skinned) animated models. We describe several applications of these ideas, and discuss their limitations.

There are four drawbacks of the work: the first is that some $O(n)$-time preprocessing is required on the CPU; the second is that the data sent to the GPU is about nine times as large as the data sent for an ordinary rendering; the third is that the thick-line-drawing algorithm, which generally fills in gaps between adjacent thick line segments, can fail to do so in some cases.; the fourth is that the size of the vertex program makes it slow down the current generation GPU, so there's a substantial loss of rendering speed.

In Appendix A, we briefly discuss two further applications of these ideas: generating Zorin-Hertzmann-style smooth contours, and generating suggestive contours.

1.1 Definition of Edge Features

An *edge feature* is, loosely speaking, an edge we wish to stroke in a line-drawing of a polyhedral object. This includes contour, valley-crease, ridge-crease, marked, and boundary edges, each of which we describe here.

Figure 2 shows an edge between adjacent faces $A = \langle v_0, v_1, v_2 \rangle$ and $B = \langle v_3, v_1, v_0 \rangle$ with unit face normals

n_A = normalize($[v_1 - v_0] \times [v_2 - v_0]$) and

n_B = normalize($[v_3 - v_0] \times [v_1 - v_0]$).

Te edge connects vertices v_0 and v_1, which have per-vertex surface normals n_0 and n_1 respectively. When exactly one of A and B is a

* e-mail: {morgan, jfh}@cs.brown.edu

[1] We also briefly mention an algorithm for drawing hidden contours.

front face and the other is a back face with respect to the viewer, we call the an edge a *contour edge*[2]. A *ridge-crease-edge* is one where the internal angle between A and B is less than the user-specified threshold angle θ_R. A *valley-crease-edge* is one where the external angle between A and B is less than the user-specified threshold angle θ_V. (Together these last two are called *crease edges.*) A *marked* edge is one that has been selected by a human modeler to be always considered a contour edge, which is useful for creating divisions between differently textured regions of a model or for highlighting details. Finally, some models are not closed—they have edges that have only a single adjacent face, which are called *boundary* edges; following Buchanan and Sousa [Buchanan00] we count these, too, as feature edges.[3]

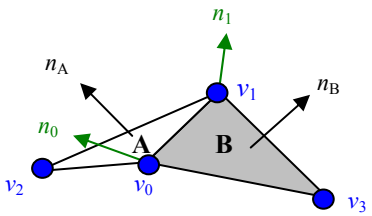

Figure 2: The edge from v_0 to v_1 with adjacent faces A and B.

1.2 Target Architecture

The techniques described in this paper are designed for use on modern programmable graphics hardware implementation, a so-called graphics processing unit (GPU) that is accessed through an API like OpenGL or DirectX. We describe the relevant vertex processing features and limitations of today's GPUs.

The GPU pipeline consists of four sequential units: vertex processor, rasterizer, fragment (pixel) processor, and combiner and a large block (hundreds of megabytes) of dedicated *video memory*. This memory is used to store the frame buffer, textures, and scene geometry. The bus connecting the CPU to the GPU is often too slow [NVIDIA] to transfer the texture and geometry data needed every frame; when it *is* fast enough, demands increase to once again make it too slow. This imposes a major restriction necessary for performance: only a small amount of data can be dynamically updated by the CPU, and a majority must be static and preloaded before interactive rendering begins. This bandwidth limitation is so severe that performance can drop by a factor of 10 to 100 when too much data is dynamic. The goal of moving graphics processing from the CPU to GPU is therefore not just to lessen the CPU load but to avoid the bandwidth limitation.

Geometry is stored in *vertex buffers* and fed to the *vertex processor* as a stream of vertices. The order of vertices in the stream can, at the programmer's choice, either match the order of their appearance in the buffer or be random access dictated by a separate index array. The vertex processor is responsible for transforming vertices from object to homogeneous clip space and computing the quantities to be interpolated across a triangle shading like the diffuse and specular lighting components. The transformation is typically accomplished by multiplying by the "model-view-projection" matrix *MVP*.

The vertex processor has two design characteristics that limit flexibility but allow incredible throughput. First, each vertex must be processed independently. No information can be stored in registers between the handling of one vertex and the next, and the vertex processor is incapable of writing back to the input buffer. Second, vertices can neither be created nor destroyed. We discuss the challenges these limitations present for contour determination at the end of this section.

Each *vertex* structure has many fields known as *attributes* that are 4-tuples. Usually these are used to store 3D position, surface normal, color, and multiple sets of texture coordinates at the vertex but a programmer is free to use them for other purposes such as encoding animation data. Although the CPU cannot modify data per-frame, the vertex processor *can* distort the input during the object-to-homogeneous-space transformation.

We write the attributes of a vertex using tuple notation. A mesh vertex typical of real-time applications that stores 3D position v in attribute 0, 4-component color c in attribute 1, 3D normal n in attribute 2, and two 2D texture coordinates t_0 and t_1 in attributes 3 and 4 is written as $<v, c, n, t_0, t_1>$. The unused w-components of attributes 0, 2, and 3 and the unused z-component of attribute 3 are ignored; they are present when these attributes are loaded into registers but are not explicitly stored in the vertex array. Boolean and small integer values can be stored in the attributes, albeit encoded as floating point numbers.

The vertex processor offers the normal mathematical, branch, and logic operators but is severely limited in terms of memory. It has no stack and is limited to a handful (16 for the current generation) of general purpose 4-component floating point registers. Relative addressing is only available against a small (255) set of constant registers that are written only by the CPU between batches of vertices. These are used to store parameters that are *uniform* across all vertices as well as numeric constants.

The *rasterizer* collects the transformed vertex stream from the vertex processor and converts it to primitives, clips those primitives against the clipping planes, then rasterizes the remaining portions into fragments which are fed to the fragment processor. It is not programmable; the programmer chooses only the desired primitive type (every two vertices is a line, every three is a triangle, or every four is a quad) prior to sending each batch of vertices. For additional performance, the primitives can be connected as *strips*, where only one vertex is needed per line segment or triangle and two for each quad. Note that the rasterizer is the only unit in the pipeline that has a notion of a primitive: the vertex processor sees only a stream of vertices and the fragment processor only a stream of fragments.

The *fragment processor* handles per-fragment texture lookup and shading. It cannot move the location of the fragment being shaded or access adjacent fragments, but can abort then rendering of a fragment. The *combiner* unit performs final alpha blending and writes fragments to the frame buffer.

We designed our contour determination algorithm for the vertex processor because it is the only programmable stage where 3D geometry is available. The limitations of the vertex processor make contour determination challenging for three reasons. First, it is a *vertex* processor but contours are *edge* features. Second, the definition of a contour depends on not only the edge's vertices but on adjacent vertices as well, and the vertex processor cannot access these in the one-vertex-at-a-time model. Third, the number of contour edges is viewer dependent and not known *a priori*, but the vertex processor can neither create geometry to build up a contour list or destroy geometry to cull non-contour edges.

2 Related Work and Other Applications

Packing adjacent face normals into the texture coordinates of a vertex has been previously proposed [Card02; Gooch03]. In such a scheme, each edge is rendered as a degenerate quad that is expanded to non-zero width only if the edge is determined to be a feature in the vertex processor.

[2] Such edges are sometimes called *silhouette* edges in the literature; we reserve that term for edges that lie between the object and the background; such edges are a subset of the contour edges for a closed surface.

[3] The Stanford bunny has several of these on its bottom surface, for example

We extend this idea by storing the four vertices of the two faces adjacent to each edge instead of explicit face normals. This allows us to construct correct face normals under animation. We then add a texture parameterization for artistic strokes and per-face computations for smooth silhouettes and suggestive contours. We then resolve the practical issues that arise by fixing the gaps between quadrilaterals, computing correct face normals for animated models, providing faster rejection using clip planes, addressing coherence, compressing data, and identifying shortcomings of the general technique.

Edge feature rendering methods can be classified into those that construct stroke geometry and those that operate on images. Geometry approaches such as ours construct explicit stroke geometry for contours and then deform or texture that geometry to produce stylized strokes. Previous geometry methods operated in whole or in part on the CPU. Markosian et al. [Markosian97] create strokes from contour edges detected using a randomized algorithm; Gooch et. al [Gooch99] compute silhouettes by treating face normals as points of the sphere, and edges as great arcs between these normal vectors, so that finding silhouettes amounts to intersecting this sphere mesh with a great circle; and Hertzmann and Zorin [Hertzmann00] used a dual-surface approach, in which tangent places to the surface are mapped to points in a dual space, and the contour-detection problem becomes a surface-plane intersection problem in that space, which can be solved quickly with BSP methods. From stroke geometry, a coherent texture parameterization can be developed. The coherent stylized silhouette method by Kalnins et al. [Kalnins03] provides nearly ideal frame-to-frame coherence as well as nicely placed strokes for a still image by explicit pixel flow between frames. Because their method cannot be implemented on the vertex processor we rely on a significantly less sophisticated methods. Our texture parameterizations are lower quality, but can be executed in parallel on multiple vertex units and are therefore amenable to substantially higher performance.

Image methods such as Saito and Takahashi's G-buffer [Saito90] detect depth, color, and curvature discontinuities in a rendered image. ATI engineers implemented this method in a pixel program for a real-time NPR demo shown at SIGGRAPH 2002 [Mitchell02]. Gooch et. al [Gooch99] observe that environment maps can darken the contour edges of a model but the resulting lines have uncontrolled, variable thickness. Dietrich [Dietrich99] refined this idea and used it with dithered two-tone shading to produce cartoon images on the GPU; Everitt [Everitt00] used the isotropy of MIP-maps to achieve a similar effect. Image methods have limited artistic style because there is no explicit stroke geometry to manipulate. However, they are able to detect edge features not explicitly present in the underlying mesh, for example, contours on parametric curves and the valley at the intersection of two polygons. Our method cannot detect these features.

Raskar proposed a hybrid [Raskar01; 02] that constructs a black polygonal halo around every triangle in a mesh (Dietrich also proposed an early version of this using enlarged black backfaces). This halo can be oriented so that it is exposed only along contours and is otherwise concealed inside the mesh (artists have long used a similar trick of a large, inside-out mesh to simulate cartoon outlines on video game characters). Although he describes a CPU solution, it is straightforward to implement his method using only the vertex processor, however the conversion will still rasterize many more polygons than our method. Raskar's method is limited to thin, black lines and cannot render thick or stylized strokes.

Our method for selectively turning edges into extruded quads also has applications for realistic rendering methods. We briefly outline two such applications.

A real-time, realistic fur algorithm [Lengyel01] renders individual hairs by stacking sheets of fur cross-section texture called shells. When shells are stacked nearly perpendicular to the view direction the individual layers can be seen, so Lengyel et al. add quads called *fins* textured with hairs in profile at these locations. They determine fin locations, which are near contour edges, on the CPU. We implement their algorithm entirely on the GPU. The vertex program used for shell rendering displaces each vertex slightly along its normal. For fin rendering, a separate vertex program detects locations where absolute value of the dot product of the view vector and one of the normals along an edge is greater than a threshold. As with our contour rendering algorithm, these edges are culled by moving them beyond the near clipping plane. The other edges are extruded into fins in a manner analogous to our thick line rendering algorithm. For contour rendering we extrude half-quads along the screen space perpendicular; for fur rendering, we extrude along the object space normal. The furry bunny in Figure 1b is rendered with this method.

The shadow volume method of shadow determination was introduced by Crow [Crow77]. He creates volumes bounding all points shadowed by an object by extruding the contour edges of that object away from the light source. Lengyel [Lengyel02] showed how to perform the extrusion in hardware using two copies of each the input mesh vertex distinguished from each other by a w-component of either 0 or 1. His notion of duplicate vertices displaced according to an integer attribute is the inspiration for the attribute i in our edge mesh data structure. Although he extruded the volumes in hardware, Lengyel performed contour edge determination on the CPU. Using the edge mesh we can implement both the edge determination and extrusion steps on the GPU in a single vertex program. This vertex program culls non-contour edges as previously described and creates quads stretching from contour edges to infinity, away from the light source. Again these quads are similar to thick lines, but rather than extruding a finite distance along the perpendicular we extrude an infinite distance along the negative light vector. The shadows in figure 1c were rendered with this method. The other aspects of the shadow volume method are unchanged and we refer the reader to our comprehensive shadow volume paper [McGuire03] for details.

Our shadow method is similar to the one proposed by Brennan [Brennan2003] that uses the face-normal structure of Card et al. Our enhancements to the work of Card et al. extend Brennan's shadows with the benefits of our NPR method: removal of unextruded edges earlier in the pipeline using a clipping plane and correct face normals under animation. To save space and achieve better vertex performance we render light and dark shadow caps from the original and reserve the edge mesh for the sides. This offsets the cost of computing correct face normals which Brennan mentioned in passing, but rejected as too expensive because his algorithm was vertex limited from the degenerate edges on the caps.

Brabec and Seidel [Brabec03] determined contour edges on the pixel processor using vertices encoded as color values and McCool [McCool03] computed shadow volumes from a depth map using an edge filter implemented on the pixel processor. Both require a CPU step to convert pixel values read back from the GPU into vertex values for a subsequent rendering pass. Although future hardware is likely to support this readback operation without CPU intervention, using the output of one rendering pass as the input for the next will always limit performance because shadow volume rendering cannot proceed until contour edge determination is complete.

3 Edge Mesh

Our contour determination technique sends geometry for all edges through the vertex processor and culls non-contour edges by transforming them behind the near clipping plane. This works around the vertex processor's inability to explicitly destroy geometry, and since clipping occurs before rasterization, there is little cost for the edges that are culled. The key idea is to use the vertex-list *not* as a way to transmit information about vertices to the GPU, but rather to transmit information about edges. Thus each entry in the vertex list will really hold information about an *edge* in the model (and indeed, this edge information will be listed in four successive vertices, for reasons that will become apparent). To perform a per-edge contour determination at the vertex level we pack all of the information about an edge into the attributes of a vertex. We call a set of attributes packaged in this manner an *edge vertex* and represent it as $\langle v_0, v_1, v_2, v_3, n_0, n_1, r, i\rangle$ where the first six fields are the 3D vectors from figure 2, r is a random scalar used for texture parameterization and i is an integer between 0 and 3, inclusive, that differentiates the inside, outside, start, and finish ends of a thick edge stroke.

In a pre-processing step, we compute the *edge mesh* from an input mesh of indexed triangles with per-vertex normals.[4] Every undirected edge with index j in the input mesh becomes four consecutive edge vertices with indices $4j$, $4j + 1$, $4j + 2$, and $4j + 3$ in the edge mesh. These four edge vertices are identical except for the i values, which successively have values $\{0, 1, 2, 3\}$. Because the edges are undirected in the input mesh, direction of the edge in the edge mesh is arbitrary, that is, v_0, v_2, n_0 can be swapped with v_1, v_3, n_1 for all four edge vertices and produce an equivalent mesh. We use $v_3 = v_0$ to signify a boundary or marked edge.

3.1 Memory use

Each edge vertex in the edge mesh requires 80 bytes in 32-bit floating point representation and 37 bytes if the vectors are stored with 16-bit integer precision and i and p are packed into a single byte. For comparison, an input mesh vertex with a color, position, and two 2D texture coordinates $\langle v, c, n, t_0, t_1\rangle$ requires 52 bytes per vertex in floating point representation. An input mesh with E edges produces an edge mesh with $4E$ vertices. Because the numbers of vertices, edges, and faces (for an orientable surface) are related to the genus, g, by $V - E + F = 2 - 2g$, and because $3F = 2E$, we get $3V - 3E + 2E = 3(2 - 2g)$, i.e., $E = 3(V - 2 + 2g)$. For typical closed meshes[5] the genus g is small, and we can say that E is about $3V$. Thus on the whole, the edge mesh requires about $3 \cdot 4 \cdot 37 / 52 = 9$ times as much storage as the input mesh.

3.2 Thin Line Contours – a straw-man algorithm

We need a vertex program and index array to render the edge mesh; alone it is just an array of several collocated vertices with unusual texture coordinates.

To render the contour edges of the input mesh as line segments – the simplest possible form of contour-rendering – we can render a line segment between alternate pairs of edge vertices. This is done by rendering an indexed line set using the edge mesh vertices an index array of the form $[0, 1, 4, 5, \ldots, 4E - 2, 4E - 1]$.[6]

[4] The per-vertex normals can be computed using any reasonable weighting of adjacent face normals and need not have unit length. Only the mesh geometry is significant so collocated vertices can be welded together.

[5] For orientable meshes with B boundary components (connected collections of boundary edges), Euler's formula becomes $V - E + F = 2 - 2g - B$; once again, if the number of boundary components is small, the number of edges is approximately three times the number of triangles.

[6] Note that in this algorithm, we ignore edge vertices with indices 2, 3, 6, 7, ... and hence could store only half as much data on the GPU.

Every pair of edge vertices is collocated, so these line segments are initially degenerate. We use a vertex program to displace the vertices to form an edge and cull the edge when it is not a contour. The program is simple: the output is the point $\langle 0, 0, -1, 1\rangle$, which is behind the near clipping plane, if the vertices are on an edge that is not a contour, otherwise the output is the product of matrix MVP and v_0 for edge vertices with $i = 0$ and the product of MVP and v_1 for edge vertices with $i = 1$.

A vertex is on a contour when any of the following expressions are true:

$$\begin{array}{ll} \text{Contour} & [n_A \cdot (eye - v_0) < 0] \text{ XOR } [n_B \cdot (eye - v_0) < 0] \\ \text{Ridge} & [n_A \cdot n_B < -\cos\theta_R] \text{ AND } [(v_3 - v_2) \cdot n_A \le 0] \\ \text{Valley} & [n_A \cdot n_B < -\cos\theta_V] \text{ AND } [(v_3 - v_2) \cdot n_A > 0] \\ \text{Marked} & v_3 = v_0 \\ \text{Boundary} & v_3 = v_0 \end{array} \qquad (1)$$

In Equation 1, *eye* is the eye-point—the object space position of the viewer, or center of projection—θ_R and θ_V are the ridge and valley thresholds, and n_A and n_B are the unit face normals. The eye-point and cosines of threshold angles are uniform across all vertices and can be computed once per object per frame. The face normals are recomputed in the vertex program from the face vertices for every vertex. Marked and boundary edges are recognized by the $v_3 = v_0$ convention which must be enforced when the edge mesh is created.

Although OpenGL supports line rasterization for thick lines, it does not provide line joining. A gap appears at the corner where two thick lines meet, as shown in figure 3. DirectX does not support thick line rasterization at all. This first method is therefore only suited for thin edges with a thickness of one or two pixels.

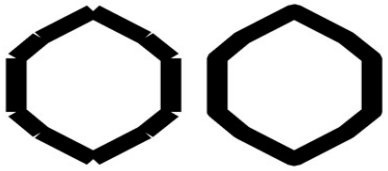

Figure 3: Thick lines leave a triangular gap where they meet (left). We introduce end caps to fill the gap (right).

3.3 Thick Contours

We render thick edges in three passes. The first pass extrudes each edge along its perpendicular to form a quad. The second and third passes create *start* and *finish caps* at the ends of each edge. When two edges meet at a vertex, as depicted in figure 4, their caps fit together to fill the gap shown in figure 3 (right), creating a smooth join.

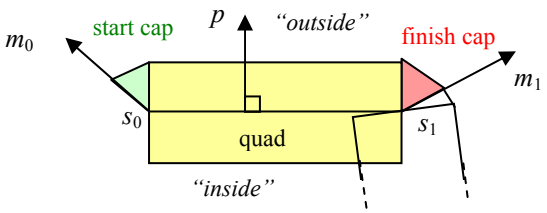

Figure 4: The edge between screen-space points s_0 and s_1 extruded into a thick quad with triangular end caps between the quad and vertex normals.

The first pass uses an index array containing the integers from 0 to $4E$, in order.[7] It uses a vertex program that displaces each of the

[7] As a shortcut, the OpenGL function *glDrawArrays* processes each vertex in an array in order, without an index array of sequential integers.

four edge vertices for an edge to a different corner of the quad. As before, if the edge is not a contour, all vertices are transformed to <0, 0, –1, 1> where they are culled by the near plane. For each edge vertex, let $s_0 = (MVP * v_0)_{xy}$ and $s_1 = (MVP * v_1)_{xy}$ be the *xy* parts of the screen space projections of v_0 and v_1. The unit length screen space perpendicular to the edge is p = normalize(<$s_{0y} - s_{1y}$, $s_{1x} - s_{0x}$>), the unit length screen space projection of the vertex normal is m_0 = normalize($MVP * [v_0 + n_0]_{xy} - s_0$), and the transformed position depends on i as follows:

$$pos_{\text{quad vertex}} = \begin{cases} s_0 - p & i = 0 \\ s_1 - p & i = 1 \\ s_1 + p & i = 2 \\ s_0 + p & i = 3 \end{cases} \quad (2)$$

This produces a degenerate quad that is culled for each non-contour edge and a non-degenerate 2-unit thick quad for each contour edge. The thickness can be adjusted by scaling p. Because it is a screen space vector, a scale factor of k divided by screen resolution produces a k-pixel thick line.

It is sometimes desirable to only draw half the quad, for example, to eliminate overdraw on the interior of an object when rendering a silhouette. Equation 3 gives coordinates for a half quad that is on the outside (side of the projected normal) of the edge.

$$pos_{\text{half quad}} = \begin{cases} s_0 & i = 0 \\ s_1 & i = 1 \\ s_1 + p\,\text{sign}(m_0 \bullet p) & i = 2 \\ s_0 + p\,\text{sign}(m_0 \bullet p) & i = 3 \end{cases} \quad (3)$$

The *sign* function returns –1 for a negative argument and +1 for a positive argument.

The second pass uses an index array containing triples of vertices of the form [0, 1, 2, 4, 5, 6, …, $4E - 4$, $4E - 3$, $4E - 2$]. These form the start-cap triangles. Equation 3 gives the transformed vertex position as a function of j, which is equal to i when m_0 and p are in the same direction (i.e., forward traversal yields a front facing cap) and $(2 - i)$ when backward traversal of the vertices is required to create a front face.

$$pos_{\text{start vertex}} = \begin{cases} s_0 & j = 0 \\ s_0 + p\,\text{sign}(m_0 \bullet p) & j = 1 \\ s_0 + m_0 & j = 2 \end{cases} \quad (4)$$

As before, the sign(m_0 • p) term ensures that the second edge vertex in each triangle is on the outside of the stroke. The third pass produces the finish-caps using the same index array and an equivalent transformation at the finish vertex:

$$pos_{\text{finish vertex}} = \begin{cases} s_1 & j = 0 \\ s_1 + p\,\text{sign}(m_1 \bullet p) & j = 1 \\ s_1 + m_1 & j = 2 \end{cases} \quad (5)$$

where m_1 = normalize($MVP * [v_1 + n_1]_{xy} - s_1$). Stroke thickness is controlled by both scaling m and p. As with the thin line and quad programs, both end cap vertex programs transform vertices that are not on contour edges to the point <0, 0, –1, 1>.

The quad and two end caps fit together with adjacent lines to form a solid thick line without gaps, as shown in the right half of figure 4. The gap between quads appears on the convex side of the projected curve. On the concave side the two quads overlap. That overlap is an incorrect rendering but looks good; it is both hard to see (even with a structured texture) and is often hidden inside the model anyway. We'd like to insert triangles connecting each quad to its neighbor to close the gap on the convex side. Because edges do not have information about neighbor edges, we use the only mutual information available: the normal at the shared vertex. Projected into screen space, this provides a common point to which both edges can connect triangle joins. We use *two* triangles because there is no way, given the limited information available in an edge vertex, to fill the gap between two thick lines with a single triangle.

This method of rendering line caps assumes that the projected vertex normal lies between the ends of the quads for adjacent lines and that it points towards the outside of the curve being stroked. Figure 5 shows a case where this assumption does not hold and gaps appear. When the end of a cylinder is viewed from a particular angle under perspective projection the vertex normals along the ridge point into the curve instead of out of it. The underlying cause is that vertex normals poorly capture the sharp curvature at this location; a beveled corner would not exhibit the same problem. The method also fails in the more rare case where the per-vertex normal is degenerate under projection. These problems do sometimes occur for creases and boundary edges; vertex normals, however, seem to reliably point the outside of a mesh at the contour edges, making them relatively immune regardless of curvature (see Appendix B.)

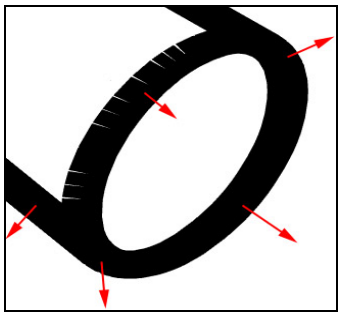

Figure 5: The line capping method fails along the ridge line of this cylinder where the projected vertex normals are a poor indication of its curvature.

3.4 Rendering Silhouettes

For a closed manifold (mesh without boundary edges), the *silhouette* between the rendered mesh and the background is a subset of the contour edges. The silhouette cannot be geometrically distinguished on the vertex processor. We therefore use the traditional approach: render a write mask to the stencil [McGuire02] or depth [Rossignack92] buffer and suppress the internal contour edges that are not on the silhouette with a per-pixel test against that mask. To avoid the expense of clearing the stencil buffer between each mesh, we extend the stencil buffer method with an incrementing test value as follows.

Before rendering the frame, initialize a variable S to 0 and clear the stencil buffer to 255. For each model:

1. Set S := (S + 1) mod 255
2. If S == 0 then clear stencil buffer to 255
3. Render the model, setting stencil to S wherever the depth test passes
4. Set the stencil test to pass where stencil == S
5. Render contour edges

For a scene with many objects, this reduces the number of times the stencil buffer must be cleared by a factor of 254. The

thin contours on the robot in figure 7c are rendered as previously described; thick boundary strokes on the silhouette were then added in a separate rendering pass with this method.

3.5 Hidden Contours

With this algorithm, we can also generate renderings in which hidden contours are rendered as dashed lines. This involves three passes. We first draw all models in the scene with a depth offset using the OpenGL *glPolygonOffset* command. Then for each hidden-contour model, we render solid contours. We then change the rendering style to "dashed," invert the sense of the depth test, and re-render the edges; only those that are occluded will appear dashed. This is essentially an easy application of Appel's "quantitative invisibility" idea [Appel79]. Figure 1a shows a mechanical part rendered with this hidden line style.

3.6 Animating the Mesh

So far we have discussed a static mesh with a variable model view transformation. The edge mesh may be animated in the same way that an ordinary input mesh is animated by transforming each of the four vertices and two normals encoded in an edge vertex. Input meshes are typically animated through keyframe and skeletal animation; we briefly describe the extension of these ideas to edge-meshes.

In hardware, the vertex stream can be composed from separate attribute streams as they are fed to the vertex processor. A common way to perform interpolated keyframe animation on the input meshes is to store the vertex positions for each frame in separate buffers and to specify the input stream as $<v, v', \ldots>$ where v is the previous frame, v' is the next frame, and the attributes not relevant to animation are not shown. A uniform parameter α controls the interpolation between these through a simple vertex program of the form $pos_{frame} = MVP*[v + \alpha (v - v')]$.

To extend this design to the edge mesh, we create a set of vertex positions and normals for each keyframe and send streams for the previous and next frame. The vertex processor now sees edge vertices of the form $<v_0, v_1, v_2, v_3, n_0, n_1, v_0', v_1', v_2', v_3', n_0', n_1', r, i>$, and interpolates corresponding data just as above.

Skeletal animation requires less data and often produces more desirable results. Fernando and Kilgard [Fernando03] describe how skeletal animation with four bone influences per vertex is performed on hardware using vertices of the form $<v, \ldots, M, \beta>$ where M is a vector of four matrices and β is a vector of corresponding blending weights. To implement skeletal animation of an edge mesh, we extend the edge vertex with four M and β values, one for each vertex. The vertex processor now sees edge vertices of the form $<v_0, v_1, v_2, v_3, n_0, n_1, M_0, M_1, M_2, M_3, B_0, B_1, B_2, B_3, r, i>$, where M_0 and B_0 are the parameters for v_0 and n_0, v_0, M_1 and B_1 are for v_1 and n_1, M_2 is for v_2, and M_3 is for v_3. Although each vertex is large, it is within the 16-attribute limit of current hardware.

4 Texture Parameterization

We map textures like those in figure 6 onto the thick lines to produce stylized strokes. These stroke textures are designed so that the horizontal texture coordinate varies from 0 to 1 along the length of the stroke from start to finish and the vertical texture coordinate varies from 0 to 1 from the outside to the inside of the stroke. The "inside" of a stroke is the side that should lie against the body of an object when stroking the silhouette and the "outside" is the side that should lie against the background. These textures tile in the horizontal direction and are clamped in the vertical direction.

We require a parameterization on the mesh that maps vertices to texture coordinates to place these textures along the strokes. A good parameterization minimizes texture distortion and provides continuous coordinates in both space and time (frame coherence). Previous NPR stroke methods [Markosian97; Kalnins03] were able to satisfy these criteria by combining adjacent edges into a single stroke and examining the previous frame because they operated on the CPU. These sorts of data are not available on the GPU vertex processor, so the available parameterizations are significantly limited.

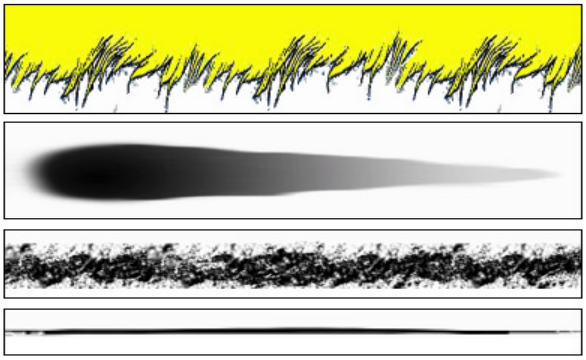

Figure 6: Brush stroke textures used to create figures 1 and 8. From top to bottom: cartoon fur scanned from *The Lorax* by Dr. Seuss, hand drawn watercolor stroke, charcoal from the Adobe Photoshop brush palette, hand drawn pen stroke.

We propose two parameterizations, one in object space and one in screen space, that are both inexpensive to compute and can execute on the vertex processor. Neither of our parameterizations is ideal. As a compromise to the lack of data available, they trade space continuity against time continuity. Under both, the vertical texture coordinate is always 0 on the outside of the stroke and 1 on the inside, so only the horizontal texture coordinates at the start and end of the stroke, u_0 and u_1, are of interest.

Let s_0, s_1, and c be the screen space projections of object space vertices v_0, v_1, and the object space origin <0, 0, 0>. Recall that

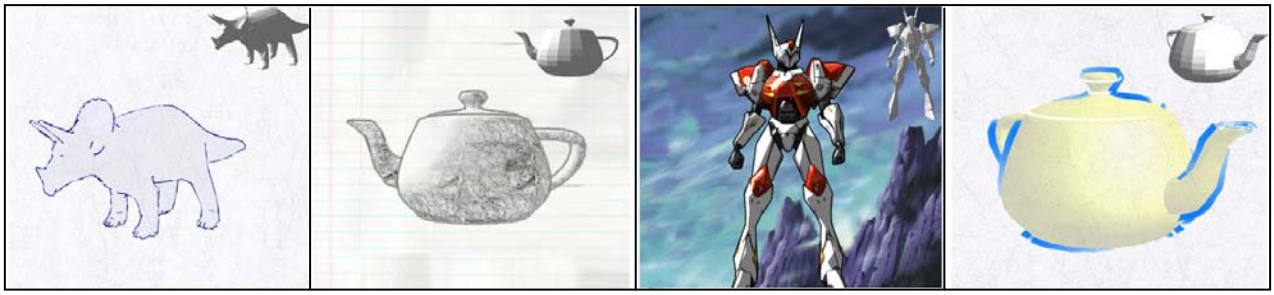

Figure 7: Rendering styles created by varying the contour stroke and mesh fill texture: pen and ink, charcoal, anime, and watercolor.

every group of four edge vertices is assigned a random scalar, r. We now use that value as a unique parameter for each edge. The *object space* parameterization assigns $u_0 = r$ at the start of a stroke and $u_1 = r + w|s_1 - s_0|$ at the end of a stroke, where w is the line width times sign($m_0 \bullet p$), which is used here to maintain the aspect ratio of the texture and flip the texture direction as needed so that strokes wind in consistently around the silhouette. The strokes rendered with this parameterization have constant tiling frequency in screen space. Spatial discontinuities occur at the ends of edges, so this parameterization is only appropriate for meshes with large screen space edges like the robot in 7c or the base of the teapot in figure 7d, or for stroke textures with little structure where texture discontinuities are unlikely to be noticed, like the charcoal and pen strokes in figure 6. This parameterization provides excellent frame coherence under animation, deformation, and translation. Under rotation, marked, boundary, and crease edges have coherent parameterizations between frames but contour edges may experience discontinuities when the model rotates far enough that one contour edge is replaced with another near-by in screen space.

An alternative *screen space* parameterization assigns $u = w(s_y - c_y)$ for both ends of a mostly vertical edge and assigns $u = w(s_x - c_x)$ for both ends otherwise. A mostly vertical edge is one where $|s_{1y} - s_{0x}| > |s_{1x} - s_{0x}|$. This parameterization has spatial discontinuities only where horizontal and vertical edges meet. It is completely independent of the tessellation of the mesh and produces good still images—the cartoon fur in figure 1d was rendered with this parameterization (the interior lines are valleys stroked with our "pen" brush to give the bunny some detail). The bunny has such high tessellation that the object space parameterization yields only noise, as shown in figure 8. Because it uses coordinates relative to the screen space projection of the object space origin, the screen space parameterization produces frame coherent results under translation perpendicular to the view vector. Most objects exhibit no frame coherence under deformation, animation, scale changes, and large rotations.

We find that contour edges are best rendered with half quad strokes where the u values map directly along the inside of a stroke and are stretched across both the caps and quad on the outside. Figure 9 demonstrates this mapping, where the on outside of the stroke u_0 and u_1 are the exterior cap texture coordinates and the quad texture coordinates are resolved by linear interpolation.

For other contours, we use the u values for both the quad corners and the caps, causing a single column of texture to stretch across the caps. This distortion is undesirable, but the alternative is to draw the quad in two pieces since no interpolated texture distortion on a single quad could texture the inside half of the quad from the top half of the texture and still make the quad's texture line up with the caps.

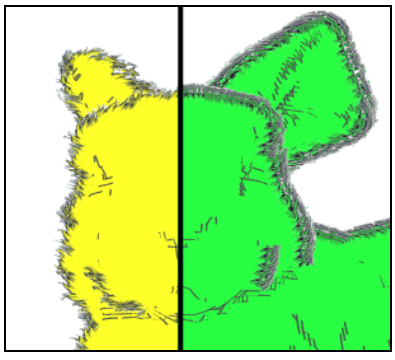

Figure 8: Screen (left) and object (right) space texture parameterization for the bunny. The object space method degenerates to noise for highly tessellated models like this.

5 Performance

Our complete algorithm with textures incurs about a factor-of-30 cost over rendering the untextured, underlying mesh. There are two main reasons: the vertex program is long and our implementation lacks significant optimizations.

Vertex programs run in time roughly linear in the number of instructions, but with a large performance drop at around 20 instructions, presumably due to the architecture of today's GPUs. Our vertex program has about 200 instructions for the quad and an additional 200 for each cap. In comparison, the fixed function pipeline requires the equivalent of about 5 instructions.

Our vertex program has not been optimized. For flexibility, it uses many slow IF statements to enable optional rendering of marked edges, creases, etc., and the program is then compiled with a pre-release compiler and run on a vertex processor that is only in its second generation.

Our goal was to move processing 100% onto the GPU by framing the (serial) CPU contour determination as a (parallel) per-vertex problem. At this point parallelism can be brought to bear on the problem by increasing the number of vertex processors, which work in parallel. As vertex processors and their compilers improve and the number of vertex processors increases, techniques like the ones we have discussed will become commonplace and dramatic performance gains will be available.

That said, limited applications of our methods are suitable for use on today's processors, even for games, with significant hand optimization. Reducing the thin-line algorithm to nearly the bare minimum for rendering contour edges (29 instructions), we can render a 163,000 (visible) polygon scene (more geometry than is visible in most games) and its contour edges at 25 fps with 4x antialiasing on an NVIDIA GeForceFX 5900 Ultra card. As another datum, we can render 100,000 polygons at a rate of 5 fps for mesh and quad contour strokes, 36 fps for mesh and contour lines, and 250 fps for the mesh alone.

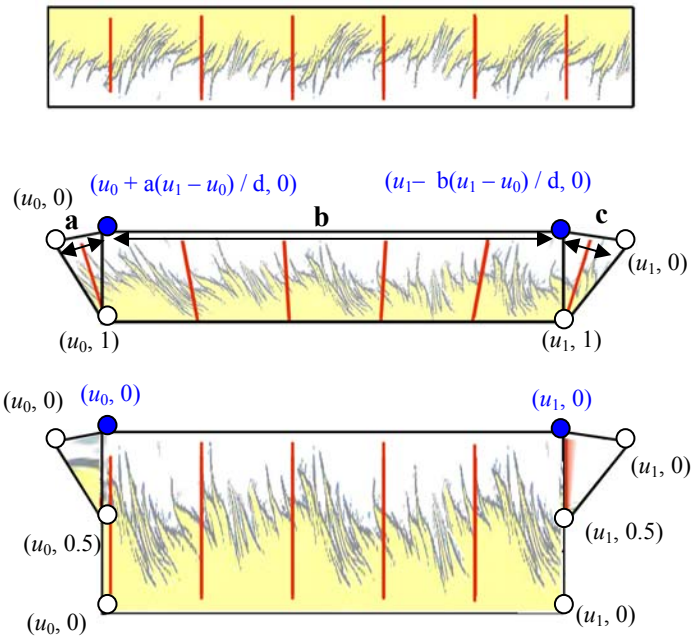

Figure 9: Mapping a stroke texture (top) to a half-quad stroke (center) and full-quad stroke (bottom). a = length of start cap, b = length of quad, c = length of finish cap, d = a + b + c. Other labels are texture coordinates at vertices.

dator

6 Discussion and Future Work

Because we create thick lines in screen space, we assume that both ends of the line can be projected to finite screen coordinates. This assumption does not hold for an edge that crosses the plane $z = 0$. Although we have not done so, presumably our method can be extended to support edges that cross this plane by clipping them at the near clip plane in the vertex program.

We have moved edge feature determination and texturing from the CPU to the GPU. This unlocks the potential for higher performance through parallel execution but comes at a reduction in quality compared to previous methods. For stroke textures with significant structure (e.g. the Dr. Seuss texture), our screen-space parameterization produces reasonable results for still images but is barely adequate for animation. The object space parameterization works very well for meshes with large edges in screen space but produces noise for highly tessellated meshes. Our end-cap rendering method for thick lines fails to render creases correctly on models with sharp corners. The straightforward solution—replacing the triangle end caps with half-disks— works well for solid-color, opaque lines, albeit at the expense of several extra rendering passes to create all of the geometry. That solution is inappropriate for textured lines, which require a unique parameterization across the end caps.

Our methods can provide high performance and attractive results for rendering relatively thin or noisy lines (figures 1a, 7ab), fur (figure 1b), and shadows (figure 1c) and are well suited for interactive applications like games and CAD. More abstract rendering styles with thick strokes like cartoon fur (figure 1d) and watercolor (figure 7d) also have high performance but require hand tuning for each model to achieve attractive results. Improving these is an interesting area for future work.

We expect the current restrictions on GPUs to be relaxed over time, leading to improved edge mesh implementations. By 2005, there should be graphics cards available that will be able to perform a memory lookup (texture reference) from the vertex processor. This will allow a more space-efficient edge mesh because each vertex and normal can be stored as one integer index instead of three floating point numbers. The DirectX Next [Microsoft03] specification, with hardware anticipated in 2006, will allow vertices to be created and destroyed on the GPU and for the vertex processor to write to memory. This will eliminate the need for culling with clipping planes, and likely enable more sophisticated frame coherence for stroke textures and hysteresis for suggestive contours. Of course at that point the GPU will be very close to a general purpose processor and the history of the "wheel of reincarnation" suggests that it will be brought closer and closer to the main processor, whereupon a new "close to the display" graphics processor will be developed and limiting the bandwidth to *that* processor will again be relevant.

Acknowledgements

John thanks the Evasion group at INRIA Rhône-Alpes, where he was on sabbatical during this project. Morgan is supported by an NVIDIA Ph.D. fellowship. Simon Green provided support for the pre-release Cg compiler and NVIDIA donated the GeForceFX 5900 Ultra graphics cards on which the images were rendered. Tomer Moscovich implemented radial curvature for suggestive contours and Tomer, Sarah, Nick, and Hari helped prepare this paper. Both authors thank the anonymous reviewers who not only identified important related work but kindly described the advantages of our approach for us; many of their words appear verbatim in section 2.

References

APPEL, A., ROHLF, F., AND STEIN, A., The Haloed Line Effect for Hidden Line Elimination. *Computer Graphics (Proc. SIGGRAPH '79)*, pp. 151-157, 1979.

BRENNAN, C. Shadow Volume Extrusion Using a Vertex Shader. in *ShaderX: Vertex and Pixel Shaders Tips and Tricks,* Wolfgang Engel editor, Wordware, May 2002.

BUCHANAN, J. W. AND SOUSA, M. C. The edge buffer: A data structure for easy silhouette rendering. In *Proc. of NPAR 2000*, pp. 39-42, 2000.

BRABEC, S. AND SEIDEL, H. Shadow Volumes on Programmable Graphics Hardware. *Proceedings of Eurographics 2003*

DREW CARD AND JASON L. MITCHELL, Non-Photorealistic Rendering with Pixel and Vertex Shaders. in *ShaderX: Vertex and Pixel Shaders Tips and Tricks*, Wolfgang Engel editor, Wordware, May 2002.

CROW, F. C. Shadow Algorithms for Computer Graphics. *Computer Graphics (SIGGRAPH '77 Proceedings)*, 11(2): 242-248, July 1977.

DECARLO, D., FINKELSTEIN, A., RUSINKIEWICZ, S., AND SANTELLA, A. Suggestive Contours for Conveying Shape. *ACM Transactions on Graphics*. 22(3):848-855, July 2003.

DIETRICH, S. Cartoon Rendering and Advanced Texture Features of the GeForce 256 Texture Matrix, Projective Textures, Cube Maps, Texture Coordinate Generation and DOTPRODUCT3 Texture Blending. NVIDIA Corporation, Austin, TX. http://developer.nvidia.com/object/Cartoon_Rendering_GeForce_256.html

EVERITT, C. One-Pass Silhouette Rendering with GeForce and GeForce2. NVIDIA Corporation, Austin, TX. http://developer.nvidia.com/object/1Pass_Silhouette_Rendering%20.html

FERNANDO, R. AND KILGARD, M. J. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*, Addison-Wesley, ISBN 0321194969, March 2003.

GOOCH, B., SLOAN, P.J., GOOCH, A., SHIRLEY, P. AND RIESENFELD, R. Interactive technical illustration. *In Proc. of 1999 ACM Symposium n Interactive 3D Graphics*, April 1999.

GOOCH, B. *in Theory and Practice of Non-Photorealistic Graphics: Algorithms, Methods, and Production Systems,* course organized by M. C. Sousa. SIGGRAPH 2003 Course notes 10.

HERTZMANN, A. AND ZORIN, D. Illustrating smooth surfaces. *SIGGRAPH 2000 Conference Proceedings*. New Orleans, Louisiana. pp. 517-526. July 23-28, 2000.

KALNINS, R. D., DAVIDSON, P. L., MARKOSIAN, L., FINKELSTEIN, A. Coherent Stylized Silhouettes. *Proc. of SIGGRAPH 2003*. pp. 856-861, 2003.

LENGYEL, J. E., PRAUN, E., FINKELSTEIN, A., AND HOPPE, H. Real-time fur over arbitrary surfaces. In *ACM Symposium on Interactive 3D Graphics*, March 2001.

LENGYEL,[8] E.. The Mechanics of Robust Stencil Shadows. Gamasutra, October 11, 2002. http://www.gamasutra.com/features/20021011/lengyel_01.htm.

MARKOSIAN, L., KOWALSKI, M., TRYCHIN, S., AND HUGHES, J. F. Real-Time Non-Photorealistic Rendering. In *SIGGRAPH 97 Conference Proceedings*, August 1997.

MCCOOL, M. D., Shadow volume reconstruction from depth maps. *ACM Transactions on Graphics*, 19(1):1–26, January 2000.

MCGUIRE, M. Object outlining. flipcode, 2002 http://www.flipcode.com/articles/article_objectoutline.shtml

MCGUIRE, M., HUGHES, J. F., EGAN, K., KILGARD, M. J., AND EVERITT, C. Fast, Practical and Robust Shadows. Brown Univ. Tech. Report. October 27, 2003.

MICROSOFT. DirectX Preview. Slides from Meltdown 2003 Developer Conference, Seattle WA. http://www.microsoft.com/downloads/details.aspx?FamilyId=3319E8DA-6438-4F05-8B3D-B51083DC25E6&displaylang=en

MITCHELL, J., BRENNAN, C., CARD, D. Real-Time Image-Space Outlining for Non-Photorealistic Rendering. SIGGRAPH 2002 Sketch. http://www.ati.com/developer/SIGGRAPH02/SIGGRAPH2002_Sketch-Mitchell.pdf

NVIDIA, NVIDIA nForce IGP TwinBank Memory Architecture, Technical Brief, undated

RASKAR, R. AND COHEN, M.. Image precision silhouette edges. In *Proc. of 1999 ACM Symposium on Interactive 3D Graphics*, April 1999.

RASKAR, R. Hardware support for non-photorealistic rendering. *Proc. of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pp. 41-46, 2001

ROSSIGNACK J. AND VAN EMMERIK, M. Hidden contours on a framebuffer. In *Proc. of SIGGRAPH/Eurographics Workshop on Graphics Hardware*. 1992.

SAITO, T. AND TAKAHASHI, T. Comprehensible rendering of 3d shapes. *Proc. of SIGGRAPH '90,* pp. 197-206. 1990.

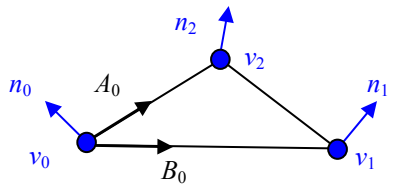

Figure 10: Suggestive-contour triangle-vertex: the locations and normals at each vertex are stored, along with the derivative of the normal at each vertex, expressed in the A_0B_0-coordinate system at v_0 and in similar coordinates at each of the other vertices.

Appendix A: Smooth Silhouettes and Suggestive Contours

With a small modification the methods described in this paper we have also generated "smooth" contour edges in the style Hertzmann and Zorin [Hertzmann00]. These are piecewise linear curves that exhibit less pronounced angles than the actual contour edges for the silhouette of the object, making it appear more smooth.

Instead of "edge-vertices" we create *triangle-vertices* that store $<v_0, n_0, v_1, n_1, v_2, n_2, r, i>$, the position and vertex-normal at each vertex of the triangle, the scalar used for parameterization, and the 0..3 index to distinguish the otherwise identical four vertices created for each triangle. From this data one can compute, for each edge of the triangle, the location p (if any) of a zero of the function $f(p) = (p - eye) \bullet n_p$, where n_p is interpolated from the normals at the ends of the edge. If this function has zeros on two edges at points p_A and p_B, we treat the line segment between those points as a contour and render it as previously described, with p_A, p_B, n_A, and n_B replacing v_0, v_1, n_0, n_1 in the edge vertex equations.[9]

Similarly, it may be possible[10] to approximately[11] implement the suggestive contours of DeCarlo et al. [DeCarlo03] in hardware; we have done so in a software simulator to produce the results shown in Figure 11. (We did not do so in hardware because the information required per vertex, for a simple implementation, slightly exceeded the current hardware's capacity, and compressing it is merely an exercise in space-hacking.) Our software implementation works like this: Again for each triangle one generates a triangle-vertex. But the information at each triangle-vertex is more complex: we send three sets of information as before, one per vertex, but for each vertex we include v_i, the vertex position, n_i, the vertex normal, *and* $d_{i,1}$, $k_{i,1}$, $d_{i,2}$, and $k_{i,2}$, where $d_{i,1}$ is the direction of greater principle curvature, and $k_{i,1}$ is the larger principle curvature, and $d_{i,2}$ and $k_{i,2}$ are the direction and value of the lesser principle curvature. From this information, we can estimate the radial curvature at each vertex as follows (dropping all i subscripts):

First, let e = eyePoint – v, and w = normalize($e - n\,(e \bullet n)$), i.e., w is the projected "look" vector at the vertex. Then we compute the radial curvature at the vertex as

$$k_{r,} = (d_1 \bullet w)^2 k_1 + (d_2 \bullet w)^2 k_2$$

With these radial curvature values at each vertex, we proceed with the smooth-silhouette algorithm, using the radial curvatures rather than $n \bullet e$. If all three have the same sign, we cull the segment by sending it behind the clipping plane. Otherwise, we determine the two edges, AB and AC, along which k_r has zeroes.

Along these two edges, we estimate the gradient of the radial curvature via

$\nabla k_{AB} = (k_B - k_A)\,(B - A) / \|B - A\|^2$
$\nabla k_{AC} = (k_C - k_A)\,(C - A) / \|C - A\|^2$
W = normalize($E - N\,(N \bullet E)$),

where N is the face normal and E is a vector from the eye to either of the two vertices (since we're assuming that all triangles are small).

If either ($\nabla k_{AB} \bullet W < 0$) or ($\nabla k_{AC} \bullet W < 0$), then the zero-contour runs through a valley, and we reject it. Otherwise, we compute and draw the zero-contour as in the smooth-silhouette case.

[8] Eric Lengyel and Jed E. Lengyel really are different people, despite sometimes working on related subjects.

[9] We ignore the zero-probability event where f is zero on the entire triangle, although one could generate output in that case.

[10] We have not yet done so; the implementation of the main algorithm was delayed until shortly before the submission deadline by Cg 1.0 compiler problems, which the NVIDIA compiler team fixed for us in Cg 2.0.

[11] The approximation involves assuming that the suggestive-contour-detection function, $D_w(n \bullet v)$ in the notation of DeCarlo et al., is approximately linear over each polygon; for models with large facets this is a bad assumption, and the method will produce meaningless results.

Unfortunately, we cannot implement the hysteresis that allows more forgiving thresholds for edges adjacent to suggestive contours, which means that our suggestive contours end up fragmented; the fragmentation is sufficiently annoying that for the time being, this method of computing suggestive contours should be regarded as a curiosity rather than a practical method.

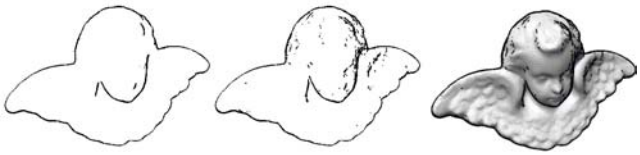

Figure 11. Left: Contours. Center: Suggestive Contours added using our method. Right: Shading added to show the mesh.

Appendix B: Projections of normal vectors.

At the end of section 3.3, we described a method for estimating a normal vector to the 2D projection of a feature curve at one of its vertices, namely, we project the (surface) normal at the corresponding vertex in 3-space, but noted that this occasionally failed. We can analyze this failure geometrically. Let us examine a feature edge *e* that's adjacent to a feature edge *f,* meeting at a vertex *v.* The unit vector **e** points along edge *e* towards *v,* and the unit vector **f** points along *f* but away from *v.* The unit outward surface normal at *v* will be denoted **s,** **d** will be a unit vector in the eye-to-*v* direction, and **b** will denote **e** × **f.**

First, if *e* and *f* are collinear, then so are their projections, and hence the adjacent quads already meet perfectly, and the "cap" is redundant.

When *e* and *f* are not collinear, there is a unique plane containing both. If the surface normal **s** lies in this plane, on the "convex side" of the bend, then the end-caps will join properly (see figure 12a). If it lies on the concave side of the bend, they will not; note that this means that even in this ideal case, when the space-normal to a feature curve is ill-aligned with the surface normal (along the ridge-line of a mountain range, for instance), there will be problems (figure 12b). Fortunately the problems generally occur on the "below the surface" part of the thickened curve, and hence are generally invisible, unless the projected surface normal points to the wrong side of the front-facing polygon that would normally obscure the error (as in figure 5).

Let's analyze this a little more carefully.

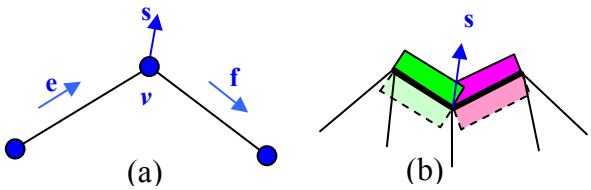

Figure 22(a) The surface normal **s** lies in the plane determined by the two feature edges, on the convex side; "capping" will work in this case. (b) The two darkened "ridge-line" edges form a feature whose image-plane normal points down, while the projection of the surface normal points up; capping will fail, but the failure will be hidden by the surface (as shown by the dotted quads).

In Figure 13, if the lower half of the thickened line were below the model surface, the problem on the left would be hidden, as often happens in practice. In particular, for contour lines the view direction is tangent to the surface in the smooth case, and lies in the tangent cone in the polyhedral case. That means that the projected normal is nearly the normal (smooth case) or nearly lies in the normal cone (polyhedral case). Its opposite is therefore very likely to lie *inside* the surface. For other feature curves, no such promise can be made: the projected normal, from a near-overhead view, may be very nearly tangent, and its opposite may well be visible. The probability of this increases when the vertex is very non-planar, but is also large when the vertex is planar, but the surface "normal vector" is far from the normal to this plane. Thus one should expect surface features on relatively smooth areas with "good" normals to show relatively few cracks; on other areas, the probability of cracks is larger. Unfortunately, for crease lines cracking can be relatively likely.

To continue the analysis, cracks can only appear when the projected normal lies on the "convex side" of the angle formed by the projection of two adjacent segments. For a random direction of projection, **d**, how likely is this? We'll assume that the random

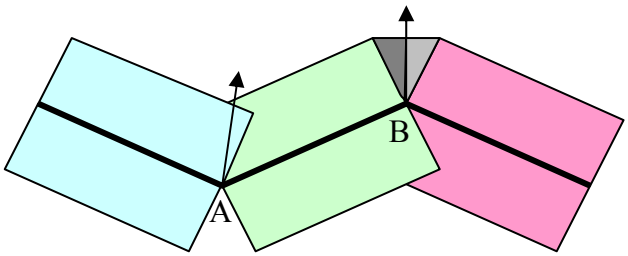

Figure 13. At vertex *A*, there's a cracking artifact because the projected feature edges curve towards the projected normal; at *B* there is not, because the edges curve away from the projected normal and the grey "caps" fill in the gap appropriately.