Real-time Curvature Estimation on Deformable Models

Wesley Griffin[*]
University of Maryland, Baltimore County

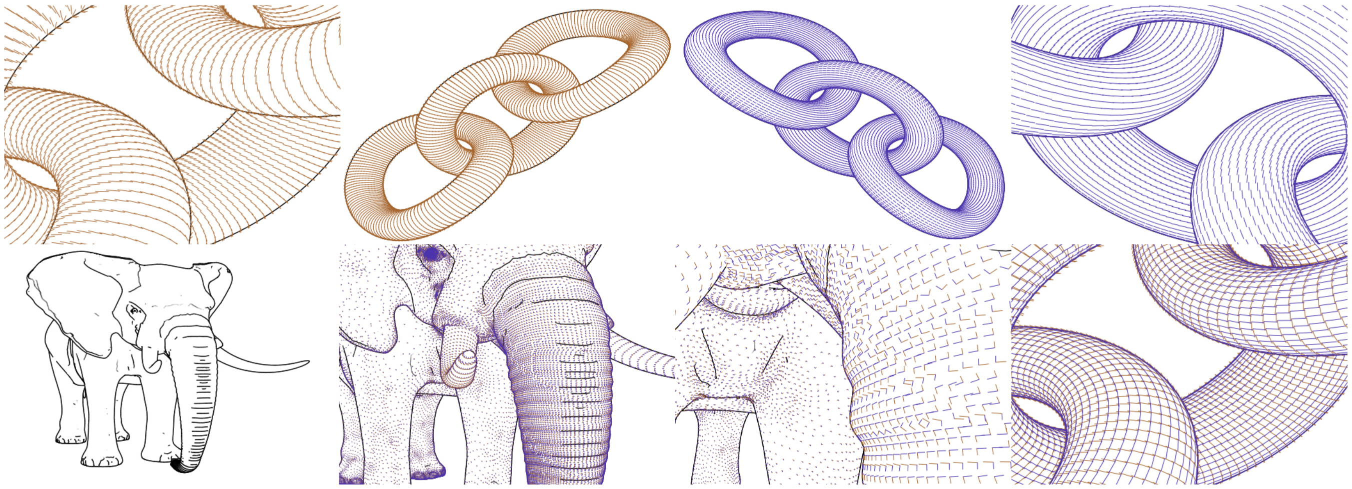
Figure 1: *Examples of estimated curvature on models. Curvature is re-computed each frame. The orange lines are the principal direction of maximum principal curvature and the blue lines are the principal direction of minimum principal direction.*

Abstract

Surface curvature and its derivative are used in a number of applications including non-photorealistic line drawing techniques. CPU algorithms for estimating curvature are a reasonable choice for static models, but curvature estimates for deformable models must be recomputed every frame. For many deformable models, for example those using vertex blending, the deformed model is only ever present in the GPU. We introduce a GPU algorithm for estimating curvature on any model using the programmable geometry shader.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Line and curve generation;

Keywords: NPR, line drawing, GPU, geometry shader, real-time rendering

1 Introduction

Surface curvature and its derivative are used in a number of applications including non-photorealistic line drawing techniques. Suggestive contours [DeCarlo et al. 2003] and apparent ridges [Judd et al. 2007] use curvature to extract contours. These techniques are object-based techniques. They work directly on the polygonal mesh representation of objects. There is another set of line drawing techniques which are image-based. These techniques render objects to a framebuffer and then process the image to draw lines.

Rusinkiewicz [2004] estimates principal curvatures and principal directions at each vertex using averages of normals over each face. We discuss this algorithm in more detail in 3. This CPU algorithm for estimating curvature is a reasonable choice for static models, but curvature estimates for deformable models must be recomputed every frame. For many deformable models, for example those using vertex blending, the deformed model is only every present in the GPU.

We introduce a GPU algorithm for estimating curvature on any model using the programmable geometry shader [Blythe 2006]. Much like Rusinkiewicz, we estimate curvature at each vertex using the averages of normals over each face. We describe our algorithm in detail in Section 4.1.

For our initial demonstration, we implemented two simple deformation primitives, tapering and bending [Barr 1984]. We also extract occluding contours from models and draw those as lines. The specific contributions of this paper are:

- A multi-pass algorithm that computes principal curvatures, principal directions of curvature, and the derivative of curvature in graphics hardware.

2 Related Work

Cole and Finkelstein present a system for rendering textured lines that runs completely on graphics hardware. While their system does not use the geometry shader, it does support silhouette edge extraction. They introduce a segment atlas to efficiently compute line visibility and then render stylized lines using textures. One limitation of their system is that it only works with pre-extracted lines from a model [Cole and Finkelstein 2009].

Lee et al. present an image-based algorithm for drawing lines. Their algorithm renders the scene in grayscale and then draws lines between areas of varying darkness and lightness. Their implementation runs at interactive rates and looks good visually. They mention a limitation of their system in that it cannot vary the stylization of lines without further post-processing [Lee et al. 2007].

Another real-time image-based system is presented by Kim et al. Their system uses shading and ray-tracing to create an intensity map, principal direction estimation and stroke direction propaga-

[*]e-mail: griffin5@cs.umbc.edu

tion to create a stroke direction map and then uses both maps along with a stroke texture in a stroke-mapping algorithm. While their system produces very good visual results, their use of ray-tracing is not well-suited to current graphics hardware. They do, however, hint at the possibility of using geometry shader hardware to estimate principal direction, but rule it out as too expensive [Kim et al. 2008].

DeCarlo et al. present a system that uses surface curvature of meshes in an object-based algorithm to create suggestive contours. Suggestive contours are lines that are near-contours from the current viewpoint and real-contours in near-by viewpoints. By rendering suggestive contours along with actual contours, their system creates much more expressive line drawings that just using contours alone. DeCarlo et al. then provide several modifications to the system to enable real-time rendering as well as mentioning that further performance improvements are likely possible if the full programmable capabilities of graphics hardware are used [DeCarlo et al. 2003; DeCarlo et al. 2004].

Another object-based method utilizing view-dependent differential geometry is presented by Judd et al. Their system uses curvature as well, but a key difference is that they derive in screen space instead of object space. Their results look very good, however the system has poor performance, especially for large meshes [Judd et al. 2007].

3 Background

Surface normals are considered first-order structure of smooth surfaces because because the normals define, along with the point $\mathbf{p}$ where the normal sits, a first-order approximation to the surface as a tangent plane [Rusinkiewicz et al. 2008].

Interesting second order structure is normal curvature. At a point $\mathbf{p}$ on a smooth surface, normal curvature $k(\mathbf{u}) = S(\mathbf{u}) \cdot \mathbf{u}$, where $\mathbf{u}$ is a unit vector tangent to the surface at $\mathbf{p}$ and $S(\mathbf{u})$ is the shape operator. Principal curvatures, k_1 and k_2, are defined as the maximum and minimum values, respectively, of $k(\mathbf{u})$ at $\mathbf{p}$. Principal directions are the directions of the principal curvatures [O'Neill 1966].

Normal curvature is often expressed in terms of the second fundamental form $\mathbf{II}$:

$$\mathbf{II} = \begin{pmatrix} D_u n & D_v n \end{pmatrix} = \begin{pmatrix} \frac{\partial n}{\partial u} \cdot u & \frac{\partial n}{\partial v} \cdot u \\ \frac{\partial n}{\partial u} \cdot v & \frac{\partial n}{\partial v} \cdot v \end{pmatrix}$$

where n is the normal and $D_u n$ is the derivative of n in the direction of u and $D_v n$ is the derivative of n in the direction of v and u and v are local coordinates in the tangent plane [Rusinkiewicz 2004].

The more interesting line drawing techniques use curvature and the differential of curvature to determine where on models to create lines [DeCarlo et al. 2004; DeCarlo et al. 2003; Judd et al. 2007]. Estimating curvature on polygonal meshes is a well understood problem and there exists an algorithm for computing the estimation at each vertex using differences in normals along edges between vertices [Rusinkiewicz 2004; Rusinkiewicz 2009].

The algorithm described by Rusinkiewicz uses vertex normals and edges to set up a set of linear constraints to solve for the second fundamental form $\mathbf{II}$:

Rusinkiewicz then approximates $\mathbf{II}$ in the discrete case by setting up the following set of linear constraints:

$$\mathbf{II} = \begin{pmatrix} e_0 \cdot u \\ e_0 \cdot v \end{pmatrix} = \begin{pmatrix} (n_2 - n_1) \cdot u \\ (n_2 - n_1) \cdot v \end{pmatrix}$$
$$\mathbf{II} = \begin{pmatrix} e_1 \cdot u \\ e_1 \cdot v \end{pmatrix} = \begin{pmatrix} (n_0 - n_2) \cdot u \\ (n_0 - n_2) \cdot v \end{pmatrix}$$
$$\mathbf{II} = \begin{pmatrix} e_2 \cdot u \\ e_2 \cdot v \end{pmatrix} = \begin{pmatrix} (n_1 - n_0) \cdot u \\ (n_1 - n_0) \cdot v \end{pmatrix}$$

where e_i are the three edges between the three vertices of a triangular face in the mesh and n_i are the vertex normals. The constraints are then used to solve for $\mathbf{II}$ using least squares. The result is then added back to each vertex weighted by the amount of the triangular face that is closest to the vertex [Rusinkiewicz 2004].

This algorithm is implemented in the *trimesh2* library [Rusinkiewicz 2009] and computes the curvature by looping over the faces of the mesh. Since the algorithm approximates $\mathbf{II}$ at each face, it has been limited to running on a host computer. With the introduction of programmable geometry shaders [Blythe 2006], however, this per-face computation can be performed directly on graphics hardware.

4 Implementation

Our algorithm for estimating curvature is a straightforward adaptation of the host-based algorithm [Rusinkiewicz 2009]. Different passes over the faces of the model become passes through the graphics pipeline. Since the curvature computed at each face is weighted and added to each vertex of the face, we use framebuffer objects and blending functions to sum the curvature contribution of each face.

4.1 Algorithm

Our algorithm consists of four passes. The output of each pass is written to a floating-point texture attached to the framebuffer. For three of the passes, we need the per-vertex computation to be summed for all faces. We accomplish this by blending values into the texture with clamping disabled.

For the passes that require summing over each face, we enable blending with `glEnable(GL_BLEND)`, leave the blending equation as `GL_FUNC_ADD` and set both the source and destination blending functions to `GL_ONE`. This results in the values being added in the normal fashion: $V_s + V_d$, where V_s is the source value (the value being written to the framebuffer) and V_d is the destination value (the value already in the framebuffer) [Shreiner et al. 2006].

Before each of the algorithm passes run, we create a 32-bit floating point format texture with dimensions of the vertex map. The texture is then attached to the framebuffer. Because we use floating point format textures, we require the hardware to support floating point format color buffers. We also disable all clamping when writing to the color buffer and textures. The textures are properly addressed by using a vertex map.

As a preprocessing step, we map each vertex of the model to a pixel in the render target. The vertex mapping step starts by calculating the dimensions of the two-dimensional map:

$$w = NP(\lfloor \sqrt{|V|} \rfloor)$$
$$h = NP(\lceil \sqrt{|V|} \rceil)$$

where $|V|$ is the number of actual vertices in the model, and $NP(x)$ returns the next power of two for x. The dimensions are then used

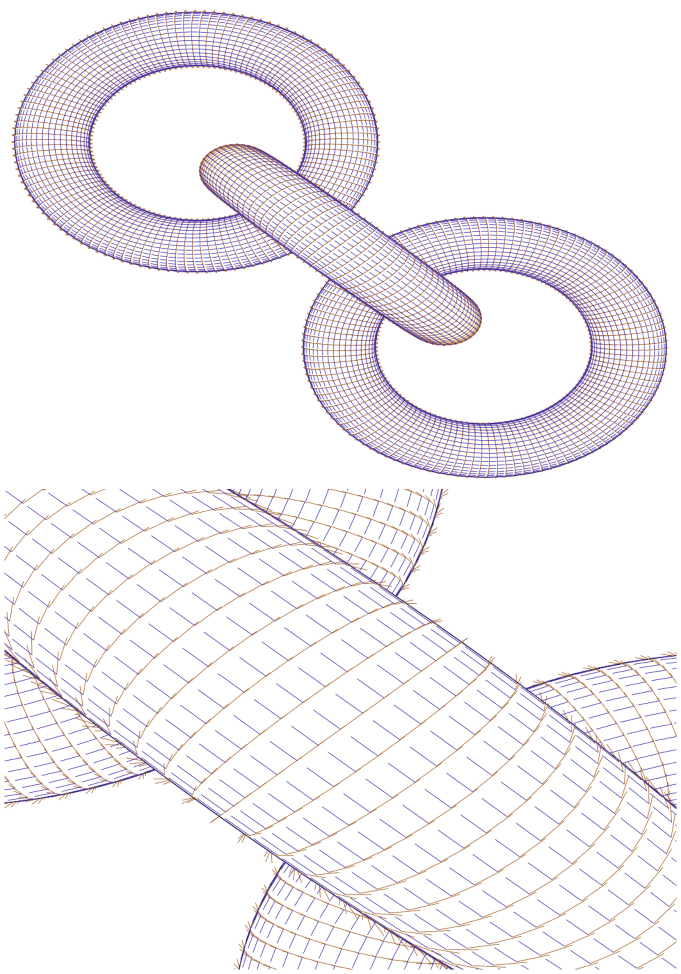

Figure 2: *Principal directions of curvature on a non-deformed model.*

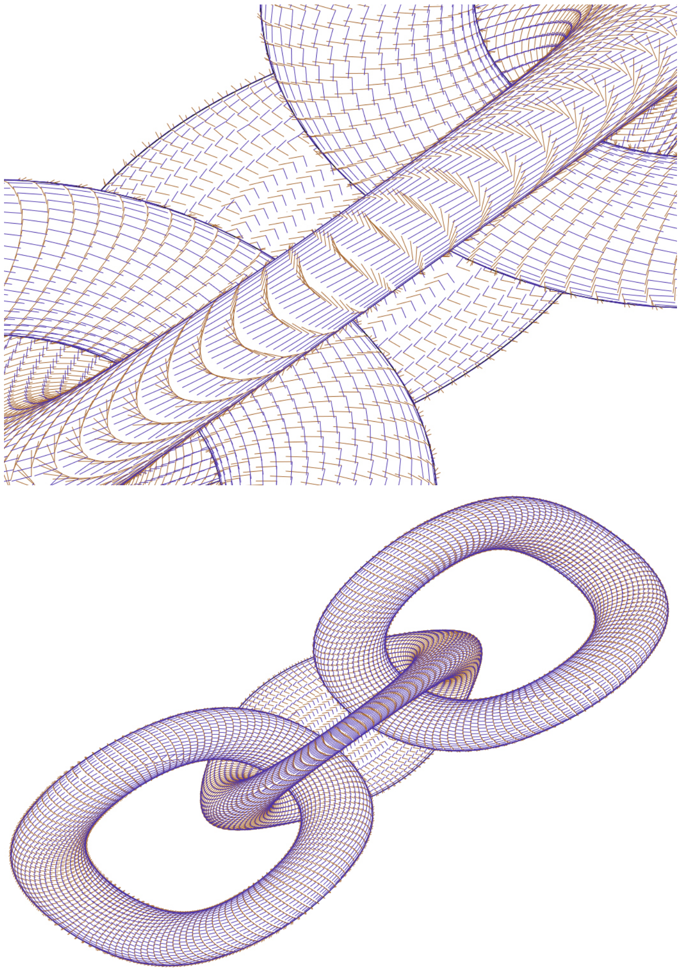

Figure 3: *Principal directions of curvature on a deformed model.*

to create a set of "vertices" from $[-1.0 - 1.0)$ which are assigned to each actual vertex in the model. The assignment is by simple position association between the two vertex arrays since the same index buffers for the triangles are used. The vertex map is also used as the texture coordinates for the passes that read the output of previous passes.

The first pass runs in the geometry shader and computes a weight for each vertex based on the area of each connected face. These weights are written for each vertex to the texture attached to the framebuffer. Since the geometry shader outputs three values for each face (the three connected vertices), and we have enabled alpha blending in the framebuffer, the weights for each vertex are summed for every face.

The second pass, also a geometry shader pass, starts by creating a tangent frame in the face. The differences between the normals at each vertex are taken and then, for each edge, a matrix is updated with dot products between the edge and the tangent frame and the normal deltas are weighted and summed. The matrix and summed delta normals are used as the linear constraints for a least-squares approximation to calculate a curvature tensor. The approximation described in more detail in 3. The curvature tensor is then projected into the tangent frame and then weighted at each vertex by the weight of the current face.

The third pass is a vertex-only pass and thus does not require blending. This pass takes the curvature tensor and uses a Jacobi rotation to find principal curvatures and directions. This pass also ensures that the principal directions at each vertex are perpendicular to the vertex normal. Figures 2 and 3 show the output of this pass. The orange lines are principal directions of maximum curvature and the blue lines are principal directions of minimum curvature. As can be seen in the bottom pictures, the directions orthogonal to each other.

The final geometry shader pass estimates the derivatives of curvature at each vertex. This pass follows a process similar to the second pass, by starting with a tangent frame in the face. The principal directions of curvature are then projected into the tangent frame. The differences between these local curvature vectors are then used, along with dot products between the edges and tangent frame to create a set of linear constraints. The linear constraints are used in a least-squares approximation to the differential curvature. The differential curvature at each vertex is weighted by the face.

4.2 Deforming Models

For our initial demonstration, we implemented two simple deformation primitives, tapering and bending [Barr 1984].

Tapering The tapering primitive is a global deformation along the z axis. From Barr, the vertex deformation is calculated as:

$$\begin{aligned} r &= f(z) = (1.0 - z^2) + 0.01 \\ X &= rx \\ Y &= ry \\ Z &= z \end{aligned} \tag{1}$$

where x, y, z are the original components of the vertex, and X, Y, Z are the deformed components. The normals are easily deformed as well using the following transformation matrix:

$$r^2 \underline{\underline{J}}^{-1T} = \begin{pmatrix} r & 0 & 0 \\ 0 & r & 0 \\ -rf'(z)x & -rf'(z)y & r^2 \end{pmatrix}$$

where $f'(z) = -2z$ from Equation 1 above.

Bending The bending primitive is a global deformation along the y axis. From Barr, the bending angle is calculated as:

$$\begin{aligned} \theta &= 0.01\hat{y} \\ C_\theta &= cos(\theta) \\ S_\theta &= sin(\theta) \end{aligned}$$

where

$$\hat{y} = \begin{cases} -1.0, & y \le -1.0 \\ y, & -1.0 < y < 1.0 \\ 1.0, & y \ge 1.0 \end{cases}$$

The vertex deformation is calculated as:

$$\begin{aligned} X &= x \\ Y &= \begin{cases} -S_\theta(z - \frac{1}{k}) + C_\theta(y + 1.0), & y < -1.0 \\ -S_\theta(z - \frac{1}{k}), & -1.0 \le y \le 1.0 \\ -S_\theta(z - \frac{1}{k}) + C_\theta(y - 1.0), & y > 1.0 \end{cases} \\ Z &= \begin{cases} C_\theta(z - \frac{1}{k}) + \frac{1}{k} + S_\theta(y + 1.0), & y < -1.0 \\ C_\theta(z - \frac{1}{k}) + \frac{1}{k}, & -1.0 \le y \le 1.0 \\ C_\theta(z - \frac{1}{k}) + \frac{1}{k} + S_\theta(y - 1.0), & y > 1.0 \end{cases} \end{aligned}$$

where x, y, z are the original components of the vertex, and X, Y, Z are the deformed components. Again, the normals are deformed using the following transformation matrix:

$$(1 - \hat{k}z)\underline{\underline{J}}^{-1T} = \begin{pmatrix} 1 - \hat{k}z & 0 & 0 \\ 0 & C_\theta & -S_\theta(1 - \hat{k}z) \\ 0 & S_\theta & C_\theta(1 - \hat{k}z) \end{pmatrix}$$

where

$$\hat{k} = \begin{cases} k, & \hat{y} = y \\ 0, & \hat{y} \ne y \end{cases}$$

4.3 Line Creation

As part of our demonstration we extract occluding contours and stroke them as simple lines. Occluding contours are the easiest type of contour to extract from a model. The naïve algorithm simply checks the angle between between the view vector (V) and normal (N) at each vertex of a triangle. If $V \cdot N < 0$ at one vertex and $V \cdot N > 0$ at another vertex, then the edge is a contour and a line is drawn. This method has the problem that it can easily cause artifacts where lines flip between edges of a triangle.

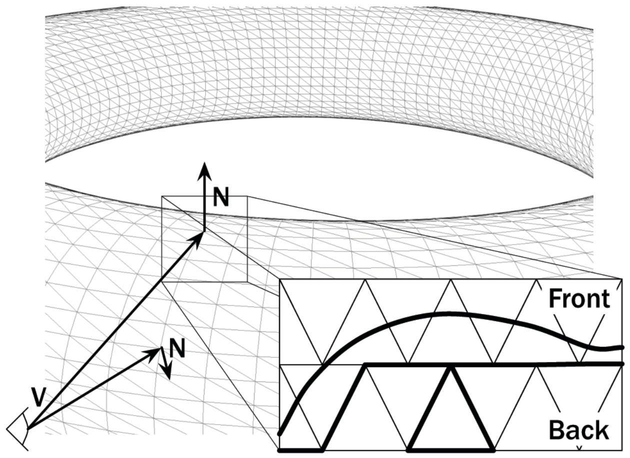

Figure 4: *Two methods to extract occluding contours on a triangular mesh. The bottom contour is the naïve method which only checks the dot product at each vertex. The top contour is an improved method that interpolates the zero crossing of the dot product across each face.*

A more advanced technique is to interpolate the dot product across the edges and find where the actual zero crossings of the dot product occur. These locations are then used to draw a line. The differences between these two techniques are illustrated in Figure 4. Figure 5 shows drawing occluding contours on a torus model.

We have an initial implementation of suggestive contours using the derivative of curvature. Figure 7 illustrates the results of these contours. While the results are promising, there are obvious artifacts, most notably the lines above the trunk and between the eyes. The implementation does, however, demonstrate that we are calculating curvature and its derivative and using those results.

4.4 Drawing Lines

Once the various types of lines have been created, they need to be drawn. We currently use the OpenGL line primitive to draw the lines. Future work will implement stylization of the lines by extruding fins in the geometry shader. Properly drawing anti-aliased lines in OpenGL is non-obvious, the most important step being to ensure writing the depth buffer is disabled with `glDepthMask(GL_FALSE)`.

5 Results

Our current implementation has not been optimized for graphics hardware execution, however, we are able to achieve moderate performance without optimization. Table 1 lists the frame rates for various models of increasing complexity. For medium complexity models, such as the elephant, we are close to interactive frame rates and we believe an optimized version of our algorithm, along with fixing the bug described below, will yield greater than interactive rates on the moderately complex models.

We have a bug that is reducing performance that we have yet to solve. In the curvature estimation passes, we are unable to load the vertex map into the graphic card memory as a buffer object. Thus, as model complexity increases, we must transfer the entire set of map vertices onto the graphics card every frame. For small models, this overhead does not affect performance, but with 80K vertices or 280K vertices, performance suffers drastically.

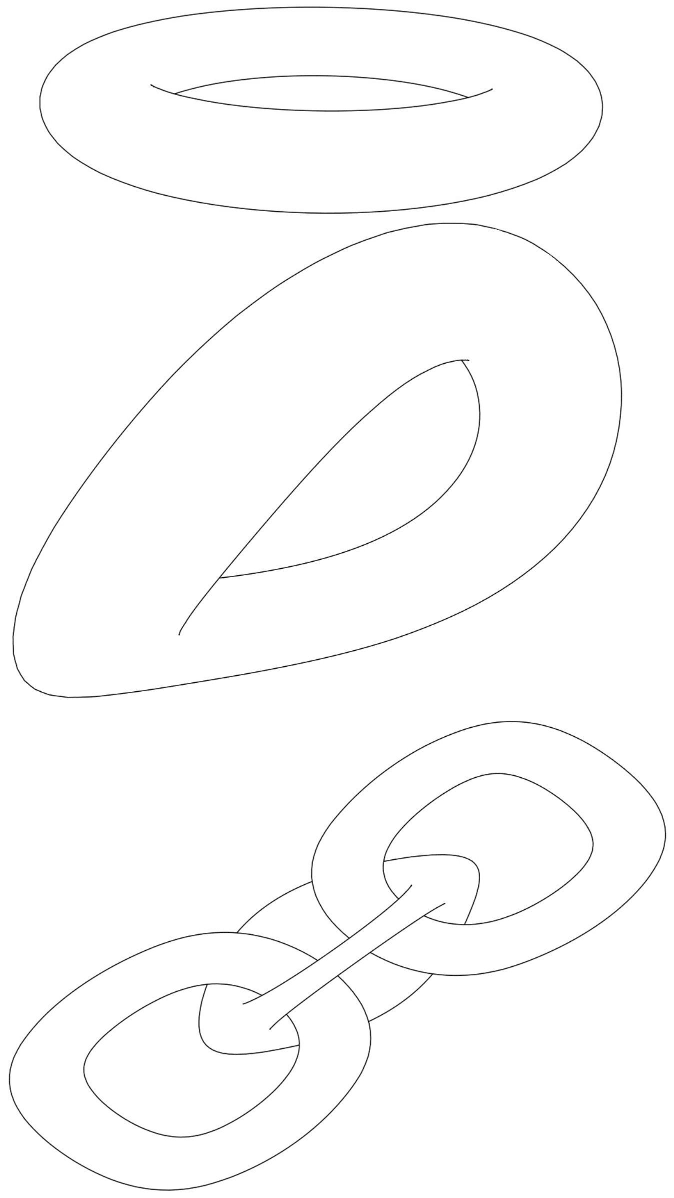

Figure 5: *Occluding contours.*

Model	Vertices	Triangles	FPS
torus	4,800	9,600	300
rings	14,400	28,800	100
elephant	78,793	157,160	10
heptoroid	286,679	573,400	0.1

Table 1: *Frame rates for various models of increasing complexity.*

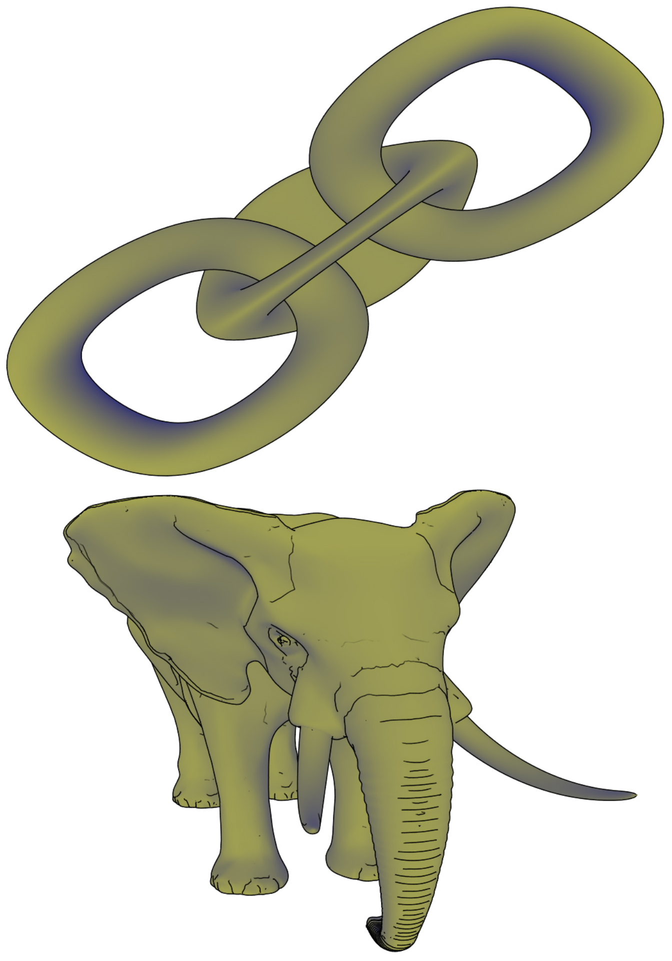

Figure 6: *Examples of our algorithm rendered with tone shading [Gooch et al. 1998].*

6 Future Work

There are several aspects to extending out initial implementation. First, we need to optimize our algorithm for execution on graphics hardware. We also need to fix the bug mentioned in 5. Both changes should increase performance to acceptable interactive frame rates for even highly complex models.

Another obvious avenue for future work is to improve our suggestive contours implementation as well as add extraction of additional contours, such as apparent ridges. Additionally, we would like to add stylization of lines by extruding fins in the geometry shader. Examples of line stylization can be found in [Kalnins et al. 2003; Rusinkiewicz et al. 2008].

We also plan to add more complex model deformation such as vertex blending to enable line drawing on more interesting animated models. Another possible extension is to implement recent work in accelerated line visibility [Cole and Finkelstein 2009].

References

BARR, A. 1984. Global and local deformations of solid primitives. *ACM SIGGRAPH Computer Graphics*, 21–30.

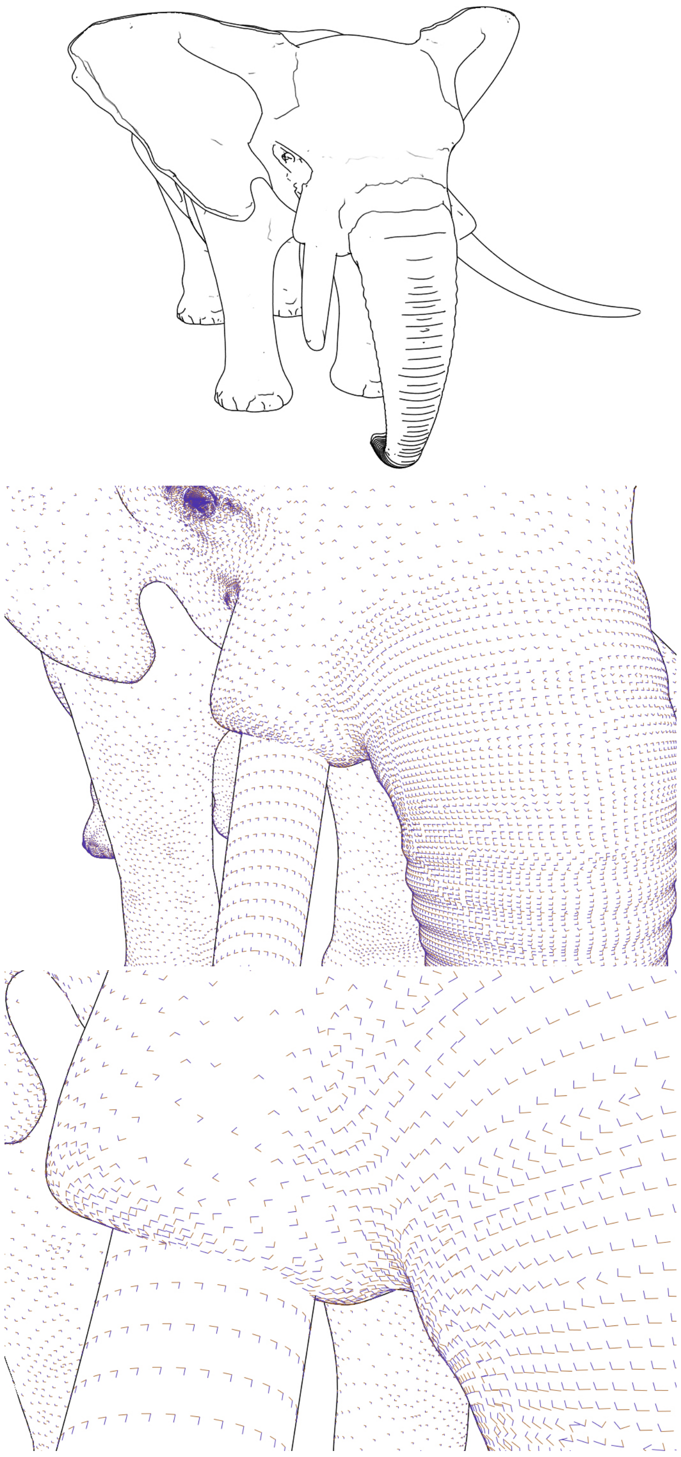

Figure 7: *Our initial attempt at implementing suggestive contours.*

BLYTHE, D. 2006. The direct3d 10 system. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, 724–734.

COLE, F., AND FINKELSTEIN, A. 2009. Fast high-quality line visibility. *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, 115–120.

DECARLO, D., FINKELSTEIN, A., RUSINKIEWICZ, S., AND SANTELLA, A. 2003. Suggestive contours for conveying shape. In *ACM SIGGRAPH 2003 Papers*, 848–855.

DECARLO, D., FINKELSTEIN, A., AND RUSINKIEWICZ, S. 2004. Interactive rendering of suggestive contours with temporal coherence. In *Proceedings of the 3rd international symposium on Non-photorealistic animation and rendering*.

GOOCH, A., GOOCH, B., SHIRLEY, P., AND COHEN, E. 1998. A non-photorealistic lighting model for automatic technical illustratio n. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer g raphics and interactive techniques*, 447–452.

JUDD, T., DURAND, F., AND ADELSON, E. 2007. Apparent ridges for line drawing. In *ACM SIGGRAPH 2007 Papers*.

KALNINS, R. D., DAVIDSON, P. L., MARKOSIAN, L., AND FINKELSTEIN, A. 2003. Coherent stylized silhouettes. *ACM Transactions on Graphics 22*, 3, 856–861.

KIM, Y., YU, J., YU, X., AND LEE, S. 2008. Line-art illustration of dynamic and specular surfaces. In *ACM SIGGRAPH Asia 2008 Papers*.

LEE, Y., MARKOSIAN, L., LEE, S., AND HUGHES, J. F. 2007. Line drawings via abstracted shading. In *ACM SIGGRAPH 2007 Papers*.

O'NEILL, B. 1966. *Elementary Differential Geometry*. Academic Press, Inc.

RUSINKIEWICZ, S., COLE, F., DECARLO, D., AND LSTEIN, A. F. 2008. Line drawings from 3d models. *SIGGRAPH 2008 Classes* (Aug), 1–356.

RUSINKIEWICZ, S. 2004. Estimating curvatures and their derivatives on triangle meshes. *Proceedings of the 2nd International Symposium on 3D Data Processing, Visualization and Transmission*, 486–493.

RUSINKIEWICZ, S. 2009. trimesh2. *http://www.cs.princeton.edu/gfx/proj/trimesh2/*.

SHREINER, D., WOO, M., NEIDER, J., AND DAVIS, T. 2006. *OpenGL Programming Guide*, fifth ed. Addison-Wesley.