GPU Curvature Estimation on Deformable Meshes

Wesley Griffin* Yu Wang* David Berrios* Marc Olano*
University of Maryland, Baltimore County

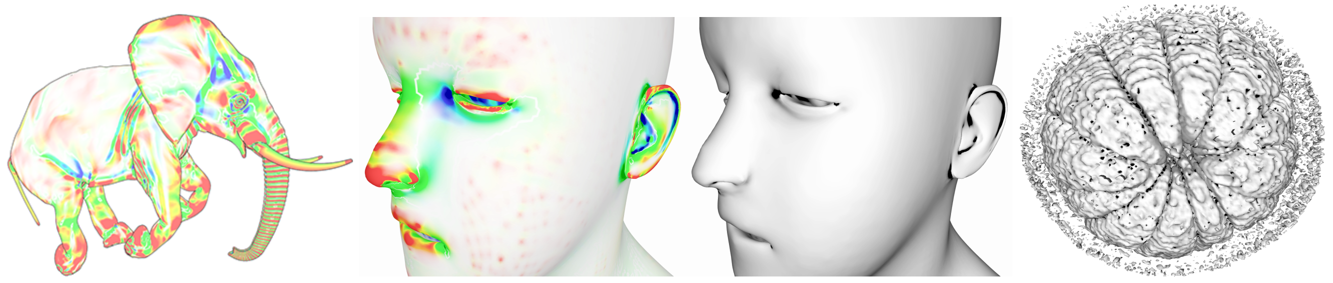

Figure 1: *On the left is one frame of an elephant animation visualizing estimated curvature. Blue indicates concave areas, red indicates convex areas, and green indicates saddle-shape areas. The middle is using surface curvature to approximate ambient occlusion. On the right is an orange volume with ∼1.5 billion vertices with Lambertian shading and ambient occlusion which we estimate curvature on in 39.3 ms.*

Abstract

Surface curvature is used in a number of areas in computer graphics, including texture synthesis and shape representation, mesh simplification, surface modeling, and non-photorealistic line drawing. Most real-time applications must estimate curvature on a triangular mesh. This estimation has been limited to CPU algorithms, forcing object geometry to reside in main memory. However, as more computational work is done directly on the GPU, it is increasingly common for object geometry to exist only in GPU memory. Examples include vertex skinned animations and isosurfaces from GPU-based surface reconstruction algorithms.

For static models, curvature can be pre-computed and CPU algorithms are a reasonable choice. For deforming models where the geometry only resides on the GPU, transferring the deformed mesh back to the CPU limits performance. We introduce a GPU algorithm for estimating curvature in real-time on arbitrary triangular meshes. We demonstrate our algorithm with curvature-based NPR feature lines and a curvature-based approximation for ambient occlusion. We show curvature computation on volumetric datasets with a GPU isosurface extraction algorithm and vertex-skinned animations. Our curvature estimation is up to ∼18x faster than a multi-threaded CPU benchmark.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Line and curve generation; I.3.3 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

Keywords: real-time rendering, GPU, geometry shader, curvature, line drawing, ambient occlusion

1 Introduction

Surface curvature is used in many areas in computer graphics, including texture synthesis and shape representation [Gorla et al. 2003]; mesh simplification [Heckbert and Garland 1999]; surface modeling [Moreton and Séquin 1992]; and artistic line drawing methods, such as suggestive contours [DeCarlo et al. 2003], apparent ridges [Judd et al. 2007], and demarcating curves [Kolomenkin et al. 2008]. These methods operate on a discrete polygonal representation of a continuous surface and must estimate curvature.

Recent research has looked at accelerating ambient occlusion [Zhukov et al. 1998; Landis 2002] for real-time use [Loos and Sloan 2010; McGuire 2010] even on dynamic models [Kontkanen and Alia 2006; Kirk and Arikan 2007]. Hattori et al. [2010] use a quadric surface approximation to analytically solve for ambient occlusion. The quadric approximation requires estimated curvature.

Video games often use vertex-blended animations where a pose mesh is transferred to the GPU once and then modified each frame such that the transformed mesh exists only on the GPU. Given the strict real-time requirements of video games, transferring animated models to the CPU to estimate curvature is too costly.

GPU Computing has increased the performance of scientific simulation computations and visualizations. When working with volumetric data, these applications might run a GPU-based surface reconstruction algorithm that generates an isosurface that only exists on the GPU. With GPU memory increasing, GPUs can store higher resolution volumes or more time steps of time-varying datasets. In situations where an isosurface needs to be computed for every time step or recomputed interactively, extraction on a GPU can offer improved performance for visualizing the data in real-time.

Algorithms exist to estimate curvature in real-time on deforming models, but the methods either work in image-space [Kim et al. 2008] or require a pre-processing step to learn the curvature function with training data [Kalogerakis et al. 2009]. While both techniques are useful, the ability to estimate curvature in object-space and without pre-processing would provide much greater flexibility.

We introduce a GPU algorithm for estimating curvature that works in object-space and does not require any pre-processing. Given an arbitrary triangular mesh in GPU memory, our algorithm can estimate the curvature on the mesh in real-time. Since our algorithm runs completely on the GPU and works on triangular meshes, it can be easily adapted to existing rendering systems.

To demonstrate our algorithm, we implement an ambient occlusion

*e-mail: {griffin5, zf89375, david.berrios, olano}@umbc.edu

approximation based on Hattori et al. [2010], a vertex-blending animation system [Akenine-Möller et al. 2008] and a GPU-based isosurface computation system.

The specific contributions of this paper are:

- A GPU algorithm to compute principal curvatures, principal directions of curvature, and the derivative of curvature.
- A CUDA marching cubes algorithm that creates fused vertices.
- A demonstration of curvature-based NPR feature lines and curvature-based ambient occlusion approximation.

The rest of the paper is organized as follows: Section 2 covers background and related work, Section 3 discusses our algorithm, and Section 4 presents our results.

2 Background and Related Work

Our system combines curvature estimation, ambient occlusion estimation, line drawing, and isosurface reconstruction.

2.1 Curvature

Below we briefly discuss curvature and refer the reader to O'Neill [2006] for more detail and relevant proofs of theorems.

At a point $\mathbf{p}$ on an continuous, oriented surface $\mathbf{M}$, curvature describes how the scale of the tangent plane changes around $\mathbf{p}$. An oriented surface is a surface where a consistent direction for the normal vector at each point has been chosen. Surface normals are considered first-order structure of smooth surfaces: at $\mathbf{p}$ a normal vector $\mathbf{N_p}$ defines a tangent plane $\mathbf{T_p}(\mathbf{M})$ to a surface $\mathbf{M}$.

Curvature is typically defined in terms of the shape operator $S_\mathbf{p}(\mathbf{u})$, which is the rate of change of a unit normal vector field, U, on the surface M in the direction $\mathbf{u}$, which is a tangent vector at $\mathbf{p}$.

$$S_\mathbf{p}(\mathbf{u}) = -\nabla_u U \tag{1}$$

The shape operator is a symmetric linear operator such that

$$S_p(\mathbf{u}) \cdot \mathbf{v} = S_p(\mathbf{v}) \cdot \mathbf{u} \tag{2}$$

for any pair of tangent vectors $\mathbf{u}$ and $\mathbf{v}$ to $\mathbf{M}$ at $\mathbf{p}$ [O'Neill 2006].

Since the shape operator is a symmetric linear operator, it can be written as a 2×2 symmetric matrix, S, for each $\mathbf{p}$ given an orthonormal basis. This matrix has real eigenvalues, λ_1 and λ_2 (principal curvatures), and eigenvectors, $\mathbf{v_1}$ and $\mathbf{v_2}$ (principal directions).

Gauss Curvature, K, and Mean Curvature, H, are then defined as:

$$K(\mathbf{p}) = \lambda_1\lambda_2 \qquad = \det S \tag{3}$$

$$H(\mathbf{p}) = \tfrac{1}{2}(\lambda_1 + \lambda_2) \qquad = \tfrac{1}{2}\text{trace } S \tag{4}$$

Another way to represent curvature is normal curvature, $k(\mathbf{u})$. A theorem relates normal curvature to the shape operator:

$$k(\mathbf{u}) = S(\mathbf{u}) \cdot \mathbf{u} \tag{5}$$

The maximum and minimum of $k(\mathbf{u})$ at $\mathbf{p}$ are principal curvatures, k_1 and k_2, and the directions in which the maximum and minimum occur are principal directions. Normal curvature is second-order structure and defines a quadric approximation to $\mathbf{M}$ at $\mathbf{p}$:

$$z = \tfrac{1}{2}(k_1x^2 + k_2y^2) \tag{6}$$

The second fundamental form is defined using the shape operator:

$$\mathbf{II}(\mathbf{u}, \mathbf{v}) = S(\mathbf{u}) \cdot \mathbf{v} \tag{7}$$

$\mathbf{II}$ is also called the curvature tensor and can be written using the directional derivatives of normals [Rusinkiewicz 2004]:

$$\mathbf{II}(\mathbf{u}, \mathbf{v}) = \begin{pmatrix} D_\mathbf{u}n & D_\mathbf{v}n \end{pmatrix} = \begin{pmatrix} \frac{\partial n}{\partial \mathbf{u}} \cdot \mathbf{u} & \frac{\partial n}{\partial \mathbf{v}} \cdot \mathbf{u} \\ \frac{\partial n}{\partial \mathbf{u}} \cdot \mathbf{v} & \frac{\partial n}{\partial \mathbf{v}} \cdot \mathbf{v} \end{pmatrix} \tag{8}$$

2.2 Curvature Estimation

On a discrete surface curvature must be estimated and there has been substantial work in estimating surface curvature [Taubin 1995; Petitjean 2002; Goldfeather and Interrante 2004; Rusinkiewicz 2004; Tong and Tang 2005]. Gatzke and Grimm [2006] divide the algorithms into three groups: surface fitting methods, discrete methods that approximate curvature directly, and discrete methods that estimate the curvature tensor. In the group that estimates the curvature tensor, Rusinkiewicz [2004] and Theisel et al. [2004] estimate curvature on a triangular mesh at each vertex using differences of surface normals (Equation 8) and then average the per-face tensor over each adjacent face to the vertex. These algorithms have been limited to running on the CPU where the algorithm can access adjacent faces (the one-ring neighborhood of the vertex).

Recently Kim et al. [2008] introduced an image-based technique that estimates curvature in real-time on the GPU. Their method traces rays based on the normals at a point to estimate the direction of curvature. Kalogerakis et al. [2009] also present a real-time method for estimating curvature specifically on animated models. Their method requires a mesh with parameterized animations, such as per-frame transformation matrices. A pre-processing step learns a mapping from the parameterization to curvature. Given a mapping, their algorithm predicts curvature in real-time on an unseen set of animation parameters.

2.3 Ambient Occlusion

Ambient occlusion (AO) is "shadowing" of ambient illumination due to local occlusion either by other objects or by self-occlusion. Introduced by Zhukov et al. [1998] as one part of ambient obscurance, AO has been extensively researched for use in dynamic environments with static models. The approaches can be divided into screen-space methods [Mittring 2007; Bavoil et al. 2008; Kajalin 2009; Bavoil and Sainz 2009] or world-space methods [Kontkanen and Laine 2005; McGuire 2010; Loos and Sloan 2010].

Ambient occlusion is formulated as the integral of a visibility function over the positive hemisphere at each point on a surface:

$$AO = 1 - \frac{1}{\pi}\int_{\Omega_+} V(x, \omega) \cdot (\widehat{\omega_i} \cdot \hat{n}) d\widehat{\omega_i} \tag{9}$$

Screen-space methods are able to compute AO on deforming meshes simply because the algorithm runs as a post-process on every frame. Little work exists on real-time computation of AO directly on deforming meshes. Kontkanen and Aila [2006] use pre-computed AO values at each vertex for a set of reference pose meshes. By learning a mapping from animation parameters to AO values, the algorithm can approximate AO on unseen pose meshes. Where Kontkanen and Aila learn a linear mapping over the entire parameterization space, Kirk and Arikan [2007] learn a multilinear mapping over subsets of the parameterization space using k-means clustering and principal component analysis.

Building on Hattori et al. [2010], we take a different approach for computing AO on deforming meshes. Using normal curvature, a surface can be approximated with a quadric at each discrete point (Equation 6). We can now analytically solve for AO using this quadric defined by k_1 and k_2 and a unit-radius positive hemisphere:

$$AO(k_1, k_2) = 1 - \frac{1}{2\pi} \int_0^{2\pi} \int_0^{\Theta} sin\widehat{\theta}_i d\widehat{\theta}_i d\widehat{\phi}_i \quad (10)$$

$$\Theta = arccos \left(\frac{-1 + \sqrt{1 - A^2}}{A} \right) \quad (11)$$

$$A = k_1 cos^2 \widehat{\phi}_i + k_2 sin^2 \widehat{\phi}_i \quad (12)$$

The $^1/_{2\pi}$ term scales the double integral to the range $(0, 1)$.

2.4 Line Drawing

Line drawing algorithms can be subdivided into two classes: image-based and object-based. Image-based techniques rasterize the scene and use image processing techniques to find surface properties, find feature lines, and to estimate curvature. Image-based algorithms are easier to implement than object-based methods but have some deficiencies. First, image-based algorithms must handle neighboring pixels that are not part of the same object. Second, the algorithms are limited to the resolution of the image and cannot account for sub-pixel detail. Temporal coherence can also be an issue and the shower-door effect must be managed. Finally, stylization can be difficult as feature lines are not represented with geometry.

Object-based methods typically use second- and third-order surface properties to extract feature lines directly from a polygonal model in world-space. Object-based techniques such as suggestive contours [DeCarlo et al. 2003], apparent ridges [Judd et al. 2007], and demarcating curves [Kolomenkin et al. 2008] can be divided into two groups: view-dependent and view-independent. View-dependent methods include the viewing direction when extracting feature lines while view-independent methods do not. These techniques must estimate curvature on a mesh in object-space.

2.5 Isosurface Reconstruction

Surface reconstruction is one of the most frequently used methods to analyze and visualize volumetric data. Marching cubes [Lorensen and Cline 1987] is the most widely used algorithm for computing a triangular surface or model from volumetric data. Treece et al. [1999] use a variation of marching cubes, marching tetrahedra, that is easier to implement and handles non-gridded data. Geiss [2007] introduces a marching cubes implementation using the programmable graphics pipeline. To improve the speed of his implementation, he introduces a method for pooling vertices and creating an indexed triangle list. Our parallel algorithm does require shared vertices, but we also want to support varying voxel sizes and Geiss' method is limited to a fixed voxel size. We introduce (Section 3.3) a CUDA marching cubes algorithm that uses a hash map to fuse vertices.

3 Parallel Algorithm

Our algorithm is based on the CPU algorithm by Rusinkiewicz [2004]. Rusinkiewicz creates a set of linear constraints to solve for curvature over a single face. The constraints use the differences between normals along edges (Equation 8) and are solved using least squares. To find the curvature at a vertex, the per-face curvatures of each adjacent face are averaged together.

The algorithm is composed of several iterative steps that can be grouped based on the type of input to each step: per-face or per-vertex. Each step builds on the computation from the previous step, but within each step the computation is independent over the input. Our parallel algorithm exploits this computational independence to parallelize the work within each step. The steps are:

1. **Normals.** Normals at each vertex are computed by averaging the per-face normals of the one-ring.
2. **Areas.** A weighting is calculated to determine how much a face contributes to each of the three vertices.
3. **Initial Coordinates.** An orthonormal basis is generated at each vertex to rotate the curvature tensor.
4. **Curvature Tensor.** An over-specified linear system for the tensor of a face is created using Equation 8 and solved using LDL^T decomposition and least squares fit. The tensor is rotated into a local coordinate system at each vertex using the orthonormal basis and weighted by the area of the vertex. The weighted tensors are summed across the one-ring to compute the averaged curvature tensor at each vertex.
5. **Principal Directions.** The curvature tensor is diagonalized by the Jacobi method and estimates eigenvalues (principal curvatures) and eigenvectors (principal directions).
6. **Curvature Differential.** The principal curvature differential at each vertex is estimated using linear constraints based on principal curvatures and averaging across the one-ring.

3.1 Computational Primitives

Based on the previous discussion, we define two computational primitives to implement our algorithm: a per-vertex primitive and a per-face primitive with the output averaged or summed over the one-ring. In a parallel algorithm, the data parallel threads will be distributed across both processors and time. Data written by one thread that will be read in another thread must force a barrier for all threads to complete. Our per-vertex computations are independent and need no barrier, but our per-face computations, which are averaged or summed, require synchronization.

3.2 Primitive Implementation

There are two possibilities for implementing our computational primitives on the GPU: with the general purpose APIs such as CUDA or with shaders in the graphics pipeline. Functionally, either choice is equivalent, as they both execute on the same hardware. The primary differences in the approaches relative to this work are more flexible shared memory access in CUDA and special purpose hardware for blending in the graphics pipeline.

The algorithm needs to sum across several threads, which could be accomplished with a CUDA atomic add or graphics pipeline blending. In either case a pass or kernel barrier must be introduced before using the results. The writes to the output buffers cannot be constrained to a single thread block, so a CUDA implementation would need a full kernel synchronization point, not just the lighter-weight *syncthreads* barrier. In contrast, GPUs have special hardware for blending to output buffers (render targets) with dedicated Arithmetic Logic Units (ALUs) optimized for throughput.

Curvature estimation is a geometric problem. Treating it as such, i.e. using the geometric pipeline, allows use of the more efficient blending hardware to synchronize access instead of atomic operations accompanied by an immediate kernel end. Additionally, when the geometry already exists in a format that the graphics pipeline expects, i.e. indexed vertex buffers, using the pipeline automatically handles setting up the geometry for the shader instances.

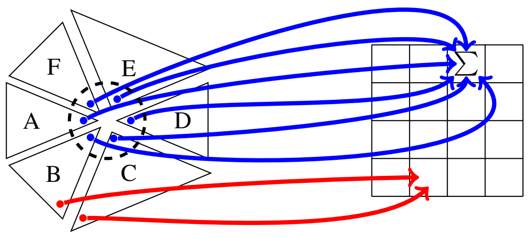

Figure 2: *The contribution of the one-ring neighborhood of a vertex is averaged by blending into a single pixel of a render target. A single vertex with the one-ring of faces is indicated by the dashed circle. The same vertex from each face is mapped as a point primitive to a single pixel of a render target.*

There are cases, however, when a general purpose API is the better choice. We describe in Section 3.3 a marching cubes CUDA implementation where there is no benefit to using the geometric pipeline. In fact, we implement a hash map to fuse vertices while extracting the isosurface using the atomic operations in CUDA.

Given our choice to compute curvature using the graphics pipeline, the per-vertex primitive is implemented in a vertex shader and the per-face primitive is implemented in a geometry shader that outputs a set of point primitives for each vertex of the face. Note that adjacency information provided as input to a geometry shader does not provide access to the one-ring of a vertex. Since the operation across the one-ring is either averaged or summed, the computation can be accomplished by using additive blending into a render target.

Figure 2 shows how each vertex in a mesh is mapped to a single pixel in a render target. Inside a geometry shader, the per-face computation is output as a point primitive. The point primitives are then written to a render target in a pixel shader program using additive blending, where the current value in the shader is added to the existing value in the target. Using this technique, the algorithm can easily average or sum values of the one-ring neighborhood around a vertex. This mapping technique requires input meshes with fused vertices. In Section 3.3 we describe a GPU vertex fusing algorithm applied to GPU marching cubes.

3.3 Input Transformations

We support two types of input: volumetric datasets and skinned animation. **Isosurface Extraction** computes an isosurface for a volumetric dataset and **Skin** transforms vertex-skinned animations.

Isosurface Extraction. For a volumetric dataset we first compute an isosurface using a CUDA marching cubes implementation that outputs an indexed, fused triangle list to support mapping the one-ring around each vertex in our curvature algorithm.

After classifying the voxels and determining the number of vertices per voxel, two sequential kernels are run: a triangle kernel and an index kernel. The triangle kernel generates the vertex buffer, a “full” index buffer with each vertex separately indexed, and a hash buffer with each vertex separately hashed. We sort the hash and index buffers as key/value pairs using radix sort [Satish et al. 2009]. Figure 3 shows the buffers both before and after sorting.

To fuse vertices, the index kernel uses the hash buffer to locate the first index of a vertex in the vertex buffer and outputs that index to the fused index buffer. After generating a reference vertex and hash, the kernel uses binary search to find the reference hash in the hash buffer. As Figure 3 shows, there will be duplicate hash keys due to both duplicate vertices and hash collisions. The binary search will terminate somewhere within the set of duplicate hash keys (thick

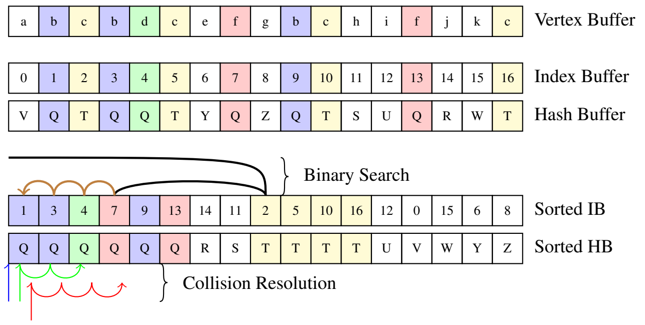

Figure 3: *Vertex, index, and hash buffers before and after radix sorting used in* ***Isosurface Extraction****. The yellow columns are one set of duplicate vertices. The blue columns are a second set of duplicate vertices. The red columns are a third set of duplicate vertices where the hash collides with the blue columns. The green column is also a hash collision with the blue and red columns.*

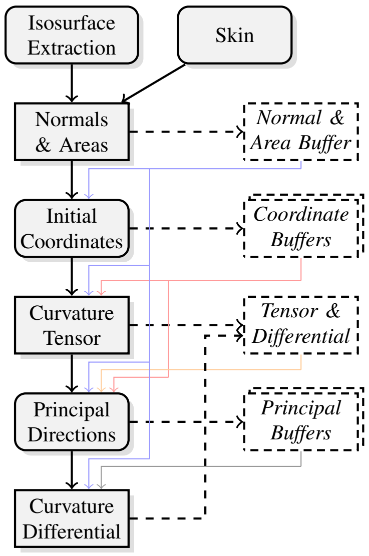

Figure 4: *Algorithm flowchart. See Section 3.4 for discussion.*

black line in the figure). Once the binary search completes, the algorithm iterates backwards in the hash buffer (thick brown line) to find the first instance of the hash code. The index of this hash instance is returned from the binary search algorithm.

After the binary search, the index kernel must deal with hash collisions. As the blue, green, and red columns in Figure 3 show, the collisions are not guaranteed to be sorted. The binary search returns the position of the first hash code in the hash buffer. To resolve collisions, the index kernel iterates forward in the hash buffer, dereferencing the value in the vertex buffer and comparing it to the reference vertex (green and red lines in the figure). Once the vertices match, the forward iterative search terminates and the corresponding value in the index buffer is output as the vertex index.

Skin. For an animated or static model we use a vertex shader to skin each vertex based on a set of keyframe matrices. The deformed vertices are written to a new buffer via stream out.

3.4 GPU Curvature Algorithm

Figure 4 is a flowchart of our parallel algorithm. Since each step of the algorithm described in Section 3 builds on the computation from the previous step, synchronization points between each step of our algorithm are necessary. Given our implementation choice of the graphics pipeline blending hardware, the synchronization barrier between each step is a pass through the pipeline.

Each pass, shown on the left in Figure 4, is an instance of one of the computational primitives described in Section 3.1. The passes with square-corners average or sum the output over the one-ring neighborhood, while the passes with round-corners do not. Render targets are shown on the right in italics font.

Normals & Areas. The curvature algorithm requires high quality vertex normals. Since the input data may be changing every frame, this pass recomputes vertex normals and areas for each face in a geometry shader. Each vertex normal is the weighted average of face normals across the one-ring. Like Rusinkiewicz [2004], vertex area is the Voronoi area described by Meyer et al. [2003]. Using the one-ring computation technique described in Section 3.1, the face normals of the one-ring at a vertex are averaged together and the areas of the one-ring are summed. The vertex normals and areas are packed into a single four-channel render target.

Initial Coordinates. An orthonormal basis for each vertex is required to transform the curvature tensor of a face into a local coordinate system. If the bases were computed as needed, then different faces sharing a vertex would use a different basis. This pass computes a single constant basis at each vertex in a geometry shader which writes each vector to a separate four-channel render target.

Curvature Tensor. To estimate the curvature tensor at each vertex, a geometry shader uses least squares to solve a linear system created with the differences of normals along the edges of a face (Equation 8). Using the one-ring computation technique, the weighted tensor at each face is averaged across the one-ring to estimate the curvature tensor at a vertex. The tensor is three floating-point values and is stored in a four-channel render target.

Principal Directions. This pass computes minimum and maximum curvatures and principal directions in a vertex shader with no averaging. The minimum curvature and principal direction are packed into one four-channel render target and the maximum curvature and principal direction are packed into a second render target.

Curvature Differential. The derivative of principal curvatures is estimated in a geometry shader. The derivative is essentially a third-order property, incorporating information from a two-ring around the vertex (i.e. a one-ring of data computed on a one-ring around a vertex). Thus, the differences of principal curvatures along the edge of a face are used in Equation 8 instead of the differences of normals. The per-face differentials are averaged over the one-ring of faces. The derivative of principal curvatures is represented with a four-component floating-point value and the *Tensor & Differential* render target is reused.

3.5 Approximating Ambient Occlusion

One application of estimated curvature is to approximate ambient occlusion. Like Hattori et al. [2010], we pre-compute ambient occlusion using Equation 11 for values of curvature where the quadric is concave in both directions. We then create a lookup texture indexed by minimum and maximum principal curvatures that can be used directly in a pixel shader. To save a texture lookup, however, we take the diagonal of the lookup texture and fit a second-degree polynomial to the values:

$$AO = 1.0 - 0.0022 * (k_1 + k_2) + 0.0776 * (k_1 + k_2) + 0.7369$$

The ambient term in a lighting equation is then multiplied by the ambient occlusion term AO to darken the ambient illumination. Equation 11 used a unit-radius hemisphere, but we can approximate varying the radius of the hemisphere by scaling the minimum and maximum principal curvatures. Scaling the curvature values makes the quadric approximation more or less flat, which has the same effect as decreasing or increasing hemisphere radius.

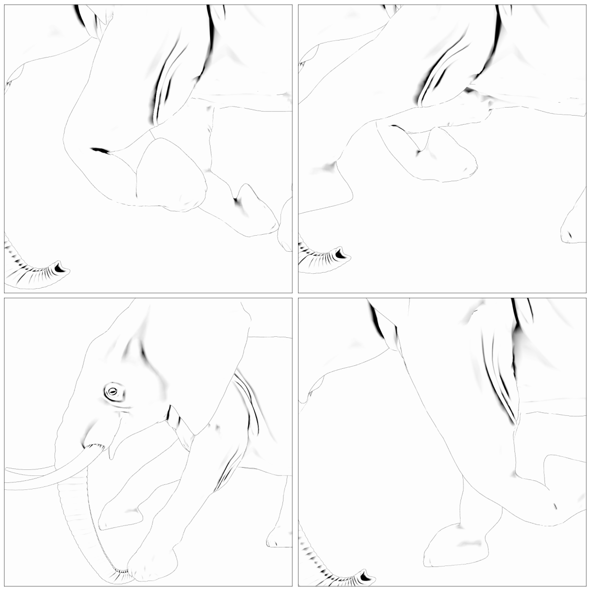
Figure 5: *Four frames of a vertex-blended animation of an elephant. The top row and right column are close-ups of the elephant's front legs. Notice the shadow behind the front knee in the top-left frame disappears in the top-right frame as the leg straightens. In the bottom-right frame the shadow begins to lightly reappear.*

3.6 Drawing Lines

Another application of estimated curvature is to extract and stroke occluding and suggestive contours on the input data. Feature lines are extracted in segments that are created per-face in the geometry shader as line primitives that are rasterized by the hardware. The extracted line segments do not have a global parameterization so any stylized stroking of the segments cannot rely on a parameterization. We leave stylized strokes for future work.

4 Results

4.1 Performance

Tables 1 and 2 and Figure 6 show that our parallel curvature algorithm achieves real-time performance for small- to medium-sized models, even on an older-generation consumer GPU. Results are shown for three different NVIDIA GPUs. To benchmark our algorithm, we took an existing CPU algorithm and threaded it to run in parallel. The CPU algorithm was run on a Core2 Quad Q8200 workstation with 8GB of RAM and each core running at 2.33GHz. The implementation used one thread per core, for a total of four threads on the workstation, to execute the algorithm.

Table 1 shows extraction and algorithm times for various volumetric datasets on the Quadro FX5800. Even on a surface with $\sim$1.5 billion vertices, our curvature algorithm runs in 39.3 milliseconds, although the surface extraction does take considerably longer.

Table 2 shows memory usage and frame times for models. The GeForce 9800GT and Quadro FX5800 results are at 1200$\times$1600 resolution and the GTX480 results are at 1920$\times$1200 resolution. All results use hardware 8$\times$ MSAA.

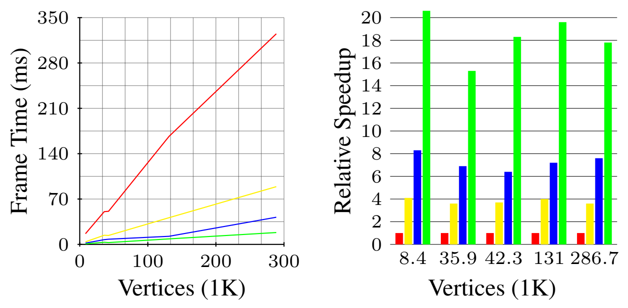

Figure 6: *Frame times and relative speedup for curvature estimation on the non-animated models. Red is a multi-threaded CPU algorithm at 1280×1024 resolution, yellow is an NVIDIA GeForce 9800GT, blue is an NVIDIA Quadro FX5800 both at 1600×1200, and green is an NVIDIA GTX480 at 1920×1200.*

Model	**Vertices**	**Iso**	**Extraction**	**AO**	**Lines**	**S+L**	**Alg**
		value	(ms)	(ms)	(ms)	(ms)	(ms)
bucky	47,208	0.095	32.375	3.4	3.7	3.7	3.3
H atom	113,256	0.095	78.750	4.1	4.6	4.6	4.1
spheres	436,668	0.110	284.875	12.8	15.0	15.0	12.8
orange	1,558,788	0.150	924.125	39.4	46.9	46.9	39.3

Table 1: *Frame times for different volumes on the NVIDIA Quadro FX5800.* ***AO*** *is the ambient occlusion approximation.* ***Lines*** *is drawing line primitives.* ***S+L*** *is Lambertian shading with ambient occlusion and drawing lines.* ***Alg*** *is only the curvature algorithm.*

For the largest animated model, our curvature estimation algorithm runs in 2.9 milliseconds (344 frames per second) on the GTX480. Even on the GeForce 9800GT, our algorithm still runs in 11.9 milliseconds (84 frames per second). The frame times on the Quadro FX5800 and GTX480 indicate that our curvature algorithm scales well with increasing hardware resources. Even on the largest model with ∼286,000 vertices, our curvature algorithm runs in 18.3 milliseconds (54 frames per second) on the GTX480.

The **RAM** column in Table 2 lists the memory usage for the geometry buffers and render targets for each model. The memory usage is independent of the screen resolution or anti-aliasing quality.

4.2 Curvature Estimation Error

To verify the accuracy of our parallel algorithm, we compare our results to a baseline CPU estimation algorithm on a torus model. Table 3 lists the maximum absolute error over the entire torus mesh.

Rusinkiewicz [2004] compares the robustness and accuracy of his CPU algorithm to other estimation algorithms. Our parallel algorithm is based on his CPU algorithm, so the error we report here is the error introduced by our GPU implementation. As Table 3 shows, the absolute errors are very small. To visualize where the errors are occurring, we have scaled the absolute error and mapped the error at each vertex to the torus mesh (Figure 7).

	FX5800	**9800GT**		**FX5800**	**9800GT**
	Max Error	**Max Error**		**Max Error**	**Max Error**
Max Dir	1.406×10^{-6}	4.34×10^{-7}	Deriv 1	9.512×10^{-5}	7.07×10^{-5}
Max Crv	7.391×10^{-6}	3.34×10^{-6}	Deriv 2	3.198×10^{-5}	3.47×10^{-5}
Min Dir	1.281×10^{-6}	4.17×10^{-7}	Deriv 3	2.933×10^{-5}	2.67×10^{-5}
Min Crv	4.053×10^{-6}	2.50×10^{-6}	Deriv 4	4.163×10^{-5}	4.13×10^{-5}
Gauss	1.574×10^{-5}	1.10×10^{-5}	Mean	5.007×10^{-6}	1.91×10^{-6}

Table 3: *Maximum absolute error in the curvature attributes.*

5 Conclusions and Future Work

We have presented a GPU algorithm for estimating curvature in real-time on arbitrary triangular meshes along with results showing real-time performance in a vertex-skinned animation and GPU isosurface extraction system. We demonstrate the use of real-time curvature estimation for approximating ambient occlusion and extracting silhouette and occluding contours.

Including additional types of contours, such as apparent ridges [Judd et al. 2007], is an obvious extension of our work. Because the algorithm outputs line primitives from the geometry shader, it can coexist nicely with the algorithm of Cole et al. [2009] for line visibility and stylization. Finally, the extracted line segments do not have any global parameterization, so stroking stylized lines is another area of future work.

Acknowledgments

Thanks to the reviewers, whose comments improved this paper. Szymon Rusinkiewicz for the trimesh2 library [2009]. Christopher Twigg for the Skinning Mesh Animations data [2009]. Stefan Röttger for the volume data sets [2010]. Lee Perry-Smith for the head scan [2010]. NVIDIA for providing the Quadro FX5800. Maryland Industrial Partnerships (MIPS) for providing support.

References

AKENINE-MÖLLER, T., HAINES, E., AND HOFFMAN, N. 2008. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA.

BAVOIL, L., AND SAINZ, M. 2009. Multi-layer dual-resolution screen-space ambient occlusion. In *SIGGRAPH 2009: Talks*, ACM, 1–1.

BAVOIL, L., SAINZ, M., AND DIMITROV, R. 2008. Image-space horizon-based ambient occlusion. In *SIGGRAPH 2008: Talks*, ACM, 1–1.

COLE, F., AND FINKELSTEIN, A. 2009. Fast high-quality line visibility. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, ACM, 115–120.

DECARLO, D., FINKELSTEIN, A., RUSINKIEWICZ, S., AND SANTELLA, A. 2003. Suggestive contours for conveying shape. *ACM Transactions on Graphics 22*, 3, 848–855.

GATZKE, T. D., AND GRIMM, C. M. 2006. Estimating curvature on triangular meshes. *International Journal of Shape Modeling (IJSM) 12*, 1 (June), 1–28.

GEISS, R. 2007. *GPU Gems 3*. Addison-Wesley Professional, ch. 1: Generating Complex Procedural Terrains Using the GPU, 7–37.

GOLDFEATHER, J., AND INTERRANTE, V. 2004. A novel cubic-order algorithm for approximating principal direction vectors. *ACM Transactions on Graphics 23*, 1, 45–63.

GORLA, G., INTERRANTE, V., AND SAPIRO, G. 2003. Texture synthesis for 3D shape representation. *IEEE Transactions on Visualization and Computer Graphics 9*, 4, 512–524.

HATTORI, T., KUBO, H., AND MORISHIMA, S. 2010. Curvature depended local illumination approximation of ambient occlusion. In *ACM SIGGRAPH 2010 Posters*, ACM, 1–1.

					CPU	GeForce 9800GT (512MB)					Quadro FX5800 (4GB)					GeForce GTX480 (1536MB)				
	Model	Vertices	Faces	RAM	Alg	AO	Lines	S+L	Alg		AO	Lines	S+L	Alg		AO	Lines	S+L	Alg	
				(MB)	(ms)	(ms)	(ms)	(ms)	(ms)	(x)	(ms)	(ms)	(ms)	(ms)	(x)	(ms)	(ms)	(ms)	(ms)	(x)
Anim.	horse	8,431	16,843	26.4	N/A	4.0	5.1	5.1	4.0	N/A	2.3	2.6	2.7	2.3	N/A	0.8	0.9	0.9	0.8	N/A
	elephant	42,321	84,638	107.9	N/A	13.9	17.3	17.4	13.8	N/A	8.1	9.3	9.4	8.0	N/A	2.9	3.3	3.4	2.9	N/A
Non Animated	horse	8,431	16,843	24.3	16.5	4.0	5.0	5.0	4.0	4.1	2.0	2.3	2.3	2.0	8.3	0.8	0.9	0.9	0.8	20.6
	bunny	35,947	69,451	99.1	50.5	14.1	16.0	16.0	14.1	3.6	7.3	8.1	8.2	7.3	6.9	3.4	3.8	3.9	3.3	15.3
	elephant	42,321	84,638	110.0	51.1	13.8	17.2	17.2	13.8	3.7	8.1	9.3	9.4	8.0	6.4	2.9	3.3	3.3	2.8	18.3
	head	131,150	262,284	240.1	166.7	41.3	46.3	46.3	41.2	4.0	22.5	25.1	25.3	22.2	7.5	8.7	10.3	10.3	8.5	19.6
	heptoroid	286,678	573,440	1188.0	325	89.3	101.0	101.3	89.1	3.6	45.1	51.3	52.0	45.1	7.2	18.5	20.5	20.7	18.3	17.8

Table 2: *Frame times and memory usage for various animations and models. The GeForce 9800GT and Quadro FX5800 are at 1600×1200 resolution and the GeForce GTX480 is at 1920×1200 resolution.* ***RAM*** *is for the render targets and geometry buffers (which are independent of resolution).* ***AO*** *is the ambient occlusion approximation.* ***Lines*** *is drawing line primitives.* ***S+L*** *is Lambertian shading with ambient occlusion and drawing lines.* ***Alg*** *is only the curvature algorithm.* ***CPU*** *is a multi-threaded benchmark CPU algorithm running four threads on four cores. The (x) columns show the relative speedup over the CPU algorithm.*

HECKBERT, P., AND GARLAND, M. 1999. Optimal triangulation and quadric-based surface simplification. *Journal of Computational Geometry: Theory and Applications 14*, 1–3, 49–65.

JAMES, D., AND TWIGG, C., 2009. Skinning mesh animations. http://graphics.cs.cmu.edu/projects/sma. November, 2009.

JUDD, T., DURAND, F., AND ADELSON, E. 2007. Apparent ridges for line drawing. *ACM Transactions on Graphics 26*, 3, 19:1–19:7.

KAJALIN, V. 2009. *Shader X^7*. Charles River Media, ch. 6.1: Screen Space Ambient Occlusion, 413–424.

KALOGERAKIS, E., NOWROUZEZAHRAI, D., SIMARI, P., MCCRAE, J., HERTZMANN, A., AND SINGH, K. 2009. Data-driven curvature for real-time line drawing of dynamic scenes. *ACM Transactions on Graphics 28*, 1, 11:1–11:13.

KIM, Y., YU, J., YU, X., AND LEE, S. 2008. Line-art illustration of dynamic and specular surfaces. *ACM Transactions on Graphics 27*, 5, 156:1–156:10.

KIRK, A. G., AND ARIKAN, O. 2007. Real-time ambient occlusion for dynamic character skins. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, ACM, 47–52.

KOLOMENKIN, M., SHIMSHONI, I., AND TAL, A. 2008. Demarcating curves for shape illustration. *ACM Transactions on Graphics 27*, 5, 157:1–157:9.

KONTKANEN, J., AND ALIA, T. 2006. Ambient occlusion for animated characters. In *Proceedings of Eurographics Symposium on Rendering 2006*, T. Akenine-Möller and W. Heidrich, Eds.

KONTKANEN, J., AND LAINE, S. 2005. Ambient occlusion fields. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, ACM, 41–48.

LANDIS, H. 2002. Production-ready global illumination. In *ACM SIGGRAPH 2002 Course #16 Notes*, ACM.

LOOS, B. J., AND SLOAN, P.-P. 2010. Volumetric obscurance. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, 151–156.

LORENSEN, W. E., AND CLINE, H. E. 1987. Marching cubes: A high resolution 3D surface construction algorithm. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, ACM, 163–169.

MCGUIRE, M. 2010. Ambient occlusion volumes. In *Proceedings of High Performance Graphics 2010*.

MEYER, M., DESBRUN, M., SCHRÖDER, P., AND BARR, A. H. 2003. Discrete differential-geometry operators for triangulated 2-manifolds. In *Visualization and Mathematics III*, Springer-Verlag, 35–57.

MITTRING, M. 2007. Finding next gen: Cryengine 2. In *SIGGRAPH 2007: Courses*, ACM, 97–121.

MORETON, H. P., AND SÉQUIN, C. H. 1992. Functional optimization for fair surface design. In *Computer Graphics (Proceedings of SIGGRAPH 92)*, ACM, 167–176.

O'NEILL, B. 2006. *Elementary Differential Geometry*, 2nd ed. Academic Press, Inc.

PERRY-SMITH, L., 2010. Infinite, 3D head scan. http://www.ir-ltd.net/infinite-3d-head-scan-released. October, 2010.

PETITJEAN, S. 2002. A survey of methods for recovering quadrics in triangle meshes. *ACM Computing Surveys 34*, 2, 211–262.

RÖTTGER, S., 2010. The Volume Library. http://www9.informatik.uni-erlangen.de/External/vollib. January, 2010.

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. March, 2009.

SATISH, N., HARRIS, M., AND GARLAND, M. 2009. Designing efficient sorting algorithms for manycore gpus. *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, 1–10.

TAUBIN, G. 1995. Estimating the tensor of curvature of a surface from a polyhedral approximation. In *Computer Vision, Proceedings of the Fifth International Conference on*, 902–907.

THEISEL, H., ROSSI, C., ZAYER, R., AND SEIDEL, H. 2004. Normal based estimation of the curvature tensor for triangular meshes. In *Computer Graphics and Applications Proceedings of the 12th Pacific Conference on*, 288–297.

TONG, W.-S., AND TANG, C.-K. 2005. Robust estimation of adaptive tensors of curvature by tensor voting. *Pattern Analysis and Machine Intelligence, IEEE Transactions on 27*, 3, 434–449.

TREECE, G., PRAGER, R. W., AND GEE, A. H. 1999. Regularised marching tetrahedra: improved iso-surface extraction. *Computers & Graphics 23*, 4 (August), 583–598.

ZHUKOV, S., INOES, A., AND KRONIN, G. 1998. An ambient light illumination model. In *Rendering Techniques '98*, Springer-Verlag Wien New York, G. Drettakis and N. Max, Eds., Eurographics, 45–56.

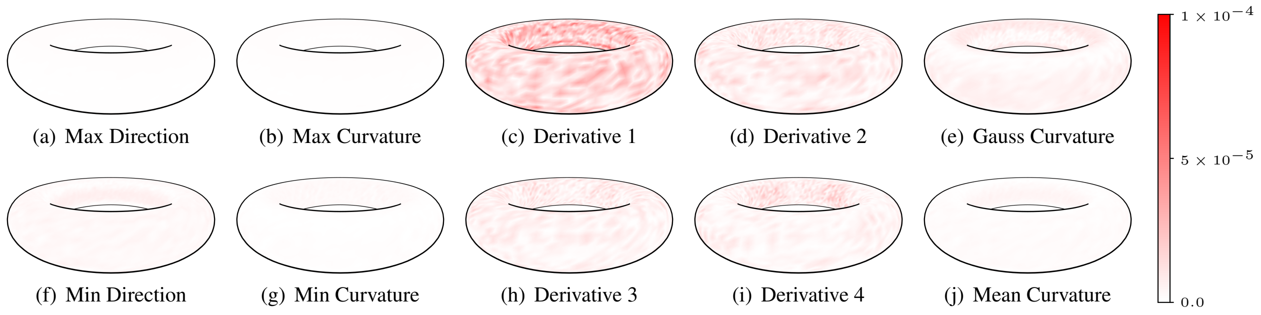

Figure 7: *Scaled absolute error in the curvature attributes on the Quadro FX5800 compared to a CPU estimation ground truth.*

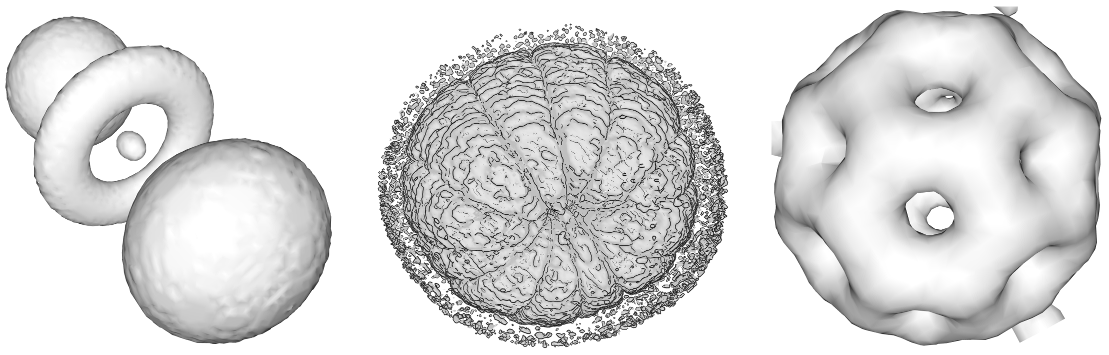

Figure 8: *On the left, a hydrogen atom volumetric data set with Lambertian shading and ambient occlusion. In the middle, is the orange mesh on which our curvature algorithm runs in 39.3 ms (25 fps) and our GPU surface extraction runs in 928 ms. On the right, is a bucky ball with Lambertian shading and ambient occlusion.*

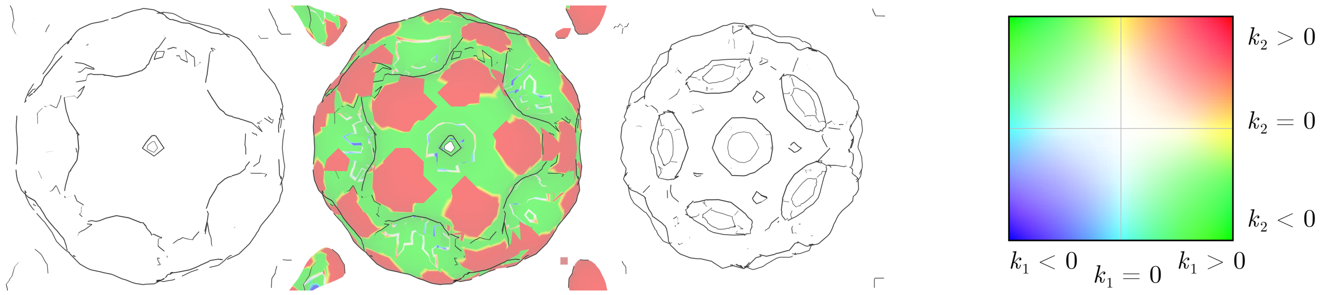

Figure 9: *The bucky ball volume with two iso values and principal curvatures. On the right, is a color map for the visualization of principal curvatures. Blue represents concave areas, red represents convex areas, and green represents saddle-shape areas.*

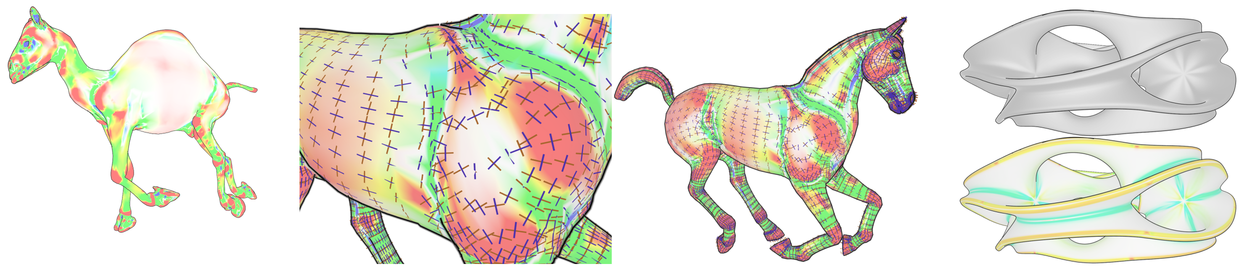

Figure 10: *On the left, the camel model with principal curvatures. In the middle, the horse model showing the principal directions of minimum (blue lines) and maximum (brown lines) curvature. On the right, is the heptoroid model. The top image shows occluding and suggestive contours with Lambertian shading, while the bottom image shows values of principal curvature.*