Real-Time GPU Surface Curvature Estimation on Deforming Meshes and Volumetric Datasets

Wesley Griffin, *Member, IEEE,* Yu Wang, David Berrios, Marc Olano, *Member, IEEE,*

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. We present a graphics pipeline and CUDA implementation. Our curvature estimation is up to $\sim$18$\times$ faster than a multi-threaded CPU benchmark.

Index Terms—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 [1]; mesh simplification [2]; surface modeling [3]; artistic line drawing methods, such as suggestive contours [4], apparent ridges [5], and demarcating curves [6]; and ambient occlusion approximation [7]. These methods operate on a discrete polygonal representation of a continuous surface and therefore must estimate 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 [8] or require a pre-processing step to learn the curvature function with training data [9]. 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 approximation based on Hattori et al. [7], a vertex-blending animation system [10] 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.

This paper extends our prior work [11] with:

- A CUDA implementation of the GPU algorithm for surface curvature estimation.
- Further error analysis and comparison to the CPU algorithm.

- *W. Griffin is with the Department of Computer Science and Electrical Engineering, University of Maryland, Baltimore County, Baltimore, Maryland, 21250.*
  E-mail: griffin5@cs.umbc.edu
- *D. Berrios is with the National Aeronautics and Space Administration.*
- *Y. Wang and M. Olano are with the Department of Computer Science and Electrical Engineering, University of Maryland, Baltimore County.*

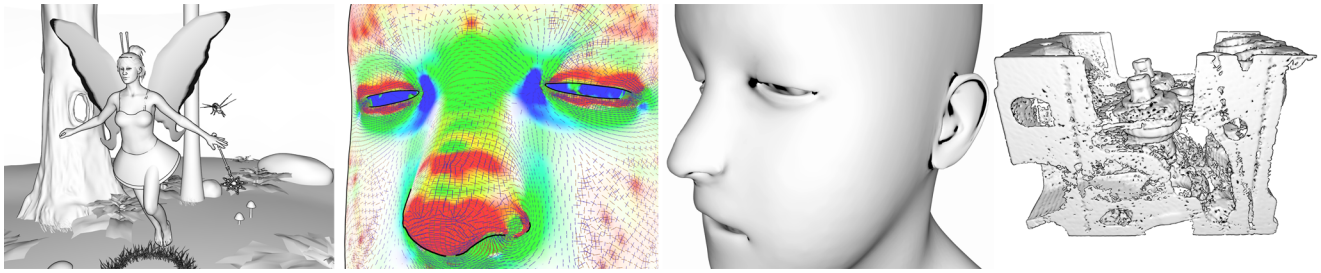


Fig. 1. On the left is one frame of an animation showing suggestive contours and approximate ambient occlusion based on real-time curvature estimation. In the middle is a visualization of the estimated curvature and approximate ambient occlusion. Blue indicates concave areas, red indicates convex areas, and green indicates saddle-shape areas. On the right is an engine volume with ∼2.7 million vertices on which our GPU algorithm estimates curvature in 26.7 ms.

- Additional animated model and volumetric dataset results.

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 [12] for more detail and relevant proofs of theorems.

At a point $\mathbf{p}$ on a 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 a 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 $\mathbf{p}$ [12].

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 a 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 [13]:

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

2.2 Curvature Estimation

Curvature can only be estimated on a discrete surface such as a polygonal model. There has been substantial work in estimating surface curvature [13], [14], [15], [16], [17]. Gatzke and Grimm [18] 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 [13] and Theisel et al. [19] estimate curvature on a triangular mesh at each vertex using differences of surface normals (8) and then average the per-face tensor over each face adjacent 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. [8] 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. [9] 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, first introduced by Cook and Torrance [20]. Zhukov et al. reintroduced AO as a part of ambient obscurance [21]. AO has been extensively researched for use in dynamic environments with static models. The approaches can be divided into screen-space [22], [23], [24], [25] or world-space methods [26], [27], [28].

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 [29] 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 [30] learn a multilinear mapping over subsets of the parameterization space using k-means clustering and principal component analysis.

Building on Hattori et al. [7], 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 (6). We can now analytically solve for AO using this quadric defined by k_1 and k_2 and a unit-radius positive hemisphere:

$$AO(k_1, k_2) = 1 - \frac{1}{2\pi} \int_0^{2\pi} \int_0^{\Theta} sin\widehat{\theta_i} d\widehat{\theta_i} d\widehat{\phi_i} \tag{10}$$

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

$$A = k_1 cos^2 \widehat{\phi_i} + k_2 sin^2 \widehat{\phi_i} \tag{12}$$

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

2.4 Line Drawing

Line drawing algorithms can be divided into two classes: image- 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 [4], apparent ridges [5], and demarcating curves [6] can be further 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 [31] is the most widely used algorithm for computing a triangular surface or model from volumetric data. Treece et al. [32] use a variation of marching cubes, marching tetrahedra, that is easier to implement and handles non-gridded data. Geiss [33] 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 3.4) a CUDA marching cubes algorithm that uses a hash map to fuse vertices.

3 Parallel Algorithm

Our algorithm is based on Rusinkiewicz's CPU algorithm [13]. Rusinkiewicz creates a set of linear constraints to solve for curvature over a single face. The constraints use the differences between normals along edges (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 (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.

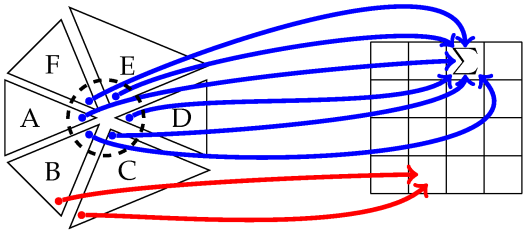


Fig. 2. The contribution of the one-ring neighborhood of a vertex is averaged in the graphics pipeline by blending into a single pixel of a render target. A single vertex with the one-ring of faces is indicated by the dashed circle. The same vertex from each face is mapped as a point primitive to a single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

We have implemented our algorithm in both the graphics pipeline and CUDA. For the graphics pipeline implementation, the per-vertex primitive is implemented in a vertex shader and the per-face primitive is implemented in a geometry shader. We describe the pipeline implementation in detail in Section 3.5. For the CUDA implementation, both computational primitives are individual CUDA kernels and each step of the algorithm is a separate kernel launch. We describe the CUDA implementation in detail in Section 3.6.

3.3 One Ring Computation

The per-face computations that are averaged or sum around the one-ring require both access to the one-ring and write synchronization of some type.

For the graphics pipeline implementation, the per-face geometry shader 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.

Fig. 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

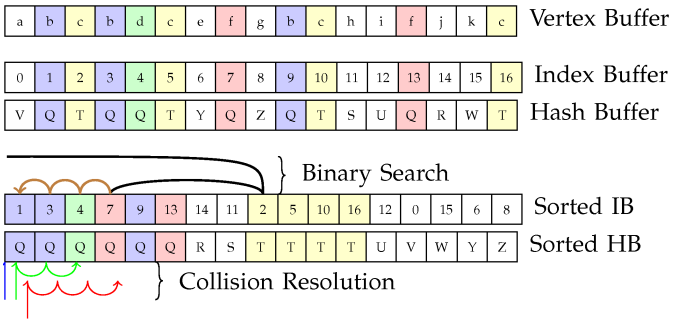


Fig. 3. Vertex, index, and hash buffers before and after radix sorting used in **Isosurface Extraction**. The yellow columns are one set of duplicate vertices. The blue columns are a second set of duplicate vertices. The red columns are a third set of duplicate vertices where the hash collides with the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

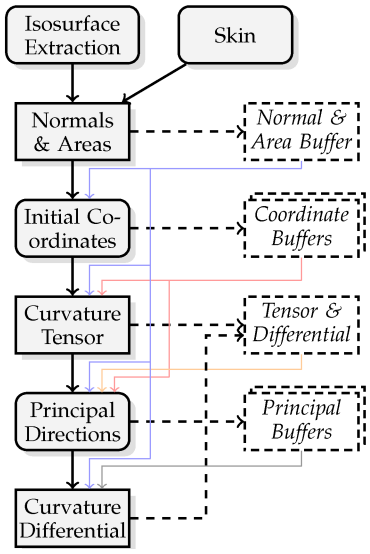


Fig. 4. Our algorithm. See Sections 3.5 and 3.6 for detailed discussion.

Section 3.4 we describe a GPU vertex fusing algorithm applied to GPU marching cubes.

For the CUDA implementation, we use the mapping coordinates from the graphics pipeline implementation. Each kernel runs over a triangle and uses the coordinates to access the render target memory. For synchronization the kernels use floating point atomic add operations.

3.4 Input Transformations

We demonstrate our method on 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 [34]. Fig. 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 Fig. 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 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 Fig. 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.

3.5 GPU Pipeline Curvature Algorithm

Fig. 4 is a flowchart of our parallel curvature algorithm. Each step of the algorithm, shown on the left in Fig. 4, is an instance of one of the computational primitives described in Section 3.1. The steps with square-corners average or sum the output over the one-ring neighborhood, while the steps with round-corners do not. The solid-outline steps are both graphics pipeline passes and CUDA kernels. The dashed-outline steps are CUDA-only kernels. Render targets are shown on the right in italics.

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 [13], vertex area is the Voronoi area described by Meyer et al. [35]. 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 (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 (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.

3.6 CUDA Curvature Algorithm

The flowchart (Fig. 4) is the same for our CUDA implementation except for one additional kernel between the input transformation and the curvature algorithm. The steps are very similar to the graphics pipeline, except that we use *float3* buffers where possible.

Max Triangle ID. The later kernels require a single, constant orthonormal basis at each vertex. We must ensure a consistent basis between frames. This kernel writes the thread ID for each vertex of a triangle using the *atomicMax* function. This results in each vertex having a triangle ID based on the largest thread ID for the set of triangles attached to the vertex.

Normals & Areas. The curvature algorithm requires high quality vertex normals. This kernel recomputes vertex normals and areas for each face. The Voronoi area and face normals are summed into a single *float4* buffer using the floating point *atomicAdd* function.

Initial Coordinates. A constant orthonormal basis is required for each vertex. This kernel computes the basis at each vertex. The kernel only writes the basis based on the ID computed in **Max Triangle ID** to a *float3* buffer.

Curvature Tensor. To estimate the curvature tensor at each vertex, a kernel solves a linear system (8). The weighted tensor is averaged around the one-ring using the floating point *atomicAdd* function and stored in a *float3* buffer.

Principal Directions. This kernel computes minimum and maximum curvatures and principal directions for each vertex of a triangle. The kernel only writes the curvatures and directions based on the ID computed in **Max Triangle ID** to two *float4* buffers.

Curvature Differential. This kernel estimates the derivative of principal curvatures and averages the weighted derivative using the floating point *atomicAdd* function into a *float4* buffer.

3.7 Approximating Ambient Occlusion

One application of estimated curvature is to approximate ambient occlusion. Like Hattori et al. [7], we precompute ambient occlusion (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:

$$AO = 1.0 - 0.0022 * (k_1 + k_2)^2 + 0.0776 * (k_1 + k_2) + 0.7369$$

The ambient term in a lighting equation is then multiplied by the ambient occlusion term AO to darken the ambient illumination. While we pre-computed with a unit-radius hemisphere (11), 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 is algebraically equivalent to decreasing or increasing hemisphere radius.

3.8 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 and Discussion

4.1 Performance

Tables 1 and 2 and Fig. 6 show that our parallel curvature algorithm achieves real-time performance for small-

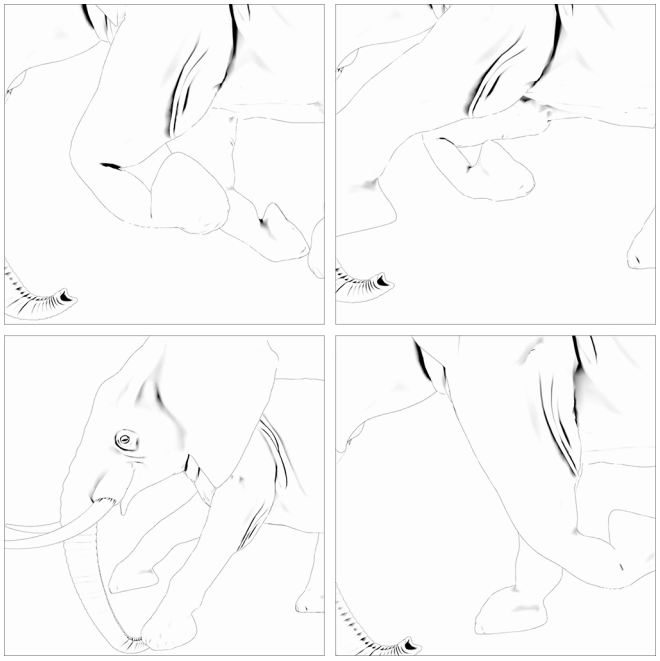


Fig. 5. Four frames of a vertex-blended animation of an elephant. The top row and right column are close-ups of the elephant's front legs. Notice the shadow behind the front knee in the top-left frame disappears in the top-right frame as the leg straightens. In the bottom-right frame the shadow begins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

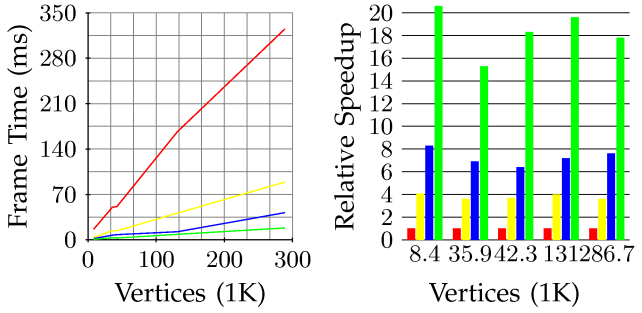


Fig. 6. Frame times and relative speedup for curvature estimation on the non-animated models. Red is a multi-threaded CPU algorithm, blue is our graphics pipeline algorithm on an NVIDIA Quadro FX5800, yellow is our CUDA algorithm and green is our graphics pipeline algorithm, both on an NVIDIA GTX480.

to medium-sized models. Results are shown for two NVIDIA GPUs. The GTX480 results include times for both our graphics pipeline and CUDA implementation. 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 executing on the workstation.

Table 1 shows memory usage (**RAM**), surface extraction (**SE**), and algorithm times for volumetric datasets. All results are at 1600×1200 resolution and use hardware 8× MSAA. Even on a surface with ∼2.7 million vertices, our curvature algorithm runs in 64.0 milliseconds on a Quadro FX5800 and 26.7 milliseconds on a GTX 480. While the actual dataset must fully reside in GPU memory, the **RAM** column in Table 1 shows that the our algorithm can allocate more memory that the GPU has available. Since we allocate several texture and buffers of various sizes but do not access them all at the same time, the GPU is managing the allocations for our algorithm.

Table 2 shows memory usage and frame times for animations and static models. All results are at 1600×1200 resolution and use hardware 8× MSAA. The toss, coin, hand, and fairy models are flipbook animations while the horse and elephant and vertex-skinned animations. For the flipbook animations, the memory usage is much larger compared to the skinned animations, because the vertex, index, and stream-out buffers hold all of the data for every frame.

For the largest animated model, our graphics pipeline curvature estimation algorithm runs in 6.7 milliseconds on the GTX480. The CUDA implementation is drastically slower on the flipbook animations because it naively computes the curvature for every vertex in every frame of the animation.

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 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.

The **CUDA** column in Tables 1 and 2 shows the algorithm time and speedup compared to the CPU benchmark for our CUDA implementation. Because we use floating point *atomicAdd* functions to sum around the one-ring and a pre-pass stage to determine a single triangle per vertex, the running time of the CUDA algorithm is significantly slower than the graphics pipeline algorithm. Additionally, the floating point atomic operations limit our implementation to Compute Model 2.0. We believe these performance results could improve with future hardware, such that the CUDA implementation could be faster than the pipeline implementation.

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. Rusinkiewicz [13] 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.

Table 3 reports the normalized root mean square error for each curvature attribute over the torus mesh using the CPU algorithm as the expected value. As the table

TABLE 1
Frame times for different volumes. **SE** is surface extraction. **AO** is the ambient occlusion approximation. **Lines** is drawing line primitives. **S+L** is Lambertian shading with ambient occlusion and drawing lines. **GP** is just the graphics pipeline curvature algorithm. **CUDA** is just the CUDA curvature algorithm.

			Quadro FX5800 (4GB)					GeForce GTX480 (1536MB)					
Volume	**Vertices**	**RAM**	**SE**	**AO**	**Lines**	**S+L**	**GP**	**SE**	**AO**	**Lines**	**S+L**	**GP**	**CUDA**
		(MB)	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)
bucky	47,208	66.4	32.375	3.4	3.7	3.7	3.3	31.25	0.9	0.9	0.9	0.9	2.2
H atom	113,256	559.2	78.750	4.1	4.6	4.6	4.1	31.25	1.5	1.5	1.5	1.5	3.9
spheres	436,668	1,224.1	284.875	12.8	15.0	15.0	12.8	62.50	4.7	4.9	4.9	4.6	13.0
daisy	620,424	1,274.5	253.906	16.1	18.0	18.0	15.9	108.40	6.5	6.7	6.8	6.5	17.5
orange	1,558,788	4,172.3	924.125	39.4	46.9	46.9	39.3	281.25	15.5	15.7	16.2	15.4	42.5
engine	2,715,000	6,505.8	959.351	64.0	70.5	70.5	64.0	337.89	27.3	27.0	28.1	26.7	69.3

TABLE 2
Frame times and memory usage for animations († is flipbook animations) and models. **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. **GP** is just the graphics pipeline curvature algorithm. **CUDA** is just the CUDA 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.

					CPU	Quadro FX5800 (4GB)					GeForce GTX480 (1536MB)						
	Model	**Vertices**	**Faces**	**RAM**	**Alg**	**AO**	**Lines**	**S+L**	**GP**		**AO**	**Lines**	**S+L**	**GP**		**CUDA**	
				(MB)	(ms)	(ms)	(ms)	(ms)	(ms)	(x)	(ms)	(ms)	(ms)	(ms)	(x)	(ms)	(x)
Animated	toss†	1,370	2,660	173.7	N/A	1.1	1.1	1.1	1.1	N/A	0.4	0.4	0.4	0.4	N/A	11.6	N/A
	coin†	2,146	4,188	544.6	N/A	2.0	2.0	2.0	2.0	N/A	0.6	0.6	0.6	0.6	N/A	30.7	N/A
	horse	8,431	16,843	26.4	N/A	2.3	2.6	2.7	2.3	N/A	0.9	0.9	0.9	0.9	N/A	2.0	N/A
	hand†	9,284	15,853	2086.9	N/A	3.8	3.9	3.9	3.8	N/A	1.0	1.0	1.0	1.0	N/A	111.1	N/A
	elephant	42,321	84,638	107.9	N/A	8.1	9.3	9.4	8.0	N/A	3.3	3.3	3.4	3.3	N/A	6.7	N/A
	fairy†	97,124	173,867	3113.4	N/A	21.7	22.1	22.4	21.7	N/A	6.4	6.4	6.5	6.4	N/A	567.8	N/A
Non Animated	horse	8,431	16,843	23.3	16.5	2.0	2.3	2.3	2.0	8.3	0.9	0.9	0.9	0.9	18.3	1.9	8.7
	bunny	35,947	69,451	94.9	50.5	7.3	8.1	8.2	7.3	7.3	3.8	3.8	3.9	3.8	13.3	8.1	6.2
	elephant	42,321	84,638	104.8	86.3	8.1	9.3	9.4	8.0	10.8	2.9	3.3	3.3	2.8	18.3	6.4	8.0
	head	131,150	262,284	224.1	156.7	22.5	25.1	25.3	22.2	7.0	10.2	9.9	10.4	9.9	15.8	20.6	7.6
	heptoroid	286,678	573,440	1153.0	319.0	45.1	51.3	52.0	45.1	7.1	20.9	20.9	21.1	20.9	15.3	27.0	11.8

TABLE 3
Normalized RMS Error in the curvature attributes.

	FX5800	**GTX480**	**CUDA**
	NRMSE %	**NRMSE %**	**NRMSE %**
Max Crv	1.656×10^{-4}	1.330×10^{-4}	1.286×10^{-4}
Min Crv	1.942×10^{-7}	1.674×10^{-7}	1.647×10^{-7}
Gauss	2.372×10^{-7}	2.004×10^{-7}	1.845×10^{-7}
Mean	4.540×10^{-7}	3.761×10^{-7}	3.797×10^{-7}

shows, both our GPU pipeline and CUDA algorithms have very little error over the CPU algorithm.

Figure 10 shows side-by-side curvature visualization comparisons between our algorithm and the CPU algorithm. Both the elephant and head results look very similiar.

4.3 Limitations and Future Work

A current limitation with our algorithm is that we do not filter the curvature and derivatives of curvature. This results in noisy results in the curvature and derivatives estimations, especially when the mesh has very few vertices. In Figure 9, for example, the horse and camel meshes only have 8,431 and 21,887 vertices respectively, which results in our algorithm estimating curvature over large triangles. Filtering the curvature would make the estimates more smooth, but would be difficult to accomplish without vertex neighborhood information over the meshes.

We also only estimate curvature around the one-ring neighborhood of each vertex and do not support larger or adaptive neighborhoods. Estimating around larger, fixed neighborhoods would be possible in the graphics pipeline algorithm by extending the mapping of vertices to pixels in a render target (Section 3.3) to include additional sets of mapping coordinates. Each set

of coordinates would allow a vertex to affect multiple faces. This would increase the memory usage of the algorithm, as each set of mapping coordinates uses two 32-bit floating point values.

These additional sets of mapping coordinates would be possible to pre-compute on the skinned animations since there is only a single, static pose-mesh for each animation. For the flipbook animations, it would be extremely difficult as the mesh vertex positions are not connected between frames and thus it would be very hard to determine corresponding vertices between frames. For the volume datasets, the surface extraction method would need to generate the neighborhoods.

Adaptive neighborhoods would further complicate the algorithm. For static models, a pre-computation step could be used to choose the neighborhood size. This pre-computation could even work for skinned animations since there is only the single pose-mesh. Whole frame animations again have the issue of no inter-frame vertex connectivity and volume datasets would require further book-keeping in the surface extraction algorithm.

A different limitation is that our algorithm only works on datasets that can reside completely on GPU memory. All of the datasets we test on obviously fit within these constraints. Table 1 indicates that we use more memory than the GPU has for some datasets. We allocate several textures and buffers of varying sizes and the GPU is managing those allocations for the algorithm. The actual buffer containing the data, however, must reside completely on the GPU, as we do not implement any data streaming method.

Including additional types of contours, such as apparent ridges [5], 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. [36] for line visibility and stylization. Additionally, the extracted line segments do not have any global parameterization, so stroking stylized lines is another area of future work.

Our choice of atomic operations is only one possible implementation of the parallel curvature algorithm. A possible different implementation would be to generate the per-face connectivity for each vertex and then use that connectivity information to sum around the one-ring. We believe a similar bottleneck in performance will still exist in generating this connectivity information.

5 Conclusions

We have presented a GPU algorithm for estimating curvature in real-time on arbitrary triangular meshes implemented in both the graphics pipeline and CUDA. We demonstrate real-time performance in a vertex-skinned animation and GPU isosurface extraction system. We show the use of real-time curvature estimation for approximating ambient occlusion and extracting silhouette and occluding contours.

Acknowledgments

Thanks to the many reviewers, whose comments have greatly improved this paper. Szymon Rusinkiewicz for the trimesh2 library [37]. Christopher Twigg for the Skinning Mesh Animations data [38]. Stefan Röttger for the volume data sets [39]. Lee Perry-Smith for the head scan [40]. White et al. for the toss and coin datasets [41]. Ingo Wald for the hand and fairy data sets [42]. NVIDIA for providing the Quadro FX5800. Maryland Industrial Partnerships (MIPS) for providing support.

References

[1] G. Gorla, V. Interrante, and G. Sapiro, "Texture synthesis for 3D shape representation," *IEEE Transactions on Visualization and Computer Graphics*, vol. 9, no. 4, pp. 512–524, 2003.

[2] P. Heckbert and M. Garland, "Optimal triangulation and quadric-based surface simplification," *Journal of Computational Geometry: Theory and Applications*, vol. 14, no. 1–3, pp. 49–65, 1999.

[3] H. P. Moreton and C. H. Séquin, "Functional optimization for fair surface design," in *Computer Graphics (Proceedings of SIGGRAPH 92)*. ACM, 1992, pp. 167–176.

[4] D. DeCarlo, A. Finkelstein, S. Rusinkiewicz, and A. Santella, "Suggestive contours for conveying shape," *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 848–855, 2003.

[5] T. Judd, F. Durand, and E. Adelson, "Apparent ridges for line drawing," *ACM Transactions on Graphics*, vol. 26, no. 3, pp. 19:1–19:7, 2007.

[6] M. Kolomenkin, I. Shimshoni, and A. Tal, "Demarcating curves for shape illustration," *ACM Transactions on Graphics*, vol. 27, no. 5, pp. 157:1–157:9, 2008.

[7] T. Hattori, H. Kubo, and S. Morishima, "Curvature depended local illumination approximation of ambient occlusion," in *ACM SIGGRAPH 2010 Posters*. ACM, 2010, pp. 1–1.

[8] Y. Kim, J. Yu, X. Yu, and S. Lee, "Line-art illustration of dynamic and specular surfaces," *ACM Transactions on Graphics*, vol. 27, no. 5, pp. 156:1–156:10, 2008.

[9] E. Kalogerakis, D. Nowrouzezahrai, P. Simari, J. Mccrae, A. Hertzmann, and K. Singh, "Data-driven curvature for real-time line drawing of dynamic scenes," *ACM Transactions on Graphics*, vol. 28, no. 1, pp. 11:1–11:13, 2009.

[10] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-Time Rendering 3rd Edition*. Natick, MA, USA: A. K. Peters, Ltd., 2008.

[11] W. Griffin, Y. Wang, D. Berrios, and M. Olano, "GPU curvature estimation on deformable meshes," in *Proceedings of the 2011 Symposium on Interactive 3D Graphics and Games*. ACM, 2011, pp. 159–166.

[12] B. O'Neill, *Elementary Differential Geometry*, 2nd ed. Academic Press, Inc., 2006.

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

[14] G. Taubin, "Estimating the tensor of curvature of a surface from a polyhedral approximation," in *Computer Vision, Proceedings of the Fifth International Conference on*, 1995, pp. 902–907.

[15] S. Petitjean, "A survey of methods for recovering quadrics in triangle meshes," *ACM Computing Surveys*, vol. 34, no. 2, pp. 211–262, 2002.

[16] J. Goldfeather and V. Interrante, "A novel cubic-order algorithm for approximating principal direction vectors," *ACM Transactions on Graphics*, vol. 23, no. 1, pp. 45–63, 2004.

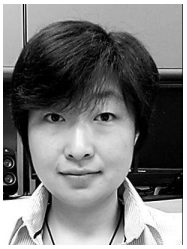
[17] W.-S. Tong and C.-K. Tang, "Robust estimation of adaptive tensors of curvature by tensor voting," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 27, no. 3, pp. 434–449, 2005.

[18] T. D. Gatzke and C. M. Grimm, "Estimating curvature on triangular meshes," *International Journal of Shape Modeling (IJSM)*, vol. 12, no. 1, pp. 1–28, June 2006.

[19] H. Theisel, C. Rossi, R. Zayer, and H. Seidel, "Normal based estimation of the curvature tensor for triangular meshes," in *Computer Graphics and Applications Proceedings of the 12th Pacific Conference on*, 2004, pp. 288–297.

[20] R. L. Cook and K. E. Torrance, "A reflectance model for computer graphics," in *Computer Graphics (Proceedings of SIGGRAPH 81)*. ACM, 1981, pp. 307–316.
[21] S. Zhukov, A. Inoes, and G. Kronin, "An ambient light illumination model," in *Rendering Techniques '98*, ser. Eurographics, G. Drettakis and N. Max, Eds. Springer-Verlag Wien New York, 1998, pp. 45–56.
[22] M. Mittring, "Finding next gen: Cryengine 2," in *SIGGRAPH 2007: Courses*. ACM, 2007, pp. 97–121.
[23] L. Bavoil, M. Sainz, and R. Dimitrov, "Image-space horizon-based ambient occlusion," in *SIGGRAPH 2008: Talks*. ACM, 2008, pp. 1–1.
[24] V. Kajalin, *Shader* X^7. Charles River Media, 2009, ch. 6.1: Screen Space Ambient Occlusion, pp. 413–424.
[25] L. Bavoil and M. Sainz, "Multi-layer dual-resolution screen-space ambient occlusion," in *SIGGRAPH 2009: Talks*. ACM, 2009, pp. 1–1.
[26] J. Kontkanen and S. Laine, "Ambient occlusion fields," in *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*. ACM, 2005, pp. 41–48.
[27] M. McGuire, "Ambient occlusion volumes," in *Proceedings of High Performance Graphics 2010*, June 2010.
[28] B. J. Loos and P.-P. Sloan, "Volumetric obscurance," in *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. ACM, 2010, pp. 151–156.
[29] J. Kontkanen and T. Alia, "Ambient occlusion for animated characters," in *Proceedings of Eurographics Symposium on Rendering 2006*, T. Akenine-Möller and W. Heidrich, Eds., 2006.
[30] A. G. Kirk and O. Arikan, "Real-time ambient occlusion for dynamic character skins," in *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*. ACM, 2007, pp. 47–52.
[31] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3D surface construction algorithm," in *Computer Graphics (Proceedings of SIGGRAPH 87)*. ACM, 1987, pp. 163–169.
[32] G. Treece, R. W. Prager, and A. H. Gee, "Regularised marching tetrahedra: improved iso-surface extraction," *Computers & Graphics*, vol. 23, no. 4, pp. 583–598, August 1999.
[33] R. Geiss, *GPU Gems 3*. Addison-Wesley Professional, 2007, ch. 1: Generating Complex Procedural Terrains Using the GPU, pp. 7–37.
[34] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore gpus," *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, pp. 1–10, 2009.
[35] M. Meyer, M. Desbrun, P. Schröder, and A. H. Barr, "Discrete differential-geometry operators for triangulated 2-manifolds," in *Visualization and Mathematics III*. Springer-Verlag, 2003, pp. 35–57.
[36] F. Cole and A. Finkelstein, "Fast high-quality line visibility," in *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*. ACM, 2009, pp. 115–120.
[37] S. Rusinkiewicz, "trimesh2," 2009, Accessed March, 2009. [Online]. Available: http://www.cs.princeton.edu/gfx/proj/trimesh2
[38] D. James and C. Twigg, "Skinning mesh animations," 2009, Accessed November, 2009. [Online]. Available: http://graphics.cs.cmu.edu/projects/sma
[39] S. Röttger, "The Volume Library," 2010, Accessed January, 2010. [Online]. Available: http://www9.informatik.uni-erlangen.de/External/vollib
[40] L. Perry-Smith, "Infinite, 3D head scan," 2010, Accessed October, 2010. [Online]. Available: http://www.ir-ltd.net/infinite-3d-head-scan-released
[41] R. White, K. Crane, and D. Forsyth, "Capturing and animating occluded cloth," *ACM Transactions on Graphics*, vol. 26, no. 3, pp. 34:1–34:8, 2007.
[42] I. Wald, "Utah 3D animation repository," 2011, Accessed October, 2011. [Online]. Available: http://www.sci.utah.edu/~wald/animrep/

Wesley Griffin received the M.S. in Computer Science from the University of Maryland, Baltimore County in 2010. He is currently a Computer Science Ph.D. student at UMBC working in the Visualization, Animation, Non-photorealistic Graphics, Object modeling, and Graphics Hardware (VANGOGH) Lab. His research interests include real-time graphics and graphics hardware. He is a member of ACM SIGGRAPH, the IEEE and the IEEE Computer Society.



Yu Wang is a PhD student in Computer Science at the University of Maryland, Baltimore County. She is currently a research assistant of the High Performance Computing Facility (HPCF) at UMBC. Her research interests include high performance computing, scientific computation, graphics hardware and graphics techniques for simulating fluid dynamics.

David Berrios received an M.S. in Computer Science from the University of Maryland, Baltimore County in 2007. He is a computer scientist in the Science Data Processing branch at NASA's Goddard Space Flight Center, working with the Community Coordinated Modeling Center. His research interests and responsibilities include science data visualizations and GPU computing.

Marc Olano received the B.S. degree in Electrical Engineering from the University of Illinois in 1990 and the Ph.D. degree in Computer Science from the University of North Carolina in 1998. He is currently an associate professor in the Computer Science and Electrical Engineering department, and Director of the Computer Science Game Development Track at the University of Maryland, Baltimore County. His Ph.D. dissertation described the first interactive graphics hardware shading language. After his Ph.D., Dr. Olano worked at SGI, creating shading languages for commercial graphics hardware. Since 2002, Dr. Olano has been at UMBC, where his research continues to revolve around all aspects of design and use of programmable graphics hardware. He is a member of ACM SIGGRAPH, Eurographics, the IEEE and the IEEE Computer Society.

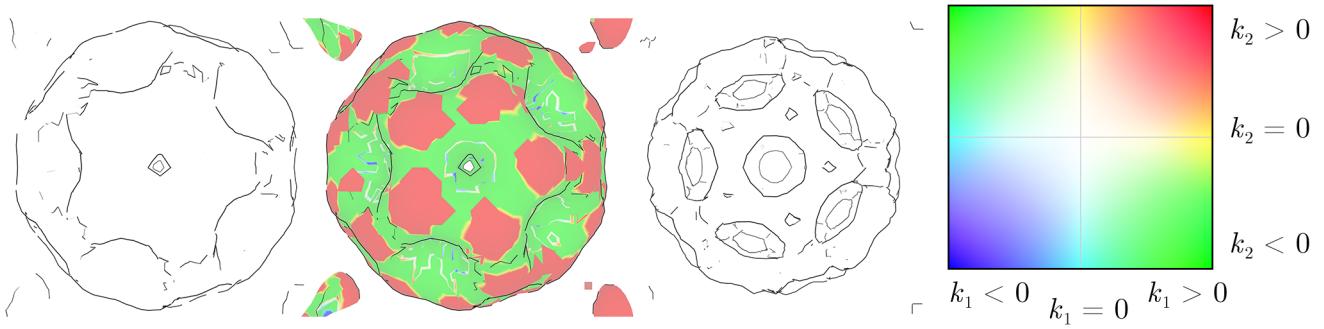


Fig. 7. The bucky ball volume with two iso values and principal curvatures. On the right, is a color map for the visualization of principal curvatures. Blue represents concave areas, red represents convex areas, and green represents saddle-shape areas.

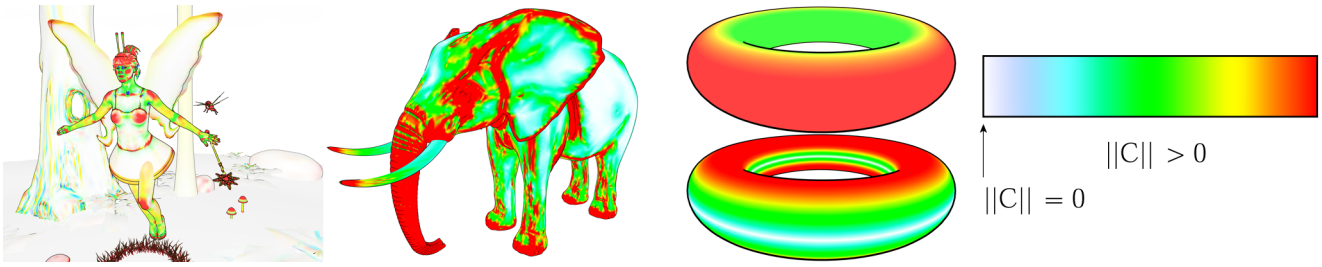


Fig. 8. On the left, curvature visualization on a frame of the fairy model. On the middle-left, a visualization of curvature differential on the elephant. On the middle-right, curvature on the torus (top) and curvature differential on the torus (bottom). On the right, is a color map for the visualization of curvature differential.

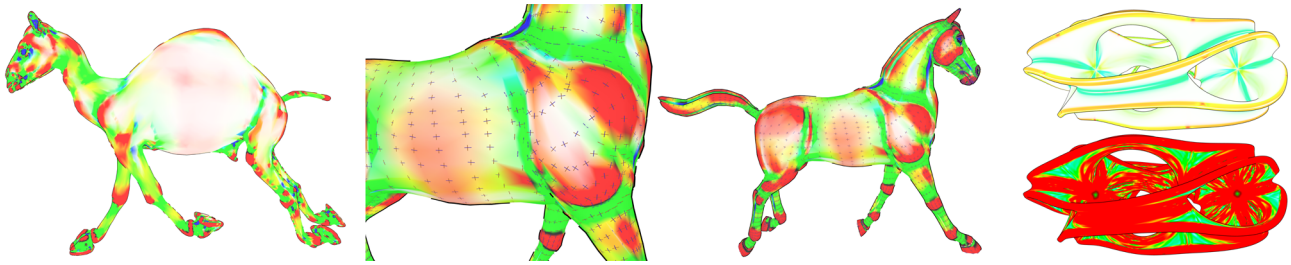


Fig. 9. On the left, the camel model with principal curvatures. In the middle, the horse model showing the principal directions of minimum (blue lines) and maximum (brown lines) curvature. On the right, is the heptoroid model. The top image visualizes curvature and the bottom image visualizes curvature differential.

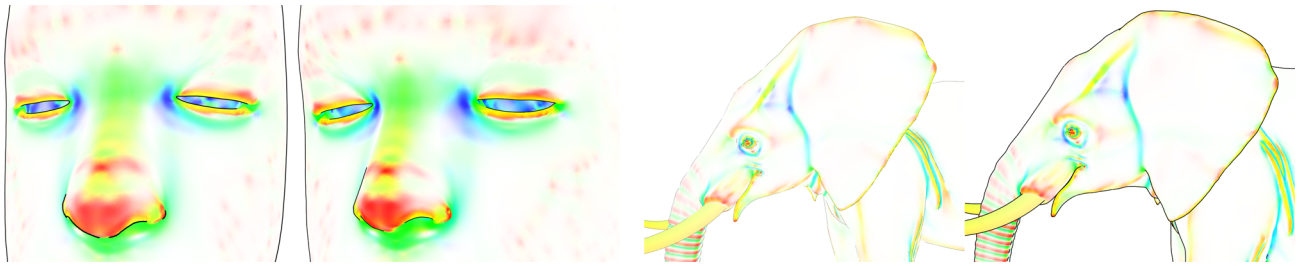


Fig. 10. On the left, the head model in our algorithm (far left) and Rusinkiewicz's algorithm (right). Curvature is visualized as well as principal curvature directions. On the right, the elephant model in our algorithm (left) and Rusinkiewicz's algorithm (far right).

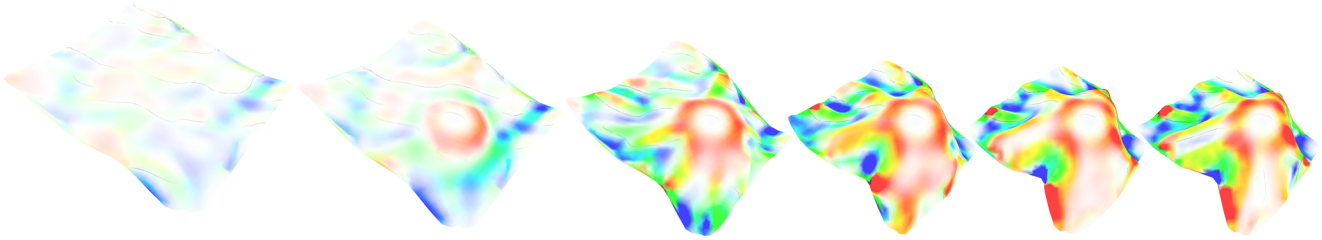


Fig. 11. Five frames from the toss animation with principal curvatures visualized. Notice how the cloth drapes over the object, creating a convex shape, which is reflected in the red areas of curvature. Our GPU algorithm estimates curvature on this animation in 0.6 ms on an NVIDIA GTX480.

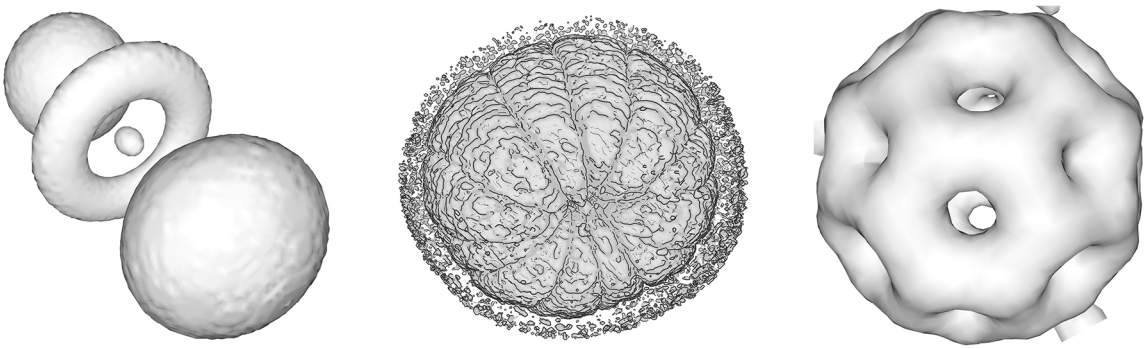


Fig. 12. On the left, a hydrogen atom volumetric data set with Lambertian shading and ambient occlusion. In the middle, is the orange mesh on which our curvature algorithm runs in 15.4 ms and our GPU surface extraction runs in 281.25 ms. On the right, is a bucky ball with Lambertian shading and ambient occlusion.

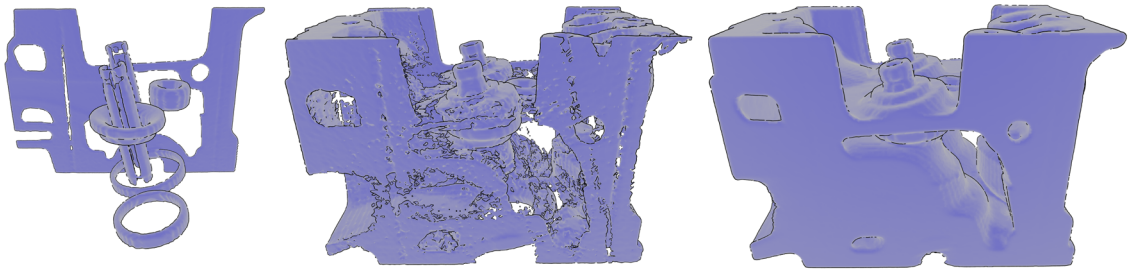


Fig. 13. Increasing iso-value surface extractions of the engine volume dataset. The extracted surface of the middle engine has 2,715,000 vertices and our GPU surface extraction runs in 337.89 ms, while our GPU curvature estimation algorithm runs in 26.7 ms.

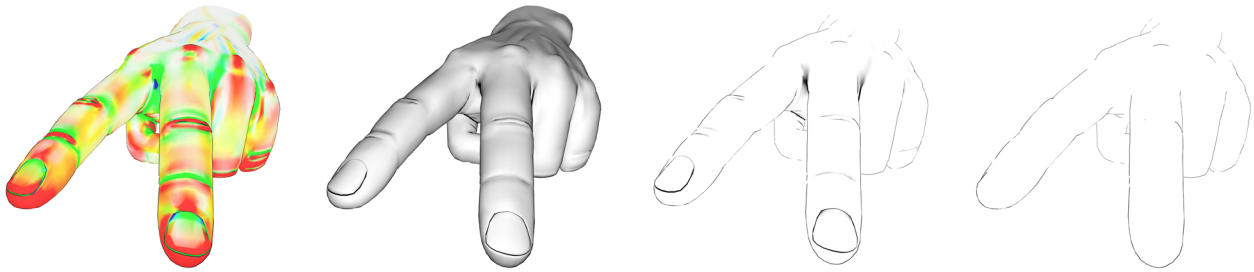


Fig. 14. A single frame from the hand model with four visualizations: curvature, shaded with ambient occlusion, just ambient occlusion, and just suggestive and occluding contours. Observe the shadows between the fingers near the knuckles caused by the ambient occlusion approximation based on curvature estimates.