Interactive 3D Distance Field Computation using Linear Factorization

Avneesh Sud Naga Govindaraju Russell Gayle Dinesh Manocha

{sud,naga,rgayle,dm}@cs.unc.edu
Department of Computer Science
University of North Carolina at Chapel Hill
http://gamma.cs.unc.edu/gvd/linfac

Abstract

We present an interactive algorithm to compute discretized 3D Euclidean distance fields. Given a set of piecewise linear geometric primitives, our algorithm computes the distance field for each slice of a uniform spatial grid. We express the non-linear distance function of each primitive as a dot product of linear factors. The linear terms are efficiently computed using texture mapping hardware. We also improve the performance by using culling techniques that reduce the number of distance function evaluations using bounds on Voronoi regions of the primitives. Our algorithm involves no preprocessing and is able to handle complex deforming models at interactive rates. We have implemented our algorithm on a PC with NVIDIA GeForce 7800 GPU and applied it to models composed of thousands of triangles. We demonstrate its application to medial axis approximation and proximity computations between rigid and deformable models. In practice, our algorithm is more accurate and almost one order of magnitude faster as compared to previous distance computation algorithms that use graphics hardware.

Keywords: distance field, generalized Voronoi diagram, graphics hardware, texture mapping, collision detection, medial-axis transform

1 Introduction

Given a set of geometric primitives, the distance field at a point is the minimum distance from the point to the closest primitive. Distance fields are used for shape representation, proximity computations, morphing, boolean operations, path planning, scientific visualization, etc. In this paper, we address the problem of interactive computation of discretized distance field along a uniform grid. The geometric primitives may correspond to open or close objects, polygonal models or point primitives.

Distance fields are closely related to the concept of Voronoi diagrams. Given a set of primitives, a Voronoi diagram partitions the space into regions, where each region consists of all points that are closer to one primitive than to any other. Many algorithms have been proposed to compute discretized Voronoi diagrams and distance fields along a grid using graphics rasterization hardware [Woo et al. 1997; Hoff et al. 1999; Sigg et al. 2003; Sud et al. 2004]. These algorithms rasterize the distance functions of the geometric primitives and use the depth buffer hardware to compute an approximation of the lower envelope of the distance functions. The algorithms for general polygonal primitives approximate the non-linear distance functions using a distance mesh. This can be expensive for complex models and the accuracy of the overall approach is governed by the tessellation error. As a result, previous techniques are unable to compute 3D distance fields of complex models at interactive rates.

Main Results: In this paper, we present a new algorithm for fast computation of 3D discretized distance fields of a set of convex polygonal primitives using texture mapping hardware. Our algorithm computes the distance field on a 2D slice by rasterizing the distance vectors from the points on the slice to the primitives. We present an elegant geometric formulation to represent the distance vectors at any point on a polygon as bilinear interpolation of the distance vectors of the polygon vertices. This *linear factorization* of distance vectors maps well to the bilinear interpolation capabilities of the texturing hardware. We compute polygons on the 2D slice bounding the Voronoi region for each primitive and express the distance vectors of the polygon vertices using the texture coordinates. We compute the magnitude of the distance vector using a fragment program. We have implemented our algorithm on a Pentium IV PC with a NVIDIA GeForce 7800 GTX GPU and use it for medial axis approximation and proximity queries between deformable models. We have observed up to *one order* of magnitude performance improvement over earlier methods. In practice, our algorithm is able to compute 3D distance fields of complex models composed of thousands of triangles at interactive rates.

Our approach is simple to implement and has many advantages:

> **Generality:** Our algorithm involves no precomputation and is directly applicable to models undergoing non-rigid motion. Moreover, we do not make any assumptions about the shape or mesh topology and our algorithm is applicable to polygon soup models.
>
> **Accuracy:** The linear factorization framework is robust and can compute the distance field up to 32-bit floating point precision on current GPUs. Furthermore, we provide tight bounds on accuracy of the discrete Voronoi diagram.
>
> **Efficiency:** The distance vectors are computed using the bilinear interpolation capabilities of GPUs. As a result, we are able to compute distance fields of complex models consisting of thousands of polygons at interactive rates.

Organization: The rest of the paper is organized as follows. We describe the related work on distance field computations

in Section 2. Section 3 gives an overview of linear factorization and distance vector evaluation. We present conservative bounds on the Voronoi regions of primitives in Section 4 and describe our GPU based distance field computation algorithm in Section 5. We describe its implementation and highlight its performance on medial axis approximation and proximity queries in Section 6. Finally, we analyze our algorithm and compare its performance with prior techniques in Section 7.

2 Related Work

In this section, we briefly survey prior work on distance field computations and GPU-based algorithms to evaluate non-linear functions.

2.1 Distance Fields

Distance field algorithms can be broadly classified based on the model representations such as images, volumes or polygonal representations. For an overview of these algorithms, refer to the surveys [Cuisenaire 1999; Aurenhammer 1991].

Algorithms for image-based data sets perform exact or approximate computations in a local neighborhood of the voxels. Danielsson [1980] used a 2D scanning approach based on the similarity between neighboring pixels. Sethian [1999] proposed the fast marching method to propagate a contour and compute distance transformations. Exact algorithms for handling 2-D and k-D images in different metrics have also been proposed [Breu et al. 1995; Maurer et al. 2003].

There is extensive work in computing the exact Voronoi diagram of a set of points. A good survey of these algorithms is given in [Aurenhammer 1991]. For geometric models represented using polygonal or higher order surfaces in 3D, many algorithms compute an approximation to the Voronoi diagram by computing distance fields on a uniform grid or an adaptive grid. A key issue is the underlying sampling rate used for adaptive subdivision [Vleugels and Overmars 1998; Teichmann and Teller 1997; Etzion and Rappoport 2002]. Many adaptive refinement strategies use trilinear interpolation or curvature information to generate an octree spatial decomposition [Shekhar et al. 1996; Frisken et al. 2000; Perry and Frisken 2001; Varadhan et al. 2003].

The computation of a discrete Voronoi diagram on a uniform grid can be performed efficiently using parallel algorithms implemented on graphics hardware. Hoff et al. [1999] rendered a polygonal approximation of the distance function on depth-buffered graphics hardware and computed the generalized Voronoi Diagrams in two and three dimensions. An efficient extension of the 2-D algorithm for point primitives is proposed in [Denny 2003]. Sud et al. [2004; 2005] presented an approach for efficiently computing k-D Voronoi diagrams of polygonal primitives by culling primitives which do not contribute to the final Voronoi diagram within each slice. A class of exact distance computation and collision detection algorithms based on external Voronoi diagrams are described in [Lin 1993]. A scan-conversion method to compute the 3-D Euclidean distance field in a narrow band around manifold triangle meshes (CSC algorithm) was presented by Mauch [2003]. The CSC algorithm uses the connectivity of the mesh to compute polyhedral bounding volumes for the Voronoi cells. The distance function for each site is evaluated only for the voxels lying inside this polyhedral bounding volume. Sigg et al. [2003] described an efficient GPU based implementation of the CSC algorithm.

2.2 GPU-Based Non-linear Computations

Many algorithms have been proposed to exploit the programmability features of GPUs to evaluate and render higher order functions or surfaces. Bolz and Schroder [**?**] used fragment programs to evaluate the Catmull-Clark subdivision surfaces. Their approach represented the control points in texture memory and used a fragment program to compute bicubic B-spline surfaces. Purcell et al. [2003] presented ray-tracing algorithms by using fragment programs to evaluate ray-primitive intersections. Kanai and Yashui [2004] presented an improved algorithm to compute per-pixel normals on subdivision surfaces. Elegant algorithms to directly render curves and parametric surfaces such as NURBS and T-spline surfaces have also been proposed [Guthe et al. 2005; Loop and Blinn 2005; Shiue et al. 2005]. In contrast with these approaches, our algorithm explicitly decomposes the distance functions into linear factors and uses bilinear interpolation capabilities of the texture mapping hardware to evaluate these functions on a planar domain.

3 Distance Fields

In this section, we give an overview of our approach to compute distance fields using texture mapping hardware. Given a collection of convex polygonal primitives, our algorithm computes the minimum distance along a 3D grid to these geometric primitives. The geometric primitives are classified into points, lines, or triangles and are known as *sites*. In order to compute the distance fields, we define the Euclidean distance functions of these sites. More specifically, the distance function computes the minimum distance of points to a given site. Formally, given a point $\mathbf{p}$ and a site o_i, the distance function $d(\mathbf{p}, o_i) = \min_{\mathbf{q} \in o_i} d(\mathbf{p}, \mathbf{q})$. The distance between the points is computed using the *Euclidean* metric. Given a set of sites $\mathcal{P} = \{o_1, o_2, \ldots, o_n\}$, the 3D distance field at a point $\mathbf{p}$ is given by $D(\mathbf{p}, \mathcal{P}) = \min_{o_i \in \mathcal{P}} d(\mathbf{p}, o_i)$.

We compute distance fields on a 3D grid of points. The 3D distance field is computed by sweeping along the Z axis. For each plane perpendicular to the Z axis, the distance field is computed using the distance from the sites to the plane [Hoff et al. 1999; Sigg et al. 2003; Sud et al. 2004]. The underlying distance function is a degree two function (i.e. a quadric surface in 3D). For example, the Euclidean distance function of a point site to a plane is one sheet of the hyperboloid, and of a line to a plane is an elliptical cone [Hoff et al. 1999].

Given a complex model with tens of thousands of sites, evaluating the non-linear distance function for each site can be expensive. Hoff et al. [1999] computed a piece-wise linear approximation of the distance function using a polygonal distance mesh. However, the linear approximation introduces tessellation error. Moreover, the overhead of computing the polygonal approximation can be high for interactive applications. Sigg et al. [2003] used the programmable capabilities of GPUs to evaluate the non-linear distance function at each point (or pixel) on the plane. They briefly mention use of

bilinear interpolation, and present an approach which uses several instructions per fragment to compute the distance function. Detailed comparisons with their approach is presented in Section 7.

Our Approach: In contrast to earlier approaches, we compute the distance function for each site by evaluating the distance vector field on the GPU. A distance vector field consists of vectors from the 3D points to the closest point on the site. The magnitude of the distance vector provides the value of the distance function of the site at a grid point. We first present a formulation to compute the distance vector at any point on a planar polygon by using the distance vectors of the polygon vertices to the site. Next, we present techniques to compute these planar polygons which bound a site's Voronoi region on a slice. Finally, we map the problem of distance vector computation to texture mapping hardware on the GPU.

3.1 Linear Factorization

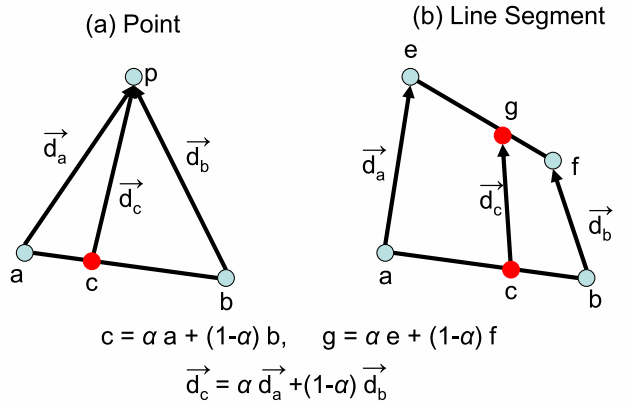

Figure 1: Distance vector computation: This figure illustrates the distance vector computation at any point on a plane. The distance vector at an interior point is a bilinear interpolant of the distance vectors at the vertices.

We use linear factorization of distance vectors to evaluate the distance functions. Formally speaking, the linear factorization expresses the distance vector at each point inside the polygon in terms of bilinear interpolation of the distance vectors of the polygon vertices. Given a convex polygon P with vertices $(v_1, \ldots, v_k)$, the linear factorization expresses the distance vector at any interior point $\mathbf{p}$ of the polygon as

$$\vec{\mathbf{d}}(\mathbf{p}, o_i) = \sum_{i=1}^{k} \alpha_i \vec{\mathbf{d}}(\mathbf{v_i}, o_i),$$

where

$$\mathbf{p} = \sum_{i=}^{k} \alpha_i \mathbf{v_i}, 0 \leq \alpha_i \leq 1 \text{ and } \sum_{i=1}^{k} \alpha_i = 1.$$

We present three key properties of distance vector computation at a point on a plane to point sites, line segment sites, and triangular sites. These properties are used to evaluate the distance functions efficiently. We first highlight the property to perform linear factorization of the distance vector of a point on a line to a point site.

Property 1. *Given two points* $\mathbf{a}$ *and* $\mathbf{b}$ *on a plane and a point site* $\mathbf{p}$*. Let* $\vec{\mathbf{d}_a}$ *and* $\vec{\mathbf{d}_b}$ *denote the distance vectors of* $\mathbf{a}$ *and* $\mathbf{b}$ *to* $\mathbf{p}$ *respectively. Then, the distance vector* $\vec{\mathbf{d}_x}$ *of any point* $\mathbf{x} = \alpha\mathbf{a} + (\mathbf{1}-\alpha)\mathbf{b}, \mathbf{0} \leq \alpha \leq \mathbf{1}$ *is the linear combination of distance vectors of* $\mathbf{a}$ *and* $\mathbf{b}$*, and* $\vec{\mathbf{d}_x} = \alpha\vec{\mathbf{d}_a} + (1-\alpha)\vec{\mathbf{d}_b}$.

Property 1 indicates that given the distance vectors of the vertices of any planar primitive to a point site, the distance vector of any interior point can be computed using a bilinear interpolation of the distance vectors of vertices.

The distance vector of a point $\mathbf{x}$ that projects orthogonally to the interior of a line segment is a vector perpendicular to the line. We use the following property to perform linear factorization of the distance vector of a point that projects onto a line segment.

Property 2. *Given two points* $\mathbf{a}$ *and* $\mathbf{b}$ *on a plane and a line segment* $\mathbf{l}$ *with end points* $\mathbf{e}$ *and* $\mathbf{f}$*. Let* $\vec{\mathbf{d}_a} = \mathbf{e} - \mathbf{a}$ *and* $\vec{\mathbf{d}_b} = \mathbf{f} - \mathbf{b}$ *denote the distance vectors of* $\mathbf{a}$ *and* $\mathbf{b}$ *to the site* $\mathbf{l}$ *respectively. Then, the distance vector* $\vec{\mathbf{d}_x}$ *of any point* $\mathbf{x} = \alpha\mathbf{a} + (\mathbf{1}-\alpha)\mathbf{b}, \mathbf{0} \leq \alpha \leq \mathbf{1}$ *is the linear combination of distance vectors of* $\mathbf{a}$ *and* $\mathbf{b}$*, and* $\vec{\mathbf{d}_x} = \alpha\vec{\mathbf{d}_a} + (1-\alpha)\vec{\mathbf{d}_b}$.

We use property 2 to compute the distance vector of any point that projects onto $\mathbf{l}$. This property indicates that given the vertices of a convex polygon whose projection lies within $\mathbf{l}$, the distance vector of any interior point in the convex polygon is the bilinear interpolation of the distance vectors of the convex polygon vertices to $\mathbf{l}$.

We extend Property 1 and Property 2 to compute the distance vector of a point $\mathbf{x}$ that projects interior to a triangular site. It can be seen that the distance vector of $\mathbf{x}$ is the normal to the triangle.

Property 3. *Given three points* $\mathbf{a}$*,* $\mathbf{b}$ *and* $\mathbf{c}$ *on a plane and a triangular site* $\mathbf{t}$ *with vertices* $\mathbf{e}$*,* $\mathbf{f}$ *and* $\mathbf{g}$*. Let* $\vec{\mathbf{d}_a} = \mathbf{e} - \mathbf{a}$*,* $\vec{\mathbf{d}_b} = \mathbf{f} - \mathbf{b}$ *and* $\vec{\mathbf{d}_c} = \mathbf{g} - \mathbf{c}$ *denote the distance vectors of* $\mathbf{a}$*,* $\mathbf{b}$ *and* $\mathbf{c}$ *to the site* $\mathbf{t}$ *respectively. Then, the distance vector* $\vec{\mathbf{d}_x}$ *of any point* $\mathbf{x} = \alpha_1\mathbf{a} + \alpha_2\mathbf{b} + (\mathbf{1}-\alpha_1-\alpha_2)\mathbf{b}, \mathbf{0} \leq \alpha_1, \alpha_2 \leq \mathbf{1}$ *is the linear combination of distance vectors of* $\mathbf{a}, \mathbf{b}$*, and* $\mathbf{c}$ *and* $\vec{\mathbf{d}_x} = \alpha_1\vec{\mathbf{d}_a} + \alpha_2\vec{\mathbf{d}_b} + (1-\alpha_1-\alpha_2)\vec{\mathbf{d}_c}$.

Property 3 indicates that given the vertices of a convex polygon projecting onto $\mathbf{t}$, the distance vector of any interior point is the bilinear interpolation of the distance vectors of the polygon vertices. Furthermore, the distance vectors of the polygon vertices are normal to the triangular site.

Lemma 1. *There exists a linear factorization to compute the distance vector of any point to a planar site.*

Proof. Trivial. Based on properties 1, 2, and 3. □

4 Domain Computation

In the previous section we showed that the distance vector from a point in the interior of a convex polygon to a site can be expressed as a bilinear interpolant of the distance vectors at the polygon vertices. In this section, we define the convex domain on a slice for which the distance function of a site is computed. We also present an approach to compute conservative bounds on the domain.

The region where the distance function of a site contributes to the distance field is exactly its Voronoi region. However, it is non-trivial to compute the exact Voronoi regions for higher order sites (i.e. lines, polygons) [Culver et al. 1998]. Moreover, the Voronoi regions are not necessarily convex. Instead of computing the exact Voronoi region, we compute a convex polygonal domain $Q_{i,k}$ on the slice s_k, which bounds the intersection of the Voronoi region of site o_i with slice s_k.

We present separate algorithms for manifold and non-manifold sites. We first present an algorithm to compute the $Q_{i,k}$ for non-manifold sites. Later, we present an improved algorithm for manifold sites that exploits the connectivity to compute tighter convex polygonal domain.

Non-Manifold Sites We present the algorithm for a triangle and it can be easily extended to a convex polygon. For a triangle t_j with vertices $\mathbf{p_1}$, $\mathbf{p_2}$, $\mathbf{p_3}$ and unit normal $\mathbf{n_t}$, consider the three (open) half-spaces given by planes through the three edges and perpendicular to the plane of the triangle. The half-spaces are given as $H_i = (\mathbf{x} - \mathbf{p_i}) \cdot \mathbf{n_i} \geq 0, i \in \{1, 2, 3\}$, where $\mathbf{n_i}$ satisfies $(\mathbf{p_i} - \mathbf{p_{i+1}}) \cdot \mathbf{n_i} = 0, \mathbf{n_t} \cdot \mathbf{n_i} = 0$. Any point in the intersection of these halfspaces is closer to the interior of the triangle, and any point not contained in the intersection $H_1 \cap H_2 \cap H_3$ is closer to one of the sites on the boundary of the triangle. The convex polygonal domain $Q_{j,k}$ is the intersection of the three half-spaces with slice s_k.

Given a line segment $\mathbf{e}_j$ with end points $\mathbf{p_1}$ and $\mathbf{p_2}$, consider the two (open) halfspaces defined by planes perpendicular to the line through the end points, $H_1 = (\mathbf{p_2} - \mathbf{p_1}) \cdot (\mathbf{x} - \mathbf{p_1}) > 0$ and $H_2 = (\mathbf{p_1} - \mathbf{p_2}) \cdot (\mathbf{x} - \mathbf{p_2}) > 0$. Any point $\mathbf{x}$ in the intersection of the two half spaces is closer to the interior of the line segment $\mathbf{e}$, and a point not in the intersection will be closer to one of the end points. Thus, the convex polygonal domain $Q_{j,k}$ is the intersection of the two half spaces H_1, H_2 with the slice s_k.

Finally, given a point $\mathbf{p_j}$, the domain $Q_{j,k}$ is the entire slice s_k.

Manifold Sites For a manifold site, we exploit the neighborhood information to compute a convex polytope G_i which bounds the Voronoi region of a site o_i [Culver 2000; Mauch 2003; Sigg et al. 2003]. For a particular slice s_k, the domain of computation of a site o_i is given by the intersection of the polytope with the slice, $Q_{i,k} = G_i \cap s_k$. For a triangle site, the bounding polytope is given by a triangular prism defined by intersection of three half spaces (as described above).

For an edge $\mathbf{e}$ incident on two triangles with normals $\mathbf{n_1}$, $\mathbf{n_2}$, the convex polytope is a wedge obtained by the intersection of four half-spaces. Two of the half-spaces are defined by parallel planes through the end vertices of the edge as shown above. The other two half-spaces are defined by planes containing the edge $\mathbf{e}$ and have normals $\mathbf{n_1}$ and $\mathbf{n_2}$.

For a point site $\mathbf{p}$, with n incident edges $\mathbf{e}_1, \ldots, \mathbf{e}_n$, the polytope $G_\mathbf{p}$ is given by intersection of half-spaces corresponding to planes through the point $\mathbf{q}$ and orthogonal to each incident edge $\mathbf{e}_i$, i.e. $G_\mathbf{p} = \bigcap_{1 \leq i \leq n} H_i, H_i = (\mathbf{x} - \mathbf{p}) \cdot \mathbf{e_i} \geq 0$. Instead of exactly computing the half-space intersection, with time complexity $O(n \log n)$, we present a simple algorithm to compute a conservative approximation of the bounding polytope $G_\mathbf{p}$ for a point site $\mathbf{p}$ in $O(n)$ time.

A point site is defined as convex iff all edges incident on the point have an internal dihedral angle less than π. Let $\mathbf{p}$ be a convex point, with incident edges $\mathbf{e}_i$, $1 \leq i \leq n$ in order. Then edges of the convex polytope $G_\mathbf{p}$ are given by $\mathbf{p} + \lambda \mathbf{n_i}$, where $n_i = \mathbf{e}_i \times \mathbf{e}_{i+1}$ (modulo n) and $\mathbf{n_i}$ is the normal of the triangle t_i containing edges $\mathbf{e}_i$ and $\mathbf{e}_{i+1}$.

This construction does not work for hyperbolic points [Peikert and Sigg 2005]. Previous approaches expand the bounding polytopes of adjacent triangles to handle hyperbolic points, however this results in a complex fragment program to compute the distance functions [Sigg et al. 2003]. We present an efficient algorithm to compute a bounding polytope $G_\mathbf{p}$ for a hyperbolic point $\mathbf{p}$ (see figure 2(a)). Let $\mathbf{n_a}$ be the average of the normals of all incident triangles. Let $\mathbf{n_j}$ be the normal which maximizes $\theta(i) = \frac{\mathbf{n_a} \times \mathbf{n_i}}{|\mathbf{n_a} \times \mathbf{n_i}|}, i = 1, \ldots, n$. We consider the case when $\theta(i) < \pi/2$. Let C be a right circular cone with axis $\mathbf{n_a}$ and opening angle $2\theta(j)$. We now prove that the bounding polytope $G_\mathbf{p}$ is a subset of cone C.

Theorem 1. *Let $\mathbf{p}$ be a manifold point, and C be a cone constructed as above. Then the convex polytope $G_\mathbf{p}$ bounding the Voronoi region of $\mathbf{p}$ is a subset of C.*

Proof. If $G_\mathbf{p} = \emptyset$, then the result trivially holds. It is sufficient to show that $G_\mathbf{p} \cap \pi_k \subseteq C \cap \pi_k$ for any plane π_k. Consider a plane π orthogonal to $\mathbf{n_a}$, and let Q be the convex polygon obtained by intersection of $G_\mathbf{p}$ with π, $Q = G_\mathbf{p} \cap \pi$. Let $\mathbf{x_i}$ be the intersection of the ray from $\mathbf{p}$ along direction $\mathbf{n_i}$. $G_\mathbf{p} \cap \pi$ is the convex region given by the intersection of 2D half spaces, each half space is given by the line through $\mathbf{x_i}$ and $\mathbf{x_{i+1}}$ (modulo n) (see figure 2(b)). For any point $\mathbf{y} \in \pi$, let $r = \max_{i=1,\ldots,n} d(\mathbf{x_i}, \mathbf{y})$. By construction, a circle c with center $\mathbf{y}$ and radius r will contain Q. Taking $\mathbf{y} = \mathbf{x_a}$, we get $c = C \cap \pi$ and $G_\mathbf{p} \subseteq C$. □

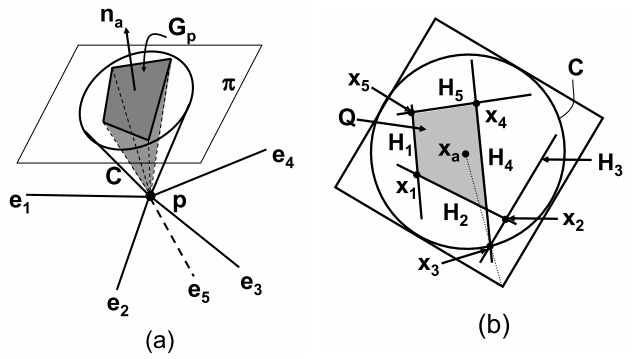

Figure 2: Bounding polytope computation for a hyperbolic point: $\mathbf{p}$ is hyperbolic point with 5 incident edges $\mathbf{e}_i$, $i = 1, \ldots, 5$. (a) The polytope $G_\mathbf{p}$ bounds the Voronoi region of $\mathbf{p}$. $G_\mathbf{p}$ is bounded by a cone C. (b) Intersection of $G_\mathbf{p}$ with plane π, showing construction of C.

Theorem 1 implies that any convex polytope containing the cone C will bound the bounding polytope $G_\mathbf{p}$ of a hyperbolic point $\mathbf{p}$. We use a square pyramid to approximate the bounding polytope as shown in figure. If $\theta(i) \geq \pi/2$, then we treat the hyperbolic point as non-manifold.

5 Distance Field Computation using GPUs

In this section, we present our algorithm for distance computation on the GPU using linear factorization. Given a site and a slice, the convex domain is computed as described

in Section 4. The distance vector to the site is computed at each vertex of the convex polygon. The distance vector is encoded as a 3D texture coordinate, and the polygon is rasterized on the GPU.

Graphics processors (GPUs) have many specialized hardware units to perform bilinear interpolation on the attributes of vertices of polygons. These vertex attributes consist of the color, position, normal, or texture co-ordinates of the vertices. These attributes are 4-D vectors of the form (v_x, v_y, v_z, v_w) and transformations are applied to the attributes in the vertex processing unit. The transformed vertices are bilinearly interpolated by the rasterization hardware and the interpolated vectors at each fragment are used for lighting computations using Gouraud or Phong shading, or environment mapping applications. In order to achieve higher performance, the rasterization hardware consists of multiple vector units to compute the interpolated vectors.

We use the fast bilinear interpolation capabilities of GPUs for distance vector computation. Based on the linear factorization formulation, we map the distance vector computation to the GPUs using vertex attributes of the polygons. We use orthographic transformations and perform a one-to-one mapping between the points on the plane and the pixels on the screen. The bilinear interpolation of the vertex attributes is used for distance vector computation at each point on the plane.

The interpolation units in current GPUs can perform computations on different vector representations. For example, the vectors can be represented using 8-bit, 16-bit or 32-bit floating point values and the vector components may be signed or unsigned. Since the components of the distance vectors are signed and represented in 32-bit floating point precision, we use the 3D texture co-ordinates of the vertices to represent the distance vectors of vertices. The interpolated texture coordinates are used in a single instruction fragment program to compute the magnitude of the distance vector. The distance field is updated to compute the minimum either using the MIN instruction in the fragment program or by using the depth test functionality.

We now present the expressions for computing the distance vectors from a convex polygon vertex to a point, an edge and a triangle. These are:

Point: Given a point site $\mathbf{p}$ and a vertex $\mathbf{v}$, the distance vector from $\mathbf{v}$ to $\mathbf{p}$ is $\vec{\mathbf{d}}(\mathbf{p}, \mathbf{v}) = \mathbf{p} - \mathbf{v}$.

Edge: For an edge $\mathbf{e}$ with end points $\mathbf{p_1}$ and $\mathbf{p_2}$, the distance vector from a vertex $\mathbf{v}$ to $\mathbf{e}$ is $\vec{\mathbf{d}}(\mathbf{e}, \mathbf{v}) = (\mathbf{p_1} - \mathbf{v}) + \lambda \frac{(\mathbf{p_2}-\mathbf{p_1})}{|\mathbf{p_2}-\mathbf{p_1}|}$, where $\lambda = \frac{(\mathbf{v}-\mathbf{p_1})\cdot(\mathbf{p_2}-\mathbf{p_1})}{|\mathbf{p_2}-\mathbf{p_1}|}$.

Triangle: Given a triangle t with a unit normal $\hat{\mathbf{n}}$, the distance vector from a vertex $\mathbf{v}$ to t is given as $\vec{\mathbf{d}}(t, \mathbf{v}) = [(\mathbf{p_i} - \mathbf{v}) \cdot \hat{\mathbf{n}}]\, \hat{\mathbf{n}}$, where $\mathbf{p_i}$ is one of the three vertices of the t.

The pseudocode for computing the distance field for a slice s_k is presented in Algorithm 1. The function *ComputePolygon*(o_i,s_k) computes the convex polygonal domain bounding the Voronoi region of site o_i on slice s_k.

The algorithm for computing the distance field in the entire domain M is presented in Algorithm 2. The function *SetOrthoProjection*(M) sets up the projection matrix to be the bounds of the (axis-aligned) domain of computation M. *NormProgram*() is a single instruction fragment program

```
Input: slice s_k, site set P
Output: distance field D(s_k, P)
foreach site o_i ∈ P do
    Q_{i,k} ← ComputePolygon(o_i, s_k)
    foreach vertex v ∈ Q_{i,k} do
        Compute distance vector d(o_i, v)
        Assign texture coordinates of v,
        (r, s, t) ← d(o_i, v)
    end
    Draw textured polygon Q_{i,k} at depth z = 0
end
```

Algorithm 1: ComputeSlice(s_k, $\mathcal{P}$): This algorithm computes the distance field for s_k for a set of primitives $\mathcal{P}$

that computes the Euclidean norm of the 3D texture coordinate at a pixel and writes it out the value to the depth buffer. The functions *StartSlice*(s_k) and *EndSlice*(s_k) setup the rendering state at the beginning and end of computation of a given slice s_k. The state setup involves enabling a floating point rendering buffer, clearing the buffer and reading it back to the CPU after computation.

```
Input: site set P, domain M, number of slices m
Output: distance field D(M, P)
Enable depth test
Set depth test function to less than
SetOrthoProjection (M)
Enable fragment program NormProgram
foreach slice s_k, k = 1, ..., m do
    StartSlice (s_k)
    ComputeSlice (s_k,P)
    EndSlice (s_k)
end
Disable fragment program NormProgram
```

Algorithm 2: Compute3D($\mathcal{P}$, M, m): Computes the 3D discretized distance field on a uniform grid M with m slices.

6 Implementation and Applications

In this section, we describe our implementation and highlight its application to medial axis approximation and proximity queries between deformable models.

6.1 Implementation

We have implemented our algorithm on a PC with a 3.2 GHz Pentium IV CPU with 2 GB memorywith an NVIDIA GeForce 7800 GPU connected via 16x PCI Express bus and running Windows XP operating system. We used OpenGL as the graphics API and used $ARB\_fragment\_program$ for implementing the fragment program. The fragment program is used to compute the magnitude of the distance vector. We computed the distance field using a tiled render texture with 32-bit floating point precision. Our run-time algorithm computed the distance vectors for the vertices of the convex polygon associated with each site. The polygons are rasterized onto the rendertexture. We incorporated the optimizations to improve the performance of our algorithm on

Model	Polygon Count	Distance Field Resolution	HAVOC Time (in sec)	Lin.Fac. Time (in sec)
Triceratops	5.6K	$256 \times 112 \times 84$	3.87	0.768
Head	22K	$78 \times 105 \times 128$	17.47	0.846
Gargoyle	11K	$256 \times 103 \times 91$	10.2	0.625
Bunny	1.5K	$256 \times 252 \times 196$	4.8	0.772
Shell Charge	4.5K	$126 \times 128 \times 126$	2.8	0.276

Table 1: This table highlights the performance of our algorithm (Lin.Fac.) and HAVOC [Hoff et al. 1999] on different polygonal models. We observe $5-10$ times speedup times on a Pentium IV PC with NVIDIA GeForce 7800 GPU.

Figure 3: Separation Distance Computation between deforming bunnies: Average query time=120ms, grid resolution= 256^3.

manifold objects. There are no precomputations and the algorithm is directly applicable to deforming models and dynamic environments. The performance of our system varies as a function of model complexity and grid resolution.

Our approach takes a fraction of a second to compute the distance field of a model with thousands of polygons on a $256 \times 256 \times 256$ grid. The performance of our algorithm to compute the global distance field for various models is highlighted in Table 1. We analyzed our implementation using Intel's vTune benchmarking software. The time spent on computation of the bounding convex polygons was approximately 12% of the total time. The observed maximum number of triangles sent to GPU was 3MTris/s; maximum number of pixels rendered was 1.6Gpixels/s and estimated memory bandwidth achieved was 26GB/s.

6.2 Proximity Computation

We have applied our algorithm to compute proximity information among deformable models. These include different proximity queries including collision detection, separation distance and penetration depth estimation, and contact normals computation. The penetration distance is defined as the minimum translation distance required to separate two overlapping objects. We first compute the distance field in the localized region of overlap between two oriented objects, O_1 and O_2. Next, we use the stencil test to compute the points on O_1 interior to O_2. These computations are performed on each slice of the distance field, and the maximum stored value at these interior points on the slices of the distance field is used to approximate the penetration depth. This test is repeated by swapping O_1 and O_2. The separation distance between two objects is the minimum Euclidean distance from the points on one object to the other object. Given two objects, we first compute the closest point P_1 on O_1 using the distance field of O_2, and the closest point P_2 on O_2 by using the distance field of O_1. Our algorithm computes the separation distance as the distance between P_1 and P_2.

Our overall proximity computation algorithm is based on PIVOT [Hoff et al. 2001] that uses a combination of object space location and distance field computation along a grid. We compute the collision response between the overlapping objects based on a constraint-based approach.

We used our algorithm for interactive proximity query on a sequence of deforming bunnies as shown in Figure C.1 (color plate). Each bunny consists of $2K$ triangles and undergoes deformation based on a mass-spring system. At each frame, we perform proximity queries using a grid resolution of 256^3. The distance computation is only performed on the localized regions computed using bounding box of each bunny. The average time to perform each proximity query is 120ms. As compared to HAVOC and PIVOT [Hoff et al. 2001], our new distance computation algorithm results in a speedup of $7-12$ times (as shown in Figure 4).

6.3 Medial Axis Approximation

We use discretized distance fields to compute an approximate medial axis. Given the polygonal boundary of a solid model, we compute the distance field and its gradient, and use them to extract the simplified medial axis [Foskey et al. 2003]. Our algorithm computes a *neighbor direction field* of the object and applies a user-specified separation angle criterion. The separation angle criterion acts like a low-pass filter and is used to eliminate noisy features in the medial axis. Our algorithm traverses through the voxels and uses the neighbor direction field to decide which voxels lie on the medial axis. The selection operation is also performed using a fragment program and the gradient field is stored in the texture memory. In Figure C.2 (color plate), we demonstrate an application of our algorithm to compute the approximate medial axis of the Gargoyle model with 11K triangles on a $256 \times 103 \times 91$ grid. Moreover, we observed 16 times speedup over HAVOC-based medial axis approximation algorithm [Foskey et al. 2003].

7 Comparison and Limitations

We have used our algorithm to compute distance fields of complex models consisting of thousands of polygons. Our algorithm is able to compute 3D discretized fields at interactive rates (i.e. $3-12$ frames per second) for 256^3 grid on complex models. Our initial benchmarks indicate that our algorithm is almost one order of magnitude faster as compared to earlier algorithms including HAVOC [Hoff et al. 1999], DiFi [Sud et al. 2004] We present qualitative comparisons with HAVOC and DiFi, and and GPU-based implementation of CSC algorithm [Sigg et al. 2003].

HAVOC and DiFi: These approaches compute a polygonal approximation of the non-linear distance function of each site. For large models, the distance mesh approximation step becomes a bottleneck. Furthermore, all the triangles are sent from the CPU to the GPU during each frame for rasterization. In contrast, our algorithm only computes the distance vector at the vertices of the convex polygons and uses the bilinear interpolation capabilities of the texture mapping hardware. Our distance computation algorithm has much lower CPU-GPU bandwidth requirements as we only transmit the distance vectors at the vertices of

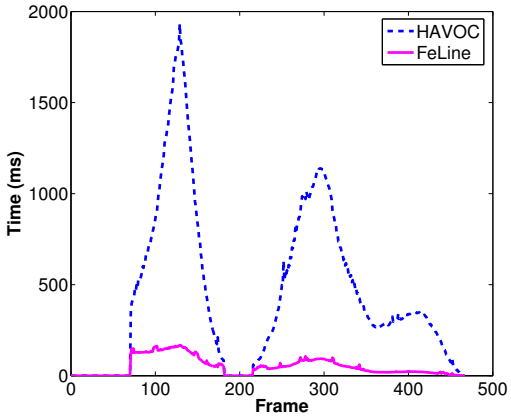

Figure 4: This graph compares the performance of our algorithm (Lin.Fac.) with HAVOC [Hoff et al. 1999] for proximity computation. We observe a speedup of 7-12 times on the deforming bunny simulation.

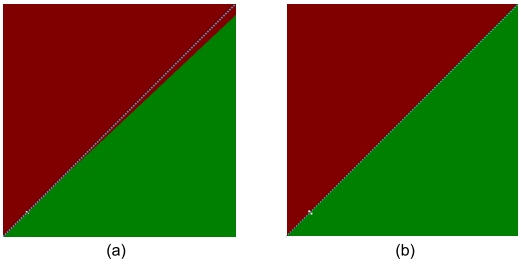

Figure 6: Improved accuracy of our algorithm: This figure highlights the error in computing the distance field and discrete Voronoi diagram using (a) HAVOC and (b) our algorithm. The exact Voronoi boundary is indicated by the dotted blue line. HAVOC introduces tessellation error due to polygonization of distance functions.

the convex polygon of each site. Our bounds on the convex polygonal domain also reduce the fill overhead during rasterization.

GPU-based CSC algorithm: Sigg et al. [2003] also mentioned the idea of using bilinear interpolation and dot products. However, they do not provide any details on their derivation or implementation. Instead they present an approach which reduces the number of polyhedra that are scan converted. The fragment program used by [Sigg et al. 2003] is more complex and increases the load on the fragment processor. Moreover, Sigg et al.'s algorithm is restricted to inputs that are closed manifolds. Further, it does not provide the complete generalized Voronoi diagram. Overall, their approach is useful for very highly tessellated models and distance field computations with low-grid resolutions and narrow band sizes. In these cases, each polyhedron can become smaller than a few voxels. The polygon transform can become a bottleneck, and reducing number of polyhedra scan-converted provides speedups.

In contrast, we provide a formal presentation of the linear factorization and the necessary details to implement it. Our algorithm is applicable to general polygonal models. Furthermore, our approach computes a pixel accurate discrete generalized Voronoi diagram (as demonstrated in Figure 6). Our fragment program is much simpler. For large grid resolutions and global computations, the distance computation

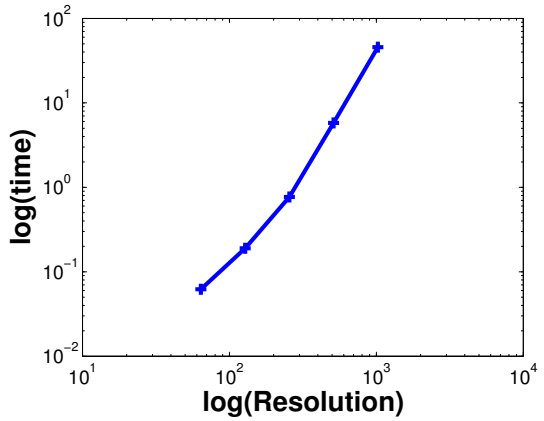

Figure 5: This log-log graph demonstrates the performance of our distance computation algorithm as a function of grid resolution. As the grid resolution increases, the running time is nearly a linear function.

on the fragment processor becomes the bottleneck and our approach is more efficient that [Sigg et al. 2003]. This has been verified by our benchmarks. The cost of slicing the polyhedra is small and the observed triangle transfer rates are significantly less than the theoretical peak. Further, our approach makes efficient use of the fragment processor.

Limitations: Our approach has some limitations. Firstly, the accuracy of the algorithm is governed by that of the graphics hardware. For example, the current hardware provides support for 32-bit floating point representation and it is not fully compatible with the IEEE floating-point standard. Secondly, our algorithm involves a readback from the GPU back to the CPU, which can have additional overhead for high resolution distance fields. Our algorithm is only useful for computing discretized distance fields and the resulting algorithms for proximity queries and medial-axis computation only perform approximate computations (up to grid resolution). For narrow bands, and highly tessellated models, polygon transformation can become a bottleneck.

8 Conclusions and Future Work

We present a new algorithm to compute discretized 3D distance fields for polygonal models. Our algorithm decomposes the distance function of each primitive into linear factors and evaluates the linear terms using bilinear interpolation capabilities of texture mapping hardware. We also present techniques to bound the Voronoi region of each primitive and evaluate the distance field within the bounded region. The accuracy of our algorithm is governed by that of the bilinear interpolation hardware, i.e. 32-bit floating point on current GPUs. In practice, our algorithm is able to compute distance fields of models composed of thousands of polygons at $3-10$ frames a second on a 256^3 grid. We have used distance field computation for collision and penetration depth computation and medial axis approximation of deforming models.

There are many avenues for future work. Our current work is limited to Euclidean distance functions and it would be useful to extend it to other distance metrics, (e.g. L_k norm). We would like to incorporate other culling techniques that

utilize spatial coherence between successive slices to compute the bounding regions for each site. The linear factorization based formulation could also be extended to higher order primitives including splines or algebraic surfaces. Moreover, we would like to perform distance field computation on adaptive grids and use our algorithm for other applications including morphing, motion planning and navigation, and simulation.

Acknowledgments

This research is supported in part by ARO Contract DAAD 19-02-1-0390, and W911NF-04-1-0088, NSF Awards 0400134, 0118743, DARPA and RDECOM Contract N61339-04-C-0043 and Intel Corporation. We thank the UNC GAMMA group for many useful discussions and support. We are also grateful to the reviewers for their feedback.

References

Aurenhammer, F. 1991. Voronoi diagrams: A survey of a fundamental geometric data structure. *ACM Comput. Surv. 23*, 3 (Sept.), 345–405.

Bolz, J., Farmer, I., Grinspun, E., and Schröder, P. 2003. Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. *ACM Trans. on Graphics (Proc. of ACM SIGGRAPH) 22*, 3.

Breu, H., Gil, J., Kirkpatrick, D., and Werman, M. 1995. Linear time Euclidean distance transform and Voronoi diagram algorithms. *IEEE Trans. Pattern Anal. Mach. Intell. 17*, 529–533.

Cuisenaire, O. 1999. *Distance Transformations: Fast Algorithms and Applications to Medical Image Processing*. PhD thesis, Universite Catholique de Louvain.

Culver, T., Keyser, J., and Manocha, D. 1998. Accurate computation of the medial axis of a polyhedron. Tech. Rep. TR98-034, Department of Computer Science, University of North Carolina. Appeared in Proceedings of ACM Solid Modeling 99.

Culver, T. 2000. *Accurate Computation of the Medial Axis of a Polyhedron*. PhD thesis, Department of Computer Science, University of North Carolina at Chapel Hill.

Danielsson, P. E. 1980. Euclidean distance mapping. *Computer Graphics and Image Processing 14*, 227–248.

Denny, M. 2003. Solving geometric optimization problems using graphics hardware. *Computer Graphics Forum 22*, 3.

Etzion, M., and Rappoport, A. 2002. Computing Voronoi skeletons of a 3-d polyhedron by space subdivision. *Computational Geometry: Theory and Applications 21*, 3 (March), 87–120.

Foskey, M., Lin, M., and Manocha, D. 2003. Efficient computation of a simplified medial axis. *Proc. of ACM Solid Modeling*, 96–107.

Frisken, S., Perry, R., Rockwood, A., and Jones, R. 2000. Adaptively sampled distance fields: A general representation of shapes for computer graphics. In *Proc. of ACM SIGGRAPH*, 249–254.

Guthe, M., Balazs, A., and Klein, R. 2005. Gpu-based trimming and tessellation of nurbs and t-spline surfaces. *ACM Trans. Graph. 24*, 3, 1016–1023.

Hoff, III, K. E., Culver, T., Keyser, J., Lin, M., and Manocha, D. 1999. Fast computation of generalized Voronoi diagrams using graphics hardware. In *Computer Graphics Annual Conference Series (SIGGRAPH '99)*, 277–286.

Hoff, K., Zaferakis, A., Lin, M., and Manocha, D. 2001. Fast and simple 2d geometric proximity queries using graphics hardware. *Proc. of ACM Symposium on Interactive 3D Graphics*, 145–148.

Kanai, T., and Yasui, Y. 2004. Per-pixel evaluation of parametric surfaces on gpu. *ACM Workshop on General Purpose Computing on Graphics Processors*.

Lin, M. 1993. *Efficient Collision Detection for Animation and Robotics*. PhD thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley.

Loop, C., and Blinn, J. 2005. Resolution independent curve rendering using programmable graphics hardware. *ACM Trans. Graph. 24*, 3, 1000–1009.

Mauch, S. 2003. *Efficient Algorithms for Solving Static Hamilton-Jacobi Equations*. PhD thesis, Californa Institute of Technology.

Maurer, C., Qi, R., and Raghavan, V. 2003. A linear time algorithm for computing exact euclidean distance transforms of binary images in arbitary dimensions. *IEEE Transactions on Pattern Analysis and Machine Intelligence 25*, 2 (February), 265–270.

Peikert, R., and Sigg, C. 2005. Optimized bounding polyhedra for gpu-based distance transform. In *Proceedings of Dagstuhl Seminar os Scientific Visualization*.

Perry, R., and Frisken, S. 2001. Kizamu: A system for sculpting digital characters. In *Proc. of ACM SIGGRAPH*, 47–56.

Purcell, T., Donner, C., Cammarano, M., Jensen, H., and Hanrahan, P. 2003. Photon mapping on programmable graphics hardware. *ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, 41–50.

Sethian, J. A. 1999. *Level set methods and fast marching methods*. Cambridge.

Shekhar, R., Fayyad, E., Yagel, R., and Cornhill, F. 1996. Octree-based decimation of marching cubes surfaces. *Proc. of IEEE Visualization*, 335–342.

Shiue, L.-J., Jones, I., and Peters, J. 2005. A realtime gpu subdivision kernel. *ACM Trans. Graph. 24*, 3, 1010–1015.

Sigg, C., Peikert, R., and Gross, M. 2003. Signed distance transform using graphics hardware. In *Proceedings of IEEE Visualization*.

Sud, A., Otaduy, M. A., and Manocha, D. 2004. DiFi: Fast 3D distance field computation using graphics hardware. *Computer Graphics Forum (Proc. Eurographics) 23*, 3, 557–566.

Sud, A., Govindaraju, N., and Manocha, D. 2005. Interactive computation of discrete generalized voronoi diagrams using range culling. In *Proc. International Symposium on Voronoi Diagrams in Science and Engineering*.

Teichmann, M., and Teller, S. 1997. Polygonal approximation of Voronoi diagrams of a set of triangles in three dimensions. Tech. Rep. 766, Laboratory of Computer Science, MIT.

Varadhan, G., Krishnan, S., Kim, Y., and Manocha, D. 2003. Feature-sensitive subdivision and isosurface reconstruction. *Proc. of IEEE Visualization*.

Vleugels, J., and Overmars, M. H. 1998. Approximating Voronoi diagrams of convex sites in any dimension. *International Journal of Computational Geometry and Applications 8*, 201–222.

Woo, M., Neider, J., and Davis, T. 1997. *OpenGL Programming Guide, Second Edition*. Addison Wesley.

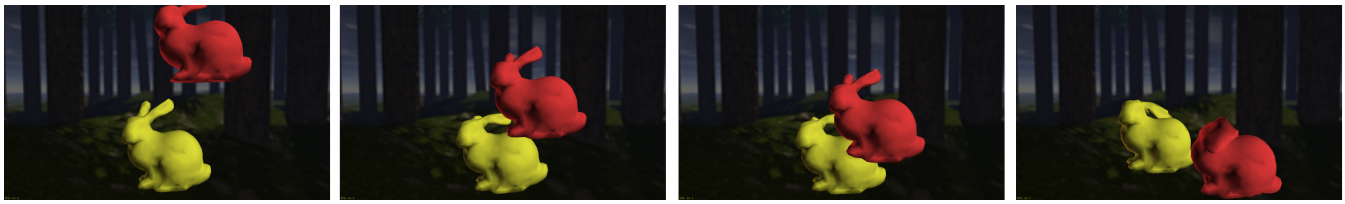

Figure C.1: Proximity Computations on Deforming Models: We use our interactive distance field computation for separation and penetration distance for dynamic simulation of deforming bunnies. Each bunny consists of $2K$ *triangles and we compute localized distance fields on a* 256^3 *grid for the simulation shown in this sequence. The average distance field and proximity query takes* $120ms$ *on a 3.2 GHz Pentium IV PC with NVIDIA GeForce 7800 GPU.*

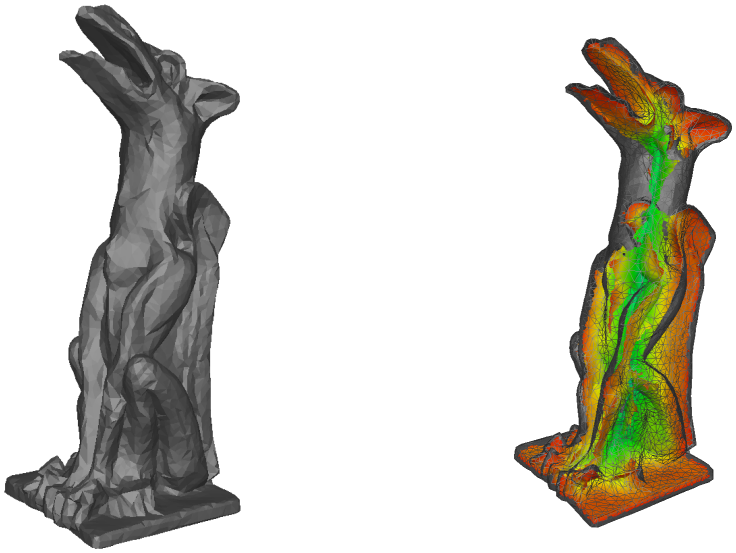

Figure C.2: Medial Axis Approximation: The left image shows the $11K$ *Gargoyle model. The right image shows its simplified medial axis using our distance field algorithm. We use a grid of size* $256 \times 103 \times 91$ *and the distance field computation takes around 625 ms on a 3.2 GHz Pentium IV PC with an NVIDIA GeForce 7800 GPU.*

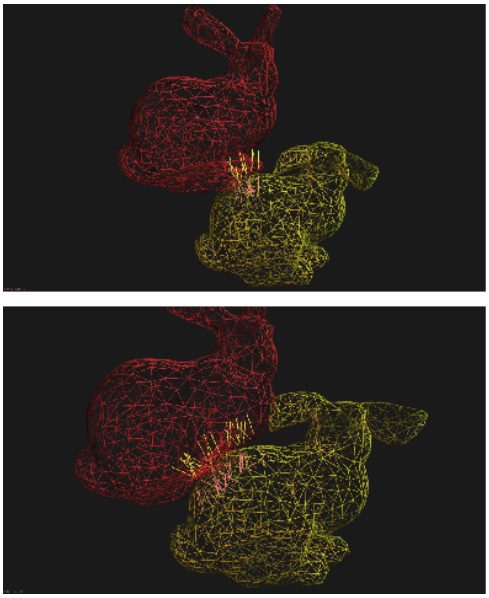

Figure C.3: Contact Normal Computation for deforming bunnies: The average query takes $120ms$ *on a* $3.2GHz$ *Pentium IV PC with NVIDIA GeForce 7800 GPU.*