DiFi: Fast Distance Field Computation using Graphics Hardware

Avneesh Sud Dinesh Manocha
Department of Computer Science
Univeristy of North Carolina
Chapel Hill, NC 27599-3175
{sud,dm}@cs.unc.edu
http://gamma.cs.unc.edu/DiFi

Abstract

We present a novel algorithm for fast computation of discretized distance fields using graphics hardware. Given a set of primitives and a distance metric, our algorithm computes the distance field along each slice of a uniform spatial grid by rasterizing the distance functions. It uses bounds on the spatial extent of Voronoi regions of each primitive as well as spatial coherence between adjacent slices to cull the primitives. As a result, it can significantly reduce the load on the graphics pipeline. We have applied this technique to compute the Voronoi regions of large models composed of tens of thousands of primitives. For a given grid size, the culling efficiency increases with the model complexity. To demonstrate this technique, we compute the medial axis transform of a polyhedral model. In practice, we are able to obtain up to one order of magnitude improvement over earlier algorithms that compute the distance fields using graphics hardware.

Keywords: distance fields, graphics hardware, Voronoi regions, proximity computations, fragment programs

1 Introduction

Given a set of geometric objects, a distance field is defined at each point by the smallest distance from the point to one of the objects. Distance Fields have been used in a number of applications including implicit surface representation, proximity queries, spatial data manipulation, shape analysis, path planning, skeletonization, pattern recognition and local optimization. The problem of computing a distance field is closely related to the problem of computing a Voronoi diagram. Once the voronoi diagram is computed, the distance field can easily be computed as the distance to the respective site. For higher order objects, such as lines and triangles, one needs to compute a generalized voronoi diagram.

Most practically efficient methods for computing the distance fields in $2-$ or $3-$space compute the distance field on a discrete grid. Essentially the methods fall into 2 categories: methods based on front propagation and methods based on Voronoi regions.

Main Contributions: In this paper we present an approach that computes discrete approximations of the 3D distance field and its application to medial axis computation using the graphics hardware. Our contributions include:

1. A novel site culling algorithm that uses Voronoi region properties to accelerate 3D distance field computation
2. A technique for reducing the fill per slice using conservative bounds on the Voronoi regions
3. Formulation of a distance field application as a SIMD process that is efficiently computed on modern graphics hardware.

The algorithm has been implemented and applied to input models consisting of tens or hundreds of thousands of triangles. The running time on a 2.5GHz PC with an nVidia GeForce FX 5800 Ultra graphics card ranges from a second for small models to tens of seconds for huge models. Our approach improves on the state-of-art in following ways:

- **Generality:** No assumption is made with regards to input set. The input models can be non-orientable, non-manifold surfaces.
- **Efficiency:** We show our approach is significantly faster than previous approaches. The culling techniques provide us with a $4 - 20$ times speedup in distance field computation over previous approaches that can handle generic models. The GPU based computation of the medial axis is $2 - 75$ times faster than the CPU based implementation.
- **Scalability:** The performance gain increases as the model size increases.

2 Related Work

The problem of computing a distance field can be broadly classified by the type of input object representation. The object can be given either as a data on a voxel grid, such as a binary image or as an explicit surface representation, such as a triangle mesh.

2.1 Voxel Data based methods

Methods to compute the distance transform of voxel data can be approximate or exact. Approximate methods consider distance transformations in a local neighborhood of each voxel. Danielsson [4] uses a scanning approach in 2D based on the assumption that the nearest object pixels for adjacent pixels are similar. The Fast Marching Method (FMM)[28] propogates a contour to compute the distance transformation from the neighbors. This provides an approximate finite difference solution to the Eikonal Equation $|\nabla u| = 1/f$.

Repeated application of the local masks of the approximate algorithms till a stable solution is reached provides exact distance transforms. A parallel algorithm for this is proposed in [31]. Efficient implementations of this store a propogation front in dynamic lists [7, 25]. Propogation methods can be augmented by storing additional information like direction vectors to nearest voxel [22] and closest features [13]. Propogating additional information along with the FMM, leads to an exact distance transform algorithm [2]. Another class of exact distance transform algorithms are based on computing partial voronoi diagrams and dimensionality reduction. A linear time algorithm for computing Euclidean distance transform os a 2-D binary image is presented in [3]. This is extended to segment lists in [11] and k-D images and further distance metrics in [21].

2.2 Surface Representation based methods

A family of methods determines the distance field for models represented by polygonal surfaces. The work by scan-converting a number of geometric primitives related to the polygonal surface and conditionally overwriting computed distance value at each voxel. Any polygonal model can be decomposed into a collection of point (vertices), line segments (edges) and triangular (faces) sites.

Hoff et al. [12] use graphics hardware to render a polygonal approximation of the distance field and compute Generalized Voronoi Diagrams in two and three dimensions. This approach works on any polygonal input and distance functions. A detailed discussion of this algorithm, and its limitations is provided in section 3.2. An efficient version of the 2-D algorithm for point sites is presented in [5]. It uses precomputed depth textures, and uses a quadtree to estimate voronoi region bounds. However, extensions of this approach to higher-order sites and dimensions are not explained.

Another scan-conversion method to compute 3-D Euclidean distance fields upto a maximum distance d for manifold triangle meshes is the Characteristics/Scan-Conversion (CSC) algorithm presented by Sean Mauch [20]. It uses the connectivity of the manifold mesh to compute polyhedral bounding volumes for the Voronoi cells. The distance function for each site is then evaluated only for the voxels lying inside this polyhedral bounding volume. An efficient GPU based implementation of the CSC algorithm is presented in [29]. The number of polygons sent to the graphics pipeline is reduced by expanding some polyhedral bounding volumes. Also the general purpose computation capability of modern GPUs is used to compute the non-linear distance functions using fragment programs. A more detailed comparison with these approaches is presnted in Section 6.2.1.

2.3 Geometric Computations on the GPU

Apart from distance fields, graphics hardware has been increasingly utilized to accelerate a number of geometric computations, including visualization of constructive solid geometry models [9, 27], interferences and cross-sections [1, 23, 26], Minkowski sums [14, 15], and applications like cloth animation [30] and virtual surgery [18].

In recent years, the performance of graphics hardware has increased more rapidly than that of CPUs. Moreover, graphics hardware has evolved from a fixed-function rendering pipeline to a pipeline with programmable vertex and fragment stages with support for upto full IEEE single-precision floating point. Programmable graphics hardware is optimized for highly-parallel code [17]. Purcell et al. [24] argue that current GPUs can be modeled as parallel stream processors highly optimized for some SIMD applications. This has led to considerable research in developing algorithms for efficient general purpose computation using graphics hardware [10], including 3-D level set solvers [16].

3 Overview

In this section, we present the basic concepts important to our approach. We give a formal definition of distance fields, the relation to generalized voronoi diagrams and review graphics hardware based computation of a discrete approximation. We then show how we may accelerate it for large input sizes.

3.1 Distance Fields and Generalized Voronoi Diagrams

Given a set of input sites $\{S_1, S_2, \ldots, S_k\}$ in n dimensional space, for any point $p \in R^n$, let $dist(p, S_i)$ denote the distance from the point p to the site S_i. $dist(p, S_i)$ is a scalar function $f : \mathbb{R}^n \rightarrow \mathbb{R}$. The *Distance Field* is the minima of all distance functions representing the distance from any point $P \in R^n$ to the closest site. The dominance region of S_i over S_j is defined by

$$Dom(S_i, S_j) = \{p \mid dist(p, S_i) \leq dist(p, S_j)\}$$

For a site S_i, the Voronoi region for S_i is defined by

$$V(S_i) = \cap_{j \neq i} Dom(S_i, S_j)$$

The partition of space into is called the *generalized Voronoi diagram*. The boundaries of the regions $V(S_i)$ are called *Voronoi boundaries*. In other words, the Voronoi region boundaries are the loci of points where distances from two or more sites is the same. The Voronoi diagram can also be treated as a minimization diagram of distance functions to the sites [6].

3.2 Graphics Hardware Based Computation

A method for fast computation of a discrete approximation of distance fields is presented in [12]. It relies on the parallel nature of interpolation-based raster graphics hardware to efficiently compute the distance fields in 2D. The distance function for each site is tessellated within given error bounds and rendered into the depth buffer using an orthographic projection. The distance field for all sites is represented by the final depth buffer. In three dimensions, the bounded volume is divided into a uniform axis-aligned voxel grid, with a set of voxels with a constant z-value constituting a *slice*. The complete distance field is obtained by sweeping slices along the z-axis over the entire volume. Note that this approach works with any type of site and distance metric as long as the distance function can be conveniently tessellated. The distance functions under L_2 norm for points, lines and planes in 3D are shown in Figure 1. For edges and triangular faces, these definitions are combined in a piecewise linear fashion to represent full distance field for the site. Any arbitrary polyhedral model can then be decomposed into a collection of these three primitives [12].

We now present a more rigorous discussion of the computational complexity of this method. Given an input model consisting of k sites and a grid resolution of $N \times N \times N$. The steps are:

1. **Distance Function Meshing**. This is done for each site per slice, hence its cost is $O(kN)$ for N slices. It also depends on the complexity of the distance function and the desired error bound. This computation is CPU limited.
2. **Vertex transfer and transformation**. Let the average number of vertices per site distance mesh is v. Then the total cost for N slices is $O(vkN)$. This computation is bus bandwidth and vertex processing limited.
3. **Rasterization**. Distance functions that have an infinite region of influence (such as the hyperboloid of a point) require computing distance values for each pixel for a slice. Thus the worst case cost for N slices of $N \times N$ pixels, with k distance meshes per slice is $O(kN^3)$. In practice this is the real bottleneck on current graphics processors due to large fill generated.
4. **Readback**. The final distance field has to be read back to the CPU. Due to low readback bandwidth and pipeline stalls, this causes significant delays for interactive applications.

3.3 Our Approach

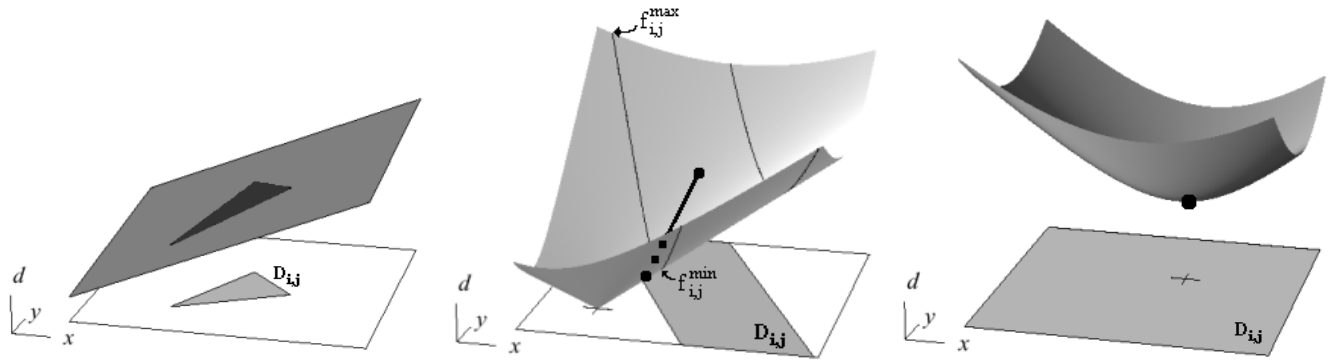
Our approach relies on two key concepts to significantly speed up 3D distance field computation under common distance metrics: *Voronoi regions are bounded and connected* and *high distance field coherence between adjacent slices*.

In Table 1, we enumerate the notations we use throughout the paper. The distance function for a site i may depend on the distance of the site to the slice j, hence notations are defined for a site i w.r.t. slice j. These notations are illustrated for a line site in Figure 1(b).

Notation	Meaning
S_i	Site i
$V(S_i)$	Voronoi region of site i
$D_{i,j}$	Region of influence of site i on slice j
$z = f_{i,j}(x, y)$	Distance function of site i to slice j
$f_{i,j}^{min}$	$min(f_{i,j}(x, y)) \forall (x, y) \in D_{i,j}$
$f_{i,j}^{max}$	$max(f_{i,j}(x, y)) \forall (x, y) \in D_{i,j}$
$Min_{i,j}$	$\{(x, y) \mid f_{i,j}(x, y) = f_{i,j}^{min}\}$ Points in $D_{i,j}$ where $f_{i,j}$ is minimum
$Max_{i,j}$	$\{(x, y) \mid f_{i,j}(x, y) = f_{i,j}^{max}\}$ Points in $D_{i,j}$ where $f_{i,j}$ is maximum

Table 1. Notation table

Monotonic Distance Functions: Under any L_k norm, the distance function is a graph, i.e. it is minimum for a closed, connected set of points in the region of influence of the site, and the function increases monotonically with the euclidean distance to this set.



(a) *Linear distance function of a polygonal site.*

(b) *Conical distance function of a line site.*

(c) *Hyperbolic distance function of a point site.*

Figure 1. Distance Functions under L_2 norm: *The shaded region on the xy-plane represents the region of influence D of the site S_i on the given slice j.*

As shown in Figure 1, this clearly holds under L_2 norm. The distance functions increase monotonically in $D_{i,j}$ as the distance to $Min_{i,j}$ increases.

Thus the distance function will have a maximum value $f_{i,j}^{max}$ at one of the corner points $Max_{i,j}$ in $D_{i,J}$. This fact is used in Section 4.3 to obtain bounds on the Voronoi region of a site in the XY plane, given a bound on the max distance $f_{i,j}^{max}$.

Bounded Connected Voronoi Regions: Within a bounded space, all voronoi regions have a bounded volume. Additionally, under distance metrics that satisfy the triangle inequality, the Voronoi region is connected. This is true for any L_k norm. Thus, the Voronoi region for any site under a L_k metric within a bounded volume is connected and bounded. Further, as the site density increases, the average spatial bounds of a voronoi site will decrease. This concept is used in Section 4 to perform culling of sites to reduce the load on the graphics pipeline.

Distance Field Coherence: Let j and $j+1$ be 2 adjacent slices in the volume at depths z_j and z_{j+1} respectively. $\delta_z = \|z_{j+1} - z_j\|$ is the fixed distance between 2 adjacent slices. Then the maximum change in distance function of a site i from one slice to the next, $\Delta_{i,j}^{max}$ is

$$\Delta_{i,j}^{max} = max \begin{cases} \|f_{i,j+1}(x,y) - f_{i,j}(x,y)\| \\ \quad \forall (x,y) \in \{D_{i,j} \cap D_{i,j+1}\} \\ \|f_{i,j+1}^{max} - f_{i,j}^{max}\| \\ \|f_{i,j+1}^{min} - f_{i,j}^{min}\| \end{cases} \quad (1)$$

Detailed analysis of $\Delta_{i,j}^{max}$ under L_2 norm is presented in the Appendix A.

In practice, for a high density of sites, $\Delta_{i,j}^{max}$ is small, exhibiting high coherence in the distance fields, and the bounds of Voronoi regions between two slices. This fact is used in Section 4 to cull sites and estimate fill bounds for each site.

4 Voronoi Region Based Site Culling

We now present our algorithm to efficiently compute 3D distance fields. The culling is performed in two steps:

1. Culling sites based on estimated Z-bounds
2. Reducing fill region based on estimated XY-bounds

4.1 Site Classification

We can assume the sweep direction is along the positive z direction. Let z_{min} and z_{max} denote the min and max z bounds of a site's voronoi region. Given a slice j at depth z_j and the sweep direction, the set of sites can be partitioned into 3 disjoint sets depending on z_{min} and z_{max} (shown in Figure 2):

1. **Intersecting**($\mathcal{I}_j$): Sites with $z_{min} \leq z_j \leq z_{max}$. The voronoi regions intersect the current slice, and their distance fields need to be rendered.
2. **Approaching**($\mathcal{A}_j$): Sites with $z_{min} > z_j$. The voronoi region of an approaching site does not intersect with current slice, but could potentially intersect in future slices.
3. **Receding**($\mathcal{R}_j$): Sites with $z_{max} < z_j$. A receding site can never become *intersecting*, hence it can be discarded for all future slices.

Note that the intersecting set for next slice $\mathcal{I}_{j+1}$ includes the intersecting set from current slice $\mathcal{I}_j$, with some approaching sites that become intersecting $(\mathcal{A}_j - \mathcal{A}_{j+1})$

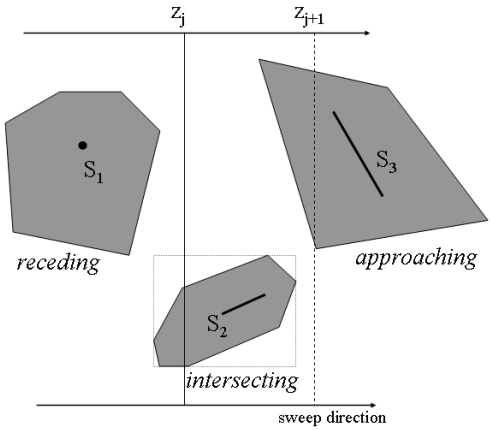


Figure 2. Site Classification: *Shaded areas represent the bounded, connected voronoi regions. For a slice at z_j, sites S_1, S_2, S_3 are classified as Approaching, Intersecting and Receding respectively. For the next slice at z_{j+1}, S_3 becomes Intersecting and S_2 becomes Receding.*

added to it, and some intersecting sites that become receding $(\mathcal{R}_{j+1} - \mathcal{R}_j)$ removed from it. Formally,

$$\mathcal{I}_{j+1} = \mathcal{I}_j + (\mathcal{A}_j - \mathcal{A}_{j+1}) - (\mathcal{R}_{j+1} - \mathcal{R}_j) \qquad (2)$$

4.2 Estimating Z-Bounds

Computing the exact set partitions is equivalent to exact voronoi computation. Instead we compute a conservative estimate of a set of *potentially intersecting* sites $\widehat{\mathcal{I}}_j$, where $\widehat{\mathcal{I}}_j \supseteq \mathcal{I}_j$. The potentially intersecting set $\widehat{\mathcal{I}}_j$ is computed using the visibility of the distance function when rendered for a given slice. We use occlusion queries commonly found in current graphics hardware. These queries scan-convert geometric primitives sent to the GPU such as bounding boxes to check whether the depth of any pixels changes. The algorithm for estimating the set partitions for slice $j+1$ proceeds as follows:

1. Refine the approaching set $\mathcal{A}_j$ to estimate $\mathcal{A}_{j+1}$. This is done by testing the bounding boxes of the distance functions all sites in $\mathcal{A}_j$ for visibility. The distance of any site $S_i \in \mathcal{A}_{j+1}$ to slice $j+1$ will be less than or equal to its distance to the slice j. Hence we use the depth buffer from slice j as an approximate occluder representation. The sites that pass the bounding box test are potentially intersecting. They are removed from $\mathcal{A}_j$ and added to $\widehat{\mathcal{I}}_{j+1}$.

2. Render the distance functions for the potentially intersecting set $\widehat{\mathcal{I}}_{j+1}$ using an occlusion query counter.

3. Refine $\widehat{\mathcal{I}}_{j+1}$ and estimate $\mathcal{R}_{j+1}$. Finally we read back the occlusion query results to determine the *exact* intersecting set $\mathcal{I}_{j+1}$. This also determines the sites that can be moved from the intersecting set to the receding set and discarded for all future slices based on the fact that voronoi regions are connected. However, for a discrete grid, the voronoi region $V(S_i)$ of site S_i need not be connected (e.g. two lines pass through same pixel). In such cases $(S_i) \cup S_i$ forms a connected region. Hence, only those sites with occluded distance functions and are behind current slice $j+1$ ($z_{max} < z_{j+1}$) are moved to the receding set.

The bounding boxes for point and line sites under L_2 are shown in Figure 3. Since the distance function of a triangle is a triangle, we do not perform bounding-box queries on triangle sites. Instead, distance functions for all approaching triangle sites are rendered.

If a bound on the maximum value f_j^{max} of the distance field for a slice j is known, it can be used to perform early CPU culling of approaching sites with $f_{i,j}^{min} < f_j^{max}$. Details on computing this conservative max depth bound and using it to reduce the fill overhead are presented in section 4.3.

4.3 Estimating XY-Bounds

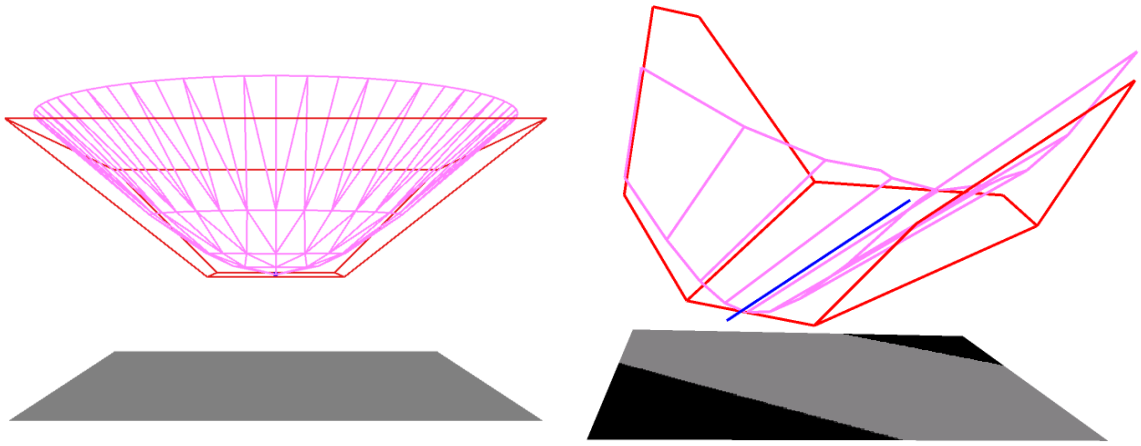
We use the monotonic nature of the distance functions and bounds on the change of maximum value as introduced in Section 3.3 to estimate conservative fill depth bounds in following steps:

1. At end of each slice, determine max depth estimate $f_{i,j}^{max}$ for each site in current slice.

2. Update the maximum depth estimate for next slice $f_{i,j+1}^{max}$, using bounds on $\Delta_{i,j}^{max}$.

3. For the next slice, given max depth $f_{i,j+1}^{max}$, compute the corner points $Max_{i,j+1}$ that give this maximum value. These corner points give us conservative bounds on the XY extent of a distance function.

We can readback the depth buffer at the end of each slice and process it to compute the maximum depth value. However, that causes costly graphics pipeline stalls and readback delays which adversely affects interactive performance. Also, this directly gives the absolute max depth f_j^{max} for the entire slice and not the max depth $f_{i,j}^{max}$ for each slice. Therefore, we use multiple occlusion queries per site to determine a bound on $f_{i,j}^{max}$ for each site too. The distance function is rendered in layers of increasing depth, using an occlusion query counter. At the end of the slice, the counter corresponding to each layer is queried and the maximum depth of the visible layer at highest depth bounds $f_{i,j}^{max}$. For a point site, the distance function of a hyperboloid is already rendered in layers by fanning out from the minimum distance value. For a line site, each radial triangle in the polygonal approximation of the elliptical cone can be



(a) *Truncated rectangular pyramid bounding box of a point site.*

(b) *Pyramidal Frustum bounding box of a line site.*

Figure 3. Bounded Boxes of distance functions under L_2 norm: *The gray region on the xy-plane represents the region of influence. The sites are shown in blue. The distance function is rendered in magenta wireframe. The bounding box is rendered in red wireframe.*

treated as a separate layer. For a triangle site, the maximum distance value is used instead of drawing it in layers. Once $f_{i,j}^{max}$ has been computed at the end of slice j, the maximum depth $f_{i,j+1}^{max}$ for next slice $j+1$ can be computed as shown above.

5 Applications

One of the disadvantages of computing distance fields on graphics hardware is that the results of the computation must be read back to the CPU for further processing. This readback is currently expensive and also stalls the graphics pipeline. Recent advances in graphics hardware, such as programmability and 32-bit floating point precision, have enabled many applications on the graphics processor(GPU) that previously required the CPU. Not only does this avoid the penalty of readback, but it allows programs to exploit the SIMD nature of graphics hardware. In this section we demonstrate how to use the distance fields to compute the medial axis of a polyhedral model.

5.1 Medial Axis

The medial axis computation, as described by [8], can be thought of as a filter applied to the distance field. Graphics hardware is well-suited to filtering, as each point where the filter is applied has an independent and simple computation. This filter removes all voxel points except for those that lie along the skeleton of the object. The filter is a separation criteria test, which tests the separation angle of the directed distance vectors between two adjacent voxels. Pseudo-code for this filter is presented in 5.1, and a detailed explanation is in [8].

FilterMedial($Slice_k, Slice_{k-1}, AngleThreshold$)

For each (i,j) in $Slice_k$
  Let $A = gradient(i,j,k)$
  for each $dir = \hat{i}, \hat{j}, \hat{k}$
    Let $B = gradient(PrevVoxel(A, dir))$
    $Thres = A \cdot B \leq AngleThreshold$
    $Tails =$ TailSeparation(A,B) $<$ HeadSeparation(A,B)
    $NotZero =$ B does not lie on model boundary
    $Result = NotZero \cdot \overline{Thresh} \cdot Tails$
  end for
if any $Result$ is true then A borders the medial axis
end for

ALGORITHM 5.1: *Pseudo code for computing the medial axis on graphics hardware.*

6 Implementation and Results

We have an OpenGL implementation the improved distance field computation algorithm running on a Pentium4 2.5GHz PC with 1GB RAM, an nVidia GeForce FX 5800 Ultra graphics card running Windows 2000. The nVidia OpenGL extension GL_NV_occlusion_query, available on GeForce 3 and higher graphics cards, is used to perform the occlusion queries. The models are stored as a list of sites with set and occlusion information.

The graphics pipeline can get stalled by querying the results of an occlusion query immediately after sending the

geometry for rendering. We avoid this by exploiting the parallelism in GL_NV_occlusion_query and batching several occlusion queries together. We also interleave rendering one set of sites and querying its visibility with the rendering another set of sites. This ensures that the graphics pipeline has finished rasterizing the geometry before we query it for visibility, thus avoiding stalls. Additionally, we maintain the list of sites sorted in order of increasing distance. This allows for front-to-back traversal of sites with respect to current slice. The front-to-back traversal allows for early termination of max depth estimation routines and also makes most efficient use of modern graphics hardware's fast Z-culling techniques.

We computed the global distance field on a number of manifold as well as non-manifold CAD models. No optimization techniques like interior masking or convex vertex/edge removal were used. Some of the test models are shown in Figures 4, 5. We compare the performance of the implementation of our algorithm (called DiFi) with HAVOC3D [12]. The timings for the distance field computation for various models are presented in Table 2. The timings for different grid resolutions for the shell model are presented in Table 3.

Model	Polys	Resolution	HAVOC3D	DiFi
Shell	4460	128x126x126	31.69	3.38
Head	21764	79x106x128	52.47	18.60
Bunny	69451	128x126x100	212.71	36.21
Cassini	90879	94x128x96	1102.01	42.90
Dragon	217852	59x91x128	1918.50	130.34

Table 2. *Times (in seconds) to compute the distance field using HAVOC3D and DiFi*

Polys	Resolution	Time(s)
4460	$32 \times 32 \times 32$	0.72
4460	$64 \times 63 \times 63$	1.56
4460	$128 \times 126 \times 126$	3.38
4460	$256 \times 252 \times 252$	7.04

Table 3. *Times (in seconds) to compute the distance field for the Shell model using DiFi with varying grid resolutions*

6.1 Medial Axis Application

We have a graphics hardware based implementation of the medial axis computation. It uses OpenGL's ARB fragment program extension, and is supported by the GeForce FX and Radeon 9500+ graphics cards. For high precision, the input distance field is copied to a floating point texture with 32 bits per channel. The output of the fragment program is 1-bit per voxel indicating its presence on the medial axis and a 32-bit distance to surface. All hardware programs were written in nVidia's Cg programming language[19]. The timings are shown in Table 4.

Model	Software			Hardware
	$\theta = 60°$	$\theta = 15°$	$\theta = 5°$	
Head	0.18	0.83	3.47	0.08
Bunny	0.68	18.5	144	0.13
Shell	3.5	19.5	65.6	0.14
Dragon	0.19	1.98	7.32	0.06
Cassini	7.59	81.7	172	0.1

Table 4. *Time to compute the medial axis in software (for an angle threshold of $60°$, $15°$, and $5°$), and in hardware (independent of threshold).*

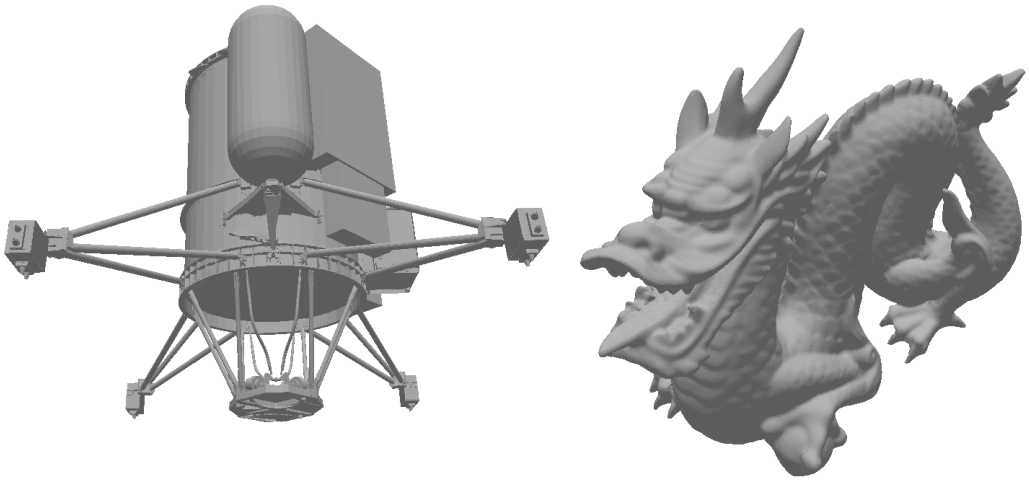
The medial axis is rendered directly from the GPU without any framebuffer readbacks to the CPU. For this, the output of the fragment program is copied to a 3D texture, with the color encoding the distance to surface. This volume is then rendered using alpha blended slices. The resulting images are shown in Figure 5.

6.2 Discussion

We now present an analysis and discussion of our approach for computing distance fields, and its application to medial axis computation.

6.2.1 Distance Field Computation

Our new algorithm for distance field computation provides a $4 - 20$ times speedup over HAVOC3D. Additionally, the speedup is more as the model size increases. This is due to the fact that average voronoi region size decreases. This leads to a higher fraction of the sites being culled and larger reduction in the fill area. The performance is also influenced by the level of occlusion in the distance functions along the swept axis. A model with high depth complexity exhibits higher occlusion in the distance function. As is evident from table 3, the distance field computation is not entirely fill limited using DiFi. For smaller resolutions, the mesh generation and graphics driver overhead cause the applicationi to be CPU limited also. We believe part of the driver overhead may also be due to the fact that the cost of adding an occlusion query during rasterisation is not negligible. The associated pipeline flushes reduce the amount of parallelism that can be achieved between the CPU and the GPU.



(a) *Cassini (90879 polygons)* (b) *Dragon (217852 polygons)*

Figure 4. Large input models used for medial computation.

Using the occlusion query has some fill overhead. Sites determined as potentially intersecting after the bounding-box tests are rasterized twice: once for the bounding-box queries and once for rendering the distance functions. This is undesirable in an application with a significant fill-bandwidth. It should be noted that this extra fill is performed only for the slice in which the sites are first determined as potentially intersecting. Moreover, our experiments show that for large input sets, the fraction of potentially intersecting sites to approaching sites is small, so the extra fill overhead is small compared to cost of meshing and transforming all approaching sites.

We now present a comparison of our approach with a graphics hardware implementation [29] of the CSC algorithm [20]. The CSC algorithm assumes the input geometry is a well-defined manifold. It has a computation cost of $O(k + rN^3)$, where k is the number of sites in the model, N^3 is the grid size and r is the degree of overlap between the bounding volumes. The $O(k)$ term reflects the setup cost of determining the polyhedral bounding volumes. The distance field is computed for a band of length d around the surface. For small values of d, r is close to unity. Thus each pixel is written approximately once, making the algorithm highly efficient for computing distance fields in narrow bands around the surface. For large values of d, to minimize the overlap cost r the polyhedral bounding volumes can be clipped during the setup stage. However, for complex non-convex shapes and large values of maximum distance d, the clipping costs can be $O(k^2)$. Also, the algorithm presented is valid only for the Euclidean distance transform.

In contrast, DiFi can take a *polygon soup* as input and does not impose any connectivty constraints. There is no setup cost, the bounding volumes of the voronoi regions are computed dynamically. This makes it well-suited for applications requiring incremental insertion of geometry. In worst case, each pixel may be written k times, making the worst case time complexity of DiFi $O(kN^3)$. However, for a completely random input, each pixel will be written a constant number of times making the time complexity $O(k + N^3)$, the $O(k)$ term reflecting the distance mesh generation and transform cost. Finally, the framework presented in DiFi is applicable to all L_k distance metrics.

6.2.2 Medial Axis Application

The computation of the medial axis on the GPU also gives us a $2-75$ times speedup. This speedup is due to the SIMD nature of the application and the GPU being optimized for such applications. The computation time for medial axis computation is proportional to the size of the grid. However, for the CPU based implementation, it also depends on the model and size of the output medial axis. This is due to conditionals in the medial axis filter and early-out tests leading to fewer instructions being executed on the CPU. For example, in [8], the θ-threshold is tested first, and no further computation is performed if the test fails. Current graphics hardware has limited support for conditionals; instead of conditional execution, it is conditional assignment. Thus our implementation executes all possible tests and assigns a value based on these results. This means that early-out tests,

such as testing for zero-valued gradients, do not effect the runtime of the application. Given the performance benefits, we expect this to change in future hardware.

For many applications the medial axis must be read back to the CPU for further computation. This requires reading back 8-bits per voxel, which can be a large depending on the grid resolution. However, the medial axis can be treated as a sparse 3D matrix. Depending on the speed of readback, it may be advantageous to perform some sort of packing to reduce this storage. For example, each face of the medial axis could be encoded as one bit in the RGB color channels of a texture, allowing eight slices to be packed into one texture. Another alternative is to perform the application of the medial axis also on the GPU and readback only the final result set, which can be expected to be 1-dimensional or 2-dimensional.

7 Conclusions and Future Work

We have presented a method for fast computation of discretized distance fields using graphics hardware. We have presented techniques to estimate bounds on the voronoi region of each primitive and use the spatial coherence between adjacent slices. We have derived the bounds for the Euclidean distance field with point, line and triangle sites. Finally, we formulate an applicatioin of the distance field in streaming SIMD framwork and show how it can be efficiently performed on modern GPUs with programmable fragment processors. Such a framework minimizes readback delays associated with methods that compute distance fields on the graphics hardware.

In the future, we would like to apply this work to various applications, including dynamic simulation and path planning. We would also like to apply more acceleration techniques to time-varying distance fields. We would also like to explore techniques for reducing the CPU limitation of current implementation.

References

[1] G. Baciu and S. Wong. Image-based techniques in a hybrid collision detector. *IEEE Trans. on Visualization and Computer Graphics*, 2002.

[2] D. Breen, S. Mauch, and R. Whitaker. 3d scan conversion of csg models into distance, closest-point and color volumes. *Proc. of Volume Graphics*, pages 135–158, 2000.

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

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

[5] M. Denny. Solving geometric optimization problems using graphics hardware. In *Proc. of Eurographics*, 2003.

[6] H. Edelsbrunner and R. Seidel. Voronoi diagrams and arrangements. *Discrete Comput. Geom.*, 1:25–44, 1986.

[7] H. Eggers. Two fast euclidean distance transformations in z^2 based on sufficient propagation. *Computer Vision and Image Understanding*, 69(1):106–116, 1998.

[8] M. Foskey, M. Lin, and D. Manocha. Efficient computation of a simplified medial axis. *Proc. of ACM Solid Modeling*, 2003. To appear.

[9] J. Goldfeather, S. Molnar, G. Turk, and H. Fuchs. Near real-time csg rendering using tree normalization and geometric pruning. *IEEE Computer Graphics and Applications*, 9(3):20–28, 1989.

[10] General purpose computation using graphics hardware. http://www.gpgpu.org.

[11] W. Guan and S. Ma. A list-processing approach to compute voronoi diagrams and the euclidean distance transform. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(7):757–761, 1998.

[12] K. Hoff, T. Culver, J. Keyser, M. Lin, and D. Manocha. Fast computation of generalized voronoi diagrams using graphics hardware. *Proceedings of ACM SIGGRAPH 1999*, pages 277–286, 1999.

[13] J. Huang, Y. Li, R. Crawfis, S. Lu, and S. Liou. A complete distance field representation. In *Proceedings of IEEE Visualization*, pages 247–254, 2001.

[14] A. Kaul and J. Rossignac. Solid-interpolating deformations: construction and animation of pips. *Computer and Graphics*, 16:107–116, 1992.

[15] Y. Kim, M. Lin, and D. Manocha. Deep: An incremental algorithm for penetration depth computation between convex polytopes. *Proc. of IEEE Conference on Robotics and Automation*, 2002.

[16] A. Lefohn, J. Kniss, C. Hanses, and R. Whitaker. Interactive deformation and visualization of level set surfaces using graphics hardware. In *Proceedings of IEEE Visualization*, page To Appear, 2003.

[17] E. Lindholm, M. Kilgard, and H. Moreton. A user-programmable vertex engine. *Proc. of ACM SIGGRAPH*, pages 149–158, 2001.

[18] J. C. Lombardo, M.-P. Cani, and F. Neyret. Real-time collision detection for virtual surgery. *Proc. of Computer Animation*, 1999.

[19] W. R. Mark, R. Glanville, K. Akeley, and M. Kilgard. Cg: a system for programming graphics hardware in a c-like language. *ACM Trans. on Graphics (Proc. of ACM SIGGRAPH)*, 2003. to appear.

[20] S. Mauch. A fast algorithm for computing the closest point and distance transform. http://www.acm.caltech.edu/~seanm/software/cpt/cpt.pdf, 2000.

[21] C. Maurer, R. Qi, and V. Raghavan. 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):265–270, February 2003.

[22] J. C. Mullikin. The vector distance transform in two and three dimensions. *CVGIP: Graphical Models and Image Processing*, 54(6):526–535, Nov. 1992.

[23] K. Myszkowski, O. G. Okunev, and T. L. Kunii. Fast collision detection between complex solids using rasterizing graphics hardware. *The Visual Computer*, 11(9):497–512, 1995.

[24] T. Purcell, I. Buck, W. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *ACM Trans. on Graphics (Proc. of SIGGRAPH'02)*, 21(3):703–712, 2002.

[25] I. Ragnelmam. Neighborhoods for distance transformations using ordered propagation. *Computer Vision, Graphics and Image Processing*, 56(3):399–409, 1992.

[26] J. Rossignac, A. Megahed, and B. Schneider. Interactive inspection of solids: cross-sections and interferences. In *Proceedings of ACM Siggraph*, pages 353–60, 1992.

[27] J. Rossignac and J. Wu. Correct shading of regularized csg solids using a depth-interval buffer. In *Eurographics Workshop on Graphics Hardware*, 1990.

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

[29] C. Sigg, R. Peikert, and M. Gross. Signed distance transform using graphics hardware. In *Proceedings of IEEE Visualization*, page To Appear, 2003.

[30] T. Vassilev, B. Spanlang, and Y. Chrysanthou. Fast cloth animation on walking avatars. *Computer Graphics Forum (Proc. of Eurographics'01)*, 20(3):260–267, 2001.

[31] H. Yamada. Complete euclidean distance transformation by parallel operation. In *Proc. of 7th International Conf. on Pattern Recognition*, pages 336–338, Montreal, Canada, 1984.

A Appendix

Determining $\Delta_{i,j}^{max}$, as defined in eqn. 1 for Point, Line and Triangle sites under L_2 norm. For simplicity of notation, $f_j \equiv f_{i,j}$, $\Delta_j \equiv \Delta_{i,j}$ below. Let j and $j+1$ be two adjacent slice at z_j and z_{j+1} respectively. Distance between the slices, $\delta = \mid z_{j+1} - z_j \mid$.

A.1 Point Site

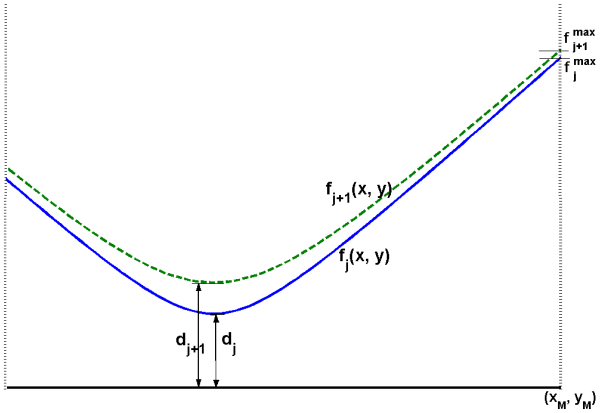


Figure 6. Point Distance Function: *Cross-section of the distance function of a point site to two adjacent slices j and $j+1$.*

Let a point site $S_i = (x_i, y_i, z_i)$ be at distances d_j and d_{j+1} to two adjacent slices, shown in Figure 6.

$$\delta = \mid d_{j+1} - d_j \mid$$

$$f_j(x,y) = \sqrt{(\Delta x)^2 + (\Delta y)^2 + d_j^2}$$

$$f_{j+1}(x,y) = \sqrt{(\Delta x)^2 + (\Delta y)^2 + d_{j+1}^2}$$

$$\Delta_j(x,y) = \sqrt{(\Delta x)^2 + (\Delta y)^2 + (d_j + \delta)^2} - \sqrt{(\Delta x)^2 + (\Delta y)^2 + d_j^2}$$

$$\text{where } \Delta x = (x - x_i), \Delta y = (y - y_i)$$

Δ_j is maximum at $\Delta x = 0, \Delta y = 0$
$\implies \Delta_{i,j}^{max} = \delta$.

A.2 Line Site

Using suitable rigid transformations, any line can be transformed to lie in the XZ plane, and passe through origin. This does not change the shape of the distance function (or the change in distance function between 2 adjacent slices). For a line in xz plane with slope $m = \tan(\theta)$, the distance function is given by

$$f_j(x,y) = \sqrt{(xc)^2 + y^2}, c = \frac{\sqrt{1+m^2}}{m} = \frac{1}{\sin\theta}$$

For a line-segment with end points $\mathbf{p1}$ and $\mathbf{p2}$, the domain of influence D_j is the strip

$$p1_x + (p1_z - z_j)\tan\theta \le x \le p2_x + (p2_z - z_j)\tan\theta$$

The shape of the distance function depends only on the slope of the line, but the domain of influence D_j depends on the distance of the line to the slice. Figure 7 shows the strip domains $[x_j^m, x_j^M]$ and $[x_{j+1}^m, x_{j+1}^M]$ for 2 consecutive slices, moving away from the line segment, with change in bounds = $x_{j+1}^M - x_j^M = \delta\tan\theta$. Note that change is bounded by the slice limits, even if $\theta \to 90°$.

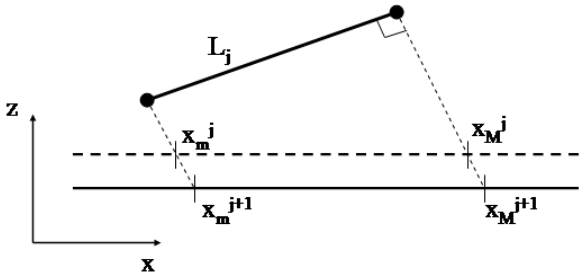


Figure 7. Region of Influence of a Line Site: *Cross-section of the region of influence of a line site to two adjacent slices j and $j+1$.*

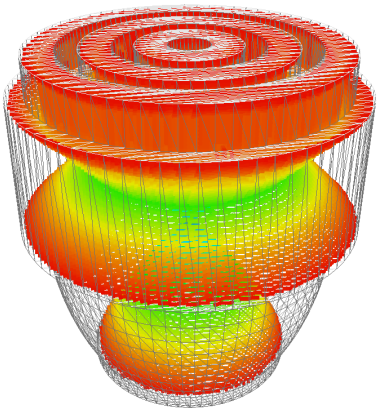
Since f_j is monotonically increasing in D_j, $x_{j+1}^M > x_j^M$,

$$\Delta_j^{max} = max\left((f_{j+1}^{max} - f_j^{max}), (f_{j+1}^{min} - f_j^{min})\right).$$

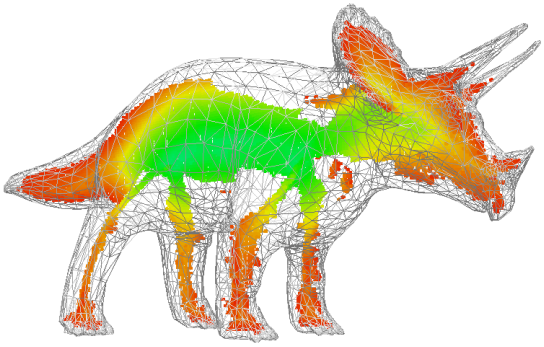
Let the regions of influence for slices j and $j+1$ be the strips D_j, D_{j+1} on the rotated slice of the XY plane. f_j, f_{j+1} are evaluated at the corner points of D_j, D_{j+1} to give $(f_j^{max}, f_{j+1}^{max})$ and $(f_j^{min}, f_{j+1}^{min})$. The maximum difference between the pairs gives Δ_j^{max}.

A.3 Triangle Site

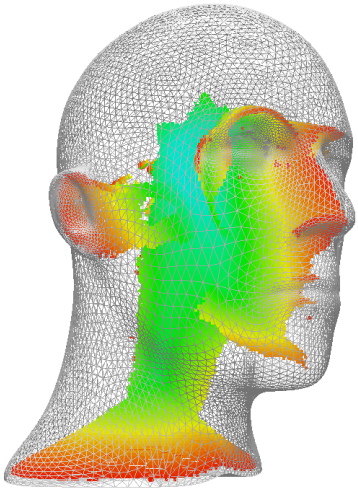
The analysis is similar to that of a line site, with the angle $\theta = \cos^{-1}(\hat{N} \cdot \hat{z})$, $\hat{N}$ = normalized triangle normal. The domain of influence D_j is the intersection of a projected triangle with the slice, as shown in Figure 1(a). The corner points of D_j, D_{j+1} are computed to give $(f_j^{max}, f_{j+1}^{max}), (f_j^{min}, f_{j+1}^{min})$ and Δ_j^{max}.



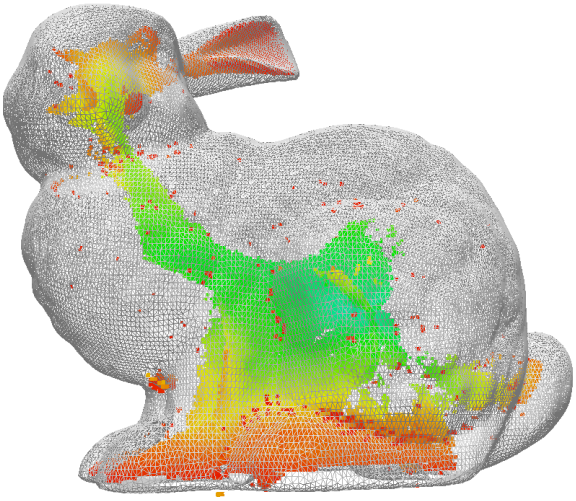
(a) *Shell Charge (*4460 *polygons)* $\theta = 30^\circ$



(b) *Triceratops (*5660 *polygons)* $\theta = 60^\circ$



(c) *Head (*21764 *polygons)* $\theta = 30^\circ$



(d) *Bunny (*69451 *polygons)* $\theta = 45^\circ$

Figure 5. Medial Axis*: The original model surface is shown in wireframe. The medial axis voxels are drawn using alpha-blended quads. The HSV color of a voxel represents the distance from the surface, with red denoting minimum distance.*