Signed Distance Transform Using Graphics Hardware

Christian Sigg Ronald Peikert Markus Gross

ETH Zürich

Abstract

This paper presents a signed distance transform algorithm using graphics hardware. The signed distance transform computes the scalar valued function of the Euclidean distance to a given manifold of co-dimension one. For a closed and orientable manifold, the distance has a negative sign on one side of the manifold and a positive sign on the other. Triangle meshes are considered for the representation of a two-dimensional manifold. The distance function is sampled on a regular Cartesian grid. The algorithm has linear complexity in the number of grid points. We assign to each primitive a simple polyhedron enclosing its Voronoi cell. The distance to the primitive is computed for every grid point inside the polyhedron. Points in regions of overlapping polyhedra are assigned the minimum of all computed distances. In order to speed up computations, points inside each polyhedron are determined by scan conversion of grid slices using graphics hardware. A fragment program is used to perform the nonlinear interpolation and minimization of distance values.

CR Categories and Subject Descriptors: I.3.3 [Computer Graphics]: Picture/Image Generation - Bitmap and framebuffer operations; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling - Curve, surface, solid, and object representations.

Additional Keywords: Distance field, distance transform, Voronoi diagram, fragment program, scan conversion.

1. INTRODUCTION

Given a set of geometry objects in 2- or 3-space, a distance field is defined at each point by the smallest Euclidean distance to a point on one of the objects. Objects can be curves in 2-space and surfaces in 3-space or, more generally, any compact subset. If an orientable and closed n-1 dimensional manifold is chosen, the distance field can be given a sign.

Signed or unsigned distance fields have many applications in computer graphics, scientific visualization and related areas, such as implicit surface representation [5,6], object metamorphosis [2], collision detection and robotics [5], skeletonization [16], accelerated volume raytracing [14], camera path planning and image registration [3]. Depending on the application, the distance field is required on a full pixel or voxel grid or only within a band of width d around the objects.

Computer Graphics Laboratory, Computer Science Dept., ETH Zürich, CH-8092 Zürich, Switzerland.

The problem of computing a 3D Euclidean distance transform exists in two flavors, distinguished by the type of object representation. The object can be given either as data on a voxel grid or in vector representation, typically a triangle mesh in the case where the object is a surface. Both problems have been studied extensively and fast methods have been developed for both of them. It is reasonable to treat the two problems separately. If the goal is to sample the exact distance to a triangle mesh, the problem cannot be stated in voxel space. Likewise, there is usually no gain in transforming the problem from voxel representation to vector representation. For triangle meshes, time complexity must depend on the number M of surface primitives (faces, edges, and vertices). Therefore, algorithms for the two different problem settings cannot be directly compared.

A method [10] which was recently presented, finds the distance transform in voxel data in $O(N)$ time, where N is the number of voxels. In the same paper, a good overview of earlier methods is given. Essentially, methods fall into two categories, propagation methods and methods based on Voronoi diagrams.

In propagation methods, the distance information is carried over to neighbor voxels, either by sweeping in all grid dimensions, or by propagating a contour. A well-known example of the latter is the Fast Marching Method [13], an upwind scheme which can solve the Eikonal Equation $|\nabla u| = 1/f$ in a single iteration and in $O(N \log N)$ operations. A signed distance field is obtained by using a constant propagation function f. However, due to the finite difference scheme, FMM is not an exact method.

Besides the distance, additional information can be stored in the distance field. Such information can be the vector pointing to the nearest object point, known as the vector distance transform [11]. Alternatively, the index of the nearest surface primitive can be attributed to each point, the resulting field is called a complete distance field representation [8].

By propagating this type of additional information, FMM and similar propagation methods can be turned into exact distance transform algorithms [1,4,15].

Another approach is to construct a Voronoi diagram, which directly leads to a complete distance field representation. Voronoi-based methods can be used for distance transform in voxel data and then have a time complexity of $O(N \log N)$. The approach seems to be more natural when the distance field to a triangle mesh is being computed. In this case, the set of Voronoi sites consists of the vertices, edges and faces. Since they are not isolated points, a generalized Voronoi diagram (GVD) has to be computed. The time for generating a diagram with M sites is $O(M \log M)$. Once a GVD is computed, the distance field can easily be computed as the distance to the respective site.

All methods mentioned thus far are image space methods. An alternative are object space methods, i.e. methods based on scan conversion. Here, the distance field is obtained by scan converting a number of geometric objects related to the triangle mesh and by

conditionally overwriting the computed voxel values. The advantage of object space methods is their sub-pixel accuracy. However, it is obvious that the relative performance degrades if the average triangle size shrinks to the size of a single voxel. It has been shown that for distance fields of triangle meshes, methods based on scan conversion are competitive. Such an algorithm was presented by Mauch et al. [9].

Starting from their algorithm, we derived a version where two-dimensional grid slices are passed to the graphics hardware for scan conversion. The problem of getting the correct nonlinear interpolation of the distance value in a grid slice was solved by a fragment program. In addition, we revised the method such that it correctly handles vertices where both convex and concave edges are adjacent.

In order to further speed up computations, we propose a modified method. By scan converting a different type of polyhedron, we were able to reduce the number of triangles to be rendered to less than a third. Our modified algorithm is significantly faster than the original software algorithm.

In Section 2, the basic definitions of a distance field and related concepts are given. In Section 3, two algorithms for computing distance fields using a scan conversion process are summarized. These two algorithms, one for graphics hardware and one for software, are the basis for our improved algorithm. Implementation issues are discussed in Section 4, and performance results are given in Section 5.

2. DISTANCE FIELDS

Given a manifold S of dimension $n-1$ in R^n, the distance field u is a a unique scalar function defined in R^n. At each point, u equals the distance to the closest point on S. If the manifold S is closed and orientable, the space is divided into inner and outer parts. Therefore, a signed distance field can be defined. A positive sign is chosen outside the surface and a negative sign inside. Thus, the gradient of the distance field on the surface is equivalent to the surface normal.

The type of distance metric which is chosen depends on the application. Common choices are chessboard, chamfer and Euclidean distance [12]. We will restrict ourselves to the Euclidean distance, which is probably the most meaningful, but it is also the most expensive to compute.

The signed distance field u is the solution to the Eikonal equation $|\nabla u| = 1$ with boundary condition $u|_S = 0$. The boundary condition shows that the definition of S as a subset of R^n and the signed distance function are equivalent descriptions. The manifold corresponds to the zero-set of the signed distance function: $S = \{x | u(x) = 0\}$. Therefore, the signed distance transform converts an explicit surface representation to an implicit one.

If S is the union of S_i with $i=1..n$, the distance field of S is the point-wise minimum of the distance fields u_i of S_i.

$$u(\bigcup S_i) = \min(u_i) \tag{1}$$

For signed distance fields, minimization must be carried out with respect to absolute values. If S is a triangle mesh in 3-space, the S_i can be chosen to represent the triangle faces. It is also possible to use a disjoint union. In the case of a triangle mesh, the S_i become the triangles (excluding the edges), the edges (excluding the endpoints) and the vertices. Collectively, faces, vertices and edges will be denoted as primitives.

In order to sample the distance field on a grid, a brute force algorithm would compute the distance of each grid point to each primitive. The distance at one grid point is chosen to be the distance to the closest primitive, thus resulting in the shortest distance. If the triangle mesh consists of a large number of triangles and the sampling grid is large, this approach is impractical. For an efficient algorithm, one needs to reduce the number of distances calculated per grid point or alternatively, per primitive.

To achieve this goal, we use the fact that only one distance per grid point is stored in the final distance field, namely the one to the closest primitive. When computing the distance field value for a sample, a primitive can be excluded from the calculation if it is known that a closer primitive exists. To quickly find a primitive that is relatively close and exclude a large number of primitives that are further away, one can store the primitives in a spatial data structure such as a BSP tree. By using this structure, one can quickly find the closest primitive to a point: While the tree is scanned for the closest primitive, one can give an upper limit of the final distance. At the same time, a lower bound of the distance can be computed for any subtree. If the lower bound of a subtree is larger than the current upper bound of the final distance, the subtree can be excluded from the search. This leads to an algorithm logarithmic in the number of primitives of the input mesh.

Alternatively, the loops can be reversed, by iterating over the primitives instead of the grid points. For each primitive, we try to reduce the computation of distances to grid points that lie closer to a different primitive. Optimally, only distances to grid points contained in the Voronoi cell of the corresponding primitive are calculated. However, the computation of Voronoi diagrams is not easier than the computation of distance fields. Still, if a point is known to lie outside of a Voronoi cell, the distance to its base primitive does not need to be calculated. An algorithm which uses this approach will be presented in Section 3.2.

3. SCAN CONVERSION BASED ALGORITHMS

Scan conversion refers to the process of sampling a geometric shape on a pixel or voxel grid, interpolating color and other attributes defined at the vertices. In computer graphics, scan conversion is a key operation in the rendering pipeline and is efficiently performed by standard graphics cards. By reading back the frame buffer data, the computing power of graphics cards becomes available for more purposes than just rendering. In recent years, the programmability of graphics cards made it possible to adapt the scan conversion operation. In particular, nonlinear interpolation functions can be programmed.

In Sections 3.1 and 3.2, we explain two algorithms using scan conversion for generalized Voronoi diagrams and for signed distance transforms, respectively. The idea of combining their strengths, the suitability for graphics hardware and the small number of triangles, brought us to the algorithm described in Sections 3.3 and 3.4.

3.1 The Generalized Voronoi Algorithm

A standard Voronoi diagram is a partitioning of a planar region into cells based on a finite set of points, the Voronoi sites. Each cell consists of the points which are closer to one particular site than to all others. Thus, Voronoi diagrams can be constructed with a graphical method by drawing the graphs of the sites' distance fields. For a single point site, the graph is a vertical circular cone which opens downwards at a fixed angle. After drawing all graphs, a view from the top shows their point-wise minimum which, by

definition, is the Voronoi diagram. For a rendering with graphics hardware, cones must be clamped at some distance d and approximated by a set of triangles. If the graphs are drawn in distinct colors, the frame buffer now contains the Voronoi diagram. The depth buffer holds the distance field.

Based on this observation, Hoff et al. [7] presented algorithms for generalized Voronoi diagrams (GVD) in two and three dimensions. In a general Voronoi diagram, the set of sites can contain more geometric objects than just points, such as line segments or triangles. In that case the cone must be replaced by the graph of their distance field. For a line segment e.g. the graph is a "tent" consisting of two rectangles and two half cones. Curved sites can be treated by piece-wise linear approximations. As a special case the sites can be the points, edges and faces of a triangle mesh.

The GVD algorithm produces unsigned distance fields. However, in cases where a sign is defined, it can easily be computed in a separate pass, by computing the vector to the nearest point on the Voronoi site and comparing it with the outward normal.

In three dimensions, the same approach can be used to obtain the distance field and the GVD. However, if done with graphics hardware, the scan conversion has to be done slice by slice. On a slice, the graphs of the distance field are again surfaces which can be tessellated and rendered. However, these surfaces are more complex than for the 2D method. In the case of a point site e.g., the graph is a hyperboloid of revolution of two sheets.

In order to bound the errors introduced by the linear interpolation performed by the graphics hardware, graphs must be finely tessellated, with up to 100 triangles for a cone [7] and even more for the hyperboloids which are doubly curved surfaces.

Because the GVD algorithm does not restrict the configuration of Voronoi sites, it is more general than needed for signed distance transforms of triangle meshes. However, each primitive of the input surface produces a large number of triangles to render. Thus, the method becomes inefficient for large meshes.

3.2 The Characteristics/Scan-Conversion Algorithm

A different approach to applying scan conversion to distance field computations was presented by Mauch et al. [9]. Their Characteristics/Scan-Conversion (CSC) algorithm computes the signed distance field for triangle meshes up to a given maximum distance d. Using the connectivity of a triangle mesh, special polyhedra serving as bounding volumes for the Voronoi cells can be computed. Therefore, only a small part of the distance field has to be calculated for each primitive.

It does not try to find exact Voronoi cells, but instead uses polyhedra which are easier to compute and are known to contain the Voronoi cell as a subset.

The algorithm computes the signed distance field up to a given maximum distance d, i.e. within a band of width $2d$ extending from both sides of the surface. Since the triangle mesh is assumed to be orientable, it is possible to classify all edges as either convex, concave, or planar. Accordingly, vertices can be classified as convex, concave, saddle or planar. The set of Voronoi sites is chosen as the open faces (triangles), the open edges, and the vertices of the mesh. According to the three type of sites, three types of polyhedra are constructed such that they contain the Voronoi cell as a subset.

• Polyhedra for the faces: 3-sided prisms ("towers") built up orthogonally to the faces (Fig. 1, left).

• Polyhedra for the edges: 3-sided prisms ("wedges") filling the space between towers (Fig. 1, right). Wedges contain an edge and extend to one side of the mesh only.

• Polyhedra for the vertices: n-sided cones which fill the gaps left by towers and wedges (Fig. 1, left). Cones contain the vertex and extend to one side of the mesh. If the vertex is a saddle, the polyhedron is no longer a cone and now has a more complex shape. This case is not explicitly treated in [9].

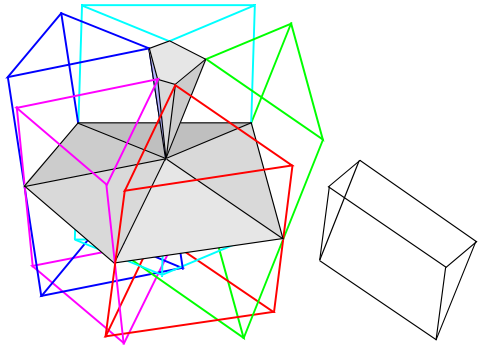

Figure 1: Polyhedra for faces, edges, and vertices: towers, wedges, and cones.

In Fig. 1, one can see that each polyhedron contains at least the generalized Voronoi cell of its primitive (i.e. face, edge or vertex). Therefore, by scan converting each polyhedron, every voxel lying within the band of width $2d$ will be assigned a distance value. Regions of intersection are scan converted once for each intersecting polyhedron and the minimum value is taken at each voxel. By this process, the intersection of two towers is divided along the dihedral angle bisector.

The goal of scan converting a polyhedron is to get the local distance field of a single mesh primitive. However, in the cases of an edge or a vertex, this field is not trilinear and thus cannot be obtained by a standard scan conversion as done by the graphics hardware. For this reason, the authors of the CSC algorithm proposed a software implementation.

3.3 A Hardware-Based Characteristics/ Scan-Conversion Algorithm

The computations for the Characteristics/Scan-Conversion algorithm described in section 3.2 can be divided into two parts. First, all polyhedra are constructed. The rest of the algorithm consists in scan converting these polyhedra and calculating the distances for each grid point inside the polyhedra. For standard mesh and grid resolutions, scan conversion and distance calculation are much more expensive than setting up the polyhedra.

There are two possible improvements which significantly speed up the algorithm. First, one could reduce the amount of polyhedron overlaps. Hence, also the number of points which are scan converted and the number of distances calculated is reduced. This approach will be discussed in Section 3.4. Second, one could exploit parallelism, for example by distributing the polyhedra on a cluster. But parallelism is limited, because one node has to gather all computed distances to build the final distance field by minimization.

Instead, a version that transfers the main workload to the graphics card was implemented. In order to display scenes with a large number of triangles at interactive rates, today's graphics hardware has been optimized for high-speed scan conversion of two-dimensional polygons. To reap the benefits of this computational power, 3D scan conversion was implemented as a series of 2D scan conversions on slices. Another feature of modern graphics cards is the facility of the graphics processor unit (GPU) to perform simple operations on a per-fragment basis. This programmability is crucial for a hardware-based implementation of CSC because the standard bilinear interpolation is not sufficient.

For each *xy*-slice of the grid, the cross sections of all polyhedra intersected by that slice are being computed. These cross sections are sent to the graphics card, which handles the remainder of the algorithm for that slice. That is, the graphics card computes the distance to the primitive for all points inside its corresponding polygon. If a point is inside more than one polygon, the minimum distance value is chosen. Similar to the algorithm described in Section 3.1, it is not possible to draw a planar polygon and use the depth buffer for the minimization process. The reason is that the distance function is not bilinear within one polygon if the base primitive is a vertex or an edge (see Fig. 2). One possible remedy would be to compute a tessellation of each polygon, approximating the radial distance value. Because the number of edges and vertices in the triangle mesh can be large, this approach leads to a vast amount of geometry data that has to be transferred to the graphics card for every slice. For a typical problem setup, this is not practical. Instead, one can use the observation that the vector to the closest point on the primitive is indeed a trilinear function within one polyhedron.

The operations available in a fragment program include normalization of vectors using cube maps and dot products. Therefore, it is possible to compute the distance value for all grid points inside a polyhedron slice without further tessellation. At each polygon edge, the vector to the closest point on the primitive is passed to the graphics card as a texture coordinate. The graphics card performs a bilinear interpolation of the texture coordinates within the polygon. For each pixel, the GPU computes the dot product of the texture coordinate and its normalized version. This value is used as the *z*-value for that pixel. The minimization process is performed by using the depth buffer. Furthermore, the stencil buffer is used to store the sign of the distance value. Because the stencil function cannot be altered within one polygon, regions of positive and negative signs are rendered one after the other. This is achieved by splitting towers into two regions, one for each sign. Distance values within wedges and cones have constant sign.

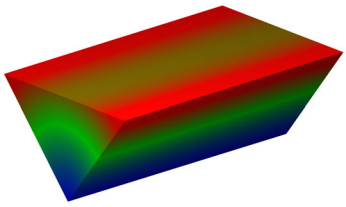

Figure 2: Distance field within a wedge type polyhedron.

The large number of polyhedra which need to be sliced and scan converted adds significantly to the overall computing time. For each polyhedron slice that has to be processed, rendering calls must be issued to define the geometric shape of the slice. The amount of time required for this operation is independent of the size of the slice. If the grid resolution is small in comparison to the triangle size, only a few distance values are calculated per polyhedron slice. Thus, sending the geometric information to the graphics hardware becomes the bottle neck and parallelism of CPU and GPU cannot be fully exploited. Therefore, the hardware version proves to be faster than the original software algorithm only if the number of triangles is relatively small compared to the voxel size.

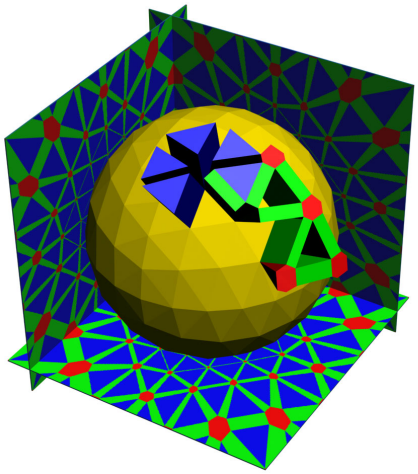

Figure 3: Sample Generalized Voronoi cells and slices generated by the HW-based CSC algorithm.

3.4 The Prism Scan Conversion Algorithm

To overcome this limitation, a modified algorithm based on fewer polyhedra is proposed in this section. The key idea is to build a three-sided pyramid frustum for each triangle that also encloses the grid points that are closest to the triangle's edges and vertices. To explain the construction, we will start with one triangle's tower from Section 3.2. This tower contains the region where the closest point on the surface is inside that triangle, up to a maximum distance. But it does not enclose any point where the closest point is on an edge or a vertex of the triangle. As a result, there is a gap on the convex side of the edge between two neighbouring triangles. This gap can be filled with a wedge as explained in Section 3.3. By using the observation that the towers of the two triangles overlap on the concave side of the edge, the gap can also be filled by tilting the side of both towers to the half angle plane between the two corresponding triangles. Because of the topology and for ease of notation, this new shape will be denoted with prism. The two prisms share one side, but do not necessarily end at a common edge. This can result in overlaps and gaps around the vertex as shown in Fig. 4. In order to ensure that the area around the surface is completely covered up to a maximum distance d, each prism is made to contain the three vertex normals of its base triangle, each scaled to the length d. This is achieved by shifting the prism sides outwards.

With this new approach, the total number of polyhedra is greatly reduced. Instead of computing one polyhedron per face, edge and slice, only one polyhedron per face is computed, reducing the overall number of polyhedra to less than one third, thereby reducing the data transfer to the graphics card per slice. The drawback of this approach is that the information about the closest primitive, i.e. face, edge, or vertex, is lost. Consequently, generalized Voronoi diagrams are no longer computed. Additionally, the vector to the closest point on the surface is no longer a trilinear

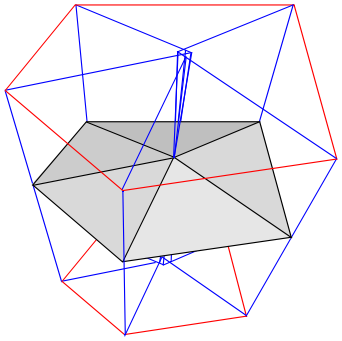

Figure 4: The polyhedra used by the modified algorithm.

function within the polyhedron. Hence, the approach of interpolating the distance vector within one polyhedron slice and computing the vector length on the GPU fails. Instead, OpenGL's ARB (Architecture Review Board) fragment program is used to calculate the distance to the triangle from a bilinearly interpolated vector on a per-fragment basis. This 3-vector defines the position of the fragment in the triangle's local coordinate frame. Therefore, the task of locating the closest point on the triangle can be performed in two dimensions. The third coordinate holds the signed distance to the triangle plane. After joining it to the difference vector of the two dimensional problem, the length of this vector equals the distance to the triangle. The sign of the distance equals the sign of the third coordinate.

The minimization process for regions of overlapping polyhedra is achieved by using the depth buffer. In order to avoid read back of more than one buffer, the signed distance is stored in a floating point pixel buffer.

4. IMPLEMENTATION

In this section, implementation details of the algorithm will be described. First, a short overview of the OpenGL implementation is given. Then, the setup of one triangle's non-orthogonal prism that has to be scan converted is described. Finally, the process of slice iteration is laid out.

As explained in section Section 3.4, OpenGL's ARB fragment program is used to calculate the distance to the triangle from a bilinearly interpolated vector on a per-fragment basis. This OpenGL extension is supported by several types of graphics hardware, such as Nvidia's GeForceFX and ATI's Radeon 9000 chipsets.

In each polyhedron, a local coordinate system *r, s* and *t,* defined by an axis frame aligned with the triangle is used. The *r*-axis is laid through the longest triangle side such that the three triangle vertices lie on the positive *r*-axis, the positive *s*-axis and the negative *r*-axis, respectively (see Fig. 5). Their distances from the origin are denoted by *a, h* and *b*. For each vertex of a polyhedron slice, the texture coordinate is used to define the position of the vertex in this local coordinate system. Texture coordinates are bilinearly interpolated within the polygon and therefore, the texture coordinate of a fragment always holds the position of the fragment in the local coordinate frame.

The task of the fragment program is now to compute the unsigned distance D(*r,s,t*) to the triangle from any point with texture coordinates *r*, *s* and *t*. The triangle itself is described by the three constants *a*, *h* and *b*.

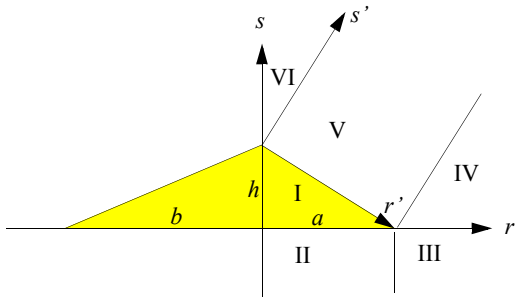

Figure 5: Distance computation in the t=0 projection plane.

First, we note that

$$D(r,s,t) = \sqrt{D(r,s,0)^2 + t^2}\,, \qquad (2)$$

so the main task is to compute *D(r,s,0)* which is a 2-dimensional problem. Furthermore, if *r* is negative, a reflection at the *s*-axis can be done. Thus, it is sufficient to treat the six regions labelled I through VI in Fig. 5. This leads to the following pseudo-code for the fragment program.

```
// Normalize to half-space r>=0
if (r<0) { r = -r; a = b; }

// Transform to a 2nd coordinate frame
lenSqr = a^2 + h^2;
r' = (a*r - h*s + h^2) / lenSqr;
s' = (h*r + a*s - a*h) / lenSqr;

// Extract components of the distance vector
r' = max(-r',r'-1,0);      // regions IV, V, VI
s' = max(s',0);            // regions I, V
r  = max(r-a, 0);          // regions II, III

// Compute the distance
if(s<0)
  dist = sqrt(r^2 + s^2 + t^2);
else
  dist = sqrt( (r'^2 + s'^2) * lenSqr + t^2);

// Place sign
dist = copysign(dist, t);
```

Figure 6: Fragment program pseudo-code which computes the distance to a triangle in local coordinates on a per-fragment basis.

In order to achieve high-precision results, a floating point pixel buffer is used to store the distance values calculated by the fragment program. Nvidia and ATI cards both support floating point buffers. Additionally, read backs from floating point buffers were found to be much faster than two subsequent read backs of depth buffer and stencil buffer, as proposed by the algorithm in Section 3.3. Yet, for a typical problem setup, the read back of the calculated distance field consumes 20-25% of the overall computational time.

The polyhedra which are sliced and then sent to the graphics card are computed for all triangles in a setup step. First, an orthogonal prism which entirely encloses all points that lie within the requested maximal distance *d* is built for all triangles. Then, the prism is clipped by the three half angle planes between the triangle

and one of its neighbors. During the clipping process, it is ensured that the topology of the prism stays constant by possibly reducing the clipped volume. This simplifies the computation of intersections during the slice iteration. As explained in Section 3.4, it is also ensured that the vertex normal is contained in all prisms of adjacent triangles.

After clipping the prism, the coordinates of the corner points in the triangle's local coordinate frame are computed. The edges and vertices of the prism are stored in a graph. During slice iteration, the graph is traversed along the z-coordinate of the vertices. An active edge table stores all edges intersected by the current slice. When a vertex is passed during slice iteration, all ingoing edges of the graph node are deleted from the active edge table and replaced with the outgoing edges.

All prisms are sorted according to their first slice intersection. The slice iteration process can now be initiated. All prisms that are intersected by the first slice are copied to an active prism table. Only prisms in this table need to be considered for the current slice. After rendering all prism intersections, the distance field of that slice stored in the pixel buffer is read from the graphics card memory. Now, the slice can be advanced. For all active edges of all active prisms, both local and world coordinates of the intersection are incrementally updated. If a corner of a prism was passed when advancing the slice, the active edge table of that prism needs to be updated. All ingoing edges of the graph node corresponding to the prism corner are deleted and replaced by the outgoing edges, where in and out is defined by the z-direction of the edge. Once the active edge table of a prism is empty, the prism is deleted from the active prism table. The distance field is computed by repeating these steps until all slices are processed.

5. RESULTS

For a performance evaluation, we compared computing times of our hardware-assisted Prism algorithm with the software CSC algorithm. The CSC algorithm was downloaded from the URL [9]. Both programs were run on a 2.4 GHz Pentium 4 equipped with 2 GB of RAM and an ATI Radeon 9700 PRO graphics card.

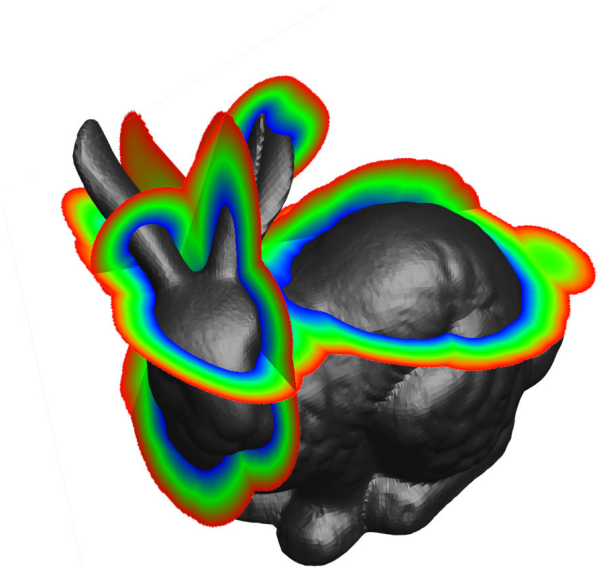

Figure 7: Three distance field slices of the Stanford Bunny computed by the HW-based Prism algorithm.

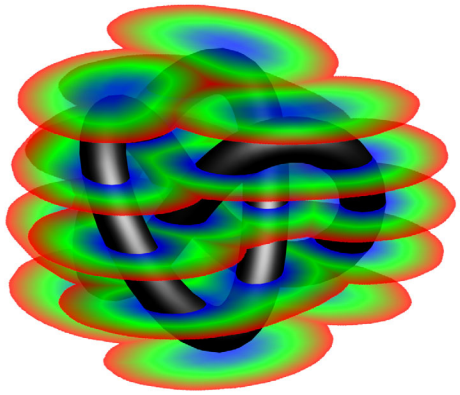

Figure 8: Five translucent distance field slices of a knot computed by the HW-based Prism algorithm.

For a first comparison, both programs were fed with the Stanford bunny dataset, a mesh with 69'451 triangles, and a torus knot with 2880 triangles. As can be seen in Table 1, the two models of different size and complexity result in a speed up greater than five compared to the original software algorithm.

Model	Triangles	Software Algorithm (seconds)	Hardware Algorithm (seconds)	Speedup
Bunny	69'451	19.408	3.737	5.19
Knot	2'880	6.437	1.176	5.47

Table 1: Timings (in seconds) for Bunny and Knot data sets. The band width is set to 10% of the model extent, the grid resolution is set to 256^3.

To analyze the performance of the two methods as a function of problem size, we chose as a simple refinable model a tessellated sphere obtained by repeatedly subdividing an octahedron. Table 2 shows a good speedup factor for coarse and medium fine meshes. Performance remains reasonable even if the triangle size comes close to the voxel size. In the mesh of 131'072 triangles, the average area is only $\pi/2$ times the voxel size.

Mesh Size	Software Algorithm	Hardware Algorithm	Speedup
2'048	5.506	0.796	6.92
8'192	5.177	0.918	5.64
32'768	6.162	1.346	4.58
131'072	10.387	3.151	3.30

Table 2: Timings (in seconds) with variable input mesh size for 0.1 band width and a 256^3 grid.

Table 3 demonstrates the effect of varying the width of the band, i.e. the computational domain. For both programs, a linear dependency is expected, plus a constant setup time. Although the hardware method needs a longer setup time, it is efficient for thin bands, too. With increasing domain size, the setup time becomes less important and the speedup factor improves.

Width of band	Software Algorithm	Hardware Algorithm	Speedup
0.1	6.162	1.346	4.58
0.2	11.701	1.785	6.56
0.4	21.009	2.546	8.25
0.8	34.489	3.724	9.26

Table 3: Timings (in seconds) with variable band width for a tessellated sphere with 32'768 triangles on a 256^3 grid.

Finally, keeping the mesh and the band constant but increasing the grid resolution shows the usefulness of our method. The speedup factor is expectedly more or less constant up to the point where the memory size becomes an issue. In contrast to the software CSC algorithm, our Prism algorithm does not have to keep the full grid in memory (see Table 4).

Grid Resolution	Software Algorithm (seconds)	Hardware Algorithm (seconds)	Speedup
$64 \times 64 \times 64$	0.901	0.244	3.69
$128 \times 128 \times 128$	1.638	0.482	3.40
$256 \times 256 \times 256$	6.162	1.346	4.58
$512 \times 512 \times 512$	109.400	6.534	16.7

Table 4: Timings (in seconds) with variable grid resolution for a tessellated sphere with 32'768 triangles and 0.1 band width.

6. CONCLUSION

In this paper we have shown that today's graphics hardware is suitable for supporting the computation of distance fields of triangle meshes on a regular grid. The construction of simple polyhedra containing the Voronoi cell of a single primitive of the input surface was presented. An algorithm was proposed to scan convert these polyhedra and compute a signed distance field up to a maximum distance. In order to exploit parallelism, slices of the polyhedra are computed by the CPU while scan conversion, distance computation, and minimization performs entirely on the GPU. By using OpenGL's ARB fragment program, it is possible to achieve the nonlinear interpolation of distance values within one polyhedron slice. This avoids fine tessellation of a polyhedron slice and reduces the number of geometry sent to the graphics card, which raises the hardware assisted version to a competitive level. Also, the slice-oriented calculation allows a smaller memory footprint than the polyhedron-oriented software version of [9]. To further reduce the CPU workload and the amount of data being sent to the graphics card, polyhedra are constructed for triangles only and are assured to completely cover the area around the mesh up to a maximum distance. The calculation of the distance to the closest point on the triangle is simple enough to be performed per-fragment on the GPU. Using full computational power of both CPU and GPU, our algorithm turns out to be significantly faster than the pure software version.

Acknowledgments

This work was partially funded by Schlumberger Cambridge Research.

References

[1] D.E. Breen, S. Mauch and R.T. Whitaker. 3D Scan Conversion of CSG Models into Distance, Closest-Point and Colour Volumes. In M. Chen, A.E. Kaufman, R. Yagel (eds.), *Volume Graphics,* Springer, London, Chapter 8, 135-158, 2000.

[2] D. Cohen-Or, D. Levin, A. Solomovici. Three-dimensional distance field metamorphosis, *ACM Transactions on Graphics*, 17(2), 116-141, April 1998.

[3] O. Cuisenaire. Distance Transformations: Fast Algorithm and Applications to Medical Image Processing, *P.h.D. thesis,* Université Catholique de Louvain, Dept. of Engineering, Louvain-la-Neuve, France, 1999.

[4] H. Eggers. Two Fast Euclidean Distance Transformations in Z^2 Based on Sufficient Propagation, *Computer Vision and Image Understanding* 69(1), 106-116, January, 1998.

[5] S.F. Frisken, R.N. Perry, A.P. Rockwood, T.R. Jones. Adaptively sampled distance fields: a general representation of shape for computer graphics, *Proceedings of the 27th annual conference on Computer graphics and interactive techniques,* July 2000.

[6] S.F.F. Gibson. Using distance maps for accurate surface representation in sampled volumes, Proceedings of the 1998 IEEE symposium on Volume visualization, October 1998.

[7] K. Hoff, T. Culver, J. Keyser, M. Lin, and D. Manocha. Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware. *SIGGRAPH 99,* 277-285, 1999.

[8] J. Huang, Y. Li, R. Crawfis, S.C. Lu, S.Y. Liou. A complete distance field representation, *Proc. 12th IEEE Visualization,* 247-254, 2001.

[9] S. Mauch. A Fast Algorithm for Computing the Closest Point and Distance Transform, http://www.acm.caltech.edu/~seanm/software/cpt/cpt.pdf .

[10] C.R. Maurer, R. Qi, V. Raghavan. A Linear Time Algorithm for Computing Exact Euclidean Distance Transforms of Binary Images in Arbitrary Dimensions, *IEEE Trans. Pattern Analysis Mach. Intell.,* 25(2), February 2003.

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

[12] A. Rosenfeld, J.L. Pfaltz. Distance Functions on Digital Pictures, Pattern Recognition, 1(1), 33-61, 1968.

[13] J.A. Sethian. A Fast Marching Level Set Method for Monotonically Advancing Fronts. In *Proc. Nat. Acad. Sci.,* vol. 94, 1591-1595, 1996.

[14] M. Sramek, A. Kaufman. Fast Ray-Tracing of Rectilinear Volume Data Using Distance Transforms, *IEEE Trans. Visualization Computer Graphics,* 6, 236-252, 2000.

[15] Y.R. Tsai. Rapid and Accurate Computation of the Distance Function Using Grids. *Technical Report, Department of Mathematics, University of California,* Los Angeles, 2000.

[16] M. Wan, F. Dachille, A. Kaufman. Distance-Field-Based Skeletons for Virtual Navigation, *Proc. 12th IEEE Visualization,* 239-245, 2001.