Eurographics Symposium on Geometry Processing (2003), pp. 1–8
L. Kobbelt, P. Schröder, H. Hoppe (Editors)

Efficient Max-Norm Distance Computation and Reliable Voxelization

Gokul Varadhan[1], Shankar Krishnan[2], Young J. Kim[1], Suhas Diggavi[2] and Dinesh Manocha[1]

[1]Department of Computer Science, University of North Carolina, Chapel Hill, U.S.A
[2]AT&T Labs - Research, Florham Park, New Jersey, U.S.A

Abstract

We present techniques to efficiently compute the distance under max-norm between a point and a wide class of geometric primitives. We reduce the distance computation to an optimization problem and use our framework to design efficient algorithms for convex polytopes, quadrics and triangulated models. We extend them to handle large models using bounding volume hierarchies, and use rasterization hardware followed by local refinement for higher-order primitives. We use the max-norm distance computation algorithm to design a reliable voxel intersection test to determine whether the surface of a primitive intersects a voxel. We use this test to perform reliable voxelization of solids and generate adaptive distance fields that provide a Hausdorff distance guarantee between the boundary of the original primitives and the reconstructed surface.

1. Introduction

The notion of a *distance function* between two elements of a set (or metric space) is fundamental in various branches of mathematics and applied sciences *e.g.*, approximation theory and numerical analysis. It is considered a fundamental problem in geometric computation and related areas including robot motion planning [22], implicit and volume modeling [11, 26, 38], surface reconstruction [8, 17], physically-based modeling [3], computer-aided design [10], etc. This problem has been actively studied in different fields and most of the algorithms have been proposed for efficient computation of Euclidean distance between two sets.

In this paper, we mainly focus on the max-norm (or l_∞) distance computation. Under this norm, the distance between two points $\mathbf{x}$ and $\mathbf{y}$ (in d dimensions) is represented as $D_\infty(\mathbf{x},\mathbf{y})$ and is defined as

$$D_\infty(\mathbf{x},\mathbf{y}) = \max_i |x_i - y_i|, \qquad i = 1,2,\ldots,d \qquad (1)$$

We can extend this definition for distance between a point $\mathbf{p}$ and a set $\mathcal{S} \subseteq \mathbb{R}^d$ †. Computing distances under the max-norm has an important difference from the Euclidean case : l_∞ is not induced by an inner product space, so notions of orthogonality for distance computation cannot be used. The max-norm distance problem arises in different application including planning under uncertainty using Markov decision processes in machine learning [15, 36], image analysis [24], dynamics and control systems [13], tolerance analysis and NC machining [10, 33] and volume graphics [11, 38]. Unlike Euclidean distance computation, no efficient and practical algorithms are known for max-norm computation.

One of our motivations for max-norm computation arises from *voxelization* of geometric primitives in $\mathbb{R}^3$. Given a geometric scene description, voxelization deals with techniques that generate a discrete set of voxels to approximate the continuous scene as faithfully as possible. Voxelization is used in ray tracing [37] and volume rendering [26, 38], implicit modeling [18, 20], shape representation [11] and model repair [29]. In order to produce an accurate voxelization and guarantee Hausdorff-distance approximation, it is essential to know whether or not some part of the geometric model passes through a voxel. We refer to this test as the *voxel-intersection test*. It is not difficult to show that an exact voxel-intersection test can be reduced to a max norm distance computation between the center of the voxel and the primitive.

Main Contributions In this paper, we present algorithms for efficient max-norm distance computations between a point and a wide class of geometric primitives. We analyze the problem of max-norm computation and reduce it to an optimization problem. Based on our optimization framework, we present efficient and specialized algorithms for convex polytopes, quadrics and polygonal models. We also present efficient techniques based on bounding volume hierarchies and rasterization hardware to extend these algorithms to large models. Overall, we show that max-norm computation is no more expensive than the Euclidean case. On the contrary, in many cases it is cheaper to compute because the corresponding distance functions are linear rather than quadratic and we utilize this property to develop efficient algorithms.

We demonstrate the application of max-norm distance computation to perform the voxel-intersection test. It is used to generate an adaptive distance field (ADF) of complex models defined using Boolean operations where the under-

† $D_\infty(\mathbf{p},\mathcal{S}) = \min_{\mathbf{s}\in\mathcal{S}} D_\infty(\mathbf{p},\mathbf{s})$

lying models consist of polyhedra, quadrics and tori. The efficient voxel-intersection tests takes a small percentage of additional time in terms of ADF generation and guarantees no missed components and a bounded Hausdorff-error on the approximated samples as well as the reconstructed surface.

Some of our new results include:

- An optimization-based framework for max-norm computation
- Specialized algorithms for convex polytopes, quadric and triangulated models.
- An efficient graphics hardware-based approximate solution for general models.
- An efficient and exact voxel-intersection test for voxelization and ADF computation.

Organization The rest of the paper is organized as follows. We briefly survey related work on distance computation and voxelization in Section 2. We reduce the max-norm computation problem to an optimization problem in Section 3 and present specialized algorithms for convex polytopes, quadric and triangulated models. We extend these algorithms using bounding volume hierarchies and graphics hardware to handle large models and non-convex primitives. We use our algorithm to perform voxel-intersection tests and ADF generation in Section 5 and highlights its performance on different benchmarks in Section 6.

2. Prior Work

In this section, we give a brief overview of prior work on distance computation, voxelization and adaptive sampling.

2.1. Distance Computation

The problem of distance computation between various primitives under Euclidean norm is well studied in computational geometry, robotics, and simulated environments. Check out a survey [23].

The distance computation under max-norm in itself has not been extensively studied in the literature. However, there is considerable amount of work for various geometric or proximity computations under l_∞ norm. These include the study of l_∞ Voronoi diagram and its combinatorial and complexity [4, 6, 12, 21, 30, 31], and l_∞ *skeleton* computations [2]. In particular, Papadopoulou *et al.* [31] have presented $O(n \log n)$ algorithms to compute the 2D l_∞ Voronoi diagram of polygons and highlighted its application to VLSI layout and manufacturing. However, no practical algorithms or implementations are known for 3D l_∞ Voronoi diagrams of point sets or higher order primitives.

2.2. Distance Fields and Voxelization

Many efficient algorithms are known to compute the distance fields and their gradients at any point in space. A good overview of these algorithms has been given in Cuisenaire's dissertation [7]. A key issue in generating discrete samples is the underlying sampling rate. Some of the common algorithms use an adaptive refinement strategy based on an octree, and only split those cells that contain a piece of the final surface in a top-down manner. However, the criterion for performing the containment test, i.e., whether the surface passes through a voxel, may not be robust. Many authors have used curvature information in generating the distance samples [14, 35]. Moreover, Frisken *et al.* [11, 32] have presented bottom-up and top-down methods for generating ADFs based on piecewise tri-linear interpolation.

3. Distance Computation under l_∞ Norm

The problem of computing the distance under any norm from a point to a set is by definition an optimization problem. Our goal is to utilize the special structure of the distance function and the underlying space $\mathcal{S}$ to formulate efficient algorithms. Computing the max norm distance of a point from a set is substantially different from the Euclidean case in several respects. First, the distance metric is not smooth with respect to its variables. Secondly, the l_∞ space is not an inner product space, unlike the l_2 space. The relationship between orthogonality and minimum distances in inner product spaces can be very powerful in formulating these problems without using optimization. In the minimum distance problem, these differences translate to changes in both the algorithmic approach and the characteristics of the solution. In the rest of this section, we first present an optimization based framework to compute the max-norm and later present specialized algorithms for convex polytopes, quadrics and triangulated models.

3.1. Optimization Framework

Let us assume that the set $\mathcal{S}$ to which we need to find the closest distance consists of points satisfying all $f_i(\mathbf{x}) \leq 0, i = 1, 2, \ldots, n$, where each f_i is a non-linear analytic function. Without loss of generality, we can assume that the point $\mathbf{p}$ from which we are computing the closest distance is the origin.

We explain our algorithm for the 2D case first. Consider partitioning the plane into regions such that the distance from any point in a region to the origin is determined by the same coordinate. This partition exists because of the definition of the norm. As shown in Fig. 1(b), the regions where the x_1-coordinate determines the l_∞ distance is given by the sets $R_{x_{11}} = \{x_1 - x_2 \geq 0 \ \wedge \ x_1 + x_2 \geq 0\}$ and $R_{x_{12}} = \{x_1 - x_2 \leq 0 \ \wedge \ x_1 + x_2 \leq 0\}$. Each region, $R_{x_{11}}$ and $R_{x_{12}}$, is bounded by two linear constraints. The regions where x_2 determines the distance, $R_{x_{21}}$ and $R_{x_{22}}$, are obtained by similar linear constraints. The four regions for the two-dimensional case is shown in Fig. 1(b).

Now let us assume that we are restricted to one such region, say $R_{x_{11}}$. By adding the additional constraint for $\mathbf{x}$ to belong to $\mathcal{S}$, our constraint space is restricted to a portion of the primitive lying inside $R_{x_{11}}$. We can find the shortest distance from the origin to this part of the surface by minimizing x_1. Note that if our constraint space was contained in $R_{x_{12}}$, our objective function would be to minimize $-x_1$. This is a simple linear function.

Extending this formulation to the d-dimensional case, we

see that the underlying space is partitioned into $2d$ regions (each region formed by $2(d-1)$ linear constraints) and each coordinate determines the distance in two regions. For example, the regions where the i^{th} coordinate determines the distance are $R_{x_{i1}} = \bigcap_{j\neq i, j=1,\ldots,d}(x_i - x_j \geq 0 \ \wedge \ x_i + x_j \geq 0)$ and $R_{x_{i2}} = \bigcap_{j\neq i, j=1,\ldots,d}(x_i - x_j \leq 0 \ \wedge \ x_i + x_j \leq 0)$. We have now reduced our minimum distance computation problem to solving $2d$ non-linear optimization programs. Each program has the form

$$\begin{aligned} \text{minimize} \quad & \mathbf{c^T x}, \\ \text{subject to} \quad & f_i(\mathbf{x}) \leq 0, i = 1,2,\ldots,n, \\ \text{and} \quad & \mathbf{g_j^T x} \geq 0, j = 1,2,\ldots,2(d-1). \end{aligned} \tag{2}$$

We use the above formulation to develop efficient algorithms for the the case of convex primitives. For the case of non-convex implicit functions, we develop a strategy based on the graphics hardware to compute a good initial guess. This is presented in section 4.2.

3.1.1. Distance Computation for Convex Primitives

In this section, we present an exact algorithm to compute the distance under l_∞ norm from a point to a convex primitive. The interior of a convex primitive satisfies $f_i(\mathbf{x}) \leq 0, i = 1,2,\ldots,n$, where each f_i is a convex function. We solve the problem by dividing it into two cases depending on whether the point $\mathbf{p}$ lies *inside* or *outside* the primitive.

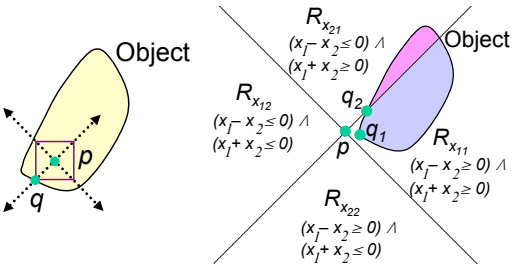
Point inside the primitive Consider the convex primitive and the point $\mathbf{p}$ in 2D as shown in Fig. 1(a). All points that are equidistant from $\mathbf{p}$ lie on the surface of an axis-aligned square centered at $\mathbf{p}$. This relation is shown by the square in the Fig. 1(a). Consider growing such a square from the point $\mathbf{p}$. The shortest distance from $\mathbf{p}$ to the surface of the object is realized by a point on the surface that first touches the growing square (point $\mathbf{q}$ in the figure). However, it is easy to see that for convex primitives only the vertices of the square are potential candidates to touch the surface first. This property reduces the task of finding the distance to that of finding the minimum from four directed distance queries. The directions in 2D are all possible combinations of the vectors $\left(\pm 1/\sqrt{2}, \quad \pm 1/\sqrt{2}\right)$.

This technique is easily extendible to the $d-$dimensional case. We can write the max-norm distance as

$$D_\infty(\mathbf{p}, \mathcal{S}) = \frac{1}{\sqrt{d}} \min_i D_{\vec{v}_i}(\mathbf{p}, \mathcal{S}), \qquad i = 1,2,\ldots,2^d,$$

where $\vec{v}_i$ is chosen from the set $\{-1/\sqrt{d}, 1/\sqrt{d}\}^d$ and $D_{\vec{v}}$ is the directed distance along vector $\vec{v}$. Algorithms to compute the directed distance between a point and a surface are efficient and well-known.

Point outside the primitive Consider the case when $\mathbf{p}$ lies outside the object as shown in Fig. 1(b). In this case, we use the optimization formulation presented in section 3.1. However in this case, the constraints described in Eq. 2 are all convex. This reduces the more general optimization formulation to a special convex programming problem. Many convex programming problems can be solved exactly using interior point methods [28]. However, the restricted class of convex



(a) (b)

Figure 1: Computing distance from a point to a convex primitive under l_∞ metric. (a) point inside primitive (b) point outside primitive

primitives that are composed of linear and quadric surfaces can be solved very efficiently. This class is rich enough to be of interest in applications like geometric modeling.

3.1.2. Distance Computation for Convex Polytopes and Quadrics

For quadrics, we can write the interior of the primitive using quadric constraints $\mathbf{x^T A x} + \mathbf{b^T x} + c \leq 0$, where $\mathbf{A}$ is a symmetric positive definite matrix, $\mathbf{b}$ is a fixed vector, and c is a constant scalar. The corresponding convex program is converted to a special case called *second-order cone program* for which a number of efficient and implementable interior-point algorithms are known [25]. These algorithms are iterative in nature, and each iteration takes time that is linear in the number of constraints. The second-order cone program that we solve has the form

$$\begin{aligned} \text{minimize} \quad & \mathbf{h^T x}, \\ \text{subject to} \quad & \| \mathbf{A_i x} + \mathbf{b_i} \|_2 \leq \mathbf{c_i^T x} + d_i, i = 1,2,\ldots,n, \\ \text{and} \quad & \mathbf{g_j^T x} \geq 0, j = 1,2,\ldots,2(d-1). \end{aligned}$$

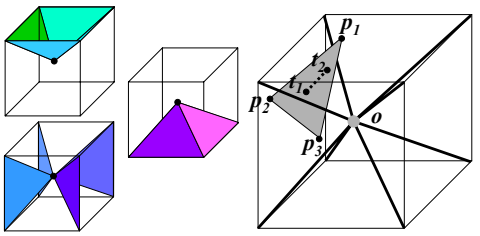
The constraints listed above also include the special case of convex polytopes (by making $\mathbf{A} = \mathbf{0}$), where the second-order cone program reduces to the more familiar linear program. Many simple and practical linear-time algorithms for solving linear programming problems in a fixed dimension are known [34]. Given a quadric primitive in 3D, we solve six cone programs (each with four linear and one quadratic constraint) and choose the minimum value among them to find the true distance.

3.2. Triangulated Models

In case of a non-convex polyhedron or triangulated models, we compute the l_∞ distance by finding distance for each polyhedral element in the primitive (i.e., polygon or triangle) and minimizing it overall. We explain how we compute l_∞ distance between a point and a triangle efficiently and also propose a hierarchical method to extend this triangle-based computation to a polyhedral primitive.

3.2.1. Distance Computation for a Triangle

In section 3.1.2, we presented a procedure to compute l_∞ distance to a convex polytope based on a linear programming



(a) 12 Partitioning Triangles (b) l_∞ Computation for a Triangle

Figure 2: Computing distance from a point to a triangle under l_∞ metric.

technique. The distance computation for a triangle $\triangle^T$ is a simple variation of the same technique. In case of a triangle, we reduce the problem to computing intersections between the target triangle $\triangle^T$ and 12 auxiliary *partitioning triangles* $\triangle^B$. In fact, these 12 $\triangle^B$'s represent the linear constraints $\mathbf{g_j}$ highlighted in Section 3.1.2; these 12 constraints are illustrated in Fig. 2(a). Notice that even though these $\mathbf{g_j}$'s form unbounded partitions of 3D space, in practice, we bound the partitions by using an axis-aligned bounding box of $\triangle^T$ such that the boundary of each partition becomes a triangle $\triangle^B$.

Once we have the $\triangle^B$'s, the next step is to compute all possible intersecting lines between $\triangle^T$ and $\triangle^B$'s, and to extract their end points. Then, the l_∞ distance from a query point to $\triangle^T$ is the minimum of l_∞ distances from the query point to all the end points as well as to the vertices comprising $\triangle^T$. For example, as illustrated in the left figure of Fig. 2(b), the distance from $\mathbf{o}$ to a triangle $\triangle^T_{\mathbf{p_1p_2p_3}}$ is the minimum of the distances from $\mathbf{o}$ to the vertices $\mathbf{p_1}, \mathbf{p_2}, \mathbf{p_3}$ as well as to $\mathbf{t_1}, \mathbf{t_2}$, which are the end points of the intersections between 12 partitioning triangles and $\triangle^T_{\mathbf{p_1p_2p_3}}$. and we take the minimum of the distance values from $\mathbf{o}$ to $\mathbf{p_1}, \mathbf{p_2}, \mathbf{p_3}$, $\mathbf{t_1}$ and $\mathbf{t_2}$.

4. Complex Models

In the previous section, we have presented efficient algorithms for max-norm distance computation to convex polytopes, quadrics and triangles. In this section, we present two algorithms to extend them to large, complex models. These are based on bounding volume hierarchies and use of graphics hardware.

4.1. Bounding Volume Hierarchy

A simple way to compute l_∞ distance for a non-convex polyhedron P is to compute the distance for every triangle $\triangle_i \in P$ and take its minimum. However, we can speed up this naive method by constructing a hierarchical bounding volume (BVH) of P and culling away unnecessary triangles by traversing the hierarchy. For the hierarchical representation, we employ a surface convex decomposition scheme similar to Ehmann *et al.* [9]. Here, a leaf node in the BVH is created by decomposing P into a collection of convex surface patches P_i and computing its convex hull. Notice that, due to the convex hull computation, the node creates some extraneous triangles that do not belong to P. Let us call these types of triangles *virtual*, and otherwise call them *real*. Then, the entire BVH is recursively built by merging children nodes in the hierarchy and computing their convex hull.

Once we have precomputed the BVH, at query-time, we traverse the BVH in a top-down manner starting from a root node. During the traversal, we maintain three types of distance values:

- U^B : Upper bound to the distance value from a given query point $\mathbf{o}$ to the polyhedron P.
- U_b : Upper bound to the distance value from $\mathbf{o}$ to the currently visited node N in the BVH. U_b is obtained by computing minimum distance only to the real triangles contained in N.
- L_b : Lower bound to the distance value from $\mathbf{o}$ to N. L_b is obtained by computing minimum distance to all the real and virtual triangles contained in N.

While we traverse the BVH, U_b is compared to U^B, and if U_b is smaller than U^B, then U^B is updated to U_b. As a result, as we go down to the deeper level of the BVH, U^B decreases and it finally computes the actual distance to P. Using U^B and L_b of a currently visited node N, we perform culling as follows: whenever we encounter N in the BVH whose L_b is greater than U^B, we can immediately reject all the triangles contained in N.

The problem of computing $D_\infty()$ gets much harder when dealing with non-convex curved or implicit primitives. To avoid solving a general non-linear optimization problem as described in section 3.1, we tessellate the primitives within some Hausdorff distance error bound ε and obtain an estimate for $D_\infty()$ using the graphics hardware. This is followed by a refinement step using local optimization. We describe the hardware algorithm next.

4.2. Distance computation using graphics hardware

Our approach is based on the algorithm presented by Hoff *et al.* [16] for constructing generalized Voronoi diagrams using graphics hardware for 3D polygonal objects. The distance field is computed by rendering the 3D polygonal mesh approximations to the distance function where the depth of the rendered mesh at a particular pixel location corresponds to the distance to the nearest polygon feature. The resulting distance field can be obtained by reading back the depth buffer. The 3D distance field is computed one slice at a time.

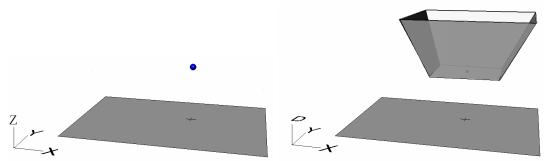
We compute a distance field under the l_∞ metric. For each site, we define a distance function, which gives, for any point, the distance to that site with respect to l_∞ metric. In contrast to l_2, the l_∞ distance functions for the case of a point, line segment and a polygon are linear. They can be represented exactly by a collection of polygons.

4.2.1. Distance functions

We present the max-norm distance functions associated with different primitives.

Points: The distance function for a point site $\mathbf{p}$ is shown in Fig. 3. Its graph is a frustum of a square pyramid. The region

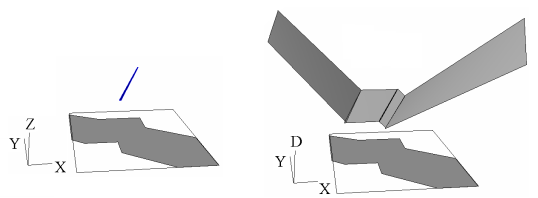
of influence for a point is the entire slice. The bottom square base of the pyramid corresponds to a region of constant distance. The four slanting faces of the pyramid correspond to the planes $x = z$, $x = -z$, $y = z$, $y = -z$. The distance at a point on the region of influence is half the length of the smallest isothetic cube centered at the point and touching **p** at one of the cube faces.



(a) (b)

Figure 3: Distance function for a point (shown in blue) is a frustum of a square pyramid. Figs (a) & (b) show the region of influence and distance function respectively. The region of influence (shaded region on the slice) is the entire slice.

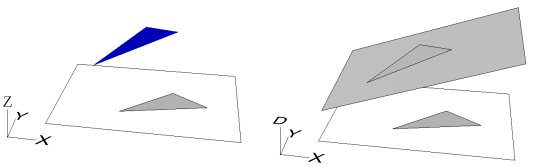
Line Segments: The distance function for a line segment l is composed of three parts: one for the segment itself and one for each endpoint. The endpoints are treated the same way as points. The distance function and region of influence for the line segment is shown in Fig. 4. The distance function is composed of four planar regions. The distance at a point on the region of influence is half the length of the smallest isothetic cube centered at the point and touching l along one of the cube edges.



(a) (b)

Figure 4: Distance function of a line segment (shown in blue): Figs (a) & (b) show the region of influence and distance function respectively. The region of influence is the shaded region on the slice. The distance function is composed of four planar regions.

Polygons: The distance function for a polygon is composed of a distance function for the polygon itself and one for each vertex and edge. The distance function for a triangle $\triangle$ is a plane as shown in Fig. 5. The region of influence is a triangle. The distance at a point on the region of influence is half the length of the smallest isothetic cube centered at the point and touching $\triangle$ at one of the cube vertices. The region of influence is obtained by projecting the vertices of the triangle onto the slice along one of four directions: $(1,1,1)$, $(-1,1,1)$, $(1,-1,1)$ and $(-1,-1,1)$. If $\hat{\mathbf{n}} = (n_1, n_2, n_3)$ denotes the normal of triangle $\triangle$, we choose the direction vector $(s_1, s_2, 1)$ where s_i $(i = 1,2)$ is 1 or -1 depending on whether n_i is greater than zero or not. If the polygon intersects the slice, the intersection is computed and the polygon is decomposed into two sub-polygons. Each sub-polygon is treated as above.



(a) (b)

Figure 5: Distance function of a triangle (shown in blue) is a plane. Figs (a) & (b) show the region of influence and distance function respectively. The region of influence is a triangle (shaded region on the slice).

4.2.2. Sources of Error

There are two sources of error in the distance computation:

- **Tessellation Error:** It arises from approximating a non-convex implicit or curved primitive by a polygonal mesh.
- **Hardware Precision Error:** This error is introduced by the limited precision of the graphics hardware.

The total error is the sum of the above two errors. We bound the tessellation error by performing a bounded-error tessellation of the non-convex or curved primitive. In this manner, we obtain a bound on the total error. We obtain conservative estimates on the distance by offsetting the distance functions of the primitives by an amount equal to the error bound.

4.3. Non-convex Implicit Primitives

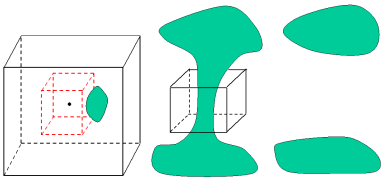
We refine the estimate obtained from the graphics hardware by performing non-linear optimization as a post-processing step. Since the estimate obtained from the hardware procedure is usually close to the right answer, this can be refined quite efficiently using a local optimization tool.

Let the implicit function surface be given by the equation $f(\mathbf{x}) = 0$. Without loss of generality, let the point from which we are computing this distance be the origin **o** and let $f(\mathbf{o}) > 0$. Under these assumptions, the constraint set that we will be using in the optimization process is $G(\mathbf{x}) : f(\mathbf{x}) \leq 0$.

We use the hardware not only to compute the distances but also to find which triangle realized the minimum distance at every point. We then use the point-triangle distance test described in section 3.2.1 to determine the exact point **q** that minimizes the distance. Now if **q** satisfies the constraint $G(\mathbf{x})$, then we use this as the starting point in the optimization. If it does not, we perturb **q** so that it does. We use the fact that the original tessellation is within a Hausdorff error of ε. If $\hat{\mathbf{n}}$ is the unit normal to the triangle containing **q**, then one of the points $\mathbf{q} \pm 2\varepsilon\hat{\mathbf{n}}$ is expected to satisfy our constraint. We use this point as our initial estimate and then refine it using a non-linear optimization solver like LOQO [1].

5. Reliable Voxelization Algorithm

A number of iso-surface extraction algorithms have been proposed for conversion from a volume representation of an object to a polygonal mesh representation of the surface. Many of these are grid-based and use the Marching



(a) (b) (c)

Figure 6: Voxel-Intersection Test: Figs (a) & (b) show a surface (in green) that passes through a voxel without intersecting any edges. The presence of such voxels can result in missed components and unwanted handles in the reconstructed surface as shown in Fig. (c). We use the l_∞ distance (indicated by the red dotted cube) to perform a voxel-intersection test. The surface intersects the voxel if and only if l_∞ distance between the center of the voxel (black dot) and the surface is less than half the voxel size.

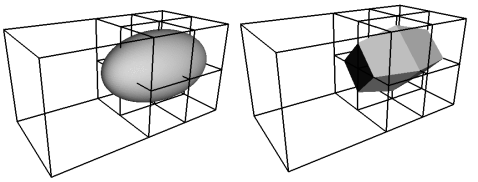
Cubes algorithm or its variants [26, 20, 18]. These algorithms detect whether a surface intersects a voxel by checking for sign change in the implicit function across the edges of the voxel. The accuracy of these algorithms is mainly dependent on the resolution of the underlying grid. Insufficient grid resolution can cause components to be missed or create unwanted handles as shown in Fig. 6. As a result, these algorithms cannot provide Hausdorff distance guarantees on the output of the reconstruction. In case of adaptive grids, it is possible that a surface passes through a coarse voxel without intersecting any edges, while it intersects the edges of a neighboring voxel that is at a finer resolution (see Fig. 7). This can result in cracks in the reconstructed surface. These problems occur because the surface intersects the voxel although the voxel doesn't exhibit a sign change. We present a voxel-intersection test and use this test to perform reliable voxelization and adaptive grid generation in order to provide Hausdorff guarantees.

5.1. Voxel-Intersection Test

The surface can pass through a cell without intersecting any of the edges. We use an exact test based on computing the $l_\infty()$ distance between the center of a voxel and the primitive. Our test is based on the fact that a voxel is intersected by the surface if the $l_\infty()$ distance at the center of the voxel is less than half the voxel size (see Fig 6). The above statement is valid even when the voxels are not regular-sized cubes. Given a voxel with dimensions a, b, c along the three coordinate axes, a weighted norm defined as $\max_i w_i|x_i - y_i|$, where $w_i = 1/a$, $1/b$, and $1/c$, for $i = 1$, 2, and 3 respectively, preserves the exactness of the voxel-intersection test.

5.2. Adaptive Grid Generation for Hausdorff Guarantee

Given a surface S, the goal of grid generation is to compute a set of discrete samples to approximate S. Suppose the reconstruction algorithm applied to the set of samples generates $\hat{S}$. A Hausdorff guarantee on $\hat{S}$ requires that given any $\varepsilon > 0$, it is possible to bound the Hausdorff distance between S and $\hat{S}$ to be less than ε. We noted earlier that we cannot provide such a guarantee if the grid has *complex* voxels, i.e, the surface intersects the voxel boundary even though the voxel does not exhibit sign change across any edge. Our algorithm generates an adaptive grid without any complex voxels. Suppose we are given an error bound ε. Note that this bound can be under any distance metric.



(a) (b)

Figure 7: Cracks: Fig. (a) shows a surface passing through a coarse voxel (left voxel) without intersecting any of the edges, while it intersects the edges of a neighboring voxel (right voxel) that is at a finer resolution. This can result in cracks in the reconstructed surface as shown in the right figure.

1. Check if the voxel is intersecting using the voxel-intersection test.
2. if no intersection, **STOP**.
3. if complex voxel or voxel size is greater than the ε, **SUBDIVIDE** else **STOP**.

We apply the Marching Cubes algorithm to each voxel of the resulting grid. The Hausdorff distance between the reconstructed surface and the actual surface is guaranteed to be than ε. Note that the voxel-intersection test provides us with an early termination condition (Step 2). This makes the adaptive grid generation algorithm very efficient.

6. Implementation and Performance

In this section, we describe the implementation of our l_∞ distance computation algorithms and highlight its performance.

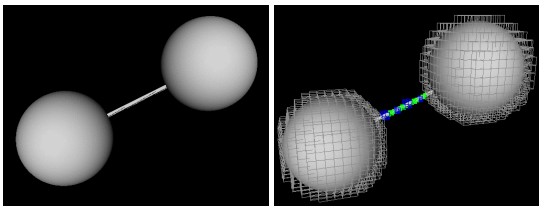
6.1. Implementation

We implemented our algorithms using C++ programming language on a 1.6 GHz Pentium IV PC with a GeForce 3 graphics card and 500 MB main memory.

6.1.1. Polyhedral Models

Our algorithm for non-convex polyhedra requires convex surface decomposition. In order to meet this requirement, we modified a public collision detection library, SWIFT++ [9], to take advantage of its decomposition scheme. We also used a public triangle-triangle intersection routine developed by Möllwer *et al.* [27] for fast intersection computations between target and partitioning triangles.

In our experiment, an average query time for a triangle takes 10 $\mu sec.$ The benchmarking results for polyhedra are also presented in Table 1. Depending on the location of a query point with respect to the polyhedron, the query time takes from 0.6 *msec* to 6.14 *msec*. When the query point is



(a) (b)

Figure 8: This figure highlights our reliable voxelization algorithm applied to an object in the shape of a dumbbell shown in Fig. (a). Fig. (b) shows its voxelization. The colors (white, blue and green in that order) represent increasing levels of subdivision. It took 11 secs to generate a voxelization based on l_∞ distance computation.

located inside the polyhedron, the query takes longer, and this query corresponds to the notion of *penetration depth* [5] for a point.

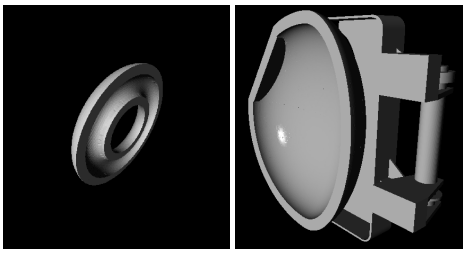
Model	Tri	Convex Pcs	In Query	Out Query
Wrinkled Torus	2000	412	2.46	6.14
Cup	500	190	0.6	3
Spoon	1344	275	1.34	4.89

Table 1: Benchmark Results for Non-Convex Polyhedra. Each column, respectively from left to right, denotes a benchmarking model, triangle counts of the model, a number of decomposed convex pieces in the model, average query time in *msec* for a point outside the model, average query time in *msec* for a point inside the model.

The advantage of using graphics hardware is its SIMD-like capability that enables us to perform queries at a number of points in parallel. It took 8 secs, 2.7 sec and 5.6 secs to compute l_∞ distance on a uniform 128x128x128 grid for the wrinkled torus, cup and spoon benchmarks respectively. Most of this time was spent in framebuffer readback. In many applications, it suffices to have accurate distance values only within a small neighborhood of a point. Given a distance bound B, we can further improve performance by employing simple culling techniques. In case of l_∞ metric, we are interested only in distance values within a cube of length $2 * B$ centered at a point. We cull away a primitive if its axis-aligned bounding box does not intersect the cube.

6.2. Adaptive Grid Generation

We applied our grid generation algorithm to different benchmarks. Fig. 8 shows the voxelization of a dumbbell. It took 11 secs to generate a voxelization using our voxel-intersection test. Fig. 9 shows the reconstruction of CAD benchmarks consisting of 1-5 solids each defined using 3-5 Boolean operations on non-convex and curved primitives including tori and ellipsoids. On an average, it took 15 secs to generate a voxelization per solid. In order to reconstruct a boundary representation, we computed signed directed distance at each of the grid points of the voxelization [20] and performed iso-surface extraction using the dual contouring



(a) (b)

Figure 9: Non-convex and curved primitives: This figure shows the reconstruction of CAD benchmarks consisting of 1-5 solids each defined using 3-5 Boolean operations on non-convex and curved primitives including tori and ellipsoids. On an average, it took 15 secs to generate a voxelization of each solid based on l_∞ distance computation.

algorithm [18]. We computed directed distances in software which took 80-90 secs. Note that the directed distance is used only for reconstruction and is different from l_∞ distance that we compute during voxelization. The reconstruction from the adaptive grid took less than a second. On an average, less than 10 % of the total time was spent inside voxel-intersection test routine. Hence it is practical to use the voxel-intersection test for adaptive grid generation.

When performing iso-surface extraction on an adaptive grid, the reconstruction algorithm often needs to perform crack patching [35]. Our grid generation algorithm generates an adaptive grid that does not require any crack patching.

6.3. Comparison with Prior Voxel-Intersection Tests

There has been prior work on determining whether an implicit surface intersects a voxel. These algorithms are based on Lipschitz condition and interval arithmetic [19]. However, these algorithms are rather slow and conservative in practice. Frisken *et al.* [32] check whether the surface passes through a voxel by comparing the Euclidean distance to the surface with half diagonal length. This is equivalent to testing if the surface passes through a bounding sphere of the voxel. This is a conservative test and can cause too much subdivision. Voxels that lie completely outside but close to the surface may intersect the bounding sphere and be unnecessarily subdivided. In contrast, we use an exact test based on the $l_\infty()$ distance which can be computed efficiently using the techniques described above.

7. Conclusion and Future Work

We have presented algorithms to efficiently perform max-norm distance computations between a point and a wide class of geometric primitives. We have demonstrated its application to perform a reliable voxel-intersection test for ADF generation of complex models. The efficient voxel-intersection test has low additional overhead, guarantees no missed components, and a bounded Hausdorff-error on the approximated samples as well as the reconstructed surface.

In the future, we would like to apply our techniques to

compute the l_∞ distance between objects. Many of the algorithms presented in this paper can be generalized to distance computation between two objects. We would also like to investigate other applications of max-norm distance.

References

1. Loqo optimization toolkit. *http://www.orfe.princeton.edu/∼loqo*, 2000.
2. Oswin Aichholzer and Franz Aurenhammer. Straight skeletons for general polygonal figures in the plane. In *Computing and Combinatorics*, pages 117–126, 1996.
3. D. Baraff and A. Witkin. *Physically-Based Modeling*. ACM SIGGRAPH Course Notes, 2001.
4. Jean-Daniel Boissonnat, Micha Sharir, Boaz Tagansky, and Mariette Yvinec. Voronoi diagrams in higher dimensions under certain polyhedral distance functions. In *Symposium on Computational Geometry*, pages 79–88, 1995.
5. S. A. Cameron and R. K. Culley. Determining the minimum translational distance between two convex polyhedra. In *Proc. of IEEE Inter. Conf. on Robotics and Automation*, pages 591–596, 1986.
6. L. Paul Chew, Klara Kedem, M. Sharir, Boaz Tagansky, and E. Welzl. Voronoi diagrams of lines in 3-space under polyhedral convex distance functions. In *Symposium on Computational Geometry*, pages 197–204, 1995.
7. O. Cuisenaire. *Distance Transformations: Fast Algorithms and Applications to Medical Image Processing*. PhD thesis, Universite Catholique de Louvain, 1999.
8. B. Curless and M. Levoy. A volumetric method for building complex models from range images. In *Proceedings of ACM Siggraph*, pages 303–312, 1996.
9. S. Ehmann and M. C. Lin. Accurate and fast proximity queries between polyhedra using convex surface decomposition. *Computer Graphics Forum (Proc. of Eurographics'2001)*, 20(3), 2001.
10. G. Farin, J. Hoschek, and M.-S. Kim, editors. *Handbook of Computer Aided Geomtric Design*. Elsevier Science, 2002.
11. S. Frisken, R. Perry, A. Rockwood, and R. Jones. Adaptively sampled distance fields: A general representation of shapes for computer graphics. In *Proc. of ACM SIGGRAPH*, pages 249–254, 2000.
12. M. Gavrilova. *Proximity and Applications in General Metrics*. PhD thesis, Department of Computer Science, University of Calgary, Canada, 1998.
13. L. El Ghaoui. Air profile optimization in the fast positive force transient control. *http://www-inst.eecs.berkeley.edu/∼ee290n/projects/pinello.ps*, 1999.
14. S. Gibson. Using distance maps for smooth representation in sampled volumes. In *Proc. of IEEE Volume Visualization Symposium*, pages 23–30, 1998.
15. C. Guestrin, D. Koller, and R. Parr. Max-norm projections for factored mdps. In *Proc. of IJCAI*, pages 673–680, 2001.
16. K. Hoff, T. Culver, J. Keyser, M. Lin, and D. Manocha. Fast computation of generalized voronoi diagrams using graphics hardw are. *Proceedings of ACM SIGGRAPH*, pages 277–286, 1999.
17. H. Hoppe, T. Derose, T. Duchamp, J. McDonald, and W. Stuetzle. Surface reconstruction from unorganized point clouds. In *Proceedings of ACM Siggraph*, pages 71–78, 1992.
18. T. Ju, F. Losasso, S. Schaefer, and J. Warren. Dual contouring of hermite data. *ACM Trans. on Graphics (Proc. SIGGRAPH)*, 21(3), 2002.
19. D. Kalra and A. H. Barr. Guaranteed ray intersections with implicit surfaces. In *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23, pages 297–306, 1989.
20. L. Kobbelt, M. Botsch, U. Schwanecke, and H. P. Seidel. Feature-sensitive surface extraction from volume data. In *Proc. of ACM SIGGRAPH*, pages 57–66, 2001.
21. V. Koltun and M. Sharir. Polyhedral voronoi diagrams of polyhedra in three dimensions. In *ACM Symposium on Computationl Geometry*, 2002.
22. J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Boston, 1991.
23. M. Lin and S. Gottschalk. Collision detection between geometric models: A survey. *Proc. of IMA Conference on Mathematics of Surfaces*, 1998.
24. W. B. Lindquist. 3dma general users manual. Technical Report SUSB-AMS-99-20, Department of Applied Math ad Statistics, SUNY - Stony Brook, 1999.
25. Miguel S. Lobo. *Robust and Convex Optimization with Application in Finance*. PhD thesis, Dept. of Electrical Engineering, Stanford University, 2000.
26. W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 163–169, 1987.
27. T. Möllwer and P. AB. A fast triangle-triangle intersection test. *Journal of Graphics Tools*, 2(2):25–30, 1997.
28. Y. Nesterov and A. Nemirovsky. *Interior-point polynomial methods in convex programming*. SIAM, Philadelphia, PA, 1994. Vol 13 of Studies in Applied Mathematics.
29. F.S. Nooruddin and G. Turk. Simplification and repair of polygonal models using volumetric techniques. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):194–205, 2003.
30. Atsuyuki Okabe, Barry Boots, and Kokichi Sugihara. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. John Wiley & Sons, Chichester, UK, 1992.
31. Evanthia Papadopoulou and D. T. Lee. The l_∞-voronoi diagram of segments and vlsi applications. In *International Journal of Computational Geometry and Applications*, pages 503–528, 2001.
32. R. Perry and S. Frisken. Kizamu: A system for sculpting digital characters. In *Proc. of ACM SIGGRAPH*, pages 47–56, 2001.
33. A.A.G. Requicha. Mathematical definition of tolerance specifications. *ASME Manufacturing Review*, 6(4):269–274, 1993.
34. R. Seidel. Linear programming and convex hulls made easy. In *Proc. 6th Ann. ACM Conf. on Computational Geometry*, pages 211–215, Berkeley, California, 1990.
35. R. Shekhar, E. Fayyad, R. Yagel, and F. Cornhill. Octree-based decimation of marching cubes surfaces. *Proc. of IEEE Visualization*, pages 335–342, 1996.
36. J. Tsitsiklis and B. Van Roy. Feature-based methods for large scale dynamic programming. *Machine Learning*, 22:59–94, 1996.
37. S. Wang and A. Kaufman. Volume sampled voxelization of geometric primitives. In *Proceedings of IEEE Conference on Visualization*, pages 78–84, 1993.
38. S. Wang and A. Kaufman. Volume-sampled 3d modeling. *IEEE Computer Graphics and Applications*, 14(5):26–32, 1994.