Efficient Max-Norm Distance Computation and Reliable Voxelization

Gokul Varadhan, Shankar Krishnan, Young J. Kim, Suhas Diggavi and Dinesh Manocha

1Department of Computer Science, University of North Carolina, Chapel Hill, U.S.A
2AT&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, implicit and volume modeling, surface reconstruction, physically-based modeling, computer-aided design, 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 $l_{\infty}$) distance computation. Under this norm, the distance between two points $x$ and $y$ (in $d$ dimensions) is represented as $D_{\infty}(x, y)$ and is defined as

$$D_{\infty}(x, y) = \max_{i} |x_i - y_i|, \quad i = 1, 2, \ldots, d \tag{1}$$

We can extend this definition for distance between a point $p$ and a set $S \subseteq \mathbb{R}^d$. Computing distances under the max-norm has an important difference from the Euclidean case: $l_{\infty}$ 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, image analysis, dynamics and control systems, tolerance analysis and NC machining and volume graphics. 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 and volume rendering, implicit modeling, shape representation and model repair. 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-
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 $l_\infty$ norm. These include the study of $l_\infty$ Voronoi diagram and its combinatorial and complexity [4, 6, 12, 21, 30, 31], and $l_\infty$ skeleton computations [2]. In particular, Papadopoulou et al. [31] have presented $O(n \log n)$ algorithms to compute the 2D $l_\infty$ Voronoi diagram of polygons and highlighted its application to VLSI layout and manufacturing. However, no practical algorithms or implementations are known for 3D $l_\infty$ 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 $l_\infty$ 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 $S$ 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 $l_\infty$ 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 $S$ to which we need to find the closest distance consists of points satisfying all $f_i(x) \leq 0, i = 1, 2, \ldots, n$, where each $f_i$ is a non-linear analytic function. Without loss of generality, we can assume that the point $p$ 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 $l_\infty$ distance is given by the sets $R_{\infty1} = \{x_1 - x_2 \geq 0 \land x_1 + x_2 \geq 0\}$ and $R_{\infty2} = \{x_1 - x_2 \leq 0 \land x_1 + x_2 \leq 0\}$.

Now let us assume that we are restricted to one such region, say $R_{\infty1}$. By adding the additional constraint for $x$ to belong to $S$, our constraint space is restricted to a portion of the primitive lying inside $R_{\infty1}$. 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 $R_{\infty2}$, 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 $R_{ik} = \bigcap_{j \neq i, j = 1, \ldots, d} (x_i - x_j \geq 0 \land x_i + x_j \geq 0)$ and $R_{ij} = \bigcap_{j \neq i, j = 1, \ldots, d} (x_i - x_j \leq 0 \land x_i + x_j \leq 0)$. We have now reduced our minimum distance computation problem to solving $2d$ non-linear optimization programs. Each program has the form

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

We use the above formulation to develop efficient algorithms for 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 $l_\infty$ norm from a point to a convex primitive. The interior of a convex primitive satisfies

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

Point inside the primitive Consider the convex primitive and the point $p$ in 2D as shown in Fig. 1(a). All points that are equidistant from $p$ lie on the surface of an axis-aligned square centered at $p$. This relation is shown by the square in the Fig. 1(a). Consider growing such a square from the point $p$. The shortest distance from $p$ to the surface of the object is realized by a point on the surface that first touches the growing square (point $q$ 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 \((\pm 1/\sqrt{2}), (\pm 1/\sqrt{2})\).

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

$$D_\infty(p, S) = \frac{1}{\sqrt{d^i}} \min_{\tilde{v}} D_\infty(p, \tilde{v}), \quad i = 1, 2, \ldots, 2^d,$$

where $\tilde{v}_i$ is chosen from the set \([-1/\sqrt{d}, 1/\sqrt{d}]^d\) and $D_\infty$ is the directed distance along vector $\tilde{v}$. 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 $p$ 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

![Figure 1: Computing distance from a point to a convex primitive under $l_\infty$ metric. (a) point inside primitive (b) point outside primitive](image)

3.1.2. Distance Computation for Convex Polytopes and Quadrics

For quadrics, we can write the interior of the primitive using quadric constraints $x^T A x + b^T x + c \leq 0$, where $A$ is a symmetric positive definite matrix, $b$ 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 \(^{23}\). 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

$$\begin{align*}
\text{minimize} & \quad b^T x, \\
\text{subject to} & \quad \| A x + b \|_2 \leq c^T x + d_i, i = 1, 2, \ldots, n, \\
\text{and} & \quad g_j^T x \geq 0, j = 1, 2, \ldots, 2(d - 1).
\end{align*}$$

The constraints listed above also include the special case of convex polytopes (by making $A = 0$), 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 $l_\infty$ 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 $l_\infty$ 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 $l_\infty$ distance to a convex polytope based on a linear programming
technique. The distance computation for a triangle $\Delta^T$ is a simple variation of the same technique. In case of a triangle, we reduce the problem to computing intersections between the target triangle $\Delta^T$ and 12 auxiliary partitioning triangles $\Delta^B$. In fact, these 12 $\Delta^B$’s represent the linear constraints $g_j$ highlighted in Section 3.1.2; these 12 constraints are illustrated in Fig. 2(a). Notice that even though these $g_j$’s form unbounded partitions of 3D space, in practice, we bound the partitions by using an axis-aligned bounding box of $\Delta^T$ such that the boundary of each partition becomes a triangle $\Delta^B$.

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

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 $l_\infty$ distance for a non-convex polyhedron $P$ is to compute the distance for every triangle $\Delta_i \in P$ 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. 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 $o$ to the polyhedron $P$.
- $U_b$: Upper bound to the distance value from $o$ 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 $o$ 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 $D_{l_\infty}(\cdot)$ 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 $\varepsilon$ and obtain an estimate for $D_{l_\infty}(\cdot)$ 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. 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 $l_\infty$ metric. For each site, we define a distance function, which gives, for any point, the distance to that site with respect to $l_\infty$ metric. In contrast to $l_2$, the $l_\infty$ 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 $p$ 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.

**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.

**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 $\mathbf{n} = (n_1, n_2, n_3)$ 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.

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

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 $G(x)$, 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 $\epsilon$. If $\mathbf{n}$ is the unit normal to the triangle containing $q$, then one of the points $q \pm 2\epsilon \mathbf{n}$ 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.

### 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
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 $\varepsilon$. 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 $\varepsilon$. Note that this bound can be under any distance metric.

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 $\varepsilon$, 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 less than $\varepsilon$. 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 $l_\infty$ 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++, to take advantage of its decomposition scheme. We also used a public triangle-triangle intersection routine developed by Möllwer et al., 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

For the wrinkled torus, cup and spoon benchmarks respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denotes a benchmarking column, respectively from left to right, denot

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 of each solid based on \( l_{\infty} \) distance computation.

When performing iso-surface extraction on an adaptive grid, the reconstruction algorithm often needs to perform crack patching. 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. However, these algorithms are rather slow and conservative in practice. Frisken et al. 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 $L_{\infty}$ 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