Fast Penetration Depth Computation Using Rasterization Hardware and Hierarchical Refinement

Young J. Kim Miguel A. Otaduy Ming C. Lin and Dinesh Manocha
Department of Computer Science
University of North Carolina at Chapel Hill
{youngkim,otaduy,lin,dm}@cs.unc.edu

Abstract: *We present a novel and fast algorithm to compute penetration depth (PD) between two polyhedral models. Given two overlapping polyhedra, it computes the minimal translation distance to separate them using a combination of object-space and image-space techniques. The algorithm computes pairwise Minkowski sums of decomposed convex pieces, performs closest-point query using rasterization hardware and refines the estimated PD by object-space walking. It uses bounding volume hierarchies, model simplification, object-space and image-space culling algorithms to further accelerate the computation and refines the estimated PD in a hierarchical manner. We highlight its performance on complex models and demonstrate its application to dynamic simulation and tolerance verification.*

Keywords: collision detection, graphics hardware, image-space computations, dynamics simulation, geometric modeling, robotics

1 Introduction

The need to perform fast proximity queries, including collision detection, tolerance checking, distance calculation and penetration computation, arises in numerous areas. Some applications include virtual environments, physically-based modeling, computer animation, robotics, haptics, and electronic prototyping. While several of these queries, such as collision detection and distance computation, have been extensively studied in the field, there is relatively little work on penetration computation that provides a measure of intersection or penetration between two overlapping models.

Given two inter-penetrating rigid polyhedral models, the penetration measure between them can be defined using different formulations. One of the widely used measures for quantifying the amount of intersection is *penetration depth*, commonly defined as the minimum translational distance required to separate two intersecting rigid models [DHKS93, CC86, Cam97]. Penetration depth (PD) is often used in contact resolution for dynamic simulation [MZ90, Mir00, ST96], force computation in haptic rendering [MPT99, GME$^+$00], tolerance verification for virtual prototyping [Req93], motion planning of autonomous agents [HKL$^+$98], etc. Most collision detection algorithms and libraries do not handle inter-penetrations and current distance computation algorithms do not provide a continuous distance measure when two objects collide. This can induce numerical problems, e.g. instability or invalid results, in dynamic simulation. Furthermore, several commonly used techniques like penalty-based methods often need to perform PD queries for imposing the non-penetration constraint for rigid body simulation. Various heuristics, such as reducing the frequency of PD computation or estimating PD based on the last closest feature pairs, are sometimes used. But, the results can be inconsistent and inaccurate. Fast and reliable PD computation is important for robust and efficient simulation of dynamical systems.

The PD between two overlapping objects can be formulated based on their *Minkowski sum*. Given two polyhedral models, say P and Q, the PD corresponds to the minimum distance from the origin of the Minkowski sum, $P \oplus (-Q)$, to the surface of this sum. However, the computational complexity of computing the Minkowski sum can be $O(n^6)$,

where n is the number of features [DHKS93]. In addition to its high computational complexity, the resulting algorithms are also susceptible to accuracy and robustness problems. Hence, no practical algorithms are yet known for accurately computing the PD between general polyhedral models.

Main Results: We present a novel approach to estimate the PD between general polyhedral models using a combination of object-space and image-space techniques. Given the global nature of the PD problem, we systematically decompose the boundary of each polyhedron into convex pieces, compute the pairwise Minkowski sums of the resulting convex polytopes and use the polygon interpolation based rasterization hardware to perform the closest point query up to image-space resolution. To further speed up this computation and improve the estimate, we use a hierarchical refinement technique that takes advantage of object-space culling, model simplification, image-space acceleration, and local refinement with greedy walking. The overall approach combines image-space accelerated queries with object-space culling and refinement at each level of the hierarchy.

This algorithm has been implemented and tested on different benchmarks. Depending on the combinatorial complexity of polyhedra and their relative configuration, its performance varies from a fraction of second to a few seconds on a 1.6GHz PC with an NVIDIA GeForce 3 graphics card. To illustrate the effectiveness of our algorithm, we demonstrate its applications to contact response computation and an improved time-stepping method for rigid-body dynamic simulation, and tolerance verification for virtual prototyping. To the best of our knowledge, this is the first practical algorithm for computing a reliable PD between general polyhedral models and it works well for different scenarios.

Organization: The rest of the paper is organized in the following manner. We give a brief summary of the related work in Section 2 and present some background material along with an overview of our approach in Section 3. Section 4 describes the underlying algorithm that uses a combination of object space and image space computations. We present a number of acceleration methods in Section 5 to improve the overall performance. Section 6 describes its implementation and performance on different configurations. Section 7 highlights its applications to dynamics simulation and tolerance verifications for virtual prototyping.

2 Previous Work

In this section, we briefly review previous work related to proximity queries, penetration depth computation and the use of graphics rasterization hardware for geometric computations.

2.1 Collision and Distance Queries

The problems of collision detection and distance computations are well studied in computational geometry, robotics, and simulated environments. Most of the prior work on polyhedra can be categorized based on the types of models, such as convex polytopes and general polygonal models.

For convex polytopes, various techniques have been developed based on Minkowski difference [Cam97, GJK88] and feature tracking using Voronoi regions [LC91, Mir98]. Some of these algorithms also utilize the spatial and temporal coherence between successive frames and perform incremental computations [Bar92, Cam97, LC91, Mir98].

For general polygonal models, bounding volume hierarchies (BVHs) have been widely used for collision detection and separation (or Euclidean) distance queries. They localize the problem and use the "divide-and-conquer" paradigm. BVHs often differ based on the underlying bounding volume or traversal schemes. These include the OBB trees [GLM96], sphere trees [Hub95], k-dops [KHM$^+$98], and convex hull-based trees [EL01]. Due to the global nature of PD problem, none of them can be directly used for PD computation between non-convex models.

2.2 Penetration Depth Computation

A few efficient algorithms to compute the penetration depth (PD) between convex polytopes have been proposed. The simplest exact algorithm is based on computing their Minkowski sum [GS87, KR92] followed by computing the closest point to its boundary from the origin. But its worst case complexity is $O(mn)$, where m and n are the number of features in each polytope. Dobkin et al. computed the directional PD using Dobkin and Kirkpatrick polyhedral hierarchy [DHKS93]. For any direction d, it finds the directional PD in $O(\log n \log m)$ time. A randomized algorithm to compute the PD is given in [AGHP$^+$00].

Given the worst-case $O(mn)$ complexity of PD computation between convex polytopes, a number of approximation approaches have been proposed for interactive applications. All of them either compute a subset of the boundary or a simpler approximation of the Minkowski sum and compute an upper or lower bound to the PD [Cam97, Ber01, OG96, KOLM01]. They also take advantages of frame-to-frame coherence and perform incremental computations.

Other approximation approaches for general polygonal models are based on discretized distance fields. These include algorithms based on fast marching level-sets for 3D models [FL01] and others based on graphics rasterization hardware and multi-pass rendering for 2D objects [HZLM01].

2.3 Graphics Hardware for Geometric Applications

Interpolation-based polygon rasterization hardware is increasingly being used for geometric applications. These include visibility and shadow computations, CSG rendering, proximity queries, morphing, object reconstruction etc. A recent survey on different applications is given in [TPK01]. All these algorithms perform computations in a discretized space (i.e. the image-space) and their accuracy is governed by the underlying pixel resolution. The set of proximity query algorithms include cross-sections and interferences [RMS92] and distance computations, including separation and local penetration estimation [HCK$^+$99, HZLM01]. An algorithm to compute a discretized approximation to the convolution of general polyhedral models using the rasterization hardware is presented in [KR92]. Algorithms for direct rendering of CSG models based on graphics rasterization hardware have been presented in [EJR89, GHF86, Wie96]. They render the CSG hierarchies using multiple passes of clipping (i.e. stencil tests) and depth tests.

3 Background and Overview

In this section, we give a brief overview of the PD computation problem and our approach to solve it.

3.1 Penetration Depth

Let P and Q be two intersecting polyhedra. The PD of P and Q, $PD(P, Q)$, is the minimum translational distance that one of the polyhedra must undergo to render them disjoint. Formally, $PD(P, Q)$ is defined as:

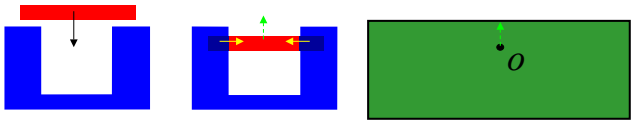
$$min\{\| \boldsymbol{d} \| \mid interior(P + \boldsymbol{d}) \cap Q = \emptyset\} \tag{1}$$

Here, $\boldsymbol{d}$ is a vector in $\mathcal{R}^3$. In practice, we represent the amount of PD, as a negative number in order to distinguish it from the ordinary Euclidean or separation distance between non-overlapping objects.

The computation of the PD between two polyhedral models is a global problem and it is rather difficult to localize it using some 'divide-and-conquer' approach. A local solution computed using intersecting features or boundaries may not be correct, as shown in Fig. 1.

3.2 Minkowski Sums

The Minkowski sum, $P \oplus Q$, is defined as a set of pairwise sums of vectors from P and Q. In other words, $P \oplus Q = \{\boldsymbol{p} + \boldsymbol{q} | \ \boldsymbol{p} \in P, \boldsymbol{q} \in Q\}$. Furthermore, $P \oplus -Q$ is defined by negating Q; i.e. $P \oplus -Q = \{\boldsymbol{p} - \boldsymbol{q} | \ \boldsymbol{p} \in P, \boldsymbol{q} \in Q\}$.



(a) (b) (c)

Figure 1: Local vs Global PD Computation. *(a) shows the situation before two polygons in 2D come into contact. (b) shows $O(nm)$ intersections after the polygons are intersected. However, a localized PD computation (denoted by solid yellow arrows) based on $O(nm)$ intersections may not provide a global PD which is denoted as a dotted green arrow in this figure. (c) shows the Minkowski sum of the two polygons in (b). The minimum distance from the origin to the surface of the Minkowski sum corresponds to the global PD.*

A general framework to compute the PD is based on Minkowski sums. It is well known that one can reduce the problem of computing the PD between P and Q to a minimum distance query on the surface of their Minkowski sum, $P \oplus -Q$ [Cam97]. More specifically, if two polyhedra P and Q intersect, then the difference vector, $\boldsymbol{O}_Q - \boldsymbol{O}_P$, between the origins[1] of P and Q is inside $P \oplus -Q$. Let us denote $\boldsymbol{O}_Q - \boldsymbol{O}_P$ by $\boldsymbol{O}_{Q-P}$. The $PD(P, Q)$ is defined as a minimum distance from $\boldsymbol{O}_{Q-P}$ to the surface of $\partial(P \ominus Q)$. For example, Fig. 1-(c) shows the Minkowski sum of the two polygons in Fig. 1-(b), and the minimum distance from the origin, $\boldsymbol{O}$, to the surface of the Minkowski sum is the global PD.

It is relatively easier to compute Minkowski sums of convex polytopes as compared to general polyhedral models. However, for non-convex polyhedra in 3D, the Minkowski sum can have $O(n^6)$ worst-case complexity [DHKS93]. There are two known approaches for computing the Minkowski sum of general polyhedral models. Both have the same underlying complexity.

The first approach is based on *convolution computation* defined on polyhedral tracings [BGRR96]. It is well known that the convolution is a superset of the surface of the Minkowski sum, and the convolution can be computed in $O(n^2)$ time in the worst case. However, in order to compute the actual boundary of the Minkowski sum, a 3D arrangement of the convolution needs to be computed and it can take $O(n^6)$ total time.

The second approach for computing Minkowski sums for general polyhedra is based on *decomposition*. It uses the following property of Minkowski computation. If $P = P_1 \cup P_2$, then $P \oplus Q = (P_1 \oplus Q) \cup (P_2 \oplus Q)$. The resulting algorithm combines this property with convex decomposition for general polyhedral models:

1. Compute a convex decomposition for each polyhedron
2. Compute the pairwise convex Minkowski sums between all possible pairs of convex pieces in each polyhedron
3. Compute the union of pairwise Minkowski sums.

After the second step, there can be $O(n^2)$ pairwise Minkowski sums and their union can have $O(n^6)$ complexity [AST97].

These approaches provide an algorithmic framework to compute the Minkowski sum. However, their practical utility is unclear. Besides the combinatorial complexity, it is a major challenge to have a robust implementation of algorithms for convolution, arrangements or union computations in 3D.

3.3 Our Approach

Our algorithm to compute the PD is based on the decomposition approach described in Section 3.2. In order to overcome its combinatorial and computational complexity, we use a *surface-based* convex decomposition of the boundary and utilize the graphics raterization hardware to estimate the PD. We do not explicitly compute the boundary of the union or any approximation to it. Rather, we perform the *closest point query* using a multipass algorithm

[1]The origin of a polyhedron refers to the origin of the local coordinate system of the polyhedron with respect to the global coordinate system. Also, throughout the rest of the paper, the origin of the Minkowski sum will refer to the difference vector of the origins of two polyhedra.

that computes the closet point from the origin to the boundary of the union of pairwise Minkowski sums. The resulting maximum depth fragment at each pixel computes an approximation to the PD, up to the image-space resolution used for this computation. Given this PD estimate, we further refine it using an object-space incremental algorithm that performs a local walk on the Minkowski sum. Each step of our approach is relatively simple to implement. However, its worst case complexity can be as high as $O(n^4)$ because of the number of pairwise Minkowski sums and the computational complexity of the closest point query.

We improve the performance of the algorithm using a number of acceleration techniques. These include hierarchical representation based on convex bounding volumes, use of model simplification algorithms, object-space culling approaches, and image-space acceleration techniques applied to the multipass algorithm. These are explained in detail in Section 5.

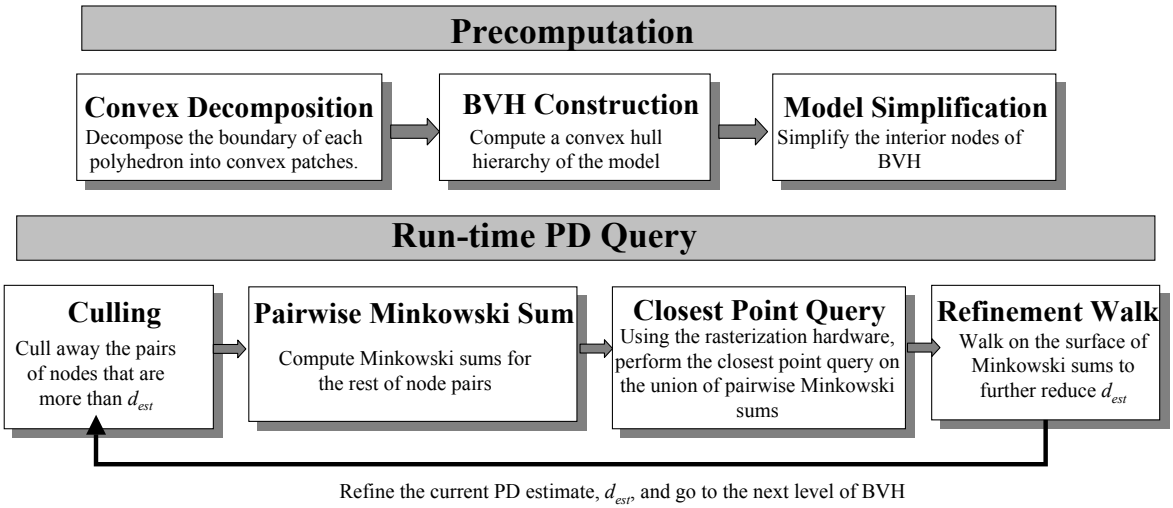


Figure 2: PD Computation Pipeline

The resulting algorithm includes a pre-computation phase as well as a runtime query. The pre-computation phase consists of the following steps:

1. Decompose the boundary of each polyhedron into convex patches using a greedy approach (Sec. 4.1).
2. Compute a bounding volume hierarchical representation of the model. Each node in the tree corresponds to a convex polytope and each leaf is a convex hull of a decomposed convex patch (Sec. 5.2).
3. Use model simplification algorithms to compute a lower polygon count approximation of the interior nodes (Sec. 5.4).

Given two polyhedra and their bounding volume representations, the runtime phase of the algorithm proceeds in the following manner:

1. Compute an upper estimate to the amount of PD. Let that estimate be d_{est}. Initially we compute an estimate based on the root nodes of each tree (Sec. 5.1).
2. At each level of the two hierarchies, repeat the following steps:
   (a) Consider all pairwise combinations of nodes at the current level. Cull away all the pairs that are non-overlapping and are more than d_{est} apart (Sec. 5.3).
   (b) Compute pairwise Minkowski sums of the rest of the node pairs that have not been culled away (Sec. 4.2).
   (c) Perform the closest point query using the rasterization hardware to compute a PD estimate (Sec. 4.3).

(d) Extract the penetration features in the object-space. Perform a local walk and refine the PD estimate using incremental algorithms (Sec. 4.4).

In step 2(c), we use a coarse pixel resolution at the top levels of the tree and refine the resolution as we traverse down the hierarchies. The entire pipeline of our PD algorithm is illustrated in Fig. 2.

3.4 Notation

We use bold-faced letters to distinguish a vector from a scalar value (e.g. the origin, $\boldsymbol{O}$). In Table 1, we enumerate the notations that we use throughout the paper.

Notation	Meaning
∂P	The boundary of P
C_i^P	A decomposed convex piece of P
$C_i^{P,l}$	A decomposed convex piece of P at level l
M_{ij}	Minkowski sum between C_i and C_j
d_{est}^k	kth refinement of the PD estimation

Table 1: Notation Table

4 Penetration Depth Computation

In this section, we present our algorithm for global PD computation. It involves decomposing the boundary of the model into convex patches, computing their pairwise Minkowski sums, and then performing closest point query using rasterization hardware and object-space refinement.

Given two intersecting polyhedra, the PD query reports a PD scalar value and direction, along with the associated PD features[2]. In this case, the origin is contained inside the Minkowski sum of the two polyhedra. Fig. 3 illustrates what the PD query attempts to report.

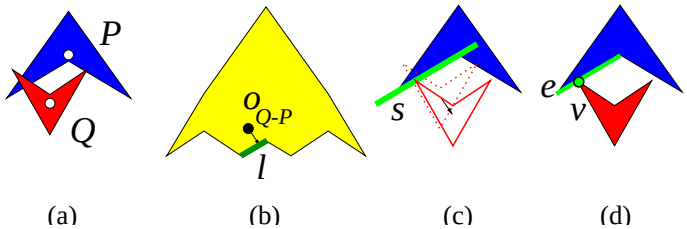
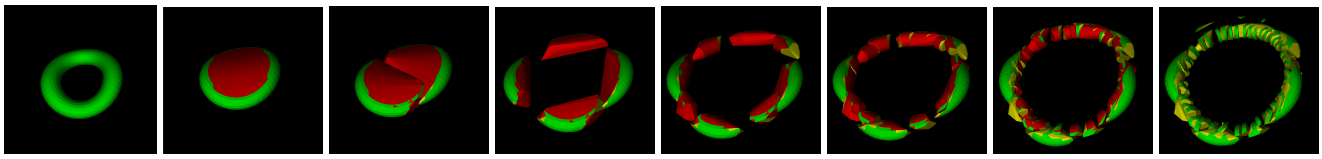


Figure 3: Simple Penetration Depth Computation in 2D. *(a) Two polygons P and Q intersect. (b) The Minkowski sum $P \ominus Q$ is computed, and the minimum distance from the origin $\boldsymbol{O}_{P \ominus Q}$ to $\partial(P \ominus Q)$ is determined. (c) When P is translated by the amount of PD, there exists a line s locally supporting both polygons. In this case, the edge/vertex feature pair, (e, v), as shown in (d) makes up the PD features. The PD features are also identified in (b) as a corresponding dark green line l.*

[2]These are a pair of features that realize the reported PD. The PD value is the same as the inter-distance between planes which locally support the PD features.



(a) (b) (c) (d) (e) (f) (g) (h)

Figure 4: Convex Surface Decomposition and Bounding Volume Hierarchy. *(a) shows an original model for a torus, and (h) shows its convex surface decomposition. From (b) to (h), the figure shows a BV hierarchy of the torus from root level to leaf level. In the figure, the green color indicates an original face in the model, the red color highlights a virtual face created by convex hull computation, and the yellow color indicates a virtual face created while converting a convex patch to a convex piece.*

4.1 Object Decomposition

We decompose the boundary of each polyhedron P into a collection of convex patches c_i. These c_i's are mutually disjoint except for their shared edges, and the union of all the c_i's covers the entire boundary of P, ∂P.

We compute the convex patches, c_i's, by dualizing the polyhedral surface and performing a graph search on it in a greedy manner. First, we construct a dual graph G of the polyhedral surface ∂P by reversing the roles of faces (F) and vertices (V) in ∂P, while using the same edges (E) in G from ∂P [CDST97]. We traverse the dual graph G by adding faces into a current convex patch c_i as long as it maintains its convexity. We repeat this process until we cover the entire boundary of ∂P. For example, Fig. 4-(h) is a convex surface decomposition for a wrinkled torus model, Fig. 4-(a).

Furthermore, we compute a convex hull of each surface patch, c_i, and denote the resulting polytope by C_i. The union of these C_i's is completely contained in the original polyhedron P. Notice that our decomposition strategy is merely a partition of ∂P, not of P. This surface decomposition is sufficient for PD computation, because we are only concerned with the surface of Minkowski sums between polyhedra.

4.2 Pairwise Minkowski Sum Computation

Our PD computation algorithm is based on the decomposition approach described in Section 3.2. The first step involves computing the pairwise Minkowski sums between all possible pairs of convex polytopes, C_i^P and C_j^Q, belonging to P and Q, respectively. Let us denote the resulting Minkowski sum as M_{ij}. Various algorithms are known for computing Minkowski sums of convex polytopes. The most efficient algorithm is based on *topological sweep* and its complexity is $O(n \log n + k)$, where n is the number of features in C_i^P and C_j^Q and k is the number of features in the resulting Minkowski sum, M_{ij} [GS87]. However, the constant factor can be high and it is non-trivial to implement it robustly, especially when the *general position* assumption is not guaranteed. Instead, we use the simpler algorithms described below.

Convex Hull Approach: We use this approach for smaller models, i.e. polyhedra with less than 50 vertices. It is based on the following property:

$$P \oplus Q = \mathbf{CH}(\{\mathbf{v}_i + \mathbf{v}_j | \mathbf{v}_i \in V_P, \mathbf{v}_j \in V_Q\}) \tag{2}$$

Here, **CH** denotes the convex hull operator, and V_P, V_Q represent the sets of vertices, respectively in polyhedra P and Q. Based on this fact, we compute the Minkowski sum as follows:

1. Compute the vector sum between all possible pairs of vertices from each polytope.
2. Compute their convex hull.

Incremental Surface Expansion: This algorithm was proposed as a part of a polyhedral morphing system based on Minkowski sums [KR92]. We use it for larger convex polytopes, i.e. more than 50 vertices. The algorithm starts with any candidate face on the resulting Minkowski sum, and incrementally expands the surface by finding the next candidate face. The incremental expansion is relatively straightforward, i.e. uses every possible face/vertex (FV), vertex/face (VF), or edge/edge (EE) combination that the current candidate face can be extended to. The worst case asymptotic time complexity of this algorithm is $O(n^2)$, but it works well in practice, with just a few degeneracies such as co-planar faces that need special handling.

4.3 Closest Point Query Using Graphics Hardware

Given all the pairwise Minkowski sums, M_{ij}, let

$$M = \bigcup_{ij} M_{ij}. \tag{3}$$

Our goal is to compute the closest point on the boundary of M, i.e. ∂M, from the origin. We use z-buffer polygon rasterization hardware to perform this query up to image-space resolution. The main idea is to visualize ∂M from the origin without computing a surface representation of ∂M explicitly. After that we compute the closest point, distance and direction.

4.3.1 Visualizing the Boundary of the Union

In order to visualize ∂M from a point $\mathbf{o}$ that is outside M, we simply display all the M_{ij}'s using the standard z-buffer visibility algorithm. The nearest or minimum depth objects at each pixel will correctly construct the ∂M from the outside with only a single pass over each M_{ij}. The resulting algorithm computes the closest point on ∂M from $\mathbf{o}$ up to image-space resolution. However, the same approach does not work if $\mathbf{o}$ is inside M. We present a new, incremental algorithm that can require m^2 passes, where m is the number of convex polytopes, M_{ij}.

Given the point $\mathbf{o}$ inside M, the normal Z-buffer minimum or maximum depth test may not suffice, since the visible internal boundaries may not even lie on ∂M. The algorithm has to remove the boundaries corresponding to the intersections between M_{ij}'s that do not belong to ∂M. If all the M_{ij}'s contain the origin, then the Z-buffer maximum depth test will construct ∂M with a single pass over each M_{ij}, and the desired result will be stored at the minimum depth pixel in the rendered image. However, we only know that at least one of the M_{ij}'s contains the origin. So, we use an incremental algorithm that constructs ∂M out from the origin.

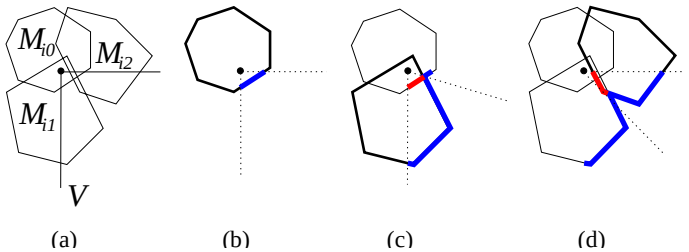


Figure 5: Visualizing the Boundary of the Union From Inside. *In (a), V is the current view-frustum. In (b), M_{i0} is rendered, and a new ∂M is constructed (blue line). In (c), when M_{i1} is rendered, it opens up a new window (dotted line), and the update region (red line) on the current ∂M is established. Thus a new ∂M (blue line) is constructed. In (d), we perform the same procedure for M_{i2}.*

Our algorithm for visualizing ∂M from a point inside is essentially a ray-shooting procedure from the origin to ∂M by incrementally expanding the front of ∂M. For example, in Fig. 5, we expand the current ∂M (blue line) by repeatedly rendering M_{i0}, M_{i1}, M_{i2}. Each time M_{i0}, M_{i1}, M_{i2} are rendered, as shown in Fig. 5(b)-(d), it opens up a new window (shown as dotted line) of the update region (red line) on the current ∂M.

The algorithm maintains the current boundary of M, ∂M^k, where k is the current iteration, and incrementally expands it with M_{ij} that intersects ∂M^k. We attempt to add M_{ij} by rendering the front faces of M_{ij}. The front faces that "pierce" the current ∂M^k open up a window through which the origin can see ∂M. After that we render the backfaces of M_{ij} into the opened window using the maximum depth test. However, we should not render the backfaces of M_{ij}, that are created by non-original (virtual) faces of P and Q. In other words, we should allow the ray to hit only ∂M.

In summary, the basic algorithm simply performs the following procedure:

1. Initialize ∂M^0 to infinity.
2. Repeat steps 3-5 m times
3. Repeat steps 4-5 for each M_{ij}
4. Render front faces of M_{ij}, and using the standard stencil operation, open a window where the depth value of the front faces is less than that of the current ∂M^k.
5. Classify the backfaces of M_{ij} into original and non-original. Render only the original back faces of M_{ij} where the depth value of the back faces is greater than that of the window. This updates the ∂M^k in the window.

After mth iteration in step 2 highlighted above, the algorithm correctly finds the portion of ∂M that is visible from the origin in the following sense. After the kth iteration in step 2, ∂M^k includes the subset of ∂M that the ray can reach with less than or equal to $k-1$ ***hops*** from the origin. Here, the ***hop*** on some point $\boldsymbol{p}$ on ∂M means how many M_{ij}'s the ray should pass through to reach $\boldsymbol{p}$. For example, ∂M^1 includes the possible contribution to the final ∂M of all M_{ij}'s that contain the origin and have zero hops. Therefore, by induction on k, we correctly find the portion of ∂M that is visible from the origin after mth iteration.

4.3.2 Computing the Closest Point

For a given view, we can compute the closest point on the boundary by simply finding the pixel with the minimum distance value. The algorithm reads back the Z-buffer to obtain the depth values for each pixel. However, these depth values have undergone the perspective depth transformation and do not contain the non-linearity that is present in the distance values. The depth transformation is applied only at the vertices and has the following geometric properties: it preserves lines and planes between the transformed vertices and preserves depth relationships with respect to an orthographic view. This is not sufficient for finding the closest point. A pixel with the minimum depth value is not necessarily the closest point in terms of distance from the origin.

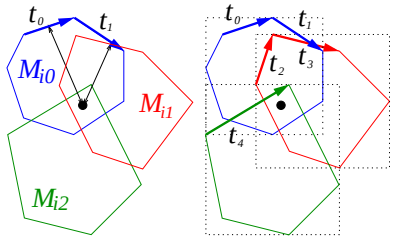
The algorithm transforms the pixel depth values into distance values based on their (x, y) coordinate positions on the viewing plane. Each pixel depth value is divided by $\cos\theta$, where θ is the angle between the vector to the (x, y) position on the viewing plane and the center viewing direction. This depth transformation is CPU-bound, and this operation typically takes a few miliseconds.

The minimum distance and direction to the closest point are derived from the pixel position containing the minimum transformed depth value. In order to examine views in all directions, we construct six views on the faces of a cube around the origin and repeat the operation.

4.4 Object Space Refinement

The accuracy of the closest point query and PD estimate is limited by the image resolution of rasterization hardware. We further refine it to improve the PD estimate by performing local ***walks*** on the boundary of the Minkowski sum of P and Q in the object-space.

We explain our walking algorithm with a simple 2D example. At any time, it maintains a notion of *current-Minkowski-sum*. Fig. 6-(a) shows that M_{i0} is the current-Minkowski-sum that contains the current PD features realized by the line t_0 (a triangle in 3D). By performing local incremental computations, the algorithm determines



(a) (b)

Figure 6: Local Refinement by Walking. *We refine the PD by iteratively minimizing the distance between the origin and a line (a triangle in 3D) on the Minkowski sum M_{i0}. Thus, in (a), there is a transition from t_0 to t_1, since the distance from the origin to t_1 is smaller than that to t_0. In (b), the feature t_3 can reduce the PD even further, the transition of the Minkowski sum from M_{i0} to M_{i1} is followed.*

that it can reduce the PD by changing the penetration feature from t_0 to t_1. After that the walking on M_{i0} stops, as subsequent features of M_{i0} are inside M_{i1} and not on the boundary of M (when t_1 intersects with t_3), shown in Fig. 6-(b). In this particular case, if the algorithm walks to the feature, t_3, as opposed to t_1, it can further reduce the PD estimate. Therefore, we need to change the current-Minkowski-sum from M_{i0} to M_{i1} and continue walking. In order to make this transition, we need to keep track of M_{i1} and M_{i2} during the walk on the features of current-Minkowski-sum M_{i0}, because their axis-aligned bounding boxes (AABBs) intersect with that of M_{i0}'s. As we change the current PD features on M_{i0} from t_0 to t_1, the closest line on M_{i1} to the current PD features also changes from t_2 to t_3, but the closest line on M_{i2} to the current PD features remains the same.

Initially the object-space refinement algorithm starts with identifying the features of P and Q that contribute to the current PD estimate. For a 2D example, in Fig. 3-(b), l on $\partial(P \ominus Q)$ contains a point that is closest to the origin, and since l was generated by $e \ominus v$, the PD features are e, v as in Fig. 3-(d). In 3D, each triangle in M_{ij} is generated by only three possible sets of feature combinations from P and Q. These include VF, FV and EE combinations [GS86], and we use that relationship to compute the actual PD features from each polyhedron that correspond to the current PD estimate.

Once the PD features and the Minkowski sum (M_{ij}), which contains them have been identified, the algorithm refines the current PD estimate by locally walking on the surface of M_{ij}, the current-Minkowski-sum. This walk proceeds by iteratively minimizing the distance from the origin to the surface of M_{ij}. We repeat this process until the algorithm reaches a local or global minimum.

As shown in Fig. 6 the algorithm needs to avoid features that are inside the volume of other Minkowski sums. Although it walks towards the interior of the volume, it sets the current-Minkowski-sum accordingly. Therefore, each time the algorithm is walking, it keeps track of which M_{ij}'s might intersect with the current PD features. We accomplish this by keeping track of a subset of Minkowski sums that can potentially intersect with the current PD features and the current-Minkowski-sum.

Let us denote the current-Minkowski-sum as M_{ij}, and also denote the subset of Minkowski sums that potentially intersect with M_{ij} as M_{ij_0}, M_{ij_1}, ..., M_{ij_l}. Here, we conservatively determine M_{ij_l}'s by intersection checks based on an AABB of the Minkowski sum. Moreover, for each M_{ij_l}, we also keep track of a closest triangle t_{k_l} to the current PD feature t_k in M_{ij}. The overall refinement algorithm proceeds as:

1. Let the triangle t_k in M_{ij} corresponds to the PD features computed based on the closest point query. Find the set of Minkowski sums $M_{ij_0}, M_{ij_1}, ..., M_{ij_l}$ that intersect M_{ij} based on checking their AABBs for overlap. Also compute $t_{k_0}, t_{k_l}, ..., t_{k_l}$, which is a set of triangles respectively on M_{ij_l}'s that is closest to t_k on M_{ij}.

2. Identify the triangles incident to t_k on M_{ij}.

3. Find a neighboring triangle, say t_{k+1}, that results in maximum decrease in the PD estimate and does not intersect with t_{k_l}'s. Change the current PD features from t_k to t_{k+1}. Also update t_{k_l} on each M_{ij_l} to the closest feature to t_{k+1}.

4. If step 4 fails, check whether there exists t_{k_l} in M_{ij_l} such that it intersects with the triangles incident to t_k or t_k itself but reduces the PD. If it exists, repeat the walk from step 1 by setting t_{k_l} as t_k and M_{ij_l} as M_{ij}.

5. Repeat the steps 2-4 until there is no more improvement in the PD.

Eventually the algorithm computes a local minimum on the boundary of the Minkowski sum, M.

5 Acceleration Techniques

The global PD computation algorithm described in Section 4 computes an upper bound on the amount of PD between two polyhedral models. However, its running time can vary based on the underlying models as well as their relative configuration. In the worst case, the convex decomposition algorithm can result in $O(n)$ patches and this can lead to $O(n^2)$ pairwise Minkowski sums, M_{ij}. Furthermore, the cost of the closest point query using rasterization hardware can be as high as $O(m^2)$, where m is the number of convex polytopes. This results in $O(n^4)$ worst case complexity for the PD estimation algorithm. In this section, we present a number of acceleration techniques to improve its performance. These include hierarchical culling, model simplification and image-space acceleration techniques.

5.1 Object Space Culling

A significant fraction of the time of the PD estimation algorithm is spent in pairwise Minkowski sum computation. The algorithm presented in Section 4.2 considers all pairs of convex polytopes, C_i^P and C_j^Q, and computes their Minkowski sum, M_{ij}. If we are given an upper bound on the PD, d_{est}, we can eliminate some pairs of convex polytopes without computing their Minkowski sum. This is based on the following lemma:

LEMMA 5.1 *Let d_{ij} be the separation or Euclidean distance between C_i^P and C_j^Q. If $d_{ij} > \| d_{est} \|$, then the closest point from the origin to ∂M lies on $\partial(M - M_{ij})$.*

Proof: Given an upper bound on the PD, d_{est}, it follows that the distance from the the closest point on ∂M to the origin is less than $\| d_{est} \|$. If we discard any feature of ∂M whose distance from the origin is more than $\| d_{est} \|$, it will not affect the outcome of closest point query to M. Since the separation distance between C_i^P and C_j^Q is d_{ij}, the distance from the origin to the closest point on M_{ij} is also d_{ij}. As a result, all points on ∂M_{ij} are farther than d_{est} from the origin and the closest point from the origin to ∂M lies on $\partial(M - M_{ij})$. Q.E.D.

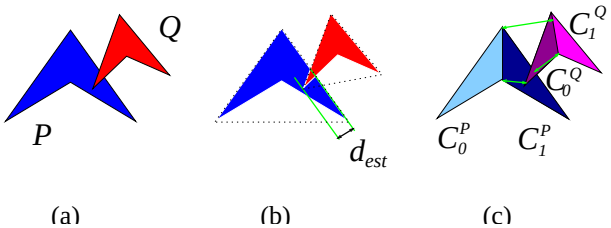


Figure 7: Object Space Culling. *(a) There are two intersecting polygons P (decomposed into C_0^P, C_1^P) and Q (decomposed into C_0^Q and C_1^Q). (b) Based on the convex hull of P and Q, we first estimate the PD as d_{est}. (c) Using d_{est}, we can cull away pairs (C_0^P, C_0^Q), (C_0^P, C_1^Q), (C_1^P, C_1^Q), whose separation distances are more than d_{est}.*

For example, in Fig. 7, there are two intersecting polygons P and Q. We estimate d_{est} based on the convex hull of P and Q (Fig. 7-(b)). Then, we can cull away the pairs whose separation distance is more than d_{est} (Fig. 7-(c)).

Based on the Lemma 5.1, we can cull away all pairs of convex polytopes, C_i^P and C_j^Q, whose separation distances are more than d_{est}. Computing separation distance between convex polytopes is relatively cheap as compared to Minkowski sum computation and a number of efficient algorithms are known [LC91, Cam97]. The efficiency of this culling approach depends on the quality of the estimate, d_{est}. Furthermore, checking all possible pairs for separation distance can take $O(n^2)$ time. We improve their performance using a bounding volume hierarchy to perform hierarchical culling.

5.2 Bounding Volume Hierarchy

We compute a bounding volume (BV) hierarchy for each polyhedron using a convex polytope as the underlying BV. Each convex polytope obtained using the decomposition algorithm explained in Section 4.1 becomes a leaf node in the hierarchy. We recursively compute the internal nodes in a bottom-up manner, by merging the children nodes and computing the convex hull of the union of their vertices. Let us define the nodes of polyhedron P at level l as $C_i^{P,l}$. The resulting hierarchy is a hierarchy of convex hulls. For example, Fig. 4-(b) $\sim$ (h) shows a BV hierarchy for the torus model, Fig. 4-(a).

This hierarchy is used in our runtime algorithm to speed up the intersection and separation distance queries for the culling algorithm. Furthermore, each level of the hierarchy provides an approximation of the model, which is used by the PD estimation algorithm.

5.3 Hierarchical Culling

We use the BV hierarchy to speed up the performance of the object-space culling algorithm. The goal is to start with an initial estimate to the PD and refine it at every level of the tree. We denote the estimate computed using level k of each BV tree as d_{est}^k.

We initially start with the root nodes of each hierarchy, $C_0^{P,0}$ and $C_0^{Q,0}$, which correspond to the convex hulls of P and Q, respectively. We compute the PD between those convex polytopes [Cam97, Ber01, KOLM01] and use that as the estimated PD at level 0. The algorithm proceeds in a hierarchical manner through the levels in each tree:

1. Consider all the pairwise nodes at level k in each tree, $C_i^{P,k}$ and $C_j^{Q,k}$. For each (i, j) pair, compute the separation distance between them. If the nodes overlap, the separation distance is zero.

2. Discard all the node pairs whose separation distances are more than d_{est}^k. Compute the Minkowski sum for the rest of the pairs.

3. Perform the closest point query on the Minkowski sum pairs and compute the new PD estimate, d_{est}^{k+1} using rasterization hardware.

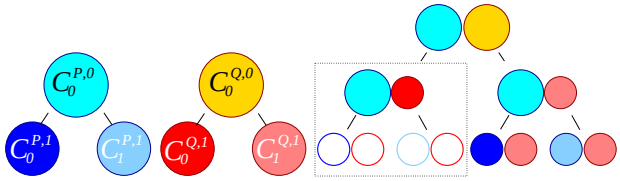
4. Refine the estimate, d_{est}^{k+1} using the object space walking algorithm presented in Section 4.4.

During each iteration, we go down a level in each tree. If we reach the maximum level in one of the trees, we do not traverse down in that tree any further. The algorithm computes an upper bound on the PD in an iterative manner and refines the bound with every traversal as: $\|d_{est}^0\| \geq \|d_{est}^1\| \geq \ldots \geq \|d_{est}^h\|$, where h is the maximum height. Finally, the algorithm returns d_{est}^l as the estimated PD between P and Q.

Fig. 8-(a) shows BV hierarchies for two different objects P and Q, and Fig. 8-(b) shows a snapshot of how the BV hierarchy traversal is performed. Without the culling scheme, one should consider all four pairs between $C_0^{P,1}$, $C_1^{P,1}$ and $C_0^{Q,1}$, $C_1^{Q,1}$. However, during the traversal, $C_0^{P,0}$ and $C_0^{Q,1}$ were found out to be non-overlapping and they are more than d_{est} apart. In this case, no more traversal is needed between the children nodes of $C_0^{P,0}$ and $C_0^{Q,1}$.

5.4 Model Simplification

Some internal nodes of the hierarchy may have a high number of vertices and that affects the complexity of pairwise Minkowski sum computation. We pre-compute a single convex simplification for each internal node in the BV tree.



(a) Two BV hierarchies (b) BV hierarchy Traversal

Figure 8: Hierarchical Culling. *(a) shows two BV hierarchies for two different objects. (b) shows a snapshot of the traversal on the BV hierarchies. During the traversal, it turns out that node $C_0^{P,0}$ and node $C_0^{Q,1}$ are non-overlapping and their inter-distance is greater than a upper bound on the current PD estimation. Thus, no more traversal is performed between the children nodes of $C_0^{P,0}$ and $C_0^{Q,1}$.*

The simplifications at each level of the BV tree provide a low polygon count approximation to the original models. We compute a simplification for each internal node in the following manner:

1. Simplify the node using any simplification error metric. We choose the quadric error metric proposed in [GH97] because of its simplicity and performance.
2. Compute the convex hull of each simplified node.
3. Scale the resulting convex polytope to enclose the internal node or the underlying geometry as tightly as possible.

5.4.1 Improved Computations using Simplified Nodes

We use the simplified BVs to improve the performance of the computations in step 2 (pairwise Minkowski sum computation) and step 3 (closest point query) of the hierarchical culling algorithm presented in Section 5.3. The simplified BVs can increase the estimated PD value, d_{est}, as compared to the original nodes computed by the BV hierarchy computation algorithm. As a result, the number of pairwise Minkowski sums that can be culled at intermediate levels of the hierarchy based on d_{est} may be reduced. However, the running time of the algorithm is significantly reduced. Also, it does not change the accuracy of the final result, as the algorithm does not simplify the leaf nodes in the BV tree.

5.5 Image Space Culling for Closest Point Query

The algorithm also spends a considerable fraction of its time in performing the closest point query using the rasterization hardware (as described in Section 4.3). Here we present a number of techniques to improve its performance.

First of all, we compute a subset of the pairs, M_{ij}'s, that contain the origin and render them only once in the algorithm described in Section 4.3.1. All the pairwise Minkowski sums in this subset have a zero *hop*. We identify this subset, say l out of total of m pairs of M_{ij}'s, by checking whether the corresponding convex polytopes, C_i^P and C_j^Q, overlap [LC91, Cam97, EL01]. Once we have computed these l M_{ij}'s, we first render them using the maximum depth test and then the remaining $(m-l)$ pairwise Minkowski sums, M_{ij}'s, $(m-l)$ times using the incremental algorithm.

Secondly, when we repeat the closest point query six times, once for each face of the cube, we apply a culling technique similar to the one discussed in Section 5.1. At each view, the algorithm maintains the current minimum depth value, d_{est}, and then as it proceeds to the next view, it culls away the M_{ij}'s whose distance from the origin is more than d_{est}, as shown in Lemma 5.1. These distances are also computed in object space. Finally, for each

view, when we render the M_{ij}'s, we perform view-frustum culling by checking whether the axis aligned bounding box of each M_{ij} lies in the current view. This object-space view frustum culling significantly reduces the number of primitives rendered during each iteration of the algorithm.

6 Implementation and Results

In this section, we describe the implementation of our PD computation algorithm and demonstrate its performance on different benchmarks and applications.

6.1 Implementation Issues

Most parts of our algorithm are fairly straightforward to implement. However, it requires quick and robust implementations of the separation distance query between convex polytopes, convex hull computation in 3D, and simplification of polyhedral models. In fact, most of the degenerate cases in our PD computation arise from these three sub-components.

We use the SWIFT++ implementation of the *Voronoi marching* technique [EL01] to efficiently perform the separation distance query. It performs distance queries between non-convex polyhedra by using a hierarchy of convex hulls. We use the public domain QHULL package [BDH93] for convex hull computation in 3D. QHULL is particularly efficient for dealing with a relatively small number of points, which is the case in our algorithm. We use the QSlim implementation [GH97] of the quadric error metric simplification algorithm to ensure that the intermediate nodes of the bounding volume trees do not have more than 50 vertices. As a result, we compute the pairwise Minkowski sums M_{ij}'s, by using the convex hull based algorithm described in Section 4.2. Our initial test results show that it outperforms the incremental surface expansion algorithm presented in [KR92] in our case.

The implementation of the object space walking algorithm uses three main subroutines. These include keeping track of neighboring Minkowski sum pairs that overlap with the current-Minkowski-sum (using AABBs), finding the features in these Minkowski sums that are closest to the current feature and performing local walk. The last step checks all the features that are incident to the current feature and computes the distance to these features from the origin. In our current benchmark we have found that the algorithm only needs to perform a few local walks before it converges to a local minimum on the boundary of ∂M.

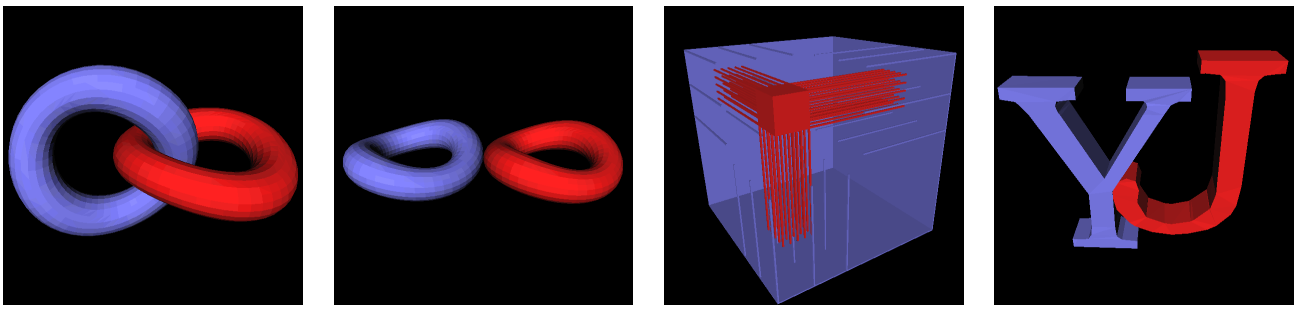


Figure 9: PD Benchmark Models. *From left to right: interlocked tori, touching tori, interlocked grates, and letters.*

We implement the closest point query operation using OpenGL graphics library. The main code to draw ∂M without any acceleration techniques is as simple as Example 1. Also, we typically set the screen space resolution to 128×128 at the intermediate step of the hierarchical refinement, then at the finest level of the refinement, we set the resolution to 256×256. For our benchmarking models, these different resolution schemes provide us with results of a reasonable accuracy, and they also balance the computation time between the object space and the image space.

```
void DrawUnionOfConvex(PairwiseMinkowski *M_ij,
                       int Num_Of_M_ij)
{
  glClearDepth(0);
  glClearStencil(0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT
          | GL_STENCIL_BUFFER_BIT);

  glEnable(GL_DEPTH_TEST);
  glEnable(GL_STENCIL_TEST);
  for (int i=0; i<Num_Of_M_ij; i++)
    for (int j=0; j<Num_Of_M_ij; j++)
    {
      glDepthMask(0);
      glColorMask(0,0,0,0);
      glDepthFunc(GL_LESS);
      glStencilFunc(GL_ALWAYS,1,1);
      glStencilOp(GL_KEEP,GL_REPLACE,GL_KEEP);
      M_ij[j].DrawFrontFaces();

      glDepthMask(1);
      glColorMask(1,1,1,1);
      glDepthFunc(GL_GREATER);
      glStencilFunc(GL_EQUAL,0,1);
      glStencilOp(GL_ZERO,GL_KEEP,GL_KEEP);
      M_ij[j].DrawBackFaces();
    }
}
```

EXAMPLE 1: OpenGL Code to Render ∂M From Inside.

6.2 Benchmark Results

We benchmark our PD algorithm with four models: interlocked tori, touching tori, interlocked "grates" and a pair of alphabet models, with their relative configuration shown in Fig. 9. We used the tori models because it is relatively difficult to compute a good convex decomposition for them. The interlocked "grates" model was chosen because the combinatorial complexity of its exact Minkowski sum is $O(m^3n^3)$. In our benchmarks, m and n are 1134 and 444, respectively. Therefore, it is a very challenging scenario for any PD computation algorithm. Earlier approaches based on localized computations or convex volumetric decomposition are unable to compute the PD efficiently and accurately on these benchmarks.

We measure the timings on a PC equipped with an Intel Pentium IV 1.6 GHz processor, 512 MB main memory and GeForce 3 graphics card. The complexity of the models varies from a few hundred faces to a few thousand faces. The number of leaf nodes, computed using the convex surface decomposition algorithm, vary from 67 pieces to 409 pieces. The running times vary based on the model complexity and the relative configuration of two polyhedra. It can vary from a fraction of a second, for the touching tori and a pair of alphabet models, to a few seconds for models that have deep penetration (e.g. interlocked tori and interlocked "grates"). Most of the time is spent in pairwise Minkowski sum computations and closest point queries using the graphics hardware. The local refinement based on the walking algorithm is quite fast and takes only a few milliseconds. Detailed timings for some levels of the hierarchy are given in Table 2. The acceleration techniques and hierarchical refinement result in several orders of magnitude improvement in the overall running time. Furthermore, the algorithm is able to compute very accurate PD estimates in these cases.

Level	Cull Ratio	Min. Sum	HW Query	$\|d_{est}\|$
3	31.2 %	0.219 sec	0.220 sec	0.99
5	96.7 %	0.165 sec	0.146 sec	0.53
7	98.3 %	1.014 sec	1.992 sec	0.50

(a) Interlocked Tori (2000 faces, 67 convex pieces each)

Level	Cull Ratio	Min. Sum	HW Query	$\|d_{est}\|$
3	98.4 %	0.135 sec	0.014 sec	0.29
7	99.9 %	0.105 sec	0.032 sec	0.29

(b) Touching Tori (2000 faces, 67 convex pieces each)

Level	Cull Ratio	Min. Sum	HW Query	$\|d_{est}\|$
3	0 %	0.66 sec	0.29 sec	6.41
7	96.9 %	0.43 sec	0.39 sec	0.63
9	99.9 %	0.03 sec	0.07 sec	0.63

(c) Grates (444 & 1134 faces, 169 & 409 pcs)

Level	Cull Ratio	Min. Sum	HW Query	$\|d_{est}\|$
2	50.0 %	0.055 sec	0.021 sec	0.06
4	56.2 %	0.099 sec	0.062 sec	0.03
6	97.6 %	0.080 sec	0.161 sec	0.01

(d) Alphabets (144 & 152 faces, 42 & 43 pcs)

Table 2: Benchmark Results. *We show the performance of our PD algorithm for various models. We also break down the performance to the object space culling rate, the pairwise Minkowski computation time and the closest point query time on some of the levels of the hierarchy.*

6.3 Performance Speedup by Acceleration Techniques

In Table 3, we also compare our accelerated PD algorithm presented in Section 5 with the basic algorithm presented in Section 4. As the table illustrates, the basic algorithm suffers from $O(n^4)$ computational costs, and our accelerated algorithm outperforms it by several orders of magnitude. The result is even more dramatic in a very complex scenario such as the interlocking grates model.

6.4 Accuracy of PD Computation

Our algorithm always computes an upper estimate on the amount of PD. In other words, the algorithm may be conservative and the computed answer may be more than the global minima defined in Equation 1. The tightness of the upper bound varies based on the underlying precision of the object-space and image-space computations. The algorithms for decomposition, Minkowski sum computations, simplifications and object-space culling is governed

Type	Without Accel.	With Accel.
Interlocked Tori	4 hr	3.7 sec
Touching Tori	4 hr	0.3 sec
Grates	177 hr	1.9 sec
Alphabets	7 min	0.4 sec

Table 3: Performance Speedup by Acceleration Techniques

by the precision of floating-point CPU-based hardware, which typically has 53 bits of mantissa. However, the dominant source of error are the rasterization errors as part of image-space computations.

The rasterization errors are generated from two main sources:

1. The discretization of ray directions to lie on a pixel grid for each view.

2. The fixed precision of the Z-buffer.

Increasing the resolution of the grid decreases the worst-case angular error that is proportional to the distance between adjacent pixels. Moreover, constructing tighter bounds on the minimum and maximum distances in each view (near and far plane distances), decreases the Z-buffer precision error. However, the local refinement using greedy walking in the last step of our algorithm improves the accuracy of the final solution and can substantially reduce the numerical error introduced by the use of fixed resolution image-space computations.

We also observe that the performance of our algorithm depends heavily on the extent of object-space culling, which is directly related to the amount of inter-penetration between the objects. Therefore, for applications that have spatial and temporal coherence between successive instances, our algorithm performs quite well since penetration is typically shallow during successive time steps. As a result, the algorithm is able to cull away a very high percentage of Minkowski pairs (as shown in Table 2) and quite fast in practice. Examples of such applications include dynamic simulation and tolerance verification.

7 Application

We have used our PD computation algorithm and implementation for two applications. These include efficient time stepping in dynamic simulation of rigid bodies and tolerance verification for rapid prototyping of complex structures.

7.1 Rigid Body Simulation

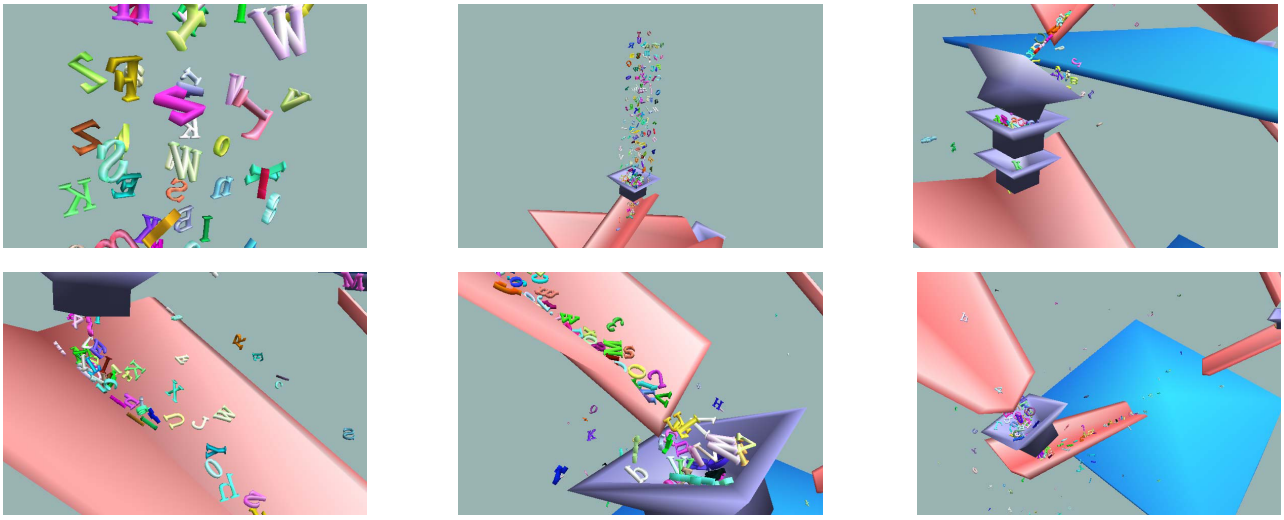


Figure 10: Application to Rigid-Body Dynamic Simulation. *Our algorithm is used to perform smarter time stepping in a dynamic simulation. A sequence of snapshots (from left to right, top to bottom) are taken from a rigid-body simulation of* 200 *models of letters and numerical digits falling onto a structure consisting of multiple ramps and funnels.*

Penetration depth (PD) computation is often needed for dynamic simulation of rigid body systems. In the physical world, objects do not occupy the same spatial extent. However, this is often unavoidable in numerical simulations. For applications involving articulated joints, stacking objects and parts assembly, bodies are nearly in contact or

actually touching each other all the time. Our PD computation algorithm provides a consistent and accurate measure of PD. Furthermore, its performance is quite fast (e.g. a fraction of a second) for shallow penetrations. In this section, we describe its application to dynamic simulation of rigid bodies, as shown in Fig. 10. This includes direct contact response computation in penalty-based methods [MZ90, MW88], as well as better time stepping technique for either impulse-based [MC95] or constraint-based simulation [Bar92, Bar94, WB97].

7.1.1 Penalty-based Methods

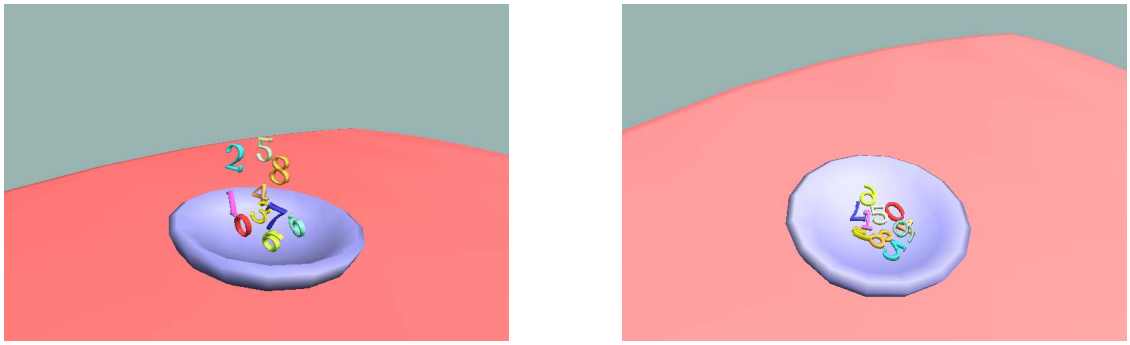


Figure 11: Challenging scenario with interlocked digits. *On the left,* 10 *digits are falling onto a bowl. On the right, a resting position of these digits is shown. Localized approaches to compute PD fail for some pairwise digits in this interlocked configuration.*

In penalty-based methods, the forces between rigid bodies are proportional to the amount of inter-penetration. Let d be the translational PD, $\boldsymbol{n}$ the direction of penetration and k a stiffness constant. The force vector $\boldsymbol{F}$ is given as:

$$\boldsymbol{F} = (k \cdot d)\boldsymbol{n} \tag{4}$$

Local vs. Global PD Computation: Localized approaches for computing the PD may fail to give a correct response for penalty-based methods. For example, in the configurations shown in Fig. 1, where we showed 2D geometric models of letters 'C' and 'I', the forces based on localized values of PD may not prevent the rigid bodies from inter-penetrating. However, our global PD computation algorithm provides a correct and robust response for such configurations. In the scenario shown in Fig. 11 several digits are interlocked. In this simulation, local approximations to the PD fail to report consistent or correct outputs for many pairs of digits.

7.1.2 Time Stepping

Unlike penalty-based methods that allow the models to inter-penetrate, some other simulation approaches impose strict non-penetration constraints. These include constraint-based simulation that distinguishes between resting contacts and colliding contacts. The resting contacts are classified based on the fact that the relative velocity is lower than a certain value. For colliding contacts, the impulsive forces are applied to prevent inter-penetration and preserve momentum properties. These impulsive forces need to be computed at the time of contact between the rigid bodies. In practice, the exact time of impact cannot be computed using analytic techniques. As a result, time stepping techniques are often used to estimate the time of collision [ST96, Mir00].

Some of the commonly used high-level scheduling algorithms for impulse-based or constraint-based rigid body simulation include retroactive detection and conservative advancement [Mir00]. Both of the techniques advance or retract the simulation time based on the separation distance between the objects and use some form of root finding algorithm to estimate the time of collision.

- **Bisection search.** Performing a simple bisection search in time is one of the commonly used technique to estimate the time of collision. This approach converges given a sufficient number of iterations. Moreover, it does not suffer from the inaccuracy of penetration depth estimation or approximations to the motion. However, it can take a long time to converge, if the time step used in the simulation is large.

- **Extrapolation.** If no information is gathered on the extent of penetration at the end of an interval, a common approach to predict the time of collision is performing extrapolation. The separation distance and the velocity of the closest features at the beginning of the interval are used for extrapolation. The main problem with this approach arises when the two objects are penetrating at the estimated time of collision. In such cases, the predicted time of collision cannot be corrected.

- **Interpolation.** If penetration information between the two objects is known, it is possible to perform interpolation. Unlike extrapolation, we make use of penetration information at the end of the interval and an iterative interpolatory scheme can be used for fast convergence to estimate the time of collision [WB97]. As shown in Fig. 1, localized estimations can deviate largely from the actual penetration depth. That can result in inaccurate estimation of the time of contact. A global and exact computation of penetration depth provides a faster and more robust convergence of the root finding scheme. Depending on the position and velocity information, the algorithm can select an appropriate order of interpolation. For example, [Mir98] performs linear interpolation using the separation distance at the beginning of the interval and an estimate of the penetration depth at the end of the interval. If we also know the velocity of the object, we can perform higher order interpolation. If only the initial velocity and the scalar value of penetration distance are known, but not the penetration direction or penetrating features, we can use quadratic interpolation. In such cases, the interpolation is performed based on the initial separation distance, and velocity, as well as the penetration depth at the end of the interval. If the algorithm also knows the penetrating features, then we can compute the relative velocity and perform cubic interpolation.

In our implementation of the time stepping scheme for dynamic simulation, we utilize the knowledge about PD features and direction and use a cubic interpolation scheme to estimate the time of collision. In general, using higher order interpolation allows us to take large steps in the simulation [WB97]. However, sometimes it is not possible to take large time steps in the simulation because of the following reasons:

- The stability of numerical integration.

- The frequency of collision events. Even if the system is numerically stable, a high frequency of contacts between the objects make the effective time step small. In such cases, the time stepping can not benefit from higher order interpolation.

Given two inter-penetrating objects, our PD algorithm computes the penetration depth d, the direction $\boldsymbol{n}$ and the penetrating features (as shown in Fig. 3). The motion in the last time step is approximated to a one-dimensional motion by projecting it onto the penetration direction. We compute a cubic interpolation of the motion, using the separation distance s and the relative velocity of the closest features at the beginning of the interval $\boldsymbol{v_s}$, along with the penetration depth and the relative velocity of the penetration features at the end of the interval $\boldsymbol{v_d}$. The cubic function is expressed as:

$$x(t) = At^3 + Bt^2 + Ct + D.$$

We treat $x(t)$ as the one dimensional distance function between the closest features or penetrating features of the rigid bodies. The parameters A, B, C and D are generic constants of a cubic polynomial that are computed by solving the following set of linear equations:

$$
\begin{aligned}
x(0) &= s = D, \\
x(T) &= d = AT^3 + BT^2 + CT + D, \\
\dot{x}(0) &= \boldsymbol{v_s} \cdot \boldsymbol{n} = C, \\
\dot{x}(T) &= \boldsymbol{v_d} \cdot \boldsymbol{n} = 3AT^2 + 2BT + C,
\end{aligned}
$$

where T is the size of the time step and $\boldsymbol{n}$ is the direction of penetration. After computing the coefficients of the cubic polynomial, we compute its first real root in the interval $[0, T]$.

Based on this approach, our algorithm is able to compute the time of collision using fewer iterations, as compared to earlier methods. If there is any error, e.g. the cubic polynomial has no real root in the interval $[0, T]$, then we use bisection search to estimate the time of collision.

In one of the example scenarios, as shown in the accompanying video and in Fig. 10, geometric models of 200 letters and digits fall along a structure consisted of multiple funnels and ramps. The letters and digits have an average complexity of 250 triangles in their boundary surface, which is decomposed into roughly 60 convex pieces. Each frame (at 30fps) of the dynamic simulation for this complex scenario takes about two minutes to compute.

For the second scenario in the supplementary video and in Fig. 11, there are 10 digit models dropped into a bowl. The geometric model of each digit has an average complexity of 250 triangles. The bowl consists of 176 triangles and its surface is decomposed into roughly 65 convex pieces. Some of the digits come into an interlocking position as they fall into the bottom of the bowl. These are challenging scenarios for contact computation and response. Each frame of this simulation takes about 18 seconds to compute.

7.2 Tolerance Verification

We have also tested our prototype implementation on a tolerance verification application, as seen in Fig. 7.2. A hammer composed of $1,692$ triangles follows a planned path through a complex virtual machine room. Its convex decomposition has 425 patches. The machine room has 897 objects and their triangle count is $195,926$. The application checks for tolerances, including separation distance and penetration depth, along a given path. In our sample path, the hammer comes into contact with 17 objects that consist of $3,810$ triangles.

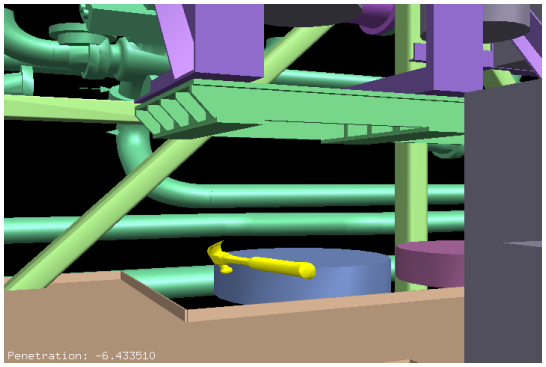


Figure 12: Tolerance Verification Scenario: *Our algorithm is used to check for positive and negative distances between a tool (yellow hammer, about 400 units long) and the nearby structures along a planned maintenance path in a virtual machine room. The amount of penetration is indicated as a negative number on the lower left corner of the scene.*

The path for routine maintenance is generated interactively with a human in the loop using a haptic device as a

motion tracker. Force feedback in the proximity of objects helps the user to avoid deep penetration. However, there still are situations where penetration between the hammer and other objects occur.

The path is played back for performing tolerance verification between the hammer and the machine room structure. At each step, the distance to the closest object is computed. When penetration occurs, the penetration distance and direction are used to help modify the planned path for machine maintenance or to improve the design of the structure layout for the room.

8 Summary and Future Work

We present a fast, global algorithm to estimate penetration depth between polyhedra using both image-space acceleration techniques and object-space culling and refinement algorithms. The resulting algorithm has been tested on difficult benchmarks and applied to time-stepping methods for dynamic simulation and tolerance verification for virtual prototyping.

There are several areas for future work. The performance of our algorithm can be further improved by exploring more optimizations. These include faster implementations of the closest point query using new features of the high-end graphics cards, as well as better hierarchical decompositions. Currently our algorithm only computes the minimum translational distance to separate two overlapping objects. It would be useful to extend it to handle rotational penetration depth.

References

[AGHP+00] P. Agarwal, L. J. Guibas, S. Har-Peled, A. Rabinovitch, and M. Sharir. Penetration depth of two convex polytopes in 3d. *Nordic J. Computing*, 7:227–240, 2000.

[AST97] Boris Aronov, Micha Sharir, and Boaz Tagansky. The union of convex polyhedra in three dimensions. *SIAM J. Comput.*, 26:1670–1688, 1997.

[Bar92] D. Baraff. *Dynamic simulation of non-penetrating rigid body simulation*. PhD thesis, Cornell University, 1992.

[Bar94] David Baraff. Fast contact force computation for nonpenetrating rigid bodies. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94*, pages 23–34. ACM SIGGRAPH, 1994. ISBN 0-89791-667-0.

[BDH93] B. Barber, D. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hull. Technical Report GCG53, The Geometry Center, MN, 1993.

[Ber01] G. Bergen. Proximity queries and penetration depth computation on 3d game objects. *Game Developers Conference*, 2001.

[BGRR96] J. Basch, L. Guibas, G. Ramkumar, and L. Ramshaw. Polyhedral tracings and their convolutions. In *Proc. Workshop on the Algorithmic Foundations of Robotics*, 1996.

[Cam97] S. Cameron. Enhancing gjk: Computing minimum and penetration distance between convex polyhedra. *Proceedings of International Conference on Robotics and Automation*, pages 3112–3117, 1997.

[CC86] S. Cameron and R. K. Culley. Determining the minimum translational distance between two convex polyhedra. *Proceedings of International Conference on Robotics and Automation*, pages 591–596, 1986.

[CDST97] Bernard Chazelle, D. Dobkin, N. Shouraboura, and A. Tal. Strategies for polyhedral surface decomposition: An experimental study. *Comput. Geom. Theory Appl.*, 7:327–342, 1997.

[DHKS93] D. Dobkin, J. Hershberger, D. Kirkpatrick, and Subhash Suri. Computing the intersection-depth of polyhedra. *Algorithmica*, 9:518–533, 1993.

[EJR89] D. Epstein, F. Jansen, and J. Rossignac. Z-buffering rendering from csg: The trickle algorithm. Technical report, IBM Research Report RC15182, 1989.

[EL01] 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.

[FL01] S. Fisher and M. C. Lin. Deformed distance fields for simulation of non-penetrating flexible bodies. *Proc. of EG Workshop on Computer Animation and Simulation*, 2001.

[GH97] M. Garland and P. Heckbert. Surface simplification using quadric error bounds. *Proc. of ACM SIGGRAPH*, pages 209–216, 1997.

[GHF86] Jack Goldfeather, Jeff P. M. Hultquist, and Henry Fuchs. Fast constructive-solid geometry display in the Pixel-Powers graphics system. In *Proc. of ACM SIGGRAPH*, volume 20, pages 107–116, 1986.

[GJK88] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi. A fast procedure for computing the distance between objects in three-dimensional space. *IEEE J. Robotics and Automation*, vol RA-4:193–203, 1988.

[GLM96] S. Gottschalk, M. Lin, and D. Manocha. Obb-tree: A hierarchical structure for rapid interference detection. In *Proc. of ACM Siggraph'96*, pages 171–180, 1996.

[GME+00] A. Gregory, A. Mascarenhas, S. Ehmann, M. C. Lin, and D. Manocha. 6-dof haptic display of polygonal models. *Proc. of IEEE Visualization Conference*, 2000.

[GS86] Leonidas J. Guibas and R. Seidel. Computing convolutions by reciprocal search. In *Proc. 2nd Annu. ACM Sympos. Comput. Geom.*, pages 90–99, 1986.

[GS87] L. Guibas and R. Seidel. Computing convolutions by reciprocal search. *Discrete Comput. Geom*, 2:175–193, 1987.

[HCK+99] 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.

[HKL+98] D. Hsu, L. Kavraki, J. Latombe, R. Motwani, and S. Sorkin. On finding narrow passages with probabilistic roadmap planners. *Proc. of 3rd Workshop on Algorithmic Foundations of Robotics*, 1998.

[Hub95] Philip M. Hubbard. Collision detection for interactive graphics applications. *IEEE Trans. Visualization and Computer Graphics*, 1(3):218–230, September 1995.

[HZLM01] K. Hoff, A. Zaferakis, M. Lin, and D. Manocha. Fast and simple geometric proximity queries using graphics hardware. *Proc. of ACM Symposium on Interactive 3D Graphics*, 2001.

[KHM+98] J. Klosowski, M. Held, Joseph S. B. Mitchell, K. Zikan, and H. Sowizral. Efficient collision detection using bounding volume hierarchies of k-DOPs. *IEEE Trans. Visualizat. Comput. Graph.*, 4(1):21–36, 1998.

[KOLM01] Y. Kim, M. Otaduy, M. Lin, and D. Manocha. 6-dof haptic display using localized contact computations. *Proc. of Haptics Symposium*, 2001.

[KR92] A. Kaul and J. Rossignac. Solid-interpolating deformations: construction and animation of pips. *Computer and Graphics*, 16:107–116, 1992.

[LC91] M.C. Lin and John F. Canny. Efficient algorithms for incremental distance computation. In *IEEE Conference on Robotics and Automation*, pages 1008–1014, 1991.

[MC95] B. Mirtich and J. Canny. Impulse-based simulation of rigid bodies. In *Proc. of ACM Interactive 3D Graphics*, Monterey, CA, 1995.

[Mir98] Brian Mirtich. V-Clip: Fast and robust polyhedral collision detection. *ACM Transactions on Graphics*, 17(3):177–208, July 1998.

[Mir00] B. Mirtich. Timewarp rigid body simulation. *Proc. of ACM SIGGRAPH*, 2000.

[MPT99] W. McNeely, K. Puterbaugh, and J. Troy. Six degree-of-freedom haptic rendering using voxel sampling. *Proc. of ACM SIGGRAPH*, pages 401–408, 1999.

[MW88] Matthew Moore and Jane Wilhelms. Collision detection and response for computer animation. In John Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 289–298, August 1988.

[MZ90] Michael McKenna and David Zeltzer. Dynamic simulation of autonomous legged locomotion. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 29–38, August 1990.

[OG96] C. J. Ong and E.G. Gilbert. Growth distances: New measures for object separation and penetration. *IEEE Transactions on Robotics and Automation*, 12(6), 1996.

[Req93] A.A.G. Requicha. Mathematical definition of tolerance specifications. *ASME Manufacturing Review*, 6(4):269–274, 1993.

[RMS92] Jarek Rossignac, Abe Megahed, and Bengt-Olaf Schneider. Interactive inspection of solids: Cross-sections and interferences. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 353–360, July 1992.

[ST96] D. E. Stewart and J. C. Trinkle. An implicit time-stepping scheme for rigid body dynamics with inelastic collisions and coulomb friction. *International Journal of Numerical Methods in Engineering*, 39:2673–2691, 1996.

[TPK01] T. Theoharis, G. Papaiannou, and E. Karabassi. The magic of the z-buffer: A survey. *Proc. of 9th International Conference on Computer Graphics, Visualization and Computer Vision, WSCG*, 2001.

[WB97] A. Witkin and D. Baraff. *Physically Based Modeling: Principles and Practice*. ACM Press, 1997. Course Notes of ACM SIGGRAPH.

[Wie96] T F Wiegand. Interactive rendering of csg models. *Computer Graphics Forum*, 15(4):249–261, 1996.