Incremental algorithms for collision detection between solid models *

Madhav K. Ponamgi Dinesh Manocha Ming C. Lin
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599
{ponamgi,manocha,lin}@cs.unc.edu

Abstract: Fast and accurate collision detection between general solid models is a fundamental problem in solid modeling, robotics, animation and computer-simulated environments. Most of the earlier algorithms are either restricted to a class of solid models, say convex polytopes, or are not fast enough for practical applications. We present an incremental algorithm for collision detection between general B-rep solid models in dynamic environments. The algorithm combines a hierarchical representation with incremental frame to frame computation to rapidly detect collisions. It makes use of coherence between successive instances to determine efficiently the number of object features interacting. For each pair of objects, it tracks the closest features between them on their respective convex hulls using the [LC91] algorithm. It detects when these objects penetrate using a new psuedo-internal Voronoi data structure and constructs the penetration region, identifying the regions of contact on the convex hulls. The features associated with these regions are represented in a precomputed hierarchy. The algorithm uses a coherence based approach to quickly traverse the pre-computed hierarchy and check for collisions between the features. The algorithm works well in practice and its complexity is output sensitive.

1 Introduction

A realistic visual simulation system, which couples geometric modeling and physical prototyping, can provide a useful toolset for applications in robotics, CAD/CAM design, molecular modeling, manufacturing design simulations, etc. Such systems create electronic representations of mechanical parts, tools, and machines, which need to be tested for interconnectivity, functionality, and reliability. The goal of these *virtual and electronic simulation* systems is to save processing time and manufacturing costs by avoiding the production of actual physical prototypes [Hop88, HH87]. This is similar to the goal of CAD tools for VLSI. It requires a complete test environment for simulating hundreds of parts interacting.

*Supported in part by a Sloan fellowship, University research council grant, NSF grant CCR-9319957, ONR contract N00014-94-1-0738, ARPA contract DABT63-93-C-0048, NSF/ARPA Science and Technology Center for Computer Graphics & Scientific Visualization and NSF Prime contract No. 8920219

A visual simulator can also be a significant aid in engineering analysis. Experiments which are too costly to construct or impractical to perform can be simulated, such as support design for tunnels, where the engineers can visualize and test effect of natural phenomena on the proposed tunnel support and watch the interactions among the blocks and supports under stress. Another example is automobile crash tests which can be done at much lower cost and under more varied conditions using computer aided simulation.

Both interactive applications and simulations of physical systems strive to present accurate simulation with realistic visualization at interactive rates. For a simulation system to mimic a true physical process, it needs not only to render realistic images according the material's physical and geometric properties, but also to model object interactions *precisely*. The interactions may involve objects in the simulation environment pushing, striking, or smashing other objects. *Detecting collisions and determining contact points* is a crucial step in portraying these interactions accurately.

The most challenging problem in a dynamic and visual simulation, namely the collision phase, can be separated into three parts: collision detection, contact area determination, and impact response. In this paper, we address two key elements of dynamic simulation by presenting a general purpose collision detection and contact area determination algorithm for simulations. Our collision detection routine reports the contact area and thus enables the application to compute an appropriate response.

Our algorithm not only addresses interaction between a pair of general non-convex objects, but also large environments consisting of hundreds of moving parts. We make no assumptions regarding the geometry of the objects nor do we assume the motions of the objects to be expressed as a closed form function of time. Our collision detection scheme is efficient and accurate (to the resolution of the models).

Main Contribution: We present an efficient algorithm for exact collision detection between general polyhedral models. Our algorithm combines a coherence based approach with a hierarchical algorithm to effectively prune down the number of object pairs interacting and to reduce the total number of features we examine between a colliding object pair. Our algorithm is output sensitive and its complexity is determined by the number of object pairs in close proximity of each other and the number of features of each such pair. The complexity of determining the number of objects in close proximity is $O(k+N)$, where N is the number of moving objects in the environment and k is the total number of object pairs whose axis-aligned bounding boxes overlap. For each pair of objects whose bounding boxes overlap, the complexity is $O(m+n)$, where m is the number of feature pairs between the objects that are close to each other. The n features of a polyhedron consist of vertices, edges and faces. In particular, m correspond to number of feature pairs whose axis-aligned bounding boxes overlap. In practice k and m are small. In the worst case, k can be $O(N^2)$ and m can be $O(n^2)$. We highlight its performance on a threaded screw insertion and a chain of toroidal rings.

Organization: The rest of the paper is organized as follows: Section 2 reviews some of the previous work in collision detection. Section 3 gives an overview of our approach, including some of our previous work and the concept of coherence. Section 4 presents the algorithm

along with a number of *implementation* issues. We highlight its performance in Section 5 and address a number of robustness issues in Section 6.

2 Previous Work

Collision detection has been extensively studied in geometric modeling, computer graphics, robotics, and computational geometry. Since collision detection is needed in a wide variety of situations, many different methods have been proposed. Most of them make specific assumptions about the objects of interest and design a solution based on object geometry or application domain.

Most robotics literature deals with collision detection in the context of path planning. Using sophisticated mathematical tools, several algorithms have been developed that plan collision-free paths for a robot in restricted environments [CC86, Can86]. This differs from virtual prototyping and simulation-based applications, where motion is subject to dynamic constraints or external forces and cannot be expressed as a closed form function of time.

Computational geometry literature typically deals with collision detection of objects in a static environment. Objects are at a fixed location and orientation, and the algorithms determine whether they are intersecting [CD87, DK85]. In most modeling and graphics applications, where many objects are in motion, such an approach would be inefficient. Moreover, the objects move only slightly from frame to frame and the collision detection scheme should take advantage of the information from the previous frame to initialize the computation for the current frame [Bar92, LC91]. Several solutions based on this idea of coherence have been proposed in [Lin93].

Approaches that combine collision response with detection can be found in [Bar92, WG93, MW88]. The methods in [WG91, WG93] make use of boundary representation to detect collisions.

Collision detection for multiple moving objects has recently become a popular research topic with the increased interest in large-scaled virtual prototyping environments. For example, a vibratory parts feeder can contain up to hundreds of mechanical parts moving simultaneously under periodical force impulses in a vibratory bowl or tray. There may be N moving objects and M stationary objects. Each of the N moving objects can collide with the other moving objects, as well as the stationary ones. Keeping track of $\binom{N}{2} + NM$ pairs of objects at every time step can become time consuming as N and M get large. To achieve interactive rates, the total number of pairwise intersection tests must be reduced before performing exact collision tests on the object pairs, which are in the close vicinity of each other. Several methods dealing with this situation are found in [Cam91, CLMP95, DZ93]. Most methods use some type of a hierarchical bounding box scheme. Objects are surrounded by bounding boxes. If the bounding boxes overlap, indicating the objects are near each other, a more precise collision test is applied.

Approximate collision detection schemes using a series bounding boxes, spheres, or voxels can be found in [AANJ94, HBZ90]. More recently, Hubbard [Hub93] has formulated an approach where accuracy is traded for speed. The approximation schemes can be used

to deal with general non-convex objects. A non-convex object can be broken into these sub-units which are tested for collisions. However, to achieve desired accuracy using these approaches in simulation is computationally expensive.

Many other techniques have been proposed on spatial partitioning methods for collision detection among multiple object pairs [Tur89]. These include binary space partition trees (BSP's), octrees, and approximation hierarchies [Cam91, FKN80, NAT90, Nay92]. However, they do not make use of coherence between successive instances and are relatively slow in practice. Algorithms based on interval arithmetic for collision detection are described in [Duf92, ea93]. These algorithms expect the motion of the objects to be expressed as a closed form function of time. Moreover, the performance of interval arithmetic based algorithms is too slow for interactive applications. Coherence based algorithms for curved models are presented in [LM95].

2.1 Algorithm Overview

In this paper, we present a fast algorithm for contact determination between non-convex polyhedra. Initially, we apply the distance computation algorithm of [LC91] to the convex hulls of the polyhedra. The algorithm described in [LC91] may go into a cyclic loop when convex hulls interpenetrate. To circumvent this problem, we introduce the notion of *pseudo internal Voronoi regions* for a convex polyhedron. These are used to detect when the convex hulls of a pair of objects penetrate. This is different from computing the amount of penetration between two objects, as shown in [GO94]. The modified algorithm with penetration detection has the same complexity as the original algorithm in [LC91].

We classify the features of the overlapping convex hulls into those belonging to the original non-convex objects, and those constructed while computing the convex hull of the non-convex objects. The "fictitious" faces constructed by the convex hulls, cover the subset of features we wish to test for collision between the pair of objects. Rather than performing a brute-force $O(n^2)$ all possible pairs test, we employ a hierarchical *sweep and prune* technique based on bounding boxes for these features. This reduces to sorting bounding boxes in the three-dimensional space by taking their projections on one-dimensional axes. Based on coherence, the overall complexity is $O(n + m)$ at each instance, where m is the number of overlapping bounding boxes of the features. These m feature pairs are eventually tested for contacts. In practice, the overall complexity is output sensitive. Typically m is a small constant or a linear function of n. Our algorithm is suitable for large environments and simulations consisting of hundreds of moving objects. We make no assumptions regarding the geometry or the motions of the objects.

3 Background

We review our previous algorithm for multiple moving convex polytopes in complex environments. Coherence combined with incremental computation is a major theme of our work. By exploiting coherence, we are able to incrementally trim down the number of pair-

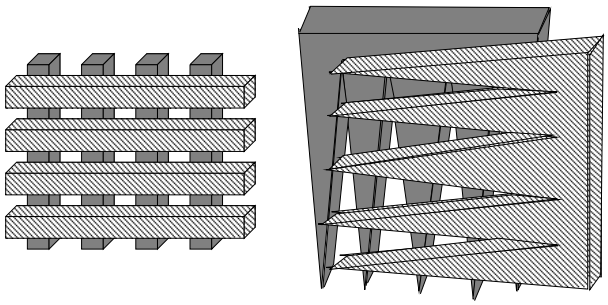

Figure 1: Worst case scenarios for collision detection.

wise object and feature tests involved in each iteration. For many applications in robotics and modeling, assuming coherence is reasonable [Bar90, Lin93].

Definition: *Temporal and geometric coherence* is the property that the state of the application does not change significantly between successive time steps or simulation frames. The objects move only slightly from frame to frame. This slight movement of the objects translates into geometric coherence, since their geometry, defined by the vertex coordinates, changes slightly between frames.

For a configuration of N objects, the worst case running time for any collision detection algorithm is $O(N^2)$ where N is the number of objects. For instance, consider Fig. 1(a) where all object pairs are colliding. A similar situation can occur for all feature pairs between a pair of colliding objects as shown in Fig. 1(b) with "comb polyhedra". However, evidence suggests that these cases rarely occur in real models [CLMP95, Lin93]. So our algorithm uses a *Sweep and Prune* technique to eliminate testing object pairs that are far apart, and later we show that the technique can be extended to eliminate testing features that are far apart between two colliding objects.

3.1 Sweep and Prune

We use a bounding box based scheme to reduce the $O(N^2)$ bottleneck of testing all possible pairs of objects for collisions. In most realistic situations, an object has to be tested against a small fraction of all objects in the environment for collision. For example, in a simulation of a vibratory parts feeder most objects are in close proximity to only a few other objects. It would be pointless and expensive to keep track of all possible interactions between objects at each time step.

Sorting the bounding boxes surrounding the objects is the key to our *Sweep and Prune* approach [CLMP95]. It is not intuitively obvious how to sort bounding boxes in 3-space to determine overlaps. We use a *dimension reduction* approach. If two bounding boxes collide in 3-D, then their orthogonal projections on the x, y, and z axes must overlap. The sweep and prune algorithm begins by projecting each 3-D bounding box surrounding an object onto the x, y, and z axes. Since the bounding boxes are axially-aligned, projecting them onto the coordinate axes results in intervals. We are interested in overlaps among these intervals, because a pair of bounding boxes can overlap *if and only if* their intervals overlap

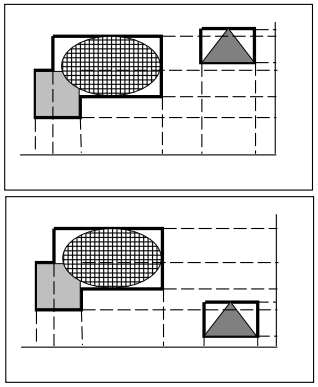

Figure 2: Bounding box overlaps in two dimensions.

in all three dimensions. Fig. 2 shows the overlapping intervals of the triangular object between different timesteps.

We construct three lists, one for each dimension. Each list contains the values of the endpoints of the intervals in each corresponding dimension. By sorting these lists, we can determine which intervals overlap. In the general case, such a sort would take $O(N \log N)$ time, where n is the number of objects. We can reduce this time bound by keeping the sorted lists from the previous frame, changing only the interval endpoints. In environments where the objects make relatively small movements between frames, the lists will be nearly sorted, so we can sort using *insertion sort* in expected $O(N)$ time.

In addition to sorting, we need to keep track of changes in overlap status of interval pairs (i.e. from overlapping in the last time step to non-overlapping in the current time step, and vice-versa). This can be done in $O(N + e_x + e_y + e_z)$ time, where e_x, e_y, and e_z are the number of exchanges along the x, y, and z axes while sorting [Bar92]. Since the exchanges are done during the sorting stage, the running time has the same expected complexity of $O(N)$. At the end of this stage, we have a list of pairs of objects whose bounding boxes overlap. The number of pairs correspond to k. We pass this list onto the next stage of the algorithm that determines for each pair if the convex hulls of the objects are overlapping.

3.2 Collision Detection between Convex Polytopes

We use the algorithm described in [LC91, Lin93] to keep track of closest features for a pair of convex polytopes. The algorithm maintains a pair of closest features for each convex polytope pair and calculates the Euclidean distance between the features to detect collisions. This approach can be used in a static environment, but is especially well-suited for dynamic environments in which objects move in a sequence of small, discrete steps. The method takes advantage of coherence: the closest features change infrequently as the polytopes move along finely discretized paths. In most situations, the algorithm runs in *expected constant time* if the polytopes are not moving swiftly.

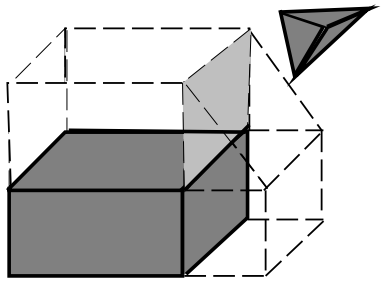

Figure 3: A walk across Voronoi cells.

3.2.1 Voronoi Regions

Each convex polytope is pre-processed into a modified boundary representation. The polytope data structure has fields for its features (faces, edges, and vertices) and corresponding *Voronoi regions*. Each feature is described by its geometric parameters and its neighboring features, i.e. the topological information of incidences and adjacencies.

Definition: A *Voronoi region* associated with a feature is a set of points closer to that feature than any other [PS85].

The Voronoi regions form a partition of the space outside the polytope, and form the generalized Voronoi diagram of the polytope. Note that the generalized Voronoi diagram of a convex polytope has linear number of features and consists of polyhedral regions. A *cell* is the data structure for a Voronoi region of a single feature. It has a set of constraint planes which bound the Voronoi region with pointers to the neighboring cells (which share a constraint plane with it) in its data structure. If a point lies on a constraint plane, then it is equi-distant from the two features which share this constraint plane in their Voronoi regions. For more details on this construction and its properties, please refer to [Lin93].

3.2.2 Closest Feature Tests

Our method for finding closest feature pairs is based on Voronoi regions. We start with a candidate pair of features, one from each polytope, and check whether the closest points lie on these features. Since the polytopes and their faces are convex, this is a local test involving only the neighboring features of the current candidate features. If either feature fails the test, we step to a neighboring feature of one or both candidates, and try again. As the Euclidean distance between feature pairs must always decrease when a switch is made, cycling is impossible for non-penetrating objects. An example of the algorithm is given in Fig. 3.

Given a pair of features *Face 1* and *vertex* V_b, on objects A and B, as the closest features we test to see if vertex V_b lies within *Cell 1* of of *Face 1*. V_b violates the constraint plane imposed by *CP* of *Cell 1*. The constraint plane *CP* has a pointer to its adjacent cell *Cell 2*, so the walk proceeds to test the containmentship of V_b within *Cell 2*. In similar fashion, vertex V_b has a cell of its own, and we see if the nearest point P_a on the edge to the vertex V_b lies within V_b's Voronoi cell. Basically, we are performing the containmentship tests of a

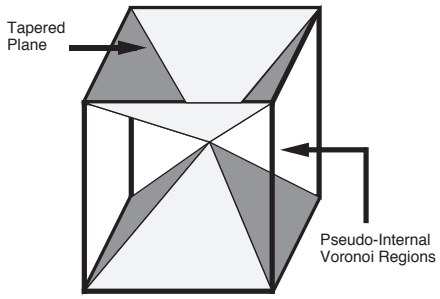

Figure 4: Pseudo Internal Voronoi regions.

point within a Voronoi region defined by the constraint planes of the region. The constraint plane causing the containmentship test to fail points the next direction for the algorithm to *advance* in the search of a new and *closer* feature. Eventually, we must reach the closest pair of features.

4 Collision Detection for General Polyhedral Model

The original collision detection algorithm goes into a cyclic loop when penetration occurs because it tries to compute the closest feature-pair using only the exterior Voronoi regions. A penetration implies that the polytopes have entered into each other's interior where the space has previously *not* been partitioned. This violates the algorithm's basic assumption that two convex polytopes are separated and therefore results in the cyclic loop.

4.1 Penetration Detection for Polytopes

The key to detecting penetrations lies in partitioning the *interior* as well as the exterior of the polytope. We are operating on the convex hull object, so the penetration detection algorithm is used for determing when the convex hulls of the objects penetrate. For internal partitioning, internal Voronoi regions can be used. The internal Voronoi regions can be constructed for any convex polytope by computing all the equi-distant hyperplanes between two or more facets on the polytope. However the general construction of the internal Voronoi regions is a non-trivial computation [PS85]. To detect a penetration – as opposed to knowing *all* the closest features – it is unnecessary to construct the exact internal Voronoi regions.

To simply detect a penetration, it is unnecessary to construct the exact internal Voronoi regions. We are not interested in knowing *all* the closest features between two interpenetrating polytopes, but only the fact that they are penetrating.

An approximation to internal Voronoi regions is calculated by first computing the centroid of each convex polytope – the weighted average of all vertices. Then, a plane from each edge is extended towards the centroid. The extended plane tapers to a point, forming a pyramid-type cone over each face. If all the faces are equi-distant from the centroid, as in the cuboid, these hyperplanes form the exact Voronoi diagram inside the polytope.

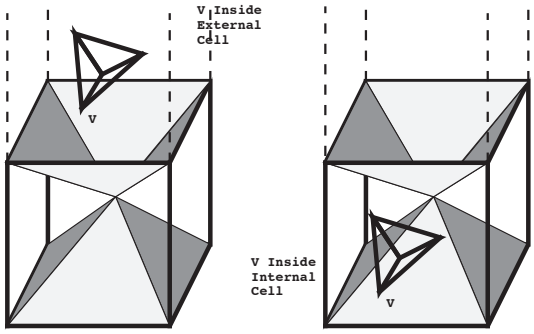

Figure 5: Walk from external to internal Voronoi regions.

The data structure of these *pseudo* internal Voronoi regions is similar to the Voronoi regions described in the previous section. Each region associated with each face has $e + 1$ hyperplanes defining it, where e is the number of edges in the face's boundary. Each hyperplane has a pointer directing to its neighboring region where the algorithm will step to next, *if* the constraint imposed by this constraint plane is violated. In addition, a *type* field is added in the data structure of a Voronoi cell to indicate whether it is an interior (pseudo) or exterior Voronoi region.

The original collision detection algorithm goes into a cyclic loop when penetration occurs because it tries to compute the closest feature-pair using only the exterior Voronoi regions. A penetration implies that the polytopes have entered into each other's interior where the space has previously *not* been partitioned. This violates the algorithm's basic assumption that two convex polytopes are separated and results in the cyclic loop.

The pseudo internal Voronoi data structure takes care of this situation. Each of the faces of the given polytope is now used as a constraint plane. If a candidate feature fails the constraint imposed by the face (indicating the closest feature pair lies possibly behind this face), the algorithm stepping "enters" inside of the polytope. If at any time we find one point on a feature of one polytope is contained within the pseudo internal Voronoi region (i.e. this point satisfies the constraints posed by an pseudo internal Voronoi region of the other polytope), it corresponds to a penetration. Otherwise, the walk progresses as before and the closest feature pair will be found between these two polytopes. Several possible scenarios are indicated in Fig. 5.

In Fig. 5, initially two polytopes are separated. The closest pair of features between them are V and the face cell. (Note that V lies inside the Vornoi cone of the face and vice versa.) As the pyramid moves closer toward the cuboid and penetrates the face, now V lies inside of the internal region . This *region switch* is indicated when V fails the constraint imposed by the face (i.e. V lies beneath face). At this point, the face and V will be returned as the critical feature pair where the penetration occurs. The face then points to its pseudo-internal Vornoi region as the next candidate region. Once the algorithm verifies the containment of V inside the internal region, a penetration is reported as in Fig. 5.

The internal and external Voronoi regions are constructed during the pre-processing

stage. Each Voronoi region defined by a large number of constraint planes is subdivided into smaller region and each new region has a constant number of hyperplanes. Therefore, all the Voronoi regions are defined by a constant number of constraint planes. This permits us to claim that each verification test (i.e. whether a point is contained within a Voronoi region) takes only constant time.

In most of applications, we check for collisions at regular time intervals. During each interval, the objects' motions are relatively small and generally preserve the geometric relationships. This implies that tracking the new closest feature pairs between objects involves traversing only a few Voronoi regions per object. Hence, the augmented distance tracking algorithm with penetration detection runs as the original algorithm in nearly constant time.

4.2 General Collision Detection

Building on the algorithms described in the previous section and the penetration dection algorithm, we present an incremental output sensitive algorithm for collision detection between general polyhedral models. The simplest algorithms are based on convex decomposition along with algorithms for convex polytopes and multiple object pairs. However, we chose *not* to use this approach for a number of reasons. The problem of computing an optimal convex decomposition of 3-D objects has been shown to be NP-Hard [Cha84]. Recently algorithms for computing convex decomposition have been described in [BT92]. They are fairly non-trivial to implement and in many cases generate too many internal or "fictitious" features (edges and faces). For example, consider a polygonal representation of a torus (Fig. 6). In such a case, any convex decomposition will generate $O(n)$ internal features and decompositions.

We make use of the coherence between successive frames. For each non-convex polyhedral object, we compute its convex hull and the axis-aligned bounding box enclosing the convex hull. A four level hierarchical algorithm is used to detect collisions:

1. For each pair of objects, we determine if the bounding boxes are overlapping using the *Sweep and Prune* technique of Section 3.1.

2. For each intersecting bounding box pair, we check if the convex hulls of the objects are colliding using the penetration detection algorithm of Section 4.1.

3. For each intersecting convex hull pair, we compute the areas of intersection on the convex hulls. The faces comprising these areas are either actual features on the original object or are faces (introduced by the convex hull) covering features of the object.

4. The features on the areas of intersection are represented as a pre-computed hierarchies. They are traversed using a hierarchical version of the *Sweep and Prune* technique to find exact collision.

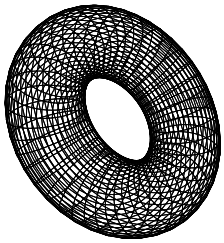

Figure 6: Example of difficult object to convex decompose.

After the first and second stage of the algorithm, we can determine whether the convex hull of two non-convex objects are colliding. Next, we need to identify the regions of contact, so that we can examine the interior of the convex hulls for collisions between features not on the convex hulls. The features internal to the convex hull are organized in a pre-processed data structure for efficient traversal.

We first describe the pre-processing used to construct the data structures for the extended *Hierarchical Sweep and Prune* traversal. Then in section 4.7 we explain how this technique is used in a dynamic environment.

4.3 Classifying the Features

Each non-convex object is described as a set of faces. Each face consists of a list of vertices describing the edges of the face. The faces, edges, and vertices comprising an object are called its features. Our first step in pre-processing is to categorize these features. Constructing the convex hull of the object separates its features into three distinct regions (as in Fig. 7). The features of the object that are coincident with the faces of the convex hull are identified as the *hull features* of the object. The features that do not lie on the convex hull are identified as the *concavity features*. The vertices and edges bordering the hull and concavity features are designated as the *boundary features*. Formally, we define $\mathbf{CH(A)}$ as the convex hull of object A, $\mathbf{HF(A)}$ as the hull features of object A, $\mathbf{CF(A)}$ as the concavity features of object A, and $\mathbf{BF(A)}$ as the boundary features of object A. The relationship among these features can be represented using set operations:

$$\mathbf{HF(A)} = \mathbf{A} \cap \mathbf{CH(A)}.$$

$$\mathbf{CF(A)} = \mathbf{CH(A)} - \mathbf{HF(A)}.$$

$$\mathbf{BF(A)} = \mathbf{HF(A)} \cap \mathbf{CF(A)}.$$

Although we have grouped features of the same type in a single set, in reality each single set is a union of disjoint sets of the same type of features. We identify these subsets using a depth-first search technique. Any face that is a member of the hull feature set has to have all of its vertices on the convex hull. Using this observation, we begin with a face belonging to the object lying on the convex hull and determine if any of its neighboring

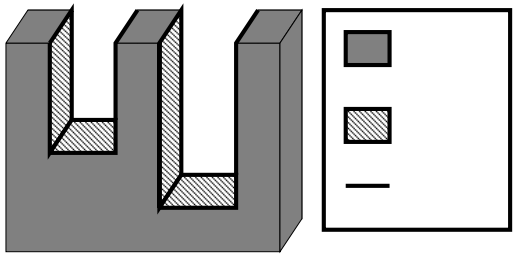

Figure 7: Example of Hull, Concavity, and Boundary Features.

faces also lie on the convex hull. This operation is repeated recursively. The end product is a single hull feature set composed of faces of the object on the convex hull. If there are any additional faces of the object that lie on the convex hull, but are not members of this set, then they are part of at least one other set. The depth-first search technique is applied again to all such faces until all the faces lying on the convex hull are associated with a hull feature set. Applying this method for an arbitrary object A results in a set of disjoint hull feature sets. This collection is denoted as $\mathbf{HF(A)}$.

In a similar fashion, the concavity features are identified. Any face that has one or more of its vertices *not* on the convex hull, is a member of the concavity feature set. Here too, multiple disjoint concavity feature sets can result. For example, a sphere with multiple dimples or dents (like a golf ball) is an object with multiple disjoint concavities.

The boundary feature set is the intersection of the hull feature set and the concavity feature set. The vertices and edges of the boundary can be identified by looking at the faces of the hull features which border a concavity. So any face that is part of a concavity, but has at least one vertex/edge on the convex hull contains a boundary vertex/edge. To identify the boundary vertices as a connected series of segments a three-step process is followed:

1. All the boundary vertices are identified by examining faces belonging to the concavity feature set bordering faces belonging to the hull feature set. We find them by comparing the vertices of the convex hull with the vertices of the object. Faces that are part of a concavity but have a vertex on the convex hull, contain a boundary vertex.

2. We begin from one boundary vertex and determine if any faces from the concavity feature set adjoining it contain boundary vertices. For each face that does, the edges leading from this vertex to those other boundary vertices form part of a boundary loop. This operation is applied recursively. When this operation is applied to all boundary vertices, the end result is a series of boundary loops.

3. All boundary loops that lie on the same plane are grouped together. It is possible that a boundary loop has only a portion on a common plane with other boundary loops. The loops and loop segments that lie on a common plane also lie on the same face of the convex hull of this object. So for each face of the convex hull, the boundary loops and loop segments resting on it are identified.

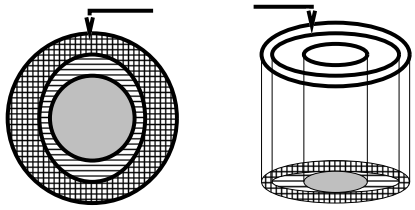

Figure 8: Nested boundary loops.

In addition, each concavity feature subset has associated with it a set of boundary loops. In particular, these are the loops that lie on the convex hull faces covering this concavity feature. The boundary loops can form complicated 3-D structures.

- The boundary loops bordering a concavity are 3-dimensional structures shown in Fig. 7.
- The boundary loop can have holes in the sense that they can be arbitrarily nested. See for example Fig. 8. The rims of a series of connected cylindrical shells of the same height but decreasing diameter that are nested within each other form a nested collection of boundary loops.
- The boundary loop can form self-intersecting structures.

4.4 Computing Caps

We want to compute a "cap" over each concavity feature subset using its boundary loops. These caps "hide" the features of the concavity. Multiple caps need to be computed because an object may be composed of disjoint concavity feature subsets. The dimpled sphere described earlier requires multiple caps because it consists of multiple concavity feature subsets connected by a single hull feature set. Using our former notation, a cap of object A is denoted as $\mathbf{CAP(A)}$:

$$\mathbf{CAP(A) = CH(A) - A}.$$

There has been some work done on computing 2-dimensional caps (caps for non-convex polygons) [Lou66]. A cap algorithm for simple 3-dimensional objects has been described in [AP76]. Fig. 9 shows an example of a cap.

The caps are computing the "missing" regions of the object so as to form its convex hull. The convex hull of an object is unique, and every face of the convex hull is a convex polygon. However, an original face of the object may be coincident with only a portion of the convex hull face covering it. These portions have to be carefully subtracted from the convex hull faces. The end result is that arbitrary non-simple polygons may be formed out of the convex hull faces with this subtraction process (See Fig. 9 for a simple example). This leads to the following $O(n^2)$ algorithm for computing caps, provided the faces of the object and convex hull are bounded by a constant number of edges:

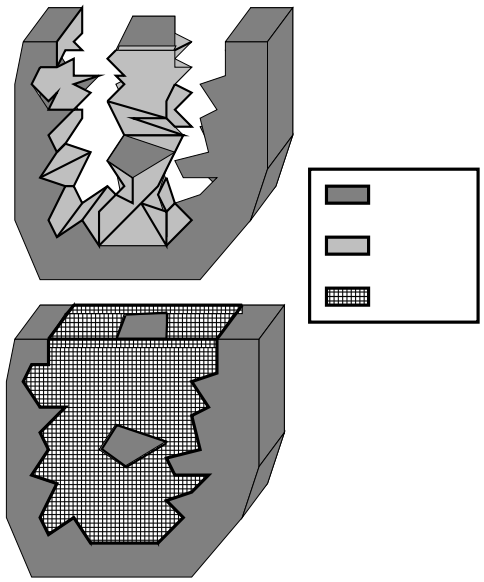

Figure 9: Cap Example.

1. Construct the convex hull of the object.

2. Each face of the object lying on the convex hull is subtracted from the convex hull by finding the polygons it intersects or overlaps.

3. Repeat steps 1 and 2 until no faces are left on the object that lie on the convex hull.

4. We identify each cap with the concavity features it covers. For each concavity feature we use an algorithm described earlier to determine its boundary vertices. A cap that uses the same vertices is its cover.

The $O(n^2)$ complexity of the algorithm comes from convex hull computation. In our implementation we use incremental algorithms of quadratic complexity as opposed to optimal algorithms of complexity $O(n \log n)$ [PS85]. Moreover, the resulting non-simple polygons are triangulated. The implementation of this algorithm has some numerical robustness problems for long thin faces. An implementation less susceptible to this is a brute force triangulation of the boundary vertex sets. All pairs of vertices resting on a common convex hull face are considered as possible new edges. If a new edge crosses an existing edge or rests on an existing face of the object it is rejected. Otherwise, it is added to the list of legal edges. Its worst case complexity can be $O(n^3)$, but it is simpler to implement.

After the cap computation, one final data structure is constructed. Each concavity feature subset is represented as an axis-aligned bounding box hierarchy. The bounding box hierarchy is traversed using an extended version of the *Sweep and Prune* algorithm when trying to compute collisions between the features that do not lie on the convex hull.

4.5 Constructing the Bounding Box Hierarchy

We have used a minor variation of octrees based on vertex partitioning. We refer to as an *octree variant* (OTV). Initially, for both of them, the vertices of each concavity are surrounded by an axis aligned bounding box. For an octree hierarchy, this bounding box is split into smaller bounding boxes with uniformly spaced cutting planes along the x, y, and z axes. For example, splitting the box in half along the x, y, and z axes results in eight children for the top-level box. The number of cuts along each axis is a user-definable parameter. The subdivision of child boxes continues recursively until each child box contains a threshold number of vertices. Then the faces associated with vertices are introduced.

Each face is surrounded by an axis aligned bounding box that contains all of its orientations. Usually these face bounding boxes do not fit completely inside the leaf box containing the vertices. So each such face is trimmed against the sides of the leaf box. The trimmed features are introduced as new faces into the boxes they spill over. This operation is carried out over all boxes at the leaf nodes and at the end of it a hierarchical representation of the concavity subset is constructed. An alternative to splitting the faces that straddle the boundaries of the partitioning walls is to evenly divide the faces among the bounding boxes they overlap. This results in overlapping boxes, but no new faces are introduced.

A binary-space-partitioning (BSP) tree was not used, primarily because for our sweep and prune technique to work, axis-aligned bounding boxes are required. For the octree variant tree (OTV), the cutting planes used to divide the vertices are *not* uniformly distributed over the bounding box volume. The cutting planes are introduced in a manner similar to $k-d$ trees [PS85]. For example, a single cut along each of the x, y, and z axes is implemented as follows:

- The median of the x-coordinates is found. A cutting plane perpendicular to the x-axis splits them into two groups: X_1 and X_2.
- The median y-coordinates of the vertices of X_1 and X_2 are found along the y-axis and split separately into two sub-groups each: XY_{11}, XY_{12}, XY_{21}, and XY_{22}.
- Each of the four groups from the y-cut median is found along the z-axis, and split into two group each. This results in eight total sub-groups for the original box.

The bounding box hierarchy computed by this algorithm is applied recursively. Its termination condition is the same as that of the octree's. When a bounding box has vertices below a threshold, the cutting procedure stops and faces are introduced at these leaf boxes. Again the faces are trimmed against the sides of the leaf boxes as in the octree's construction or the faces are divided evenly among the boxes. This tree is more expensive to construct because the cuts require finding the median of the vertices.

A total hierarchy for each polytope is constructed by building sub-hierarchies for each hull feature subset and concavity feature subset. We have experimented with different

configurations for these hierarchies by varying the number of cuts along the axes for both types of hierarchies.

As the objects move, the bounding box hierarchy has to be updated. Each bounding box examines its children and finds the minimum and maximum coordinates of its children along each axis and uses them to determine its dimensions. The operation is repeated recursively up the tree until the root node is reached. Several optimizations are used for performance improvement. The first is to make the boxes containing the faces large enough to contain the faces at any orientation. Thus, updating the hierarchy amounts translating the box coordinates. We have implemented this method for the bounding boxes containing the convex hulls of the objects with excellent results [CLMP95]. Another approach is to pre-compute the hierarchy dimensions at common orientations and interpolate between these.

4.6 Determining Contact Regions between Polytopes

When the convex hulls of two polytopes are determined to be colliding based on the penetration algorithm, we enter the third stage of the algorithm. The collision between the objects' the convex hulls has three cases as illustrated in Fig. 10:

1. The collision is between hull feature subsets on both the objects. This is a "real" collision, and the objects are not permitted to penetrate any further because the objects are physically touching each other.

2. The collision is between a cap of one object and a convex feature subset of the other object. We want to check if the convex feature subset collides with any of the faces underneath the cap belonging to the concavity.

3. The collision is between two caps. We want to determine if any faces underneath both caps collide.

There can be various combinations of these cases. Determining the interpenetration volume of the convex hulls delineates which caps (and the features underneath them) and which hull features from both objects are involved in the collision. A optimal linear time convex polytope penetration algorithm is described in [Cha89]. The algorithm pre-computes a hierarchy of simplified representations for each polytope (the Dobkin-Kirkpatrick hierarchy [DK85]) and traces the intersection through this hierarchy. The pre-processing and the algorithm are quite complex, so we present two simpler methods that based on coherence.

When two convex hulls A and B interpenetrate, then at least one feature of $a \in A$ is contained within the pseudo internal Voronoi region of B and vice-versa for feature $b \in B$. These two features can be used as the starting basis to construct the complete interpenetrated volume with a depth-first-search technique. Each adjacent feature of a is tested against the pseudo Voronoi cell containing a and eventually the pseudo Voronoi cell containing it is determined. This is repeated recursively for adjacent features of a and b. After the initial cost of constructing this interpenetration volume, the update time for each

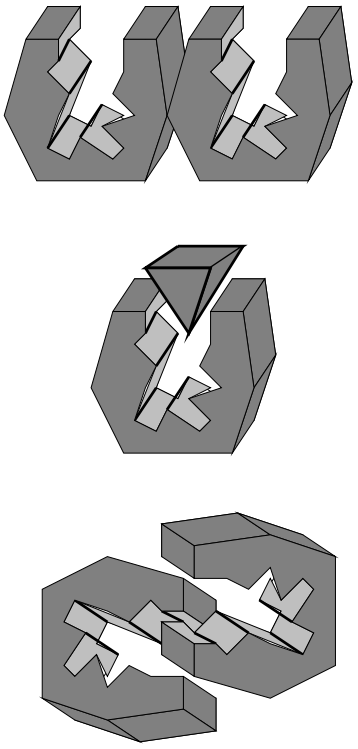

Figure 10: Collisions involving different feature sets.

feature, as each of these features move slightly from frame to frame, is an expected constant due to coherence (for each feature). Moreover, only the penetrating features on both objects are considered so the overall cost is linear in the number of features penetrating. The total expected running time using the internal penetration walk is $O(n)$.

There is another simple approach to compute the penetrating region. The initial interpenetration point a can used to construct the intersection volume in another way. We designate a point $p = (p_x, p_y, p_z)$ that is an ϵ offset from a and is internal to both convex hulls to be the origin. The vertices of the convex hulls and normals of each face are recomputed relative to this origin.

Each convex hull can be thought of as the intersection of the half-spaces described by the faces. The half space of a face f is described by the inequality:

$n_x(f)x + n_y(f)y + n_z(f)z \leq d(f)$

where $n_x(f), n_y(f), n_z(f)$, and $d(f)$ are the normal and skew away from the origin of the face. We can normalize $d(f)$ to be 1. We now define a dual transform T such that the face planes of the convex hull are transformed to points by dualizing their normals. Vertex points of the convex hull are dualized to face planes.

A point $q = (q_x, q_y, q_z)$ originally a distance $d = \sqrt{q_x^2 + q_y^2 + q_z^2}$ under the transformation becomes a plane $1/d$ from the origin on the other side [DF78]. A similar transformation occurs for face planes. Because we have defined a new origin, internal to both convex hulls, the closer faces move further away and the further ones closer with this transformation. To

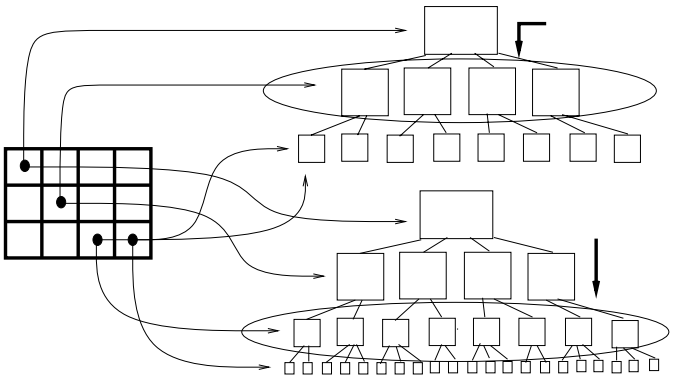

Figure 11: Tree Hierarchies.

find the interpenetrating volume, we simply find the convex hull of the dualized face planes (which are points) and transform them back into face planes on the objects [DF78].

With this computation we now know which caps have been penetrated as well as which features of the hull features have penetrated. The hull and concavity features are in a pre-computed hierarchy, so we traverse the two hierarchies to determine the contact status between the two objects. As the objects move, this overlapping convex volume changes little from frame to frame, so the convex hull computation can use the previous convex hull vertices as the starting basis for faster results.

Once the algorithm detects interpenetration, it applies hierarchical *Sweep and Prune* between the object representations to determine precise contacts in successive instances. It uses the inital penetration detection algorithm's features as the starting features and traverses upwards in object hierarchies to see what extent the objects are overlapping. This process continues recursively until we reach a high enough level at which the hierarchies do not overlap. Then from this point, we traverse using downwards in both hierarchies to the face bounding box level to find the exact points of contact. The hierarchical *Sweep and Prune* is explained in the next section.

4.7 Hierarchical Sweep and Prune

Consider two object hierarchies of levels 3 and 4 as shown in the Fig. 11. Initially, we have discovered, via the penetration detection algorithm, that the root level bounding boxes object A and object B overlap. The children of these bounding boxes are sorted using the *Sweep and Prune* technique to determine if they too overlap. Sorting and determining children's overlap status is done recursively until we reach the leaf boxes which contain the faces of both hierarchies. If the hierarchies are of the same height, then we test the boxes at the levels $(1,1),(2,2),(3,3)$, etc successively. If the hierarchies are of different heights, then the last level of the shallower hierarchy is tested against the descending levels of the other hierarchy. For example, in Fig. 11 the bounding boxes of levels $(1,1),(2,2),(3,3)$, and $(3,4)$ are tested because the second hierarchy has a greater depth. The hierarchical sort manager keeps track of these level pairs between a pair of objects.

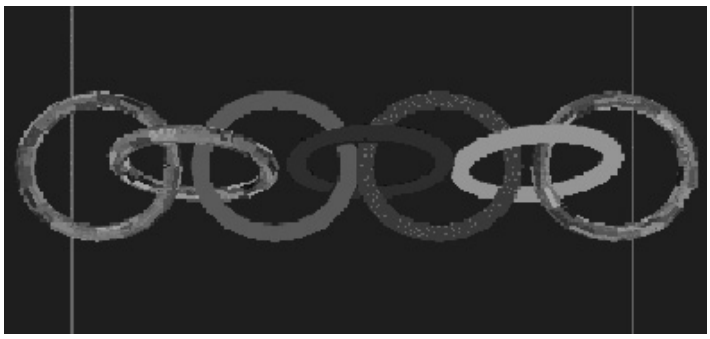

Figure 12: Chain interlocked tori.

When both hierarchies are at the leaf level, the bounding boxes enclosing faces are sorted and tested for overlaps. At this last level, we check to see if the the faces of the relevant leaf nodes are intersecting. An intersection consisting of one or more faces indicates a collision between the objects. The exact contact points are computed by finding the intersection of the polygons and passed to the application to plan an appropriate response.

The level by level sorting of the bounding box hierarchies is tricky to manage both in terms of efficiency and memory usage for objects composed of thousands of polygons. Multiple lists have to be sorted at each level since not all children overlap. Moreover, this information needs to be managed efficiently. We use a bounding box management scheme that allocates the space for the data once, but can have multiple pointers to it. So instead of allocating this data value at multiple levels in the hierarchy, it efficiently swaps pointers during the sort. It also allocates only as much memory per pair of objects as the number of features of the objects that are interacting (as opposed to allocating memory for all possible feature pairs for the objects). This is *critical* because an inefficient all pairs allocation quickly results in memory exhaustion for objects of thousands of polygons.

The data structure also keeps the data from each sorting pass to use on the next pass. The previous frame's sorted bounding box lists, which indicate the coordinates of objects' features, usually change little from frame to frame. Therefore, the lists in the current frame, with the updated coordinates, should require only a minimal amount of swapping. Exploiting this coherence is critical to the performance of our approach. We have implemented several simulations using this hierarchy data structure with promising results.

5 Applications

The Fig. 12 shows a still from one of the simulations we used to test our collision detection algorithm.

The tori simulation consists of seven interlocked tori (a toroidal chain). We polygonize each torus into 400 polygons and check for collision between the polyhedral approximations. After the polyhedral models overlap, a local algebraic solver checks for exact collision [LM95]. Each ring was initially given an arbitrary translation and rotation vector.

Tori	Tree Type	Levels	X-Y-Z Cuts	Iteration Time
2	None	N/A	N/A	0.045 secs.
2	Octree	1	2-2-2	0.014 secs.
2	OTV	1	2-2-2	0.012 secs.
8	Octree	2	3-2-2/2-2-2	0.046 secs.
8	OTV	2	3-2-2/2-2-2	0.038 secs.

Table 1: Performance of the collision detection algorithm on the toroidal chain

Their motions resulted in elastic collisions. The interlocked configuration generated many collisions involving features not on the convex hull. This problem was specifically chosen to illustrate that our algorithm performs well in difficult simulations.

We tried various configurations of the bounding box hierarchy with the tori chain: no hierarchy, octree tree variant (OTV) vs octree, and variable number of children per level. Table 5 is a synopsis of our results. These numbers do *not* include rendering time.

The first entry shows the time for a brute force collision scheme where all pairs of face bounding boxes were sorted. So for two interlocked tori, we applied an initial sweep and prune over all bounding boxes surrounding the faces of the two objects. Those overlapping were further tested for face-face intersections. The second and third entries constructed a one level hierarchy for each torus. Each torus was split into eight parts and the bounding boxes surrounding the faces were assigned to one of these eight boxes depending on whether they fit inside the boxes. The octree version split the faces uniformly whereas the octree variant partitioned the faces depending on vertex position. The time for two tori improved by a factor of three with the hierarchies. The third and fourth entries constructed a two level hierarchy of 12 children at the first level and then 8 children per node on the second level. These last two configurations were used on an interlocked chain of eight tori.

5.1 Application to Threaded Insertion

We have applied the contact determination algorithm for dynamic simulation of a threaded insertion. According to an earlier study by Nevins and Whitney [NW80], the insertion and tightening of threaded fasteners is one of the twelve most commonly assembled tasks. More recently, Nicolson and Fearing have presented the first algorithm and system for dynamic simulation of screw insertion [NF91]. In this section, we briefly describe the geometric model of the screw based on the formulation in [NF91].

A screw thread is a ridge of a thread profile, wrapped in a helical fashion about a cylinder. The pitch is the spatial period of the thread profile. The external screw thread is the thread of the bolt and the internal screw thread is that on a nut. The root of the profile is the smallest diameter and the crest is the largest. Based on these definitions, a rounded crest and root thread profile can be made with the following four variables:

Figure 13: A cross-threaded bolt and nut configuration [NF91]

p: pitch.

d: internal thread basic major diameter.

a: allowance ratio, $0 \leq a \leq 1$, where the actual allowance is $a * d$

r: root and crest radius.

The ratio $\frac{d}{p}$ determines if the bolt has fine or coarse threads. Typically $(\frac{d}{p})_{\text{fine}} = 2(\frac{d}{p})_{\text{coarse}}$. The allowance ratio gives the relative radius of the bolt with respect to the nut. A cross-section of the geometric model is shown in Figure 13.

The thread profile is parameterized as a function of position, t, and thread pitch, p. Given r and a, it can be represented as a bivariate piecewise function of t and p [NF91]. This thread profile is used to formulate the surface of the screw in space as a parametric function of two variables. Given the geometric description of the thread, the dynamic simulation requires precise determination of contact points between the nut and the bolt. Since the motion of the bolt can not be expressed as a closed form function of time, it is also very useful to know the exact distance between them. This helps in choosing the appropriate time steps. There are no general purpose analytic and efficient approaches known for computing the contact points between the bolt and the nut. Nicolson and Fearing [NF91] used a set of "critical points" on the thread profile at which contact is most likely to occur. This is based on approximating the bolt by a helix, whose radius is same as that at the crest of the bolt thread. Given the set of critical points, their algorithm checks whether these critical points lie on the surface of the nut. In their simulation, collision detection accounted for almost half of the simulation time and misses some contacts at times [Nic94].

We have applied our collision detection algorithm to a polygonal approximation of the bolt and nut. The polygonal approximation is specified to an ϵ tolerance of the original model. For our measurements $\epsilon = 0.04$ and as a result we obtain 154 polygons corresponding to each thread of the bolt and the nut. The top and the bottom of the bolt are modeled as cylinders and their polygonal approximation consists of 100 polygons each. For a bolt with T_b threads and a nut with T_n threads, this reduces to computing all the contacts between $200 + 154T_b$ polygons corresponding to the bolt and $154T_n$ polygons of the nut. The motion of the bolt is a cylindrical function specified by the controller. The algorithm at each instance is able to detect all possible contacts between the polygons. In case there

T_b (Bolt Threads)	T_n (Nut Threads)	Bolt Polygons	Nut Polygons	Coll. Det. Time
3	3	562	462	0.059 sec.
6	6	1024	924	0.099 sec.
9	9	1486	1386	0.191 sec.

Table 2: Performance of Collision Detection Algorithm on Threaded Insertion

are no contacts, it is able to compute the closest distance between the polygonal features. In Table 2, we have highlighted the average performance of the *collision detection* routines for the given simulation. All simulations were performed on an HP 735 with 64 MB of main memory.

At each instance the position of the nut is fixed and the bolt is undergoing rigid motion. The algorithm updates the positions of each vertex on the bolt and uses it to compute the necessary boundary boxes for the sweep and prune method. The overall running time of the contact determination algorithm increases linearly, despite the fact the number of possible interactions increases quadratically. As a result, we are able to accurately compute all the contacts efficiently.

6 Robustness Issues

All the algorithms described in this paper have been implemented in C. We use floating-point arithmetic. In practice we have encountered many degenerate geometric problems while testing our algorithm on many *real-world* models. We are listing a few of them here.

- The model may have coincident polygons. This affects the feature classification algorithm and at times the algorithm may miss collision.
- The model may have co-planar faces. In several parts of the overall algorithm, we compute the convex hull of an object. It is a well-known problem that taking the convex hull of co-planar faces (hence vertices) causes numerical problems [O'R94].
- The model may have T-vertices, where one edge intersects another edge somewhere in the middle. We add extra edges to remove the T-vertices.
- Problems in constructing of the bounding box hierarchy. While grouping the faces into boxes, we use orthogonal cutting planes which cut the existing faces. These cuts can result in long thin faces which combined with floating point arithmetic cause representation problems.

For many of these conditions, we wrote pre-processing code to remove or identify them. In several instances, we have introduced ϵ tolerance tests. However, it is rather difficult

to guarantee robustness based on these tests. Future work includes perturbation-based approaches to handle degeneracies.

7 Conclusions

We have presented an incremental algorithm for contact determination between general polyhedral solid models. Its running time is a function of the complexity of the collision status between the models. The algorithm efficiently and accurately computes all the contact points between moving objects. We have demonstrated it on simulations of high geometric complexity.

References

[AANJ94] A.Garica-Alonso, N.Serrano, and J.Flaquer. Solving the collision detection problem. *IEEE Computer Graphics and Applications*, 13(3):36–43, 1994.

[AP76] A.Appel and P.Will. Determining the three-dimensional convex hull of a polyhedron. *IBM Journal of Research and Development*, pages 590–600, 1976.

[Bar90] D. Baraff. Curved surfaces and coherence for non-penetrating rigid body simulation. *ACM Computer Graphics*, 24(4):19–28, 1990.

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

[BT92] C.L. Bajaj and T.Dey. Convex decomposition of polyhedra and robustness. *SIAM Journal of Computing*, 21:339–364, 1992.

[Bob89] J. E. Bobrow. A direct minimization approach for obtaining the distance between convex polyhedra. *International Journal of Robotics Research*, 8:65–76, 1989.

[Cam91] S. Cameron. Approximation hierarchies and s-bounds. In *Proceedings. Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 129–137, Austin, TX, 1991.

[Can86] J. F. Canny. Collision detection for moving polyhedra. *IEEE Trans. PAMI*, 8:pp. 200–209, 1986.

[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 pp. 591–596, 1986.

[CD87] B. Chazelle and D. P. Dobkin. Intersection of convex objects in two and three dimensions. *J. ACM*, 34:1–27, 1987.

[Cha84] B. Chazelle. Convex partitions of polyhedra: a lower bound and worst-case op timal algorithm. *SIAM J. Comput.*, 13:488–507, 1984.

[Cha89] B. Chazelle. An optimal algorithm for intersecting three-dimensional convex polyhedra. In *Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 586–591, 1989.

[CLMP95] J. Cohen, M. Lin, D. Manocha, and M. Ponamgi. I-collide: An interactive and exact collision detection system for large-scale environments. In *Proc. of ACM Interactive 3D Graphics Conference*, Monterey, CA, 1995, pp. 112–120.

[CP90] B. Chazelle and L. Palios. Triangulating a non-convex polytope. *Discrete Comput. Geom.*, 5:505–526, 1990.

[DF78] D.Muller and F.Preparata. Finding the intersection of two convex polyhedra. *Theoretical Computer Science*, 8(2):217–236, 1978.

[DK85] D. P. Dobkin and D. G. Kirkpatrick. A linear algorithm for determining the separation of convex pol yhedra. *J. Algorithms*, 6:381–392, 1985.

[Duf92] Tom Duff. Interval arithmetic and recursive subdivision for implicit functions and constructive solid geometry. *ACM Computer Graphics*, 26(2):131–139, 1992.

[DZ93] P. Dworkin and D. Zeltzer. A new model for efficient dynamics simulation. *Proceedings Eurographics workshop on animation and simulation*, pages 175–184, 1993.

[ea93] J. Snyder et. al. Interval methods for multi-point collisions between time dependent curved surfaces. In *Proceedings of ACM Siggraph*, pages 321–334, 1993.

[FKN80] H. Fuchs, Z. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. In *Proc. of ACM Siggraph*, volume 14, pages 124–133, 1980.

[GH89] E. G. Gilbert and S. M. Hong. A new algorithm for detecting the collision of moving objects. *IEEE Conference on Robotics and Automation*, pages pp. 8–14, 1989.

[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:pp. 193–203, 1988.

[GO94] E.G. Gilbert and C.J. Ong. New distances for the separation and penetration of objects. In *Proceedings of International Conference on Robotics and Automation*, pages 579–586, 1994.

[HBZ90] B. V. Herzen, A. H. Barr, and H. R. Zatz. Geometric collisions for time-dependent parametric surfaces. *Computer Graphics*, 24(4):39–48, 1990.

[HH87] C.M. Hoffmann and J. E. Hopcroft. Simulation of physical systems from geometric models. *IEEE Journal on Robotics and Automation*, 3(3):194–206, 1987.

[Hop88] J.E. Hopcroft. Electronic prototyping. *IEEE Transactions on Aerospace and Electronic Systems*, 24(2):663–667, 1988.

[Hub93] P. M. Hubbard. Interactive collision detection. In *Proceedings of IEEE Symposium on Research Frontiers in Virtual Reality*, October 1993.

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

[Lin93] M.C. Lin. *Efficient Collision Detection for Animation and Robotics*. PhD thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, December 1993.

[LM95] M.C. Lin and Dinesh Manocha. Efficient contact determination between geometric models. *International Journal of Computational Geometry and Applications*, 1995. To appear.

[Lou66] P. Loutrel. Determination of hidden edges in polyhedral figures. Technical Report NYU-3-12-1966, Department of Computer Science, New York University, 1966.

[Meg83] N. Megiddo. Linear-time algorithms for linear programming in r^3 and related problems. *SIAM J. Computing*, 12:pp. 759–776, 1983.

[MW88] M. Moore and J. Wilhelms. Collision detection and response for computer animation. *Computer Graphics*, 22(4):289–298, 1988.

[NAT90] B. Naylor, J. Amanatides, and W. Thibault. Merging bsp trees yield polyhedral modeling results. In *Proc. of ACM Siggraph*, pages 115–124, 1990.

[Nay92] B. Naylor. Interactive solid geometry via partitioing trees. In *Proc. of Graphics Interface*, pages 11–18, 1992.

[NF91] E. Nicolson and R. Fearing. Dynamic simulation of a part-mating problem: Threaded fastener insertion. In *Proceedings, IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 30–37, 1991.

[Nic94] E. Nicolson. Personal Communication, 1994.

[NW80] J.L. Nevins and D.E. Whitney. Assembly research. In *Factory Automation*, volume 2, Maidenhead, England, 1980.

[O'R94] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1994.

[PS85] F.P. Preparata and M. I. Shamos. *Computational Geometry.* Springer-Verlag, New York, 1985.

[Qui94] S. Quinlan. Efficient distance computation between non-convex objects. In *Proceedings of International Conference on Robotics and Automation*, pages 3324–3329, 1994.

[Sei90] R. Seidel. Linear programming and convex hulls made easy. In *Proc. 6th Ann. ACM Conf. on Computational Geometry*, pages 211–215, Berkeley, California, 1990.

[THK91] F. Thomas, J. Hamlin, and R.B. Kelley. Spherical object representation and fast distance computation for robotic applications. In *Proceedings of International Conference on Robotics and Automation*, 1991.

[TT94] F. Thomas and C. Torras. Interference detection between non-convex polyhedra revisited with a practical aim. In *Proceedings of International Conference on Robotics and Automation*, pages 587–594, 1994.

[Tur89] G. Turk. Interactive collision detection for molecular graphics. Master's thesis, Computer Science Department, University of North Carolina at Chapel Hill, 1989.

[WG91] W.Bouma and G.Vanecek. Collision detection and analysis in a physically based simulation. *Proceedings Eurographics workshop on animation and simulation*, pages 191–203, 1991.

[WG93] W.Bouma and G.Vanecek. Modeling contacts in a physically based simulation. *Second Symposium on Solid Modeling and Applications*, pages 409–419, 1993.

[WK93] D.Waco and Y.Kim. Geometric Reasoning for Machining Features Using Convex Decomposition *Second Symposium on Solid Modeling and Applications*, pages 323–332, 1993.