Collision Detection: Algorithms and Applications

Ming C. Lin, *U.S. Army Research Office and University of North Carolina, Chapel Hill, NC, USA*
Dinesh Manocha, *University of North Carolina, Chapel Hill, NC, USA*
Jon Cohen, *University of North Carolina, Chapel Hill, NC, USA*
Stefan Gottschalk, *University of North Carolina, Chapel Hill, NC, USA*
http://www.cs.unc.edu/~geom/collide.html

Fast and accurate collision detection between general geometric models is a fundamental problem in modeling, robotics, manufacturing and computer-simulated environments. Most of the earlier algorithm are either restricted to a class of geometric models, say convex polytopes, or are not fast enough for practical applications. We present an efficient and accurate algorithm for collision detection between general polygonal models in dynamic environments. The algorithm makes use of hierarchical representations along with frame to frame coherence to rapidly detect collisions. It is robust and has been implemented as part of public domain packages. In practice, it can accurately detect all the contacts between large complex geometries composed of hundreds of thousands of polygons at interactive rates.

1 Introduction

Collision detection is a fundamental problem in robotics, computer animation, physically-based modeling, molecular modeling and computer simulated environments. In these applications, an object's motion is constrained by collisions with other objects and by other dynamic constraints. The problem has been well studied in the literature.

A realistic 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. A fundamental component of such a system is 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 simulation, namely the collision phase, can be separated into three parts: collision detection, contact area determination, and collision response. In this paper, we address the first two elements by presenting general a purpose collision detection and contact area determination algorithm for simulations. The collision response is application dependent. The algorithm 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 polygonal objects, but also large environments consisting of hundreds of moving parts, such as those encountered in the manufacturing plants. Furthermore, we do not 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).

Given the geometric models, the algorithm precomputes the convex hull and a hierarchical representation of each model in terms of oriented bounding boxes. At runtime, it uses tight fitting axis-aligned bounding boxes to pair down the number of object pair interactions to only those pairs within *close proximity* [12]. For each pair of objects whose bounding boxes overlap, the algorithm checks whether their convex hulls are intersecting based on the closest feature pairs [22]. Finally for each object pair whose convex hulls overlap, it makes use of oriented bounding box hierarchy (OBBTree) to check for actual contact [18].

Organization: The rest of the paper is organized as follows: Section 2 reviews some of the previous work in collision detection. Section 3 outlines the algorithm for pruning the number of object pairs. We briefly describe the closest feature and contact determination algorithms in Section 4. Finally, we describe the implementation and performance on different applications in Section 5

2 Previous Work

Collision detection has been extensively studied in CAD, 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.

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 [8, 9]. However, in path planners based on potential field methods, collision detection and distance computation are still considered as major bottlenecks [21, 10].

Most computational geometry literature 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 [11, 13]. 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 [3, 23]. Several solutions based on this idea of coherence have been proposed in [22]. Approaches that combine collision response with detection can be found in [3, 33, 25]. The methods in [32, 33] make use of the 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. In a general simulation environment, 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 [7, 12, 16]. 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.

As for curved models, algorithms based on interval arithmetic for collision detection are described in [15, 17]. 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 [24].

In many CAD applications, the input models are given as collections of polygons with no topology information. Such models are also known as 'polygon soups' and their boundaries may have cracks, T-joints, or may have non-manifold geometry. In general, no robust techniques are known for cleaning such models. Many of the algorithms described are not applicable to such models. Rather techniques based on hierarchical bounding volumes and and spatial decomposition are used on such models. Typical examples of bounding volumes include axis-aligned boxes (of which cubes are a special case) and spheres, and they are chosen for to the simplicity of finding collision between two such volumes. Hierarchical structures used for collision detection include cone trees, k-d trees and octrees [31], sphere trees [20, 30], R-trees and their variants [5], trees based on S-bounds [7] etc. Other spatial representations are based on BSP's [27] and its extensions to multi-space partitions [32], spatial representations based on space-time bounds or four-dimensional testing [1, 6, 9, 20] and many more. All of these hierarchical methods do very well in performing "rejection tests", whenever two objects are far apart. However, when the two objects are in close proximity and can have multiple contacts, these algorithms either use subdivision techniques or check very large number of bounding volume pairs for potential contacts. In such cases, their performance slows down considerably and they become a major bottleneck in the simulation, as stated in [19].

3 Collision Detection between Multiple Moving Objects

We review our previous algorithm for multiple moving convex polytopes in complex environments. Coherence combined with incremental computation is a major theme of our algorithms. By exploiting coherence, we are able to incrementally trim down the number of pairwise object pairs and feature tests involved in each iteration.

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 spatial relationship does not change much between frames.

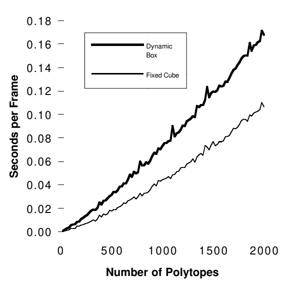
For a configuration of N objects, the worst case running time for any collision detection algorithm is $O(N^2)$ where N is the number of objects. However, evidence suggests that these cases rarely occur in simulations [3, 12, 22, 28]. So our algorithm uses a *Sweep and Prune* technique to eliminate testing 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.

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.

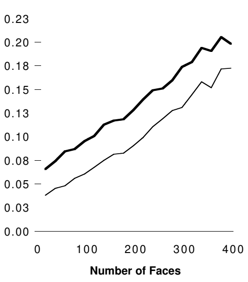
Sorting the bounding boxes surrounding the objects is the key to our *Sweep and Prune* approach [12]. It is not intuitively obvious how to sort bounding boxes in 3-space to determine overlaps. We use a *dimension reduction* approach. If two bounding boxes collide in 3-D, then their orthogonal projections 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 in all three dimensions.

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, updating only the interval endpoints. In environments where the objects make relatively small movements between frames, the lists will be nearly sorted, so we can re-sort using *insertion sort* in expected $O(N)$ time [26, 3]. Graphs 1 – 6 are timings taken from a multi-object simulation where we compare the performance of using fixed versus dynamic sized boxes [12]. Parameters such as the number of objects, the polygonal complexity of the objects, the velocity of the objects, etc. were varied as the graphs show.

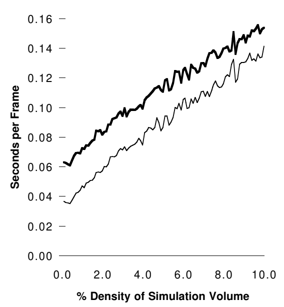
Graph 1



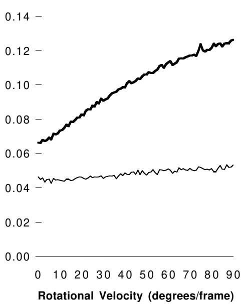
Graph 2



Graph 3



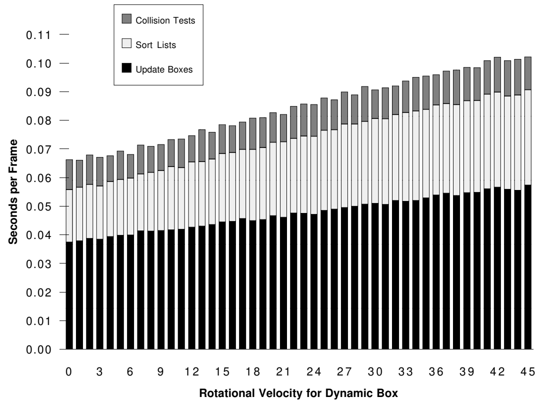
Graph 4



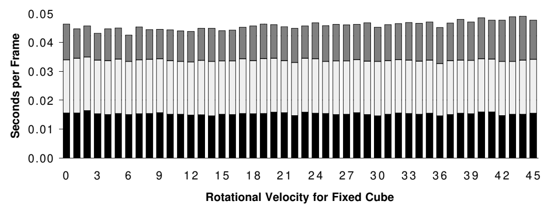
4 Exact Collision Detection

Given two objects in close proximity, the algorithm initially checks whether their convex hulls are overlapping. It incrementally computes their closest feature

Graph 5



Graph 6



pairs and check for overlap. If the convex hulls are intersecting, it checks whether the overlapping features belong to the original model or are introduced by the convex hull computation. Eventually it makes use of hierarchy of oriented bounding boxes to check for exact contact.

4.1 Collision Detection between Convex Polytopes

The algorithm computes the convex hull of all objects as part of pre-processing. Furthermore, it classifies the feature of convex hulls into *red* and *green* features. The red features correspond to the features of the original model and the green features are introduced by the convex hull computation.

We use the algorithm described in [22] 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 at large discrete steps (e.g. 180 degrees of rotation per step).

4.1.1 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 [29].

The Voronoi regions form a partition of the space outside the polytope, and they 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 [22].

4.1.2 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. 1.

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

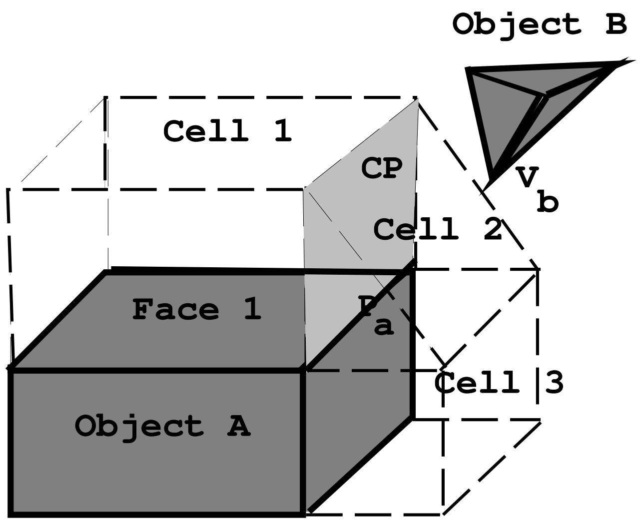


Figure 1: *A walk across Voronoi cells.*

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 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.

4.1.3 Penetration Detection for Convex Polytopes

The key to detecting penetrations lies in partitioning the *interior* as well as the exterior of the convex polytope. 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 [29]. To detect a penetration – as opposed to knowing *all* the closest features – it is unnecessary to construct the exact internal Voronoi regions.

Rather we use an approximation, labeled as pseudo-internal Voronoi region. It is calculated by first computing the centroid of each convex polytope – the weighted average of all vertices. Then a hyperplane from each edge is extended towards the centroid. The extended hyperplane tapers to a point, forming a pyramid-type cone over each face. 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 (as shown in Fig. 2). 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. A detailed discussion is presented in [28]

4.1.4 Feature Classification

The algorithm highlighted above returns all pairs of overlapping features between the convex hulls. The resulting feature pairs can be classified into:

- Red-red feature overlap (as shown in Fig. 3(a)): This corresponds to an actual collision between the original models.
- Red-green feature overlap (as shown in Fig. 3(b)): This may or may not correspond to a collision. The algorithm presented in the next section is used to check for exact contact.
- Green-green feature overlap (as shown in Fig. 3(c)): Same as above. The algorithm presented in the next section is used to check for exact contact.

4.2 Exact Contact Determination

In this section, we present a robust, efficient and general purpose algorithm to compute all contacts between geometric models composed of polygons. The algorithms computes a hierarchical representation using *oriented bounding boxes (OBBs)*. An OBB is a rectangular bounding box at an arbitrary orientation in 3-space. The resulting hierarchical structure is referred to as an OBBTree. The idea of using OBBs is not new and many researchers have used them extensively to speed up ray tracing and interference detection computations [2]. In this paper, we briefly describe the algorithms for computing tight-fitting OBBs and checking them for overlap. More details are given in [18].

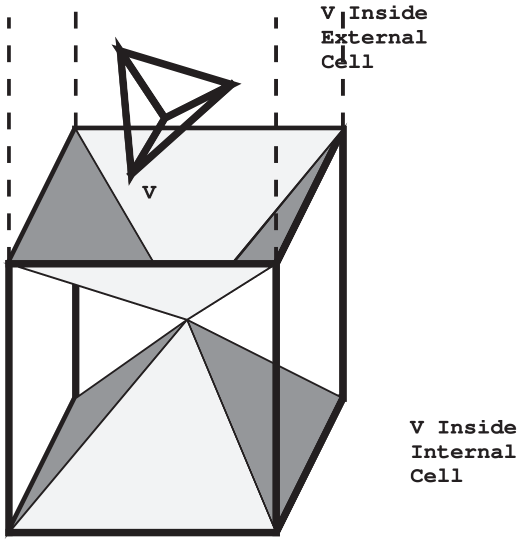


Figure 2: *Walk from external to internal Voronoi regions.*

4.2.1 Building an OBBTree

In this section we describe algorithms for building an OBBTree. The tree construction has two components: first is the placement of a tight fitting OBB around a collection of polygons, and second is the grouping of nested OBB's into a tree hierarchy.

We want to approximate the collection of polygons with an OBB of similar dimensions and orientation. We triangulate all polygons composed of more than three edges. The OBB computation algorithm makes use of first and second order statistics summarizing the vertex coordinates. They are the mean, μ, and the covariance matrix, $\mathbf{C}$, respectively [14]. If the vertices of the i'th triangle are the points $\mathbf{p}^i$, $\mathbf{q}^i$, and $\mathbf{r}^i$, then the mean and covariance matrix can be expressed in vector notation as:

$$\mu = \frac{1}{3n}\sum_{i=0}^{n}(\mathbf{p}^i + \mathbf{q}^i + \mathbf{r}^i),$$

$$\mathbf{C}_{jk} = \frac{1}{3n}\sum_{i=0}^{n}(\overline{\mathbf{p}}^i_j\overline{\mathbf{p}}^i_k + \overline{\mathbf{q}}^i_j\overline{\mathbf{q}}^i_k + \overline{\mathbf{r}}^i_j\overline{\mathbf{r}}^i_k), \qquad 1 \le j,k \le 3$$

where n is the number of triangles, $\overline{\mathbf{p}}^i = \mathbf{p}^i - \mu$, $\overline{\mathbf{q}}^i = \mathbf{q}^i - \mu$, and $\overline{\mathbf{r}}^i = \mathbf{r}^i - \mu$. Each of them is a 3×1 vector, e.g. $\overline{\mathbf{p}}^i = (\overline{\mathbf{p}}^i_1, \overline{\mathbf{p}}^i_2, \overline{\mathbf{p}}^i_3)^T$ and $\mathbf{C}_{jk}$ are the elements of the 3 by 3 covariance matrix.

The eigenvectors of a symmetric matrix, such as $\mathbf{C}$, are mutually orthogonal. After normalizing them, they are used as a basis. We find the extremal vertices along each axis of this basis, and size the bounding box, oriented with the basis vectors, to bound those extremal vertices. Two of the three eigenvectors of the covariance matrix are the axes of maximum and of minimum variance, so they will tend to align the box with the geometry of a tube or a flat surface patch.

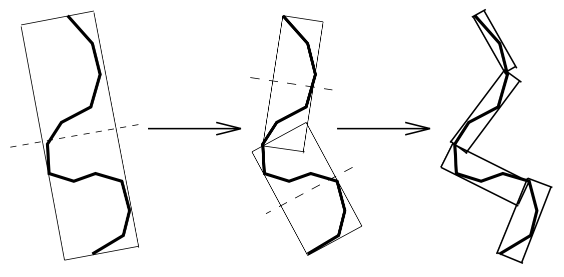


Figure 4: Building the OBBTree: recursively partition the bounded polygons and bound the resulting groups.

The basic failing of the above approach is that vertices on the interior of the model, which ought not influence the selection of a bounding box placement, can have an arbitrary impact on the eigenvectors. For example, a small but very dense planar patch of vertices

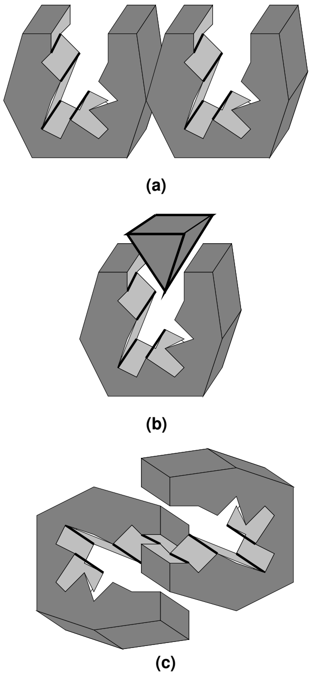


Figure 3: Feature classification based on the overlap between convex hulls

in the interior of the model can cause the bounding box to align with it.

We improve the algorithm by using the convex hull of the vertices of the triangles. The convex hull is the smallest convex set containing all the points and efficient algorithms of $O(n \lg n)$ complexity and their robust implementations are available as public domain packages [4]. This is an improvement, but still suffers from a similar sampling problem: a small but very dense collection of nearly collinear vertices on the convex hull can cause the bounding box to align with that collection.

Given an algorithm to compute tight-fitting OBBs around a group of polygons, we need to represent them hierarchically. Most methods for building hierarchies fall into two categories: bottom-up and top-down. Bottom-up methods begin with a bounding volume for each polygon and merge volumes into larger volumes until the tree is complete. Top-down methods begin with a group of all polygons, and recursively subdivide until all leaf nodes are indivisible. In our current implementation, we have used a simple top-down approach.

Our subdivision rule is to split the longest axis of a box with a plane orthogonal to one of its axes, partitioning the polygons according to which side of the plane their center point lies on (a 2-D analog is shown in Figure 4). The subdivision coordinate along that axis was chosen to be that of the mean point, μ, of the vertices. If the longest axis cannot not be subdivided, the second longest axis is chosen. Otherwise, the shortest one is used. If the group of polygons cannot be partitioned along any axis by this criterion, then the group is considered indivisible.

4.2.2 Fast Overlap Test for OBBs

Given OBBTrees of two objects, the interference algorithm typically spends most of its time testing pairs of OBBs for overlap. A simple algorithm for testing the overlap status for two OBB's performs 144 edge-face tests. In practice, it is an expensive test. OBBs are convex polytopes and therefore, algorithms based on linear programming and closest features computation can be applied to check for overlap. However, they are relatively expensive.

One trivial test for disjointness is to project the boxes onto some axis (not necessarily a coordinate axis) in space. This is an 'axial projection.' Under this projection, each box forms an interval on the axis. If the intervals don't overlap, then the axis is called a 'separating axis' for the boxes, and the boxes must then be disjoint. If the intervals do overlap, then the boxes may or may not be disjoint – further tests may be required. We make use of the *separating axis theorem* presented in [18] to check for overlaps. According to it, two convex polytopes in 3-D are disjoint iff there exists a separating axis orthogonal to a face of either polytope or orthogonal to an edge from each polytope. Each box has 3 unique face orientations, and 3 unique edge directions. This leads to 15 potential separating axes to test (3 faces from one box, 3 faces from the other box, and 9 pairwise combinations of edges). If the polytopes are disjoint, then a separating axis exists, and one of the 15 axes mentioned above will be a separating axis. If the polytopes are overlapping, then clearly no separating axis exists. So, testing the 15 given axes is a sufficient test for determining overlap status of two OBBs.

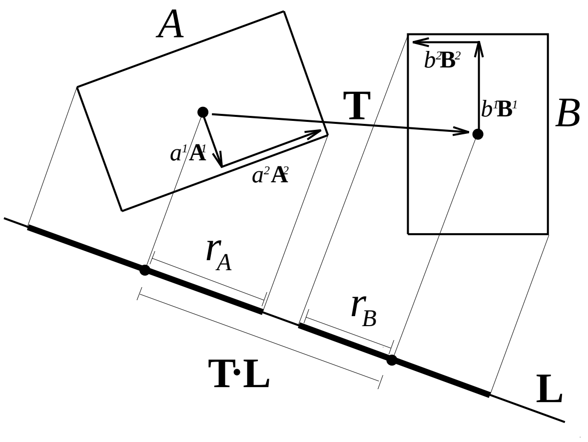


Figure 5: $\vec{L}$ is a separating axis for OBBs A and B because A and B become disjoint intervals under projection onto $\vec{L}$.

To perform the test, our strategy is to project the centers of the boxes onto the axis, and also to compute the radii of the intervals. If the distance between the box centers as projected onto the axis is greater than the sum of the radii, then the intervals (and the boxes as well) are disjoint. This is shown in 2D in Fig. 5. In practice, this corresponds to at most 200 arithmetic operations in the worst case [18]. Due to early exit (when the boxes are not overlapping), the algorithm takes about half the operations in practice. This algorithm for overlap detection between OBBs is about one order of magnitude faster than previous algorithms and implementations to check for overlap between OBBs.

5 Implementation and Performance

All these algorithms have been implemented and available as part of general purpose public domain packages. These are:

I_COLLIDE collision detection package available at **http://www.cs.unc.edu/~geom/I_COLLIDE.html**. It contains routines for the sweep and prune as well as the closest distance pair algorithm for convex polytopes. This package is applicable to environments, which can be described as union of convex polytopes. It has been widely used for dynamic simulation, architecture walkthrough and other applications.

RAPID interference detection package available at **http://www.cs.unc.edu/~geom/OBB/OBBT.html**. It contains routines for building the OBBTree data structure and fast overlap tests between two OBBTrees. It has been used for virtual prototyping and simulation-based design applications. More details on their performance and robustness issues are given in [12, 18].

In practice, the algorithms based on OBBs asymptotically perform much better than hierarchies based on sphere trees or axis-aligned bounding boxes (like Octrees). The I_COLLIDE routines are able to compute all contacts between environments composed of hundred of convex polytopes at interactive rates (about 1/20 of a second). The OBBTree based interference detection algorithm (available as part of RAPID) has been applied to two complex synthetic environments to demonstrate its efficiency (as highlighted in Table 1). These figures are for an SGI Reality Engine (90 MHz R8000 CPU, 512 MB).

A simple dynamics engine exercised the collision detection system. At each time step, the contact polygons were found by the collision detection algorithm, an impulse was applied to the object at each contact before advancing the clock.

In the first scenario, the pipes model was used as both the environment and the dynamic object, as

Scenario	Pipes	Torus
Environ Size	143690 pgns	98000 pgns
Object Size	143690 pgns	20000 pgns
Num of Steps	4008	1298
Num of Contacts	23905	2266
Num of Box-Box Tests	1704187	1055559
Num of Tri-Tri Tests	71589	7069
Time	16.9 secs	8.9 secs
Ave. Int. Detec. Time	**4.2 msecs**	**6.9 msecs**
Ave. Time per Box Test	7.9 usecs	7.3 usecs
Ave. Contacts per Step	6.0	1.7

Table 1: *Timings for simulations*

shown in Fig. 7. Both object and environment contain 140,000 polygons. The object is 15 times smaller in size than the environment. We simulated a gravitational field directed toward the center of the large cube of pipes, and permitted the smaller cube to fall inward, tumbling and bouncing. Its path contained 4008 discrete positions, and required 16.9 seconds to determine all 23905 contacts along the path. This is a challenging scenario because the smaller object is entirely embedded within the larger model. The models contain long thin triangles in the straight segments of the pipes, which cannot be efficiently approximated by sphere trees, octrees, and trees composed of axis-aligned bounding boxes , in general. It has no obvious groups or clusters, which are typically used by spatial partitioning algorithms like BSP's.

The other scenario has a complex wrinkled torus encircling a stalagmite in a dimpled, toothed landscape. Different steps from this simulation are shown in Fig. 8. The spikes in the landscape prevent large bounding boxes from touching the floor of the landscape, while the dimples provide numerous shallow concavities into which an object can enter. Likewise, the wrinkles and the twisting of the torus makes it impractical to decompose into convex polytopes, and difficult to efficiently apply bounding volumes. The wrinkled torus and the environment are also smooth enough to come into parallel close proximity, increasing the number of bounding volume overlap tests. Notice that the average number of box tests per step for the torus scenario is almost twice that of the pipes, even though the number of contacts is much lower.

We have also applied the RAPID library to detect collision between a moving torpedo on a pivot model (as shown in Fig. 6). These are parts of a torpedo storage and handling room of a submarine. The torpedo model is 4780 triangles. The pivot structure has 44921 triangles. There are multiple contacts along the length of the torpedo as it rests among the rollers. A typical collision query time for the scenario shown in Fig. 6 is 100 ms on a 200MHz R4400 CPU, 2GB SGI Reality Engine.

6 Acknowledgements

Thanks to Greg Angelini, Jim Boudreaux, and Ken Fast at Electric Boat for the model of torpedo storage and handling room. We would like to thank John Canny, Brian Mirtich and Krish Ponamgi for interesting discussions. Brian Mirtich and Krish Ponamgi also helped in the implementation of I_COLLIDE library. This research has been supported in part by a Sloan foundation fellowship, ARO Contract P-34982-MA, DARPA contract DABT63-93-C-0048, NSF grant CCR-9319957, NSF Grant CCR-9625217, ONR contract N00014-94-1-0738, NSF/ARPA Science and Technology Center for Computer Graphics & Scientific Visualization and NSF Prime contract No. 8920219.

References

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

[2] J. Arvo and D. Kirk. A survey of ray tracing acceleration techniques. In *An Introduction to Ray Tracing*, pages 201–262, 1989.

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

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

[5] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. *Proc.*

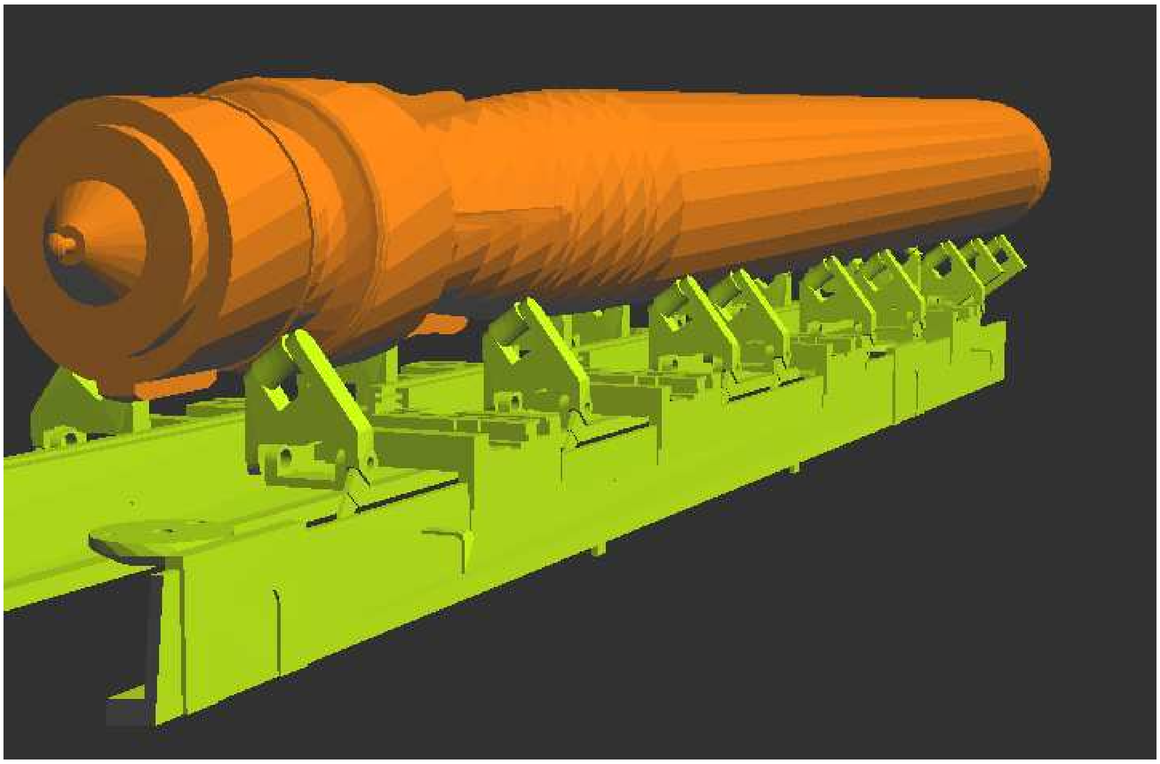


Figure 6: Interactive Interference Detection for a Torpedo (shown on the top) on a Pivot Structure – Torpedo has 4780 triangles; Pivot has 44921 triangles; Average time to perform collision query: 100 msec on SGI Reality Engine with 200MHz R4400 CPU

SIGMOD Conf. on Management of Data, pages 322–331, 1990.

[6] S. Cameron. Collision detection by four-dimensional intersection testing. *Proceedings of International Conference on Robotics and Automation*, pages 291–302, 1990.

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

[8] 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.

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

[10] H. Chang and T. Li. Assembly maintainability study with motion planning. In *Proceedings of International Conference on Robotics and Automation*, 1995.

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

[12] 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*, pages 189–196, 1995.

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

[14] R.O. Duda and P.E. Hart. *Pattern Classification and Scene Analysis*. John Wiley and Sons, 1973.

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

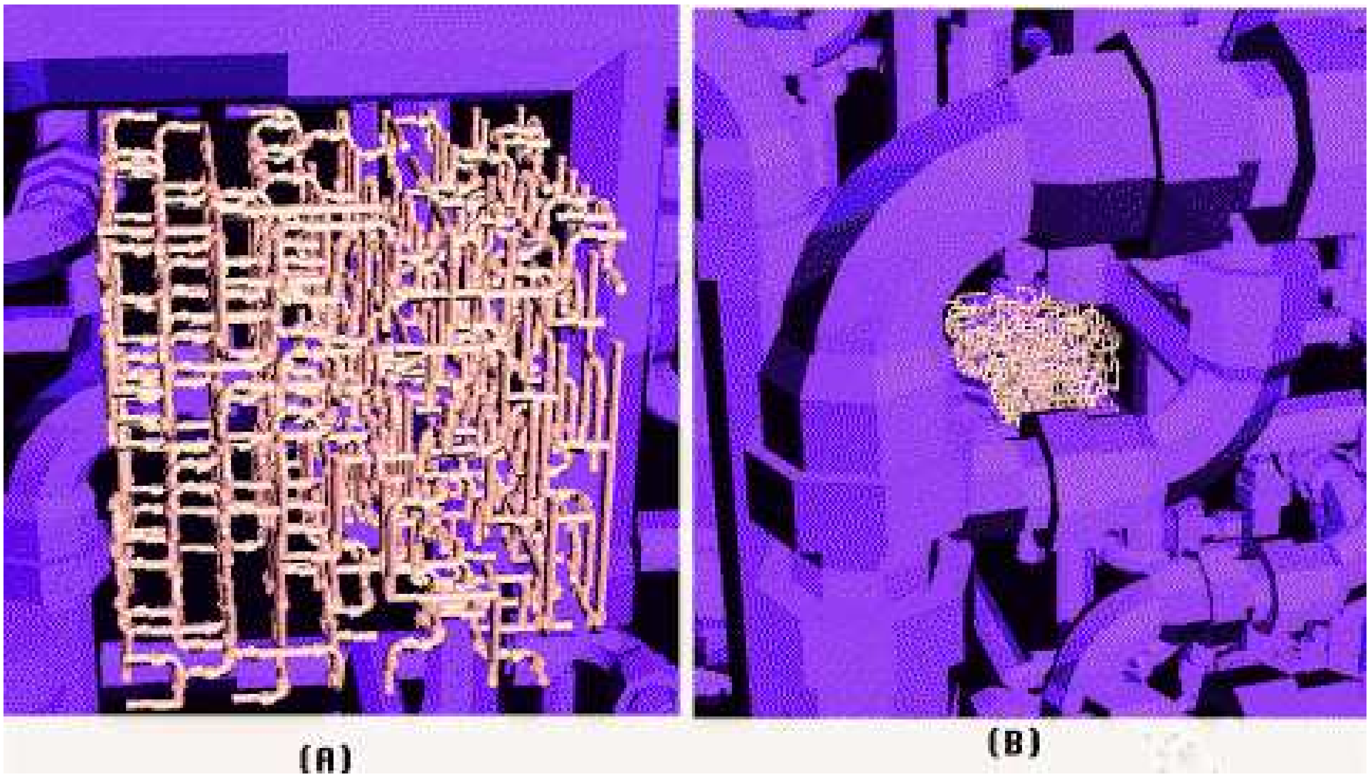


Figure 7: Interactive Interference Detection on Complex Interweaving Pipeline : 140,000 polygons each; Average time to perform collision query: 4.2 msec on SGI Reality Engine with 90MHz R8000 CPU

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

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

[18] S. Gottschalk, M. Lin, and D. Manocha. Obb-tree: A hierarchical structure for rapid interference detection. To Appear in Proc. of ACM Siggraph'96, 1996.

[19] J. K. Hahn. Realistic animation of rigid bodies. *Computer Graphics*, 22(4):pp. 299–308, 1988.

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

[21] J.C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, 1991.

[22] 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.

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

[24] M.C. Lin and Dinesh Manocha. Efficient contact determination between geometric models. *International Journal of Computational Geometry and Applications*, 1996. To appear.

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

[26] M.Shamos and D.Hoey. Geometric intersection problems. *Proc. 17th An. IEEE Symp. Found. on Comput. Science*, pages 208–215, 1976.

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

[28] M. Ponamgi, D. Manocha, and M. Lin. Incremental algorithms for collision detection between general solid models. In *Proc. of ACM/Siggraph Symposium on Solid Modeling*, pages 293–304, 1995.

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

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

[31] H. Samet. *Spatial Data Structures: Quadtree, Octrees and Other Hierarchical Methods*. Addison Wesley, 1989.

[32] 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.

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

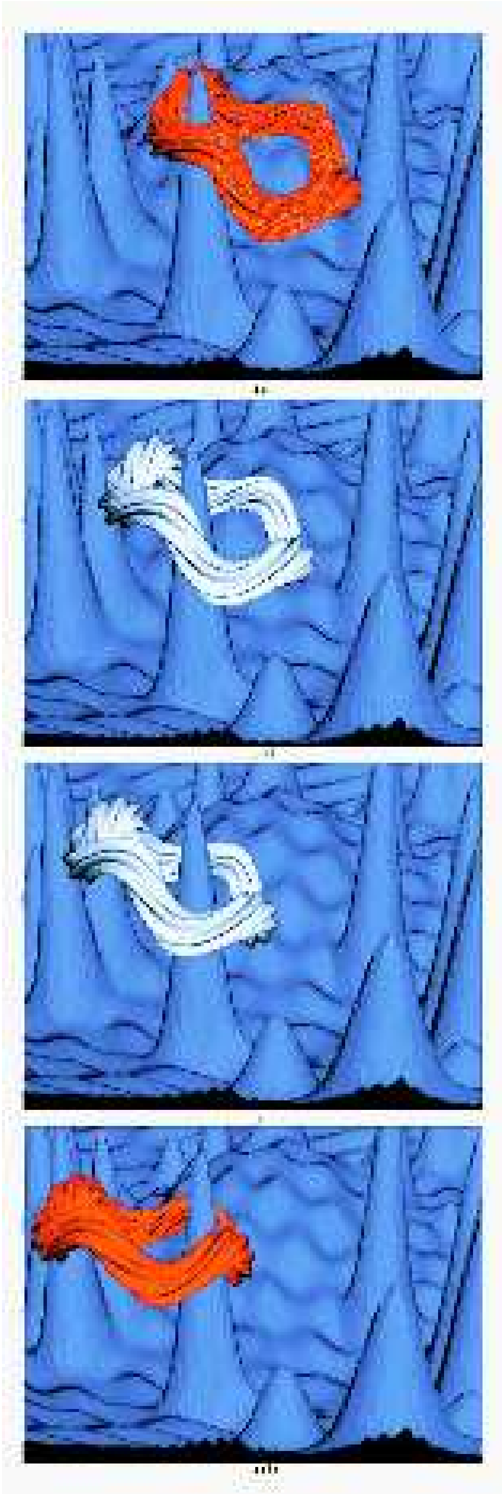


Figure 8: Interactive Interference Detection for a Complex Torus – Torus has 20000 polygons; Environment has 98000 polygons; Average time to perform collision detection: 6.9 msec on SGI Reality Engine