Efficient Distance Computation between Non-Convex Objects

Sean Quinlan

Robotics Laboratory
Computer Science Department
Stanford University

Abstract

This paper describes an efficient algorithm for computing the distance between non-convex objects. Objects are modeled as the union of a set of convex components. From this model we construct a hierarchical bounding representation based on spheres. The distance between objects is determined by computing the distance between pairs of convex components using preexisting techniques. The key to efficiency is a simple search routine that uses the bounding representation to ignore most of the possible pairs of components. The efficiency can further be improved by accepting a relative error in the returned result. Several empirical trials are presented to examine the performance of the algorithm.

1. Introduction

Computing the distance between objects is a common problem in robotics. Using a mathematical model of two objects, we find a point on each object such that the distance between the points is minimized. If one object is a robot and the other object is the union of all the obstacles in the environment, such information describes how close the robot is to collision. Distance computation has been used for real-time collision avoidance [1], real-time path modification [2], and optimal path planning [3].

Previous work on the distance computation problem has focused on convex objects. Lumelsky [4] describes an efficient algorithm for pairs of line segments. Gilbert et al. [5], Bobrow [6], and Lin and Canny [7] present algorithms that find the distance between two convex polyhedra. Each of these three algorithms iteratively finds pairs of points, one on each object, such that the distance between the points monotonically converges to the minimum. These algorithms rely heavily on the properties of convex objects and it appears difficult to extend them directly to the non-convex case.

To compute the distance between non-convex objects, we can break each object into convex components and then use one of the above algorithms to determine the distance between components. The distance between the two objects is then the smallest distance between any pair of convex components. However, a naive implementation that examines all such pairs has complexity $O(nm)$, where n and m are the number of components of each of the objects.

Collision detection is related to distance computation; two objects are in collision if and only if the distance between the objects is zero. Collision detection has been used extensively in robotics for applications such as path planning [8], and in computer graphics for physical based modeling [9]. For such applications, collision detection often consumes a significant percentage of the execution time and much research has been directed at finding efficient algorithms.

Two related approaches for efficient collision detection are hierarchical models and bounding representations. A hierarchical model, such as proposed by Faverjon [10], describes an object at various levels of detail. The collision detection algorithm uses the different levels of detail to reduce the number of components that are examined. Similarly, bounding representations approximately model an object with simple primitives such as rectangles. Efficient algorithms, such as described by Baraff [11], determine if collision has occurred between the bounding representation and only then are components of the original model examined.

In this paper, we present an efficient algorithm for distance computation between non-convex obstacles. The approach builds from the research on collision detection and convex distance computation. Objects are described as a set of convex components and we refer to this description as the underlying model. From this model we build a hierarchical bounding representation, based on spheres, that approximates the object. A search routine examines the hierarchical bounding representation of each object and determines pairs of components to compare with a convex distance algorithm. Experimental results show that only a small fraction of the possible pairs are compared, thus avoiding the $O(nm)$ complexity of the naive implementation.

To drastically increase the efficiency of our algorithm, we propose computing the distance between objects with a *relative error*. The idea is that, for some applications, it is acceptable to partially underestimate the distance between objects. We limit the error between the reported distance and the exact distance to be a user specified fraction of the exact distance; the magnitude of the error is reduced as the two objects approach each other and we never incorrectly report collision.

By using a relative error, exact distance computation and collision detection become two extremes of the same problem. When the user specifies zero relative error, we compute the exact distance. Conversely, when the acceptable relative error approaches one hundred percent, the value returned by the distance computation becomes meaningless, but a value of zero is returned if and only if the objects intersect. Providing a continuum between these two extremes enables applications to gain the benefits of some distance information with the efficiency of collision detection.

The remainder of the paper is divided into five sections. First, we describe the bounding representation and how we build it from the underlying model. Next, we examine the time to build the bounding representation and discuss how it can be done as a precomputation. We then describe the search routine used to determine the distance between objects. To examine the efficiency of the algorithm, we present the results of several empirical trials. We conclude with a summary of the two main ideas of this paper and their consequence.

2. The Hierarchical Bounding Representation

Before computing the distance between objects, we use the underlying model of the objects to build a hierarchical bounding representation.

The version of our algorithm described in this paper assumes the underlying model is a surface representation consisting of a set of convex polygons. This assumption is not fundamental and extending the approach to other representations would not be difficult. Using a surface representation implies that we will not detect collision when one object completely contains another object, but such situations are avoided in many applications.

The bounding representation is based on spheres. The sphere is the simplest geometric solid; it can be specified with a position vector and a radius. To calculate the distance between two spheres we require only seven additions, three multiplications, and one square root. Other primitives, such as rectanguloids or ellipsoids, may better approximate components of the underlying model, however, we feel the simplicity of the sphere makes it the preferred bounding shape. A collision detection algorithm by del Pobil et al [12] also uses spheres to build a bounding representation.

The bounding representation consists of an approximately balanced binary tree. Each node of the tree contains a single sphere and the tree has has the following two properties: the union of all the leaf spheres completely contains the surface of the object and the sphere at each node completely contains the spheres of its descendant leaf nodes.

The idea behind the bounding representation is as follows. The leaf spheres closely approximate the surface of the object. Interior nodes of the tree represent approximations of descendant leaf spheres. One can use the sphere at an interior node to determine a lower bound for the distance to any of the descendant leaf nodes, and hence to the object's surface. Nodes that are close to the root of the tree represent many leaf nodes, but to a coarse resolution. Conversely, nodes near the bottom of the tree closely approximate the shape of the few leaf spheres below them. The tree represents a hierarchical description of the object.

The first step to building the tree is to cover the object's surface with small spheres. These spheres will be the leaf nodes of the tree. The underlying model of the object is a set of convex polygons; to cover the surface we cover each polygon. The covering of a convex polygon is done in a process similar to scan conversion in computer graphics [13]. A regular grid of equal sized spheres covers the polygon with the center of each sphere lying in the plane of the polygon. In addition, to enable the search routine to determine which convex components to compare, we label each leaf sphere with the polygon for which it was created.

After covering the object with small spheres we use a divide and conquer strategy to build the interior nodes of the tree. The set of leaf nodes is divided into two approximately equal subsets. We build a tree for each of the subsets and these are combined into a single tree by creating a new node with each of the subtrees as children. The subtrees are built by recursively calling the same algorithm until the set consists of a single leaf node.

Each node has a sphere that contains all the spheres of the descendent leaf nodes and represents an approximation to these leaves. The two children of the node are intended to represent a slightly more accurate approximation of the same leaves. To maximize the improvement of the approximation, we desire to split a set of leaf nodes into two subsets so that the bounding sphere for each subset will be small. Although we have not found an optimal method of splitting a set of leaf nodes, the process used is simple, efficient, and effective.

To divide a set of leaf node into two subsets, we compute a bounding rectanguloid box that is aligned to the object's coordinate frame and contains the centers of all of the leaf spheres. Such a bounding box can be found by determining the minimum and maximum value for each of the three coordinates for the leaves' position vectors. Next, we select the axes along which the bounding box is longest, and divide the leaf nodes using the average value along this axis as the discriminant. Each of the resulting two subset should be rather compact and contain approximately equal numbers of elements.

After dividing the set into two subsets, we build trees for each subset by recursively invoking the algorithm, then create a new node with the two subtrees as children.

All that remains is to determine a bounding sphere that contains all the descendant leaf spheres.

There is no obvious way to compute the smallest sphere that contains a set of spheres, so we use two heuristic methods and select the smaller of the two spheres. The first method finds a bounding sphere that contains the spheres of the two children nodes and hence, by induction, all the descendant leaf nodes. The position and size of such a sphere can be determined optimally and uniquely as we are only bounding two spheres; the details are trivial and not included here. The second method directly considers the leaf spheres. We first select a center for the bounding sphere and then examine each of the descendant leaf spheres to determine the minimum radius required. The selection of the bounding sphere center is done by using the average position of the centers of the leaf spheres, which has already been calculated in the process of dividing the leaf spheres. The first method works well near the leaves of the tree, while the second methods produces better results closer to the root.

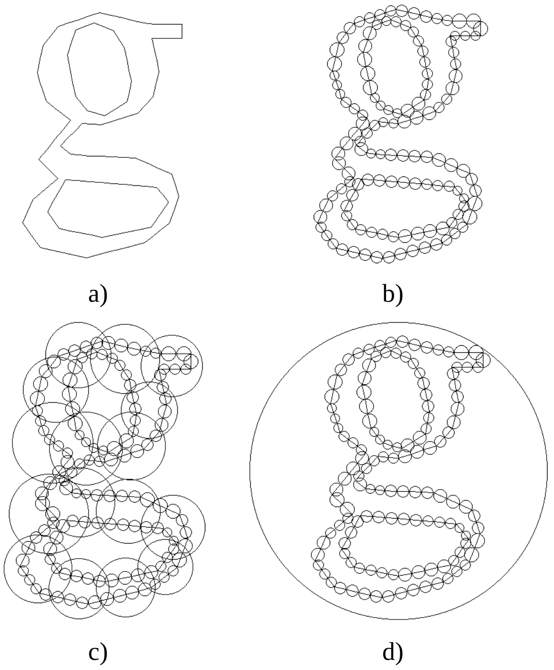


Figure 1. The bounding tree for an object.

Figure 1 attempts to illustrate a bounding tree generated by the above algorithm. Due to the difficulty of visualizing three dimensional objects, we show the analogous two dimensional version. The object is a polygon approximation to the letter ‘‘g’’ as shown in Figure 1 a). Figure 1 b) depicts the one hundred and thirty leaf spheres used to cover the outline of the object. The resulting bounding tree has eight levels. Figure 1 c) shows level five and Figure 1 d) shows the root of the tree.

3. Execution Time

The bounding tree is expected to be approximately balanced. If there are n leaf nodes, we expect there to be about n interior nodes and the depth of the tree to be about $\log n$. The time taken to split a set of nodes into two and the time needed to form the bounding sphere has order $O(n_i)$ complexity, where n_i is the number of leaf nodes descending from the ith node. A close look at the execution time of the above algorithm reveals an expected execution time of $O(n \log n)$, but a worst case of $O(n^2)$.

The worst case execution time can be reduced to $O(n \log n)$ if we use the median rather than the average to partition a set. The median can be found in $O(n)$ via the classic algorithm [14] or, more simply, if we perform an initial sort of the leaf nodes along each of the three axes and carry out some additional bookkeeping [15]. The disadvantage of these algorithms is the higher constant factor in the expected execution time. A close analogy can be drawn between the relative benefits of quicksort, which has worst case $O(n^2)$ performance, versus merge sort, which has worst case $O(n \log n)$; quicksort is preferred for its faster expected execution time.

For many applications, building the bounding representation can be performed as a precomputation step. If the two objects are rigid bodies then we compute the bounding tree for each object in their local coordinate frames. Before computing the distance between the two objects at specific positions and orientations, we augment each tree with the corresponding transformation matrix describing the positions of the object with respect to some global coordinate frame. Before a node is used in the search routine, the node’s sphere is mapped through the tree’s transformation matrix. The mapping is done only when a node is used because the search routine only examine a small fraction of the total nodes of a tree; mapping all the nodes before the search would result in lower efficiency.

A similar scheme can be devised if an object consists of several rigid bodies. A tree is built for each rigid body as a precomputation step. For a given configuration of the rigid bodies, we build a meta-tree with one leaf node per rigid body. The algorithm for building the meta-tree is identical to the algorithm for the individual rigid body trees. Each leaf node corresponds to the root of the tree for one rigid body and contains the transformation matrix needed to map the body from its local coordinate frame to the global frame. The bounding sphere for a leaf node is the bounding sphere of the root of the rigid body tree mapped into the global frame. In this fashion, only the construction of a relatively small tree need be done before each distance computation. The search routine traverses the tree in such a manner that any node can be mapped into the global frame when needed.

4. Computing the Distance between Two Objects

In this section we describe the algorithm to compute the distance between objects. As mentioned in the introduction, the goal of our algorithm is to compute the distance with a user specified relative error; to aid the presentation we first describe a version of the algorithm that has no error.

To compute the exact distance between two objects we need to find a pair of points, one on each object, such that the distance between the points is less than or equal to the distance between any other pair. In our implementation, we describe the object's surfaces as a set of convex polygons and we assume one object does not completely contain the other object; we can find the distance between the objects by finding a pair of polygons such that the distance between the polygons is less than or equal to the distance between any other pair. The distance between polygons is computed using a convex distance algorithm.

An overview of our algorithm to compute the distance, d, between objects is as follows. We initially set d to infinity. A search routine attempts to show the objects are at least a distance d apart. Suppose the search finds two polygons from the underlying model that are less than d apart; for the initial value of d this is not difficult. If the polygons intersect, then we know that the distance between the two objects is zero and we are done. Otherwise, we set d to the distance between the two polygons and continue the search with the new value of d. Eventually, the search shows that either the objects are a distance d apart or the objects intersect.

The key to the algorithm is be able to show the two objects are a distance d apart without examining all possible pairs of polygons. As each polygon is covered by a set of leaf nodes in the bounding tree, we need only examine pairs of polygons for which a corresponding pair of leaf spheres are less than a distance d apart. Of course, if we had to examine all possible pairs of leaf spheres, then we would have gained nothing, but, the hierarchical structure of the bounding tree enables us to avoid this situation.

The search routine finds pairs of leaf nodes that are less than a distance d apart. The search examines pairs of nodes in a depth-first manner starting with the root nodes of the two trees. If the distance between the nodes' spheres is greater or equal to the current value of d then, from the structure of the bounding trees, we know the distance between the two sets of descendant leaf spheres is greater or equal to d and can thus be ignored. If the two nodes are less than d apart, then we must further examine the children of the nodes. There are three cases to consider.

If both the nodes are from the interior of the tree, we split one of the nodes into its two children then recursively search the two pairs consisting of a child and the node not split. Deciding which node to split is based on the heuristic of splitting the node with the larger associated sphere. If all leaf spheres are assumed to be roughly the same size, we would expect the node with the smaller sphere to more closely approximate the shape of the underlying surface; more information about the surface will be obtained by splitting the larger sphere. Of the two subsequent recursive searches, we first examine the pair of nodes with spheres that are closer together. This heuristic aids the search in quickly lowering the value of d, resulting in fewer nodes having been searched.

In the case of one interior node and one leaf node, the interior node is split. The order of the two subsequent searches is the same as above.

In the case where two leaf spheres are less than the distance d apart, the underlying model must be examined. Each leaf sphere is labeled with the polygon that it covers. The distance between two polygons can be computed using one of the many available distance algorithms for convex objects. In the case of this implementation, we used the algorithm developed by Gilbert et al. [5]. If the distance between the two polygons is less than d, then we have found a new minimum. If the distance is zero, i.e. the polygons intersect, then we know the distance between the objects is zero and the search need not continue. Otherwise, we set d to the new distance and continue the search.

Each polygon may be covered by many leaf spheres, thus it is possible that the search routine may compute the distance between the same pair of polygons multiple times. Since such multiple identical computations are unnecessary, we record which pairs of polygons have been examined and, before computing the distance between two polygons, we check that the computation has not previously been done. Due to the large number of possible pairs of polygons, we record this information using a hash table.

The search routine can be modified to included our notion of relative error. In the modified algorithm, the user specifies a relative error α. We calculate a distance d' such that $d' \leq d$ and $d - d' \leq \alpha d$. Note that d' can equal zero only if d equals zero; we will not incorrectly report collision.

To implement the modified algorithm, the search routine must show that the objects are a distance d', rather than d, apart. The initial value of d' is set to infinity as in the exact case. However, when we find two polygons that are closer than d' apart, we set d' to be a fraction $1 - \alpha$ of the distance between them. After completing the search, we know the objects are at least d' apart. In addition, the true distance d between the objects is obviously less than or equal to the distance between the two polygons that where used to set d'. Hence, it can be shown that the error between d and d' meets the relative error specification.

5. Empirical Trials

To demonstrate the performance of the distance computation algorithm, we present the results of several empirical trials. The nature of the algorithm is such that the performance depends on many factors: the shape of the objects, the underlying representation, the specified relative error, the distance between the objects, the accuracy of the bounding tree, etc. As these factors depend on the application for which the distance algorithm is used, it is difficult to make general statements about the performance. Instead, we have chosen one scenario and examined the performance with respect to a few factors. Hopefully, these results give a reasonable impression of the behavior of the algorithm. Reported execution times are from an implementation on a DECstation 5000/240.

The scenario for which the experiments are performed is based on determining the distance between chess pieces. Six chess pieces, a king, queen, rook, bishop, knight and pawn, are randomly placed in three dimensional space. For each chess piece, we determine its distance to the union of the other five pieces. Figure 2 represents a typical configuration of the chess pieces.

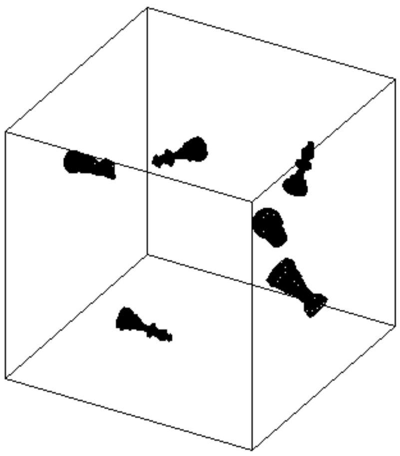


Figure 2. A typical configuration of the chess pieces.

The model of the pieces was designed by Randy Brown and is publicly available by ftp from wuarchive.wustl.edu. Each piece is described by a bounding representation consisting of roughly 2,000 triangles. The pieces are non-convex, have rather detailed features, and are roughly 100 units high. It is worth noting that the naive implementation for computing the distance from one chess piece to the other five pieces would examine all of the possible $2{,}000 \times 10{,}000$ pairs of triangles.

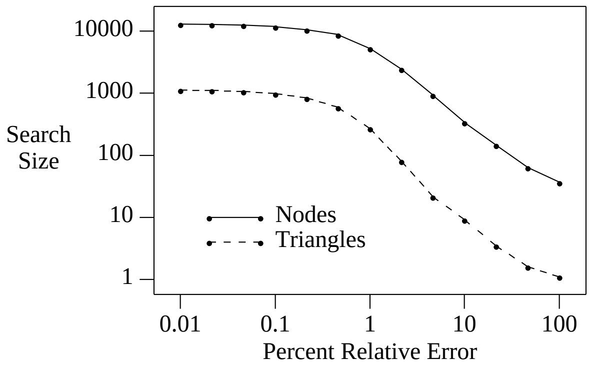
As a precomputation step, we built the bounding tree for each of the chess pieces. For the first two experiments we set the radius of the leaf spheres to 2 units resulting in a total of 34461 leaf nodes. A smaller leaf size would increase the number of nodes and the accuracy of the bounding tree. The result would be an increase in the search time and a decrease in the number of polygon comparisons. The net effect of such a change depends on the situation; in this case 2 units was found to be a reasonable size. The bounding trees for all the pieces are built in 6.4 seconds.

The objects were placed with random position and orientation. The orientation was selected from a uniform distribution of all possibles orientations. The position vector was selected from a uniform distribution within a cube measuring 500 units along the sides. Each point in the following graphs represent the average for one hundred random positions.

Before performing each distance computation, we built a meta-tree for the five chess pieces that were compared to the current piece. As the tree contains only five leaf nodes, the time to build the tree was negligible.

Graph 1 examines the effect of varying the relative error. The smaller relative errors effectively correspond to computing the exact distance between objects. Even in these cases, the number of comparisons of both nodes and triangles is far less than the possible $20{,}000{,}000$. As the relative error is increased, the number of comparisons drops dramatically. There is approximately two orders of magnitude improvement if a 20% relative error is specified.

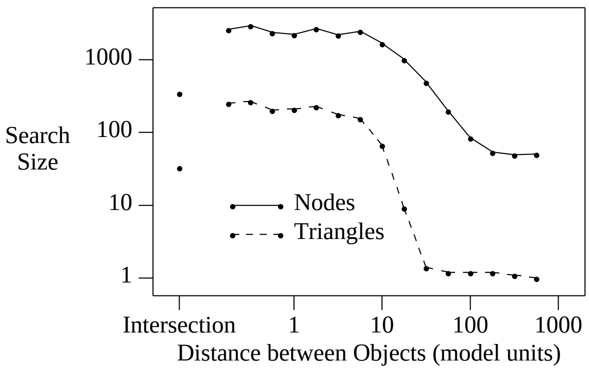
Graph 1. Search Size vs Relative Error



The current implementation can examine 80,000 pairs of nodes a second and compute the distance between 11,000 triangles a second. For a 20% relative error, the average execution time is 2.0 milliseconds. We speculate that such execution time would be comparable to the performance of a good collision detection algorithm, with the advantage that we obtain considerable distance information.

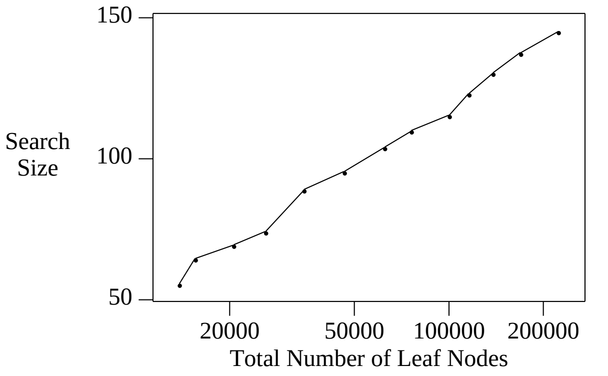
As two objects get closer, one would expect the search routine to examine more nodes and triangles. Graph 2 depicts this relationship for a relative error of 20%. As can be seen, the algorithm runs a lot slower when the objects are close. The effect on the average time to perform a distance computation depends on the distribution of distances for a given application. Note that the far left points corresponds to configurations where the chess piece intersected with one of the other five pieces.

Graph 2. Search Size vs. Distance between Objects



Graph 3 examines the effect of increasing the amount of detail used to describe each of the chess pieces. As we only have one polygonal model of the pieces, we perform this experiment by assuming the leaf spheres of the bounding representation exactly describe the object. We use the underlying model only to build the bounding trees, it is not used during the search routine. By varying the radius of the leaf spheres, we can vary the detail of the description. As can be seen, the number of nodes searched appears to be proportional to the log of the number of leaf spheres. This is very encouraging; a more detailed model of the object can be used with very little affect on the execution time. These results we performed with a relative error of 20%. Unfortunately, the same relationship does not hold for very small relative errors.

Graph 3. Search Size vs. Number of Leaf Nodes



6. Conclusion

The combination of a hierarchical bounding representation, a simple search routine, and a convex distance algorithm appears to be a powerful framework for building an efficient distance algorithm for non-convex objects.

The notion of relative error enables exact distance computation and collision detection to be unified as two extremes of a single problem. By accepting a reasonable relative error, we hope to achieve the performance of a collision detection algorithm while still obtaining useful distance information.

Efficient distance computation opens the possibility of modifying applications that currently employ collision detection. As distance computation provides information about how close objects are to collision, it is possible to reason rigorously about the collision free motion of objects. This capability is more difficult if we consider only the question of whether or not a given configuration of the objects is in collision since any motion of objects may bring them into contact.

References

1. O. Khatib, ''Real-Time Obstacle Avoidance for Manipulators and Mobile Robots,'' *The International Journal of Robotics Research* **5**(1), pp. 90-98 (Spring 1986).

2. S. Quinlan and O. Khatib, ''Elastic Bands: Connecting Path Planning and Control,'' *Proc. IEEE International Conference on Robotics and Automation,* Atlanta (1993).

3. J. E. Bobrow, ''Optimal robot plant planning using the minimum-time criterion,'' *IEEE Journal of Robotics and Automation* **4**(4), pp. 443-50 (August 1988).

4. V. J. Lumelsky, ''On Fast Computation of Distance Between Line Segments,'' *Information Processing Letters* **21**, pp. 55-61 (August 1985).

5. E. G. Gilbert, D. W. Johnson, and S. S. Keerthi, ''A Fast Procedure for Computing the Distance Between Complex Objects in Three-Dimensional Space,'' *IEEE Journal of Robotics and Automation* **4**(2) (April 1988).

6. J. E. Bobrow, ''A Direct Minimization Approach for Obtaining the Distance between Convex Polyhedra,'' *The International Journal of Robotics Research* **8**(3), pp. 65-76 (June 1989).

7. M. C. Lin and J. F. Canny, ''A Fast Alogrithm for Incremental Distance Calculation,'' *Proc. of IEEE International Conference on Robotics and Automation,* pp. 1008-1014 (April 1991).

8. J. C. Latombe, *Robot Motion Planning,* Kluwer Academic Publishers, Boston (1991).

9. D. Baraff, ''Analytical Mathods for Dynamic Simulation of Non-penetrating Rigid Bodies,'' *Computer Graphics (Proc. SIGGRAPH)* **23**(3), pp. 223-232 (July 1989).

10. B. Faverjon, *Hierarchical Object Models for Efficient Anti-Collision Algorithms*, 1989.

11. D. Baraff, ''Rigid Body Simulation,'' *SIGGRAPH Course Notes 1992* **19** (July 1992).

12. A. P. del Pobil, M. A. Serna, and J. lovet, ''A New Representation for Collision Avoidance and Detection,'' *Proc. of IEEE International Conference on Robotics and Automation*, pp. 246-251 (May 1992).

13. D. F. Rogers, *Procedural Elements for Computer Graphics,* McGraw-Hill (1985).

14. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, Reading (1983).

15. F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction,* Springer-Verlag, New York (1985).