Interactive and Exact Collision Detection for Multi-Body Environments

Jonathan D. Cohen *
Computer Science Department
University of North Carolina
Chapel Hill, NC 27599-3175
cohenj@cs.unc.edu

Ming C. Lin
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943
mlin@cs.nps.navy.mil

Dinesh Manocha †
Computer Science Department
University of North Carolina
Chapel Hill, NC 27599-3175
manocha@cs.unc.edu

Madhav K. Ponamgi ‡
Computer Science Department
University of North Carolina
Chapel Hill, NC 27599-3175
ponamgi@cs.unc.edu

Abstract: We present an algorithm for exact collision detection in interactive environments. Such environments are characterized by the number of objects undergoing motion and the complexity of the models. We do not assume that the motions of the objects are expressible as closed-form functions of time, nor do we assume any limitations on their velocities. The algorithm uses a two-level hierarchical representation for each model to *selectively* compute the *precise* contact between objects, achieving real-time performance without compromising accuracy. In large environments with n moving objects, it uses the temporal and geometric coherence that exists between successive frames to overcome the bottleneck of $O(n^2)$ pairwise comparisons. The algorithm has been successfully demonstrated in architectural walkthrough and simulated environments. In particular, the algorithm takes less than 1/20 of a second to determine all the collisions and contacts in an environment consisting of more than 1000 moving polytopes, each consisting of more than 50 faces.

Additional Keywords and Phrases: interference, contact, geometric coherence, simulations, virtual environment, walkthrough

*Supported in part by NSF MIP-9306208

†Supported in part by a Junior Faculty Award, University Research Award, NSF Grant CCR-9319957, ARPA Contract DABT63-93-C-0048 and NSF/ARPA Science and Technology Center for Computer Graphics and Scientific Visualization, NSF Prime Contract Number 8920219.

‡Supported in part by NSF MIP-9306208 and NSF Grant CCR-9319957

1 Introduction

Collision detection has been a fundamental problem in computer animation, physically-based modeling, geometric modeling, and robotics. In these applications, interactions between moving objects are modeled by dynamic constraints and contact analysis. The objects' motions are constrained by various interactions, including collisions. This paper focuses on collision detection for virtual environments and complex simulations.

A virtual environment, like a walkthrough, creates a computer-generated world, filled with virtual objects. Such an environment should give the user a feeling of presence, which includes making the images of both the user and the surrounding objects feel solid. For example, the objects should not pass through each other, and things should move as expected when pushed, pulled or grasped. Such actions require accurate collision detection, if they are to achieve any degree of realism. However, there may be hundreds, even thousands of objects in the virtual world, so a naive algorithm could take a long time just to check for possible collisions as the user moves. This is not acceptable for virtual environments, where the issues of *interactivity* impose fundamental constraints on the system [Zel92]. A fast and interactive collision detection algorithm is a fundamental component of a complex virtual environment. Fig. 1 shows a typical walkthrough environment.

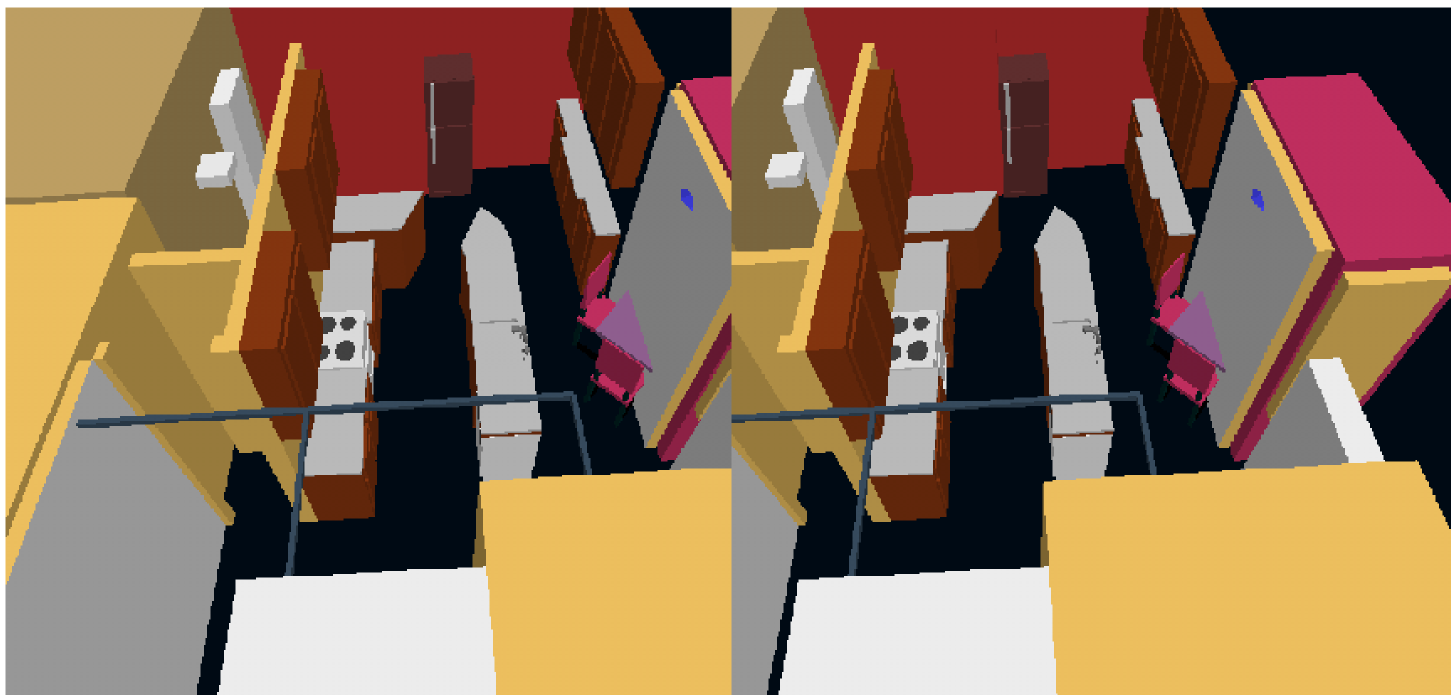


Figure 1: A Virtual Walkthrough

Complex simulations involve many objects moving according to some physical parameters. For example, virtual prototyping systems create electronic representations of mechanical parts which need to be tested for interconnectivity, functionality, and reliability. The testing process necessitates simulations containing hundreds of parts, whose complex inter-

actions are based on physics and geometry. Such simulations rely on fast, accurate collision detection schemes.

The objective of collision detection in both virtual environments and simulations is to report geometric contacts between objects. If we know the positions and orientations of the objects in advance, we can solve collision detection as a function of time. However, this is not the case in virtual environments or other interactive applications. In fact, in a walkthrough environment, we usually do not have any information regarding the maximum velocity or acceleration, because the user may move with abrupt changes in direction and speed. Because of these unconstrained variables, collision detection is currently considered to be one of the major bottlenecks in building interactive simulated environments[Pen90].

Main Contribution: We present an algorithm for interactive and exact collision detection in complex environments. In contrast to the previous work, we show that accurate, real-time performance *can* be attained in most environments if we use coherence not only to speed up pairwise interference tests, but also to reduce the actual number of these tests we perform. We are able to sucessfully trim the $O(n^2)$ possible interactions of n simulataneously moving objects to $O(n+m)$ where m is the number of objects "very close" to each other. In practice m is a small number and in the worst case can be $O(n^2)$. Our approach is flexible enough to mitigate the problem of choosing an appropriate time step common in many "static turned dynamic" detection algorithms. Moreover, we do *not* restrict ourselves to the sparse environment only, because in many interesting environments (e.g. vibrating parts feeder bowl) this assumption does not hold. Nor do we make *any* assumptions about object velocity or acceleration as required by a space-time approach [Hub93]; this type of information may be available in a simulation but is typically *absent* in a *truly interactive* virtual environment. The algorithm makes use of spatial and temporal coherence between successive instances and works very well in practice.

The rest of the paper is organized as follows. Section 2 reviews some of the previous work in collision detection. Section 3 defines the concept of coherence and describes an exact pairwise collision detection algorithm which applies it. Section 4 describes our algorithm for performing collision detection among multiple objects. This algorithm efficiently reduces the number of pairwise collision tests using a *dimension reduction* technique. Section 5 describes the implementation of our algorithm. Section 6 presents our experimental results on architectural walkthrough environments and simulations.

2 Previous Work

The problem of collision detection has been extensively studied in robotics. However, the goal in robotics has been the planning of collision-free trajectories between obstacles [Can86, Lat91, LPW79, Sha87]. This differs from virtual environments and physically-based simulations, where the motion is subject to dynamic constraints or external forces and cannot typically be expressed as a closed form function of time .

The problem has also been addressed in computational geometry. The emphasis in the computational geometry literature has been on theoretically efficient intersection de-

tection algorithms for pairs of objects at a single instance in time [AS90, Cha89, CD87, DK90, Ede83]. For convex 3-polytopes [1] linear time algorithms based on linear programming [Meg83, Sei90] and tracking closest points [GJK88] have been proposed. More recently, temporal and geometric coherence have been used to devise algorithms based on checking local features of pairs of convex 3-polytopes [Bar90, Lin93]. Alonso et al.[ASF94] use bounding boxes and spatial partitioning to test all $O(n^2)$ pairs of arbitrary polyhedral objects.

To handle curved models, algorithms based on interval arithmetic have been proposed by [Duf92, HBZ90, SWF$^+$93]. Faster algorithms using coherence have been proposed for curved models defined algebraically or using NURBs [LM93].

Algorithms for collision detection between single pairs of objects are often extended to multiple-object interactions. An early algorithm of complexity $O(n^2m^2)$, where n is the number of polyhedra with m vertices per polyhedron, is described in [MW88].

Different methods have been proposed to overcome the bottleneck of $O(n^2)$ pairwise tests in an environment of n bodies. The simplest of these are based on spatial subdivision [BF79, Lev66]. The space is divided into cells of equal volume, and at each instance the objects are assigned to one or more cells. Collisions are checked between all object pairs belonging to a particular cell. This approach works well for sparse environments in which the objects are uniformly distributed through the space. Object space coherence has been used extensively in ray tracing for the past decade [SML90].

Another approach operates directly on four-dimensional volumes swept out by object motion over time [Cam90, Hub93]. This is difficult not only to visualize, but to model as well, especially when the motion is complex and abrupt. In addition, computing the intersection of complex hyper-cones induced by acceleration or uncertainty of position is non-trivial.

None of these algorithms adequately address the issue of collision detection in an interactive virtual environment. A complex, interactive virtual environment poses a new challenge to the detection problem because it requires performance at interactive rates for thousands of pairwise tests. Very recently, Hubbard has proposed a solution to address this problem by an interactive collision detection algorithm that trades accuracy for speed [Hub93]. However, exact contact determination is required for realistic simulations, so this tradeoff is not always acceptable. In an early extension of their work Lin and Canny [LC92] proposed a scheduling scheme to handle multiple moving objects. Dworkin and Zeltzer extended this work for a sparse model [DZ93]. The model works well for sparse environments only and requires information about time-parameterized trajectories of the objects to predict the possible intersection.

3 Background

In this section, we will first discuss the importance of coherence. This central concept leads us to present the exact collision detection algorithm for testing pairs of objects. Later we

[1] We shall refer to a bounded d-dimensional polyhedral set as a convex d-polytope, or briefly polytope. In common parlance, "polyhedron" is used to denote the union of the boundary and of the interior in E^3.

will use this algorithm as a component of the multi-body collision detection scheme.

3.1 Temporal and Geometric Coherence

Temporal coherence is the property that the application state does not change significantly between time steps, or frames. The objects move only slightly from frame to frame. This slight movement of the objects translates into geometric coherence, because their geometry, defined by the vertex coordinates, changes minimally between frames. The underlying assumption is that the time steps are small enough that the objects to do not travel large distances between frames.

A simulation with n independently moving objects can have $O(n^2)$ possible collisions at a given time step. In general, however, most configurations of a simulation have far fewer than $O(n^2)$ collisions. Combining temporal and geometric coherence with this observation has resulted in some efficient collision detection schemes between single pairs of objects [Bar90, Lin93, LM93].

3.2 Pairwise Collision Detection for Convex Polytopes

Here we will review an efficient collision detection algorithm which tracks closest points between pairs of convex polytopes [LC92, Lin93]. This algorithm is used at the second level of collision detection to determine the exact contact status between convex polytopes. The method 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. The algorithm runs in *expected constant time* if the polytopes are not moving swiftly. Even when a closest feature pair is changing rapidly, the algorithm takes only slightly longer (the runtime is proportional to the number of feature pairs traversed, which is a function of the relative motion the polytopes undergo).

3.2.1 Voronoi Regions

We represent each convex polytope using a modified boundary representation. Each polytope data structure has fields for its *features* (faces, edges, and vertices) and *Voronoi regions* (defined below). Each feature (a vertex, an edge, or a face) is described by its geometric parameters and its neighboring features, i.e. the topological information of incidences and adjacencies. In addition, we precompute the Voronoi region for each feature.

Definition: A *Voronoi region* (as shown in Fig. 2) associated with a feature is a set of points closer to that feature than to any other [PS85].

The Voronoi regions form a partition of space outside the polytope according to the closest feature. The collection of Voronoi regions of each polytope is the generalized Voronoi diagram of the polytope. Note that the generalized Voronoi diagram of a convex polytope

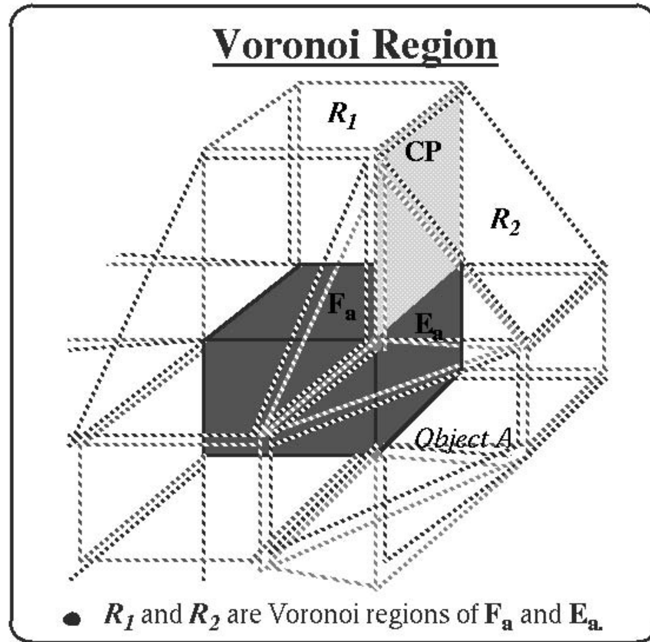


Figure 2: Voronoi Regions

has linear size and consists of polyhedral regions. A *cell* is the data structure for a Voronoi region. It has a set of constraint planes which bound the Voronoi region with pointers to the neighboring cells (which each 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. If a point P on object A lies inside the Voronoi region of the feature f_B on object B, then f_B is a closest feature to the point P.

3.2.2 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. Because 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. With some simple preprocessing, we can guarantee that every feature has a constant number of neighboring features. This is how we can verify or update the closest feature pair in *expected constant* time.

When a feature pair fails the test, the new pair we choose is guaranteed to be closer than the old one. Even if the closest features are changing rapidly, say once per step along the path, our algorithm takes only slightly longer. In this case, the running time is proportional to the number of feature pairs traversed in this process. Because the Euclidean distance between feature pairs must always decrease when a switch is made [Lin93], cycling is impossible for non-penetrating objects. An example of this algorithm is given in Fig. 3.

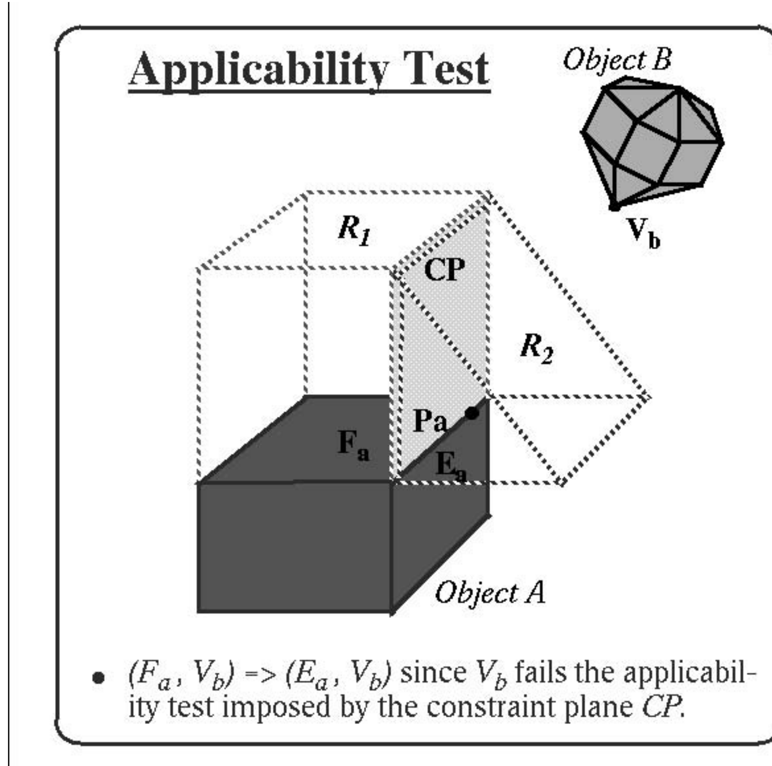


Figure 3: An Example of collision detection for convex polytopes

For example, given a pair of features, $feat_A$ and $feat_B$, on objects A and B, we first find a pair of nearest points, P_A and P_B, between these two features. Then, we need to verify that $feat_B$ is truly the closest feature on B to P_A (i.e. P_A lies inside the Voronoi region of $feat_B$) and $feat_A$ is truly the closest feature on A to P_B (i.e. P_B satisfies a similar applicability test of $feat_A$). If either test fails, a new, closer feature is substituted, and the new pair of features is tested. Eventually, we must reach the closest pair of features.

3.3 Penetration Detection for Convex Polytopes

The core of the collision detection algorithm is built using the properties of Voronoi regions of convex polytopes. As mentioned earlier in Sec 3.2.1, the Voronoi regions form a partition of space outside the polytope. When polytopes interpenetrate, some features may not fall into any Voronoi regions. This can lead to cycling of feature pairs if special care is not taken. So it is important that we have a module which detects interpenetration.

- **Pseudo Voronoi Regions:**

This module partitions the *interior space* of the convex polytopes. The partitioning does not have to form the exact Voronoi regions, because we are not interested in knowing the closest features between two interpenetrating polytopes but only detecting such a case. We can calculate a close approximation to internal Voronoi regions, then use this to detect

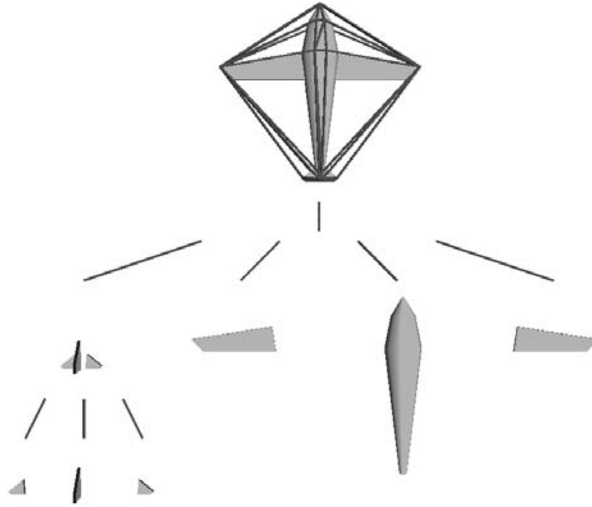


Figure 4: Subpart hierarchical tree of an aircraft

interpenetrating polytopes.

We can partition the interior of a polytope as follows. We calculate the centroid of each convex polytope, which is the weighted average of all the vertices, and then construct the constraint planes of each face. Each interior constraint plane of a face F is a hyperplane passing through the centroid and an edge in F's boundary, except one additional hyperplane containing the face F itself. If all the faces are equi-distant from the centroid, these hyperplanes form the exact Voronoi diagram for the interior of the polytope.

We can now walk from the external Voronoi regions into the pseudo Voronoi regions when necessary. If either of the closest features falls into a pseudo Voronoi region at the end of the walk, we know the objects are interpenetrating. Ensuring convergence as we walk through pseudo Voronoi regions requires special case analysis.

3.4 Extension to Non-Convex Objects

We can extend the collision detection algorithm for convex polytopes to handle non-convex objects, such as articulated bodies, by using a hierarchical representation. In the hierarchical representation, the internal nodes can be convex or non-convex sub-parts, but *all* the leaf nodes are convex polytopes.

Beginning with the leaf nodes, we construct either a convex hull or other bounding volume and work up the tree, level by level, to the root. The bounding volume associated with each node is the bounding volume of the union of its children; the root's bounding volume encloses the whole hierarchy. We precompute this bounding volume and store it in the node structure for later use. For instance, a hand may have individual joints in the leaves, fingers in the internal nodes, and the entire hand in the root. Fig. 4 shows a hierarchical representation of a simplified aircraft.

We test for collision between a pair of these hierarchical trees recursively. The collision detection algorithm first tests for collision between the two parent nodes. If there is no collision between the two parents, the algorithm reports that there is no collision. It returns the closest feature pair of their bounding volumes. If there is a collision, then the algorithm expands their children. If there is also a collision among the children, then the algorithm recursively proceeds down the tree to determine if a collision actually occurs. For complex objects, using a deep hierarchy tree with a lower branching factor will keep down the number of nodes which need to be expanded.

The number of levels in a tree is limited to the geometric complexity of the object. Exact contact status between two trees is determined by the exact collision detection between their convex sub-parts. This differs from Hubbard's approach [Hub93] which creates some number of nested levels of bounding spheres to achieve an *approximate* solution to a desired accuracy; our approach finds the *exact* collision points.

4 Collision Detection for Multiple Objects

Virtual environments contain both stationary and moving objects. For example, the human participants may walk through a building where the tables, chairs, etc. remain stationary. In such an environment, there are N moving objects and M stationary objects. Each of the N moving objects can collide with the other moving objects, as well as with the stationary ones. Keeping track of $\binom{N}{2} + NM$ pairs of objects at every time step can become time consuming as N gets large. To achieve interactive rates, we must reduce this number before performing pairwise collision tests. The overall architecture of the multiple object collision detection algorithm is shown in Fig. 5.

Sorting is the key to our *Sweep and Prune* approach. Assume that each object is surrounded by some 3-dimensional bounding volume. We would like to sort these bounding volumes in 3-space to determine which pairs are overlapping. We could then sweep over these sorted bounded volumes, pruning out the pairs that do not overlap. We only need to perform exact pairwise collision tests on the remaining pairs.

However, it is not intuitively obvious how to sort objects in 3-space. We use a *dimension reduction* approach. If two bodies collide in a 3-dimensional space, their orthogonal projections onto the xy, yz, and xz-planes and x, y, and z-axes must overlap. Using this observation, we choose axis-aligned bounding boxes as our bounding volumes. We can efficiently project these bounding boxes onto a lower dimension, and perform our sort on these lower-dimensional structures.

This approach is quite different from the typical space partitioning approaches used to reduce the number of pairs. A space partitioning approach puts considerable effort into choosing good partition sizes. But there is *no* partition size that prunes out object pairs as ideally as testing for bounding box overlaps. Partitioning schemes may work well for environments where N is small compared to M, but object sorting works well whether N is very small or very large.

Figure 5: Architecture for Multiple Body Collision Detection Algorithm

We will now present more details of our algorithm, including bounding volume methods and 1-D and 2-D sweep and prune techniques. We will also discuss some alternative methods.

4.1 Bounding Volumes

Many collision detection algorithms use bounding boxes, spheres, ellipses, etc. to rule out collisions between objects which are far apart. As stated above, our algorithm uses axis-aligned bounding boxes to efficiently identify 3-D overlaps. These overlaps trigger the *exact collision detection* algorithm.

We have considered two types of axis-aligned bounding boxes: fixed-size bounding cubes (fixed cubes) and dynamically-resized rectangular bounding boxes (dynamic boxes).

- **Fixed-Size Bounding Cubes:**

We compute the size of the fixed cube to be large enough to contain the object at *any* orientation. We define this axis-aligned cube by a *center* and a *radius.* Fixed cubes are easy to recompute as objects move, making them well-suited to dynamic environments. If an object is nearly spherical, or "fat" [Ove92], the fixed cube fits it well.

As preprocessing steps we calculate the center and radius of the fixed cube. At each time step as the object moves, we recompute the cube as follows:

1. Transform the center using one vector-matrix multiplication.

2. Compute the minimum and maximum x, y, and z-coordinates by subtracting and adding the radius from the coordinates of the center.

Step 1 involves only one vector-matrix multiplication. Step 2 needs six arithmetic operations (3 additions and 3 subtractions).

- **Dynamically-Resized Rectangular Bounding Boxes:**

We compute the size of the rectangular bounding box to be the tightest axis-aligned box containing the object at a *particular* orientation. It is defined by its minimum and maximum x, y, and z-coordinates (for a convex object, these must correspond to coordinates of up to 6 of its vertices). As an object moves, we must recompute its minima and maxima, taking into account the object's orientation.

For oblong objects rectangular boxes fit better than cubes, resulting in fewer overlaps. This is advantageous as long as few of the objects are moving, as in a walkthrough environment. In such an environment, the savings gained by the reduced number of pairwise collision detection tests outweighs the cost of moving the dynamically-resized boxes.

As a precomputation, we compute each object's initial minima and maxima. Then as an object moves, we recompute its minima and maxima as follows:

Method 1 –

We can use a modified routine from the exact collision detection algorithm described in Sec 3.2.2 or [Lin93]. We can set up 6 imaginary boundary walls. Each of these walls is located at a minimal or maximal x, y, or z-coordinate possible in the environment. Given the previous bounding volume, we can update each vertex of the bounding volume by performing only *half* of the modified detection test, because since all the vertices are always in the Voronoi regions of these boundary walls. Then we proceed as follows:

1. Find the nearest point on the boundary wall to the given vertex at its current (updated) position.

2. Check to see if the nearest point on the boundary wall lies inside of the Voronoi region of the previous vertex.

3. If so, the previous vertex is still the extremal point (minimum or maximum in the x, y, or z-axis) and we are done.

4. Otherwise, walk to the appropriate neighboring vertex returned by the modified exact collision detection algorithm and repeat the whole process.

This is a modification to the existing routine described in Sec. 3.2.2, and it preserves the properties of locality and coherence.

Method 2 –

Alternatively, we can use a simple method based on convexity. At each time step, we do the following:

1. Check to see if the current minimum (or maximum) vertex for the x, y, or z-coordinate still has the smallest (or largest) value in comparison to its neighboring vertices. If so we are finished.

2. Update the vertex for that extremum by replacing it with the neighboring vertex with the smallest (or largest) value of all neighboring vertices. Repeat the entire process as necessary.

This algorithm recomputes the bounding boxes at an expected constant rate. Once again, we are exploiting the temporal and geometric coherence, in addition to the locality of convex polytopes.

We can optimize this approach by realizing that we are only interested in *one* coordinate value of each extremal vertex, say the x coordinate while updating the minimum or maximum value along the x-axis. Therefore, there is no need to transform the other two coordinates in order to compare neighboring vertices. This reduces the number of arithmetic operations by 2/3.

4.2 One-Dimensional Sweep and Prune

The one-dimensional sweep and prune algorithm begins by projecting each three-dimensional bounding box onto the x, y, and z axes. Because the bounding boxes are axis-aligned, projecting them onto the coordinate axes results in intervals (see Fig. 6). 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 corresponding to that dimension. By sorting these lists, we can determine which intervals overlap. In the general case, such a sort would take $O(nlogn)$ 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 values of the interval endpoints. In environments where the objects make relatively small movements between frames, the lists will be nearly sorted, so we can sort in expected $O(n)$ time. *Bubble sort* and *insertion sort* work well for previously sorted lists.

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. This also runs in expected linear time due to coherence, but $e_x, e_y, and e_z$ can each be $O(n^2)$ with an extremely small constant.

Our method is suitable for dynamic environments where coherence is preserved. In computational geometry literature several algorithms exist that solve the static version of determining bounding box overlaps in $O(nlog^{d-1}n + s)$ time, where d is the dimension of the bounding boxes and s is the number of pairwise overlaps [Ede83, HSS83, SW82]. For 3-D bounding boxes we have reduced this from $O(nlog^2 n + s)$ to $O(n + s)$ by using coherence.

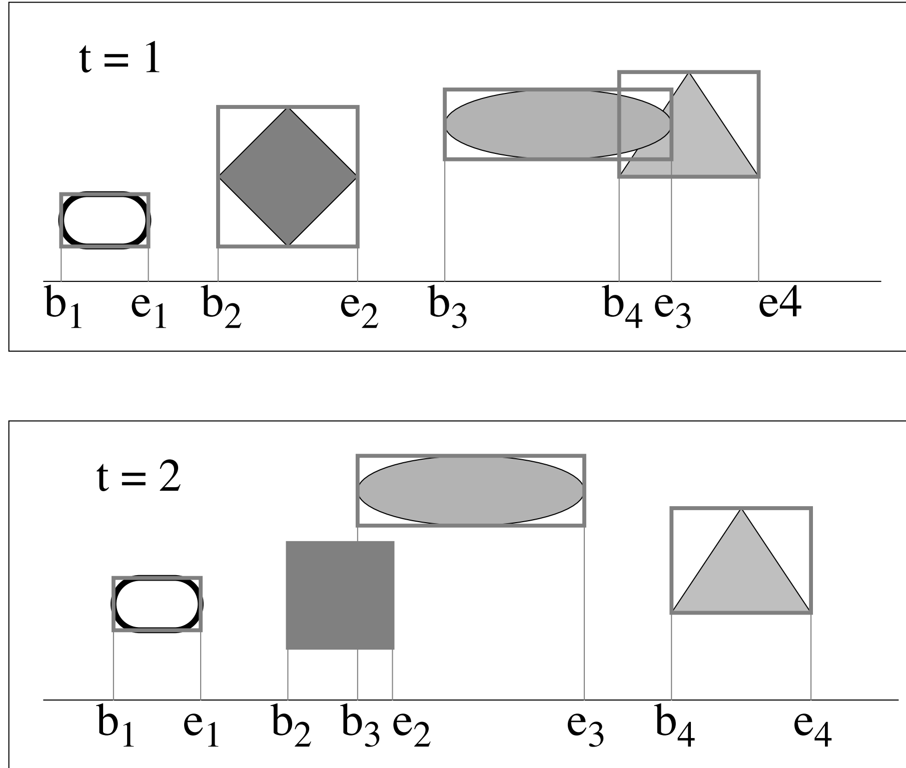


Figure 6: Bounding Box Behavior

4.3 Two-Dimensional Intersection Tests

The two-dimensional intersection algorithm begins by projecting each three-dimensional axis-aligned bounding box onto the x-y, x-z, and y-z planes. Each of these projections is a rectangle in 2-space. Typically there are fewer overlaps of these 2-D rectangles than of the 1-D intervals used by the sweep and prune technique. This results in fewer swaps as the objects move. We expect that in situations where the projections onto one-dimension result in densely clustered intervals, the two-dimensional technique will be more efficient. The interval tree is a common data structure for performing such two-dimensional range queries.

An interval tree is actually a *range tree* properly annotated at the nodes for fast search of real intervals. Assume that n intervals are given, as $[b_1, e_1], \cdots, [b_n, e_n]$ where b_i and e_i are the endpoints of the interval as defined above. The range tree is constructed by first sorting all the endpoints into a list $(x_1, \cdots, x_m)$ in ascending order, where $m \leq 2n$. Then, we construct the range tree top-down by splitting the sorted list L into the left subtree L_l and the right subtree L_r, where $L_l = (x_1, \cdots, x_p)$ and $L_r = (x_{p+1}, \cdots, x_m)$. The root has the split value $\delta = \frac{x_p + x_{p+1}}{2}$. We construct the subtrees within each subtree recursively, creating the sublists L_l and L_r based on the δ at each node, in this fashion until each leaf contains only an endpoint. The construction of the range tree for n intervals takes $O(nlogn)$ time. After we construct the range tree, we further link all nodes containing stored intervals in a doubly linked list and annotate each node if it or any of its descendants contain stored

intervals. The embellished tree is called the *interval tree*.

We can use the interval tree for static query, as well as for the rectangle intersection problem. To check for rectangle intersection using the sweep algorithm: we take a sweeping line parallel to the y-axis and sweep in increasing x direction, and look for overlapping y intervals. As we sweep across the x-axis, y intervals appears or disappear. Whenever a y interval appears, we check to see if the new interval intersects the old set of intervals stored in the interval tree, reporting all intervals it intersects as rectangle intersection, and add the new interval to the tree.

Each query of an interval intersection takes $O(logn + k)$ time where k is the number of reported intersections and n is the number of intervals. Therefore, reporting intersections among n rectangles can be done in $O(nlogn + K)$ where K is the total number of intersecting rectangles.

4.4 Alternatives to Dimension Reduction

There are many different methods for reducing the number of pairwise tests, such as binary space partitioning (BSP) trees [TN87], octrees, etc. We are currently considering other approaches for the virtual prototyping of mechanical parts/tools design and simulation, where hundreds of mechanical parts are rotating and translating in a vibrating bowl.

One of the first approaches we implemented uses a priority queue (implemented as a heap) sorted by the lower bound on the expected time to collision [LC92, Lin93]. This works well for a *sparse simulation* environment where very few collisions are occurring. However, as the number of objects increases, the number of "near misses" increases as well. It becomes progressively slow, since too many object pairs need to be re-ordered and large portions of the priority queue require to be re-shuffled. In addition, the velocity and acceleration information is not readily available in a truly interactive environment, such as the architectural walkthrough environment.

Several practical and efficient algorithms are based on uniform space division. We divide space into unit cells (or volumes) and place each object in some cell(s) [BF79, Lev66]. To check for collisions, we examine the cell(s) occupied by each object to verify if the cell(s) is(are) shared by other objects. But, it is difficult to set a near-optimal size for each cell and it requires a tremendous amount of allocated memory. If the size of the cell is not properly chosen, the computation can be rather expensive. For an environment where almost all objects are of uniform size and fat [Ove92], like a vibrating parts feeder bowl, or molecular modeling application [Lev66, Tur89], this algorithm is ideal, especially for execution on a parallel machine.

Alonso et. al. [ASF94] have an algorithm that combines space partitioning with hierarchical bounding boxes. The running time of their approach depends on the complexity and size of the objects being considered for collision detection. They perform a simple convex decomposition of the objects into voxels and place the objects into bounding boxes. They efficiently determine which top-level bounding boxes overlap, then examine them using the lower-level voxel representation.

Edahiro et. al. [ETHA87] proposed a $O(k)$ search time, $O(n)$ preprocessing time bucketing algorithm for reporting all segment intersections with a given orthogonal query segment, where n is the number of orthogonal segments and k is the number of reported intersecting segments. Although this approach cannot be applied directly to our problem, it indicates the potential for good average running time performance of the modified bucketing algorithm for 3-D intersection tests.

In fact, Overmars has shown that using a hashing scheme to look up entries and variable sizes (levels) for cubes or cells, we can use a data structure of $O(n)$ storage space to perform the point location queries in constant time [Ove92]. This approach works especially well if the objects are both fat and not intersecting, since the assumption behind the basic algorithm depends on the fact that each cube or cell can only be occupied by a fixed number of objects. If the objects are allowed to interpenetrate each other, this assumption can no longer hold. Another mechanism must be used to detect interpenetration.

5 Implementation

We implemented the collision detection algorithm in a library which deals with polytope as well as non-convex polyhedral models. A variety of applications can turn on collision detection, treating the library as a black box. In this section we will describe the implementation details of the Sweep and Prune algorithm, the exact collision detection algorithm, the multi-body simulation, and our interactive architectural walkthrough application.

5.1 Sweep and Prune

As described earlier, the Sweep and Prune algorithm reduces the number of pairwise collision tests by eliminating polytope pairs that are far apart. It involves three steps: calculating bounding boxes, sorting the minimum and maximum coordinates of the bounding boxes as the algorithm sweeps through each list, and determining which bounding boxes overlap. As it turns out, we do the second and third steps simultaneously.

We calculate bounding boxes using one of two methods. The first method involves computing the tightest fitting axis-aligned bounding box[2] for each polytope as it moves. Since the movement may involve rotation as well as translation, the dimensions of the box may vary significantly. The second method involves computing the smallest axis-aligned bounding cube that is guaranteed to contain the polytope at any orientation. We have used both types of bounding boxes in our tests. In general, fixed-size cubes are preferred over dynamically-resized boxes, except for the class of oblong objects.

Irrespective of which bounding box we choose, each resulting bounding box consists of a minimum and a maximum coordinate value for each dimension: x, y, and z. These minima and maxima are maintained in three separate lists, one for each dimension. We sort each list

[2] To account for uncertainty of model dimensions and unpredictable motion factor as well as to report a *possible intersection* instead of an *overlap*, we add in a small amount of tolerance δ when computing this bounding box.

of coordinate values using a bubble or insertion sort, while maintaining an overlap status for each bounding box pair.

Bubble and insertion sorts are useful for two reasons. First, temporal coherence makes it likely that each list is almost sorted. Both sorts operate in nearly linear time for such lists, because the number of interchanges is small. Second, the sorts swap only adjacent elements, making it convenient to maintain an overlap status for each polytope pair. We have found bubble sort works better for environments where only a few object move, such as the walkthrough, and the insertion sort works better for environments where large numbers of objects move locally.

In both sorts, the overlap status consists of a boolean flag for each dimension. Whenever all three of these flags are set, the bounding boxes of the polytope pair overlap. These flags are only modified when bubble or insertion sort performs a swap. We decide whether or not to toggle a flag based on whether the coordinate values both refer to bounding box minima, both refer to bounding box maxima, or one refers to a bounding box minimum and the other a maximum.

When a flag is toggled, the overlap status indicates one of three situations:

1. All three dimensions of this bounding box pair now overlap. In this case, we add the corresponding polytope pair to a list of active pairs.

2. This bounding box pair overlapped at the previous time step. In this case, we remove the corresponding polytope pair from the active list.

3. This bounding box pair did not overlap at the previous time step and does not overlap at the current time step. In this case, we do nothing.

When sorting is completed for this time step, the active pair list contains all the polytope pairs whose bounding boxes currently overlap. We pass this active pair list to the exact collision detection routine to find the closest features of all these polytope pairs and determine which, if any, of them are colliding.

5.2 Exact collision detection

The collision detection routine processes each polytope pair in the active list. The first time a polytope pair is considered, we select a random feature from each polytope; otherwise, we use the previous closest feature pair as a starting point. This previous closest feature pair may not be a good guess when the polytope pair has just become active. Dworkin and Zeltzer [DZ93] suggest precomputing a lookup table for each polytope to help find better starting guesses.

Our current collision detection implementation uses only the external Voronoi regions of the polytopes. Due to the lack of internal Voronoi regions in our current implementation, collisions often require us to perform some tests which may take linear time in the complexity of the polytope. These tests can be eliminated and reduced to a quick, constant expected time check by using internal Voronoi regions.

5.3 Multi-body Simulation

The multi-body simulation is an application we developed to test the collision detection library. It represents a general, non-restricted environment in which objects move in an arbitrary fashion resulting in collisions with simple impulse responses.

While we can load any convex polytopes into the simulation, we typically use those generated by the tessellation of random points on a sphere. Unless the number of vertices is large, the resulting polytopes are not spherical in appearance; they may range from oblong to fat, in all different random shapes, to present a richer set of models.

The simulation parameters include:

- The number of polytopes
- The complexity of the polytopes, measured as the number of faces
- The rotational velocity of the polytopes
- The translational velocity of the polytopes
- The density of the environment, measured as the ratio of polytope volume to environment volume
- The bounding volume method used for the Sweep and Prune (fixed-size cubes or dynamically-resized boxes)

The simulation begins by placing the polytopes at random positions and orientations. At each time step, the positions and orientations are updated using the translational and rotational velocities (since the detection routines make no use of pre-defined path, the polytopes' paths could just as easily be randomized at each time step). The simulation then calls the collision detection library and receives a list of colliding polytope pairs. It exchanges the translational velocities of these pairs to simulate an elastic reaction. It also tests for collisions with walls of the working volume and reacts by bouncing when a collision occurs.

We use this simulation to test the functionality and speed of the detection algorithm. In addition, we are able to visually display some of the key features. For example, the bounding boxes of the polytopes can be rendered at each time step. When the bounding boxes of a polytope pair overlap, we can render a line connecting the closest features of this polytope. It is also possible to show all pairs of closest features at each time step. These visual aids have proven to be useful in indicating actual collisions and additional geometric information for algorithmic study and analysis. See Frames 1-4 for examples of the simulation with several sets of parameters.

5.4 Walkthrough

The walkthrough is a head-mounted display application that involves a large number of polytopes depicting a realistic scene [ARJ90]. The integration of our library into such an

environment demonstrates that an interactive environment can use our collision detection library without affecting the application's real-time performance.

The walkthrough creates a virtual environment, such as a kitchen or a porch. The user travels through this environment, interacting with the polytopes: picking up virtual objects, changing their scale, and moving them around. Whenever the user's hand collides with the polytopes in the environment, the walkthrough provides feedback by making colliding bodies appear red.

We have incorporated the collision detection library routines into the walkthrough application. The scene is composed of polytopes, most of which are stationary. The user's hand, composed of several convex polytopes, moves through this complex environment, modifying other polytopes in the environment. Frames 5-9 show a sequence of shots from a kitchen walkthrough environment. Frame 5 shows the stereo image pair, while the rest show only the image seen by the left eye.

6 Performance Analysis

We measured the performance of the collision detection algorithm using the multi-body simulation as a benchmark. We profiled the entire application and tabulated the cpu time of only the relevant detection routines. All of these tests were run on an HP-9000/750.

The main routines involved in collision detection are those that update the bounding boxes, sort the bounding boxes, and perform exact collision detection on overlapping bounding boxes. As described in the implementation section, bounding boxes were updated by two different methods. Using fixed cubes as bounding boxes resulted in higher frame rates for the parameter ranges we tested.

In each of the first four graphs, we plot two lines. The bold line displays the performance of using dynamically-resized bounding boxes whereas the other line shows the performance of using fixed-size cubes. All five graphs refer to "seconds per frame", where a frame is one step of the simulation, involving one iteration of collision detection.

Each graph was produced with the following parameters, by holding all but one constant.

- *Number of polytopes.* The default value is a 1000 polytopes.
- *Complexity of polytopes*, which we define as the number of faces. The default value is 36 faces.
- *Rotational velocity*, which we define as the number of degrees the object rotates about an axis passing through its centroid. The default value is 10 degrees.
- *Translational velocity*, which we define in relation to the object's size. We estimate a radius for the object, and define the velocity as the percentage of its radius the object travels each frame. The default value is 10%.
- *Density*, which we define as the percentage of the environment volume the polytopes occupy. The default value is 1.0%.

In the graphs, the timing results do not include computing each polytope's transformation matrix, rendering times, and of course any minor initialization cost. We ignored these costs, because we wanted to measure the cost of collision detection alone. Our results show that collision detection will not hinder the performance of interactive applications.

We have summarized our most important results in the following graphs. Graph 1 shows how the number of seconds per frame scales with an increasing number of polytopes. We took 100 uniformly sampled data points from 20 to 2000 polytopes. The fixed and dynamic bounding box methods scale nearly linearly with a small higher-order term. The dynamic bounding box method results in a slightly larger non-linear term because the resizing of bounding boxes causes more swaps during sorting. This is explained further in our discussion of Graph 5. The seconds per frame numbers in Graph 1 compare very favorably with the work of Dworkin and Zeltzer [DZ93] as well as those of Hubbard [Hub93]. For a 1000 polytopes in our simulation, our *frame rate* is **23** frames per second using the fixed bounding cubes.

Graph 2 shows how the number faces affects the frame time. We took 20 uniformly sampled data points. For the dynamic bounding box method, increasing the model complexity increases the time to update the bounding boxes because finding the minimum and maximum values requires walking a longer path around the polytope. The time for exact collision detection also increases because our current implementation does not include internal Voronoi regions, as described in the previous implementation section. Surprisingly, the time to sort the bounding boxes decreases with number of faces, because the polytopes become more spherical and fat. As the polytopes become more spherical and fat, the bounding box dimensions change less as the polytopes rotate, so fewer swaps are need in the sweeping step.

For the fixed bounding cube, the time to update the bounding boxes and to sort them is almost constant. Only the limitations of the exact collision detection implementation, i.e. the lack of *efficient* treatment for penetration, affect the frame time.

Graph 3 shows the effect of changes in the density of the simulation volume. For both bounding box methods, increasing the density of polytope volume to simulation volume results in a larger sort time and more collisions. The number of collisions scales linearly with the density of the simulation volume. As the graph shows, the frame time scales well with the increases in density.

Graphs 4 through 6 show the effect of rotational velocity on the frame time. The slope of the line for the dynamic bounding box method is much larger than that of the fixed cube method. There are two reasons for this difference. The first reason, which we anticipated, is that the increase in rotational velocity increases the time required to update the dynamic bounding boxes. When we walk from the old maxima and minima to find the new ones, we need to traverse more features.

The second reason, which came as a more of a surprise, is the larger number of swapped minima and maxima in the three sorted lists. Although the three-dimensional volume of the simulation is fairly sparse, each one-dimensional view of this volume is much more dense, with many bounding box intervals overlapping. As the boxes grow and shrink, they cause

many swaps in these one-dimensional lists. And as the rotational velocity increases, the boxes change size more rapidly.

Graph 6 clearly shows the advantages of the static box method. Both the update bounding box time and sort lists time are *almost* constant as the rotational velocity increases.

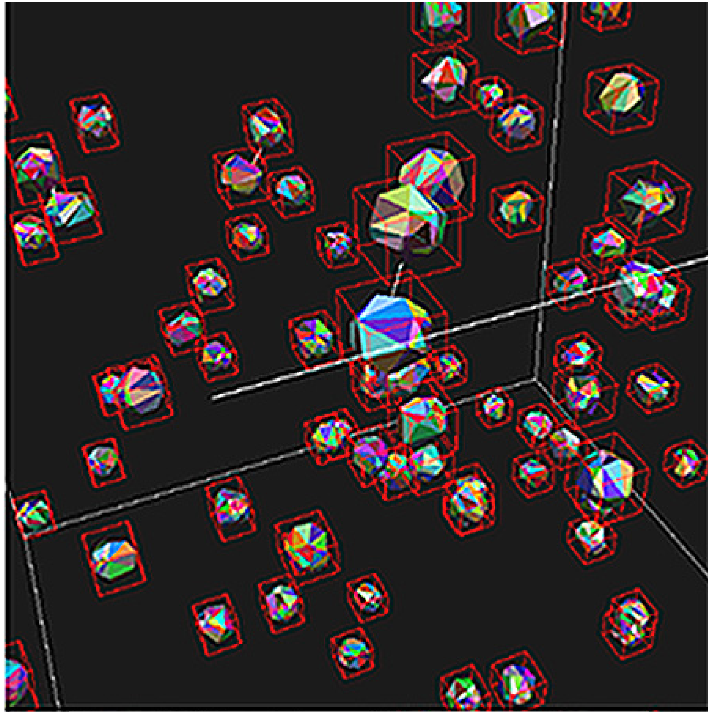
All of our tests show that high frame rates can be achieved while performing *exact* collision detection in demanding environments. The key is that the collision detection needs to be applied intelligently. The Sweep and Prune algorithm selectively applies fast, exact collision detection, and thus results in the high frame rates.

Realistic situations like the kitchen walkthrough often have many fewer *moving* objects than our simulations. Nevertheless, our algorithm showed great performance even under extreme conditions. The architectural walkthrough models showed no perceptible performance degradation when collision detection was added (as in Frame 5 to 9).

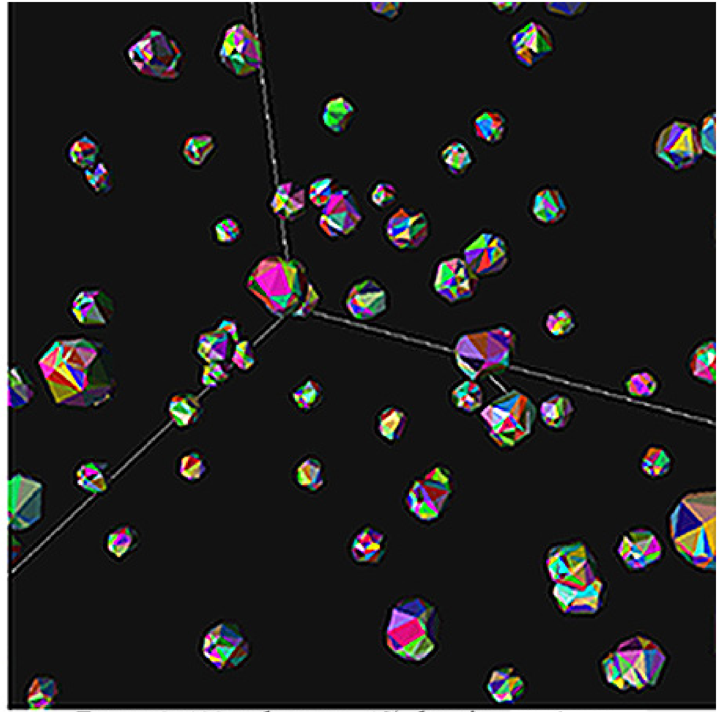
7 Conclusion

For a virtual environment, like a walkthrough, to be convincing, it needs to contain objects that respond realistically to collisions with the user and with the other objects. By making use of geometric and temporal coherence, our algorithm detects these collisions more efficiently and effectively than any algorithms we have known. Under many circumstances our algorithm produces frames rates over 20 hertz for environments with over a 1000 moving polytopes. Our walkthrough experiments showed no degradation of frame rates when collision detection was added.

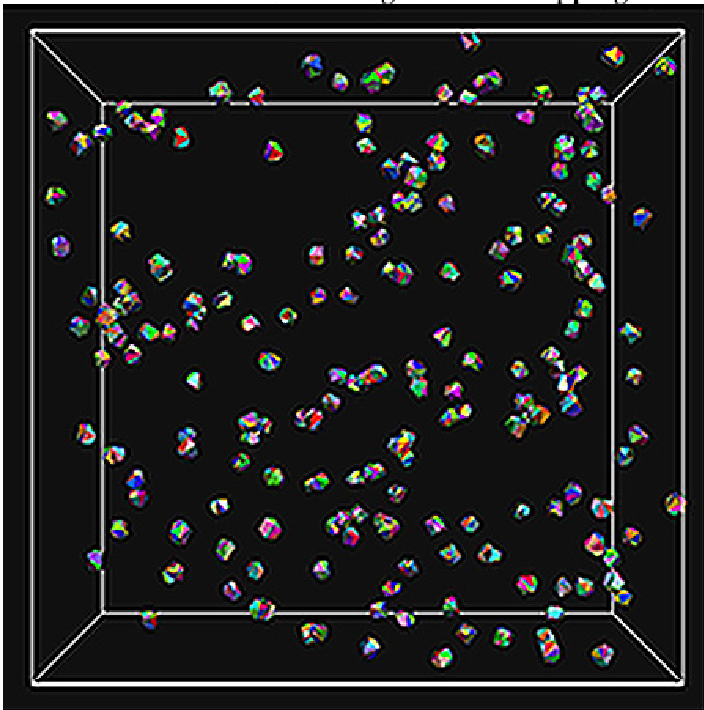
We plan to add to our current system complete penetration detection using internal Voronoi regions. This should result in better performance for dense environments, where collisions are more common. In addition, we would like to implement deeper hierarchy to explore methods of grouping concave polytopes. It should also be interesting to combine spatial decomposition with the Sweep and and Prune algorithm. Such a hybrid algorithm might be well-suited for a parallel implementation. We also feel that exploiting geometric coherence further might yield improved algorithms in the future.



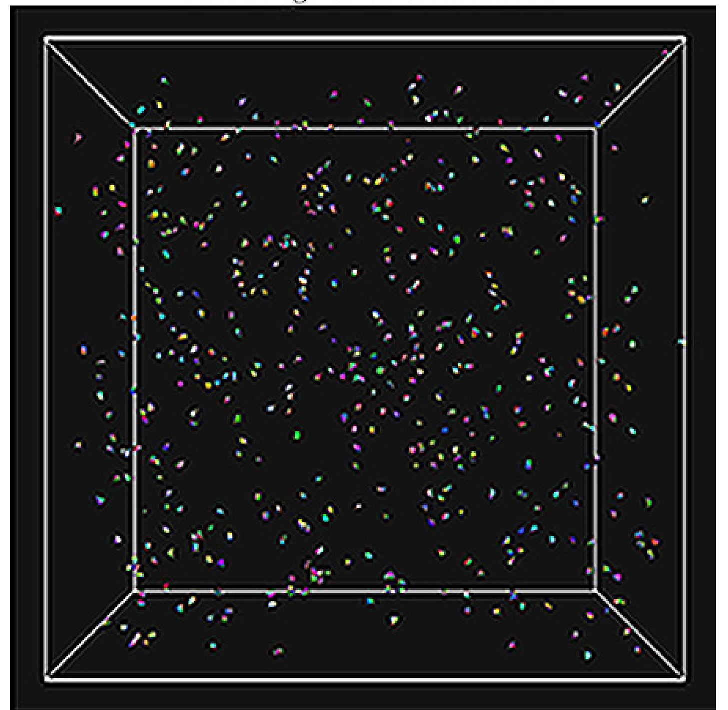
Frame 1: 100 polytopes, 1% density, 56 faces
Pair of bounding boxes overlapping



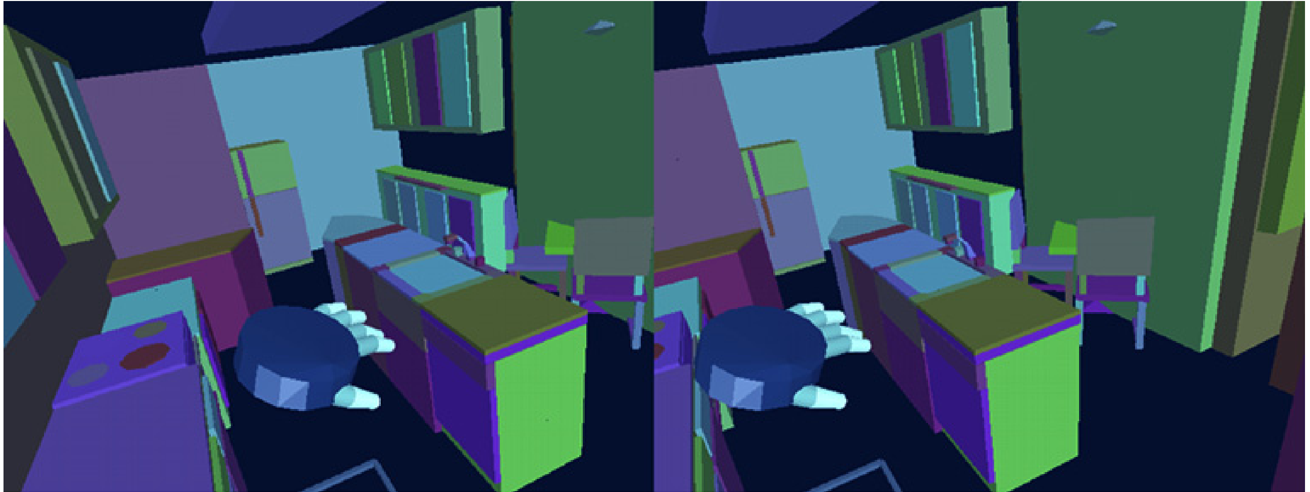
Frame 2: 100 polytopes, 1% density, 56 faces
Bounding boxes not rendered.



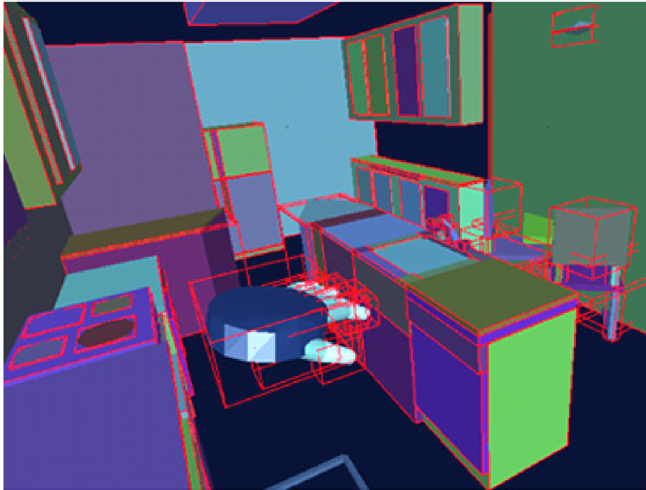
Frame 3: 200 polytopes, 1% density, 56 faces



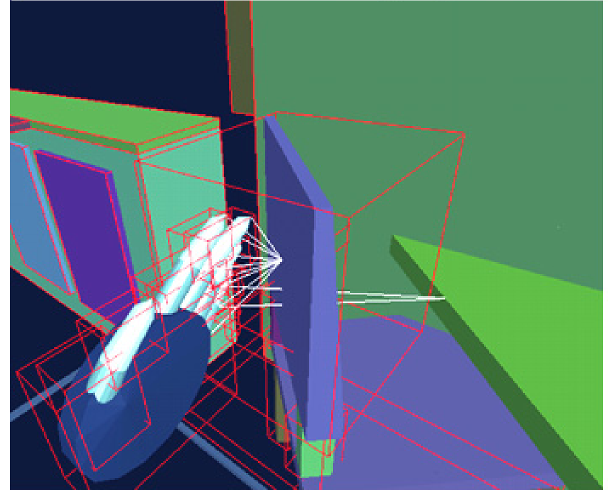
Frame 4: 500 polytopes, 0.1% density, 16 faces



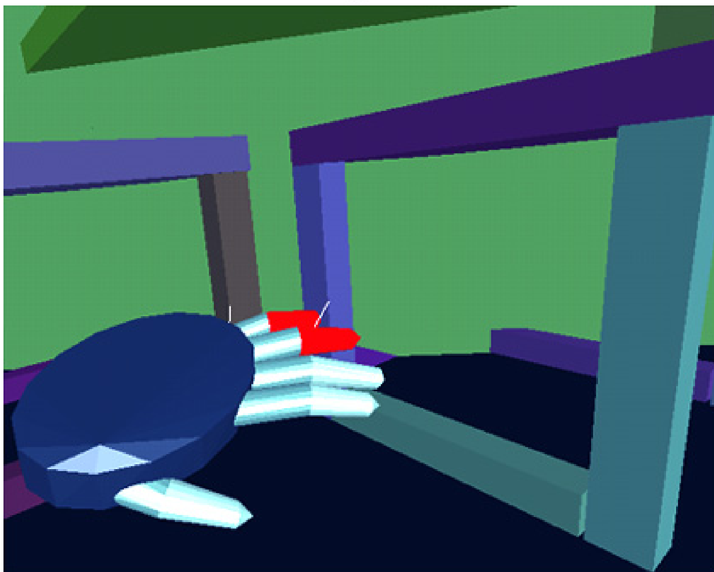
Frame 5: A multi-polytope hand moves through a kitchen walkthrough environment.
The user sees this stereo image pair in a head-mounted display.

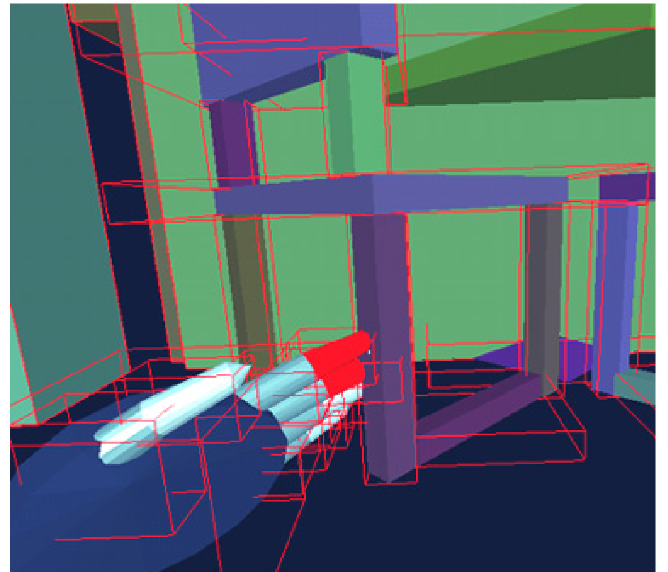


Frame 6: Walkthrough environment with bounding boxes rendered.

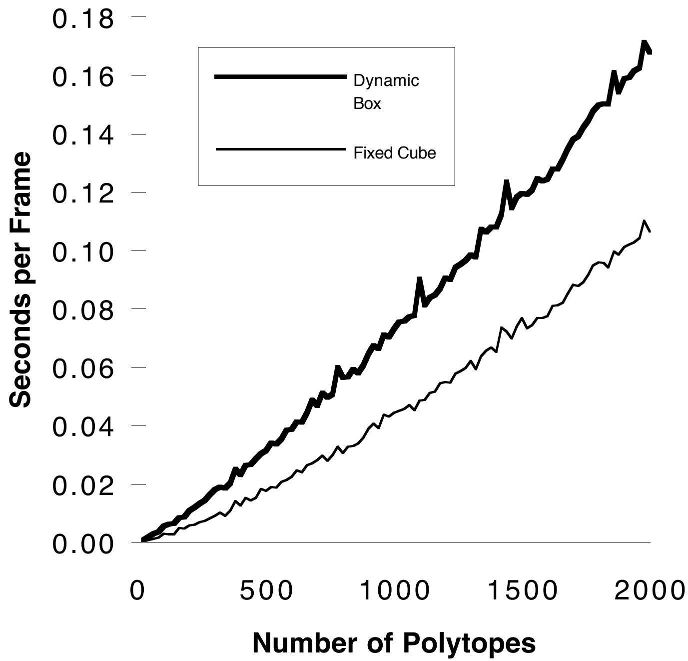


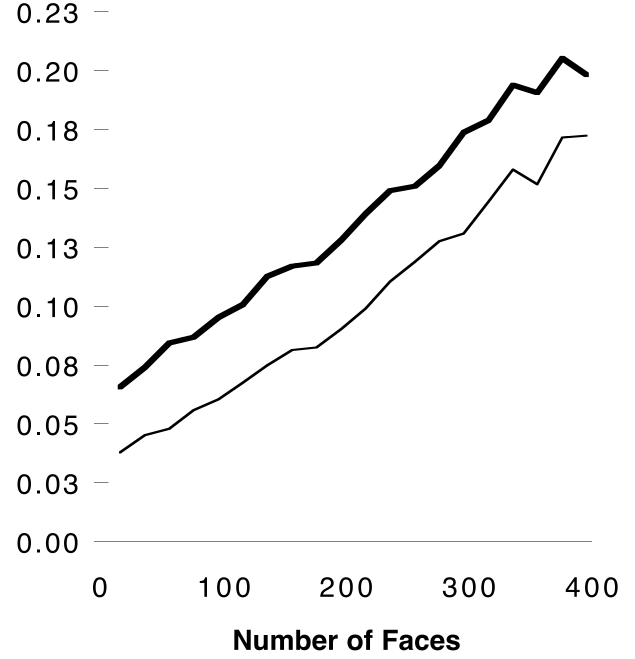
Frame 7: When bounding boxes overlap, closest feature pairs appear.

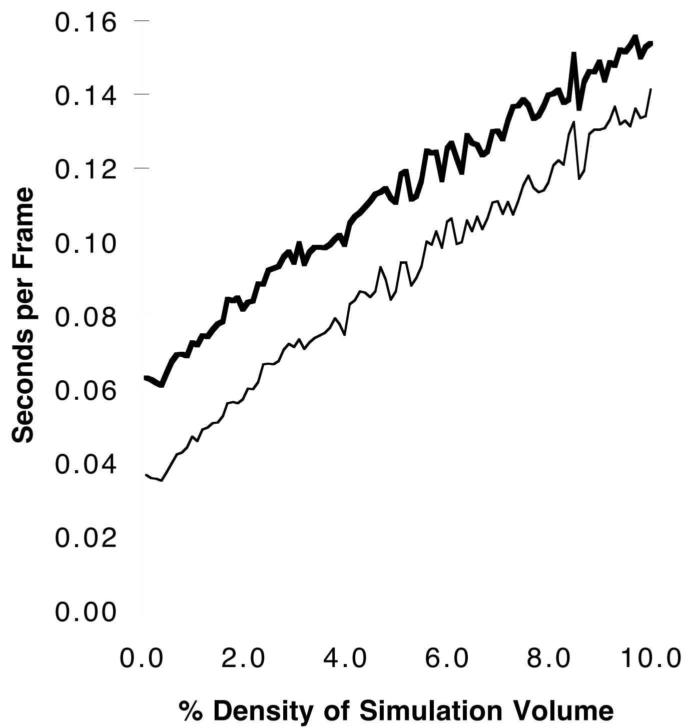


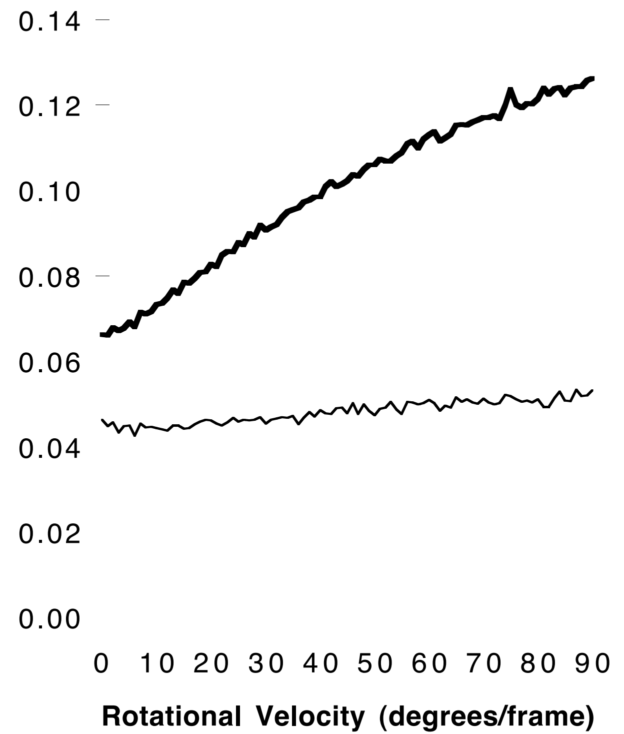


Frames 8 and 9: More examples of the hand interacting with objects in the kitchen.
Red fingers indicate polytopes in collision.


Graph 1
Seconds per Frame
0.18
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
Dynamic Box
Fixed Cube
0
500
1000
1500
2000
Number of Polytopes


Graph 2
0.23
0.20
0.18
0.15
0.13
0.10
0.08
0.05
0.03
0.00
0
100
200
300
400
Number of Faces


Graph 3
Seconds per Frame
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
0.0
2.0
4.0
6.0
8.0
10.0
% Density of Simulation Volume


Graph 4
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
0
10
20
30
40
50
60
70
80
90
Rotational Velocity (degrees/frame)

Graph 5

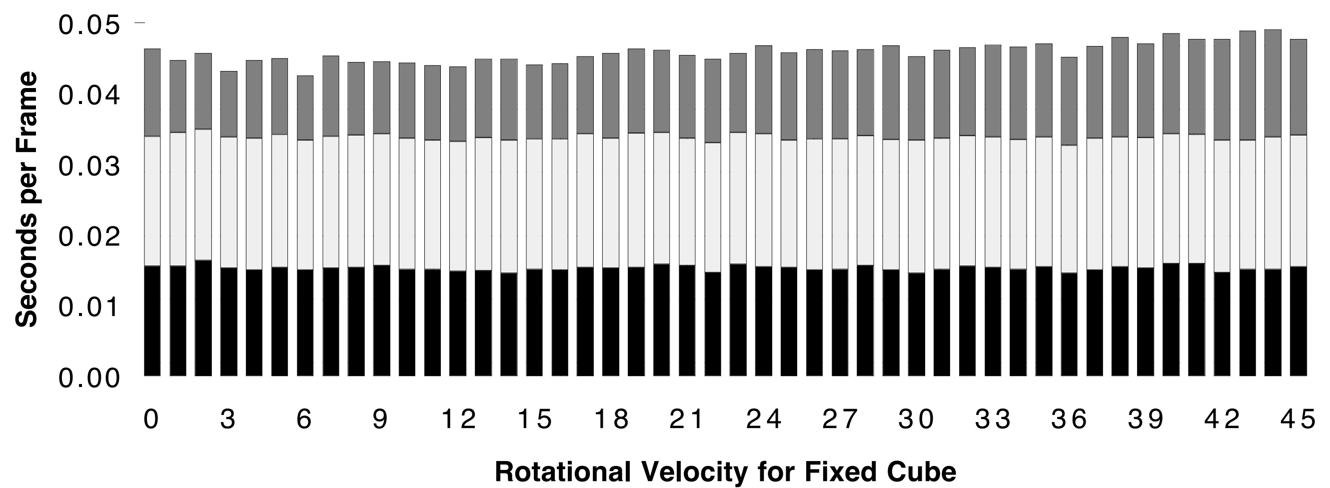
Collision Tests
Sort Lists
Update Boxes

Seconds per Frame: 0.00, 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.10, 0.11

0 3 6 9 12 15 18 21 24 27 30 33 36 39 42 45

Rotational Velocity for Dynamic Box

Graph 6



References

[ARJ90] John M. Airey, John H. Rohlf, and Frederick P. Brooks Jr. Towards image realism with interactive update rates in complex virtual building environments. *ACM SIGGRAPH Special Issue on 1990 Symposium on Interactive 3D Graphics*, 24(2):41–50, 1990.

[AS90] P. K. Agarwal and M. Sharir. Red-blue intersection detection algorithms, with applications to motion planning and collision detection. *SIAM J. Comput.*, 19:297–321, 1990.

[BV91] W. Bouma and G. Vanecek, Jr. Collision Detection and Analysis in Physically Based Simulations Proceedings of the Eurographics Workshop on Animation and Simulation Vienna Austria, pp. 191-203, September, 1991.

[ASF94] A. Garcia-Alonso, N. Serrano, and J. Navarre Solving the Collision Detection Problem *IEEE Computer Graphics and Applications*, pp. 36-43, May, 1994.

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

[BB88] R. Barzel and A. Barr. A modeling system based on dynamic constraints. *ACM Computer Graphics*, 22(4):31–39, 1988.

[BF79] J. L. Bentley and J. H. Friedman. Data structures for range searching. *Computing Surveys*, 11(4), December 1979.

[Cam90] S. A. Cameron. Collision detection by four-dimensional intersection testing. *IEEE Trans. on Robotics and Automation*, 6(3):291–302, 1990.

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

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

[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.

[DK90] D. P. Dobkin and D. G. Kirkpatrick. Determining the separation of preprocessed polyhedra – a unified approach. In *Proc. 17th Internat. Colloq. Automata Lang. Program.*, volume 443 of *Lecture Notes in Computer Science*, pages 400–413. Springer-Verlag, 1990.

[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 of the Fourth Eurographics Workshop on Animation and Simulation*, 1993.

[Ede83] H. Edelsbrunner. A new approach to rectangle intersections, Part I. *Internat. J. Comput. Math.*, 13:209–219, 1983.

[ETHA87] M. Edahiro, K. Tanaka, R. Hoshino, and Ta. Asano. A bucketing algorithm for the orthogonal segment intersection search problem and its practical efficiency. In *Proc. 3rd Annu. ACM Sympos. Comput. Geom.*, pages 258–267, 1987.

[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.

[Hah88] J. K. Hahn. Realistic animation of rigid bodies. *Computer Graphics*, 22(4):pp. 299–308, 1988.

[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.

[Hof89] C.M. Hoffmann. *Geometric and Solid Modeling.* Morgan Kaufmann, San Mateo, California, 1989.

[HSS83] J.E. Hopcroft, J.T. Schwartz, and M. Sharir. Efficient detection of intersections among spheres. *The International Journal of Robotics Research*, 2(4):77–80, 1983.

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

[KWBB93] Kass, Witkin, Baraff, and Barr. An introduction to physically based modeling. Course Notes 60, 1993.

[Lat91] J.C. Latombe. *Robot Motion Planning.* Kluwer Academic Publishers, 1991.

[LC92] M.C. Lin and John F. Canny. Efficient collision detection for animation. In *Proceedings of the Third Eurographics Workshop on Animation and Simulation*, 1992. Cambridge, England.

[Lev66] C. Leventhal. Molecular model-building by computer. *Scientific American*, 214(6), June 1966.

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

[LM93] M.C. Lin and Dinesh Manocha. Interference detection between curved objects for computer animation. In *Models and Techniques in Computer Animation*, pages 43–57. Springer-Verlag, 1993.

[LPW79] T. Lozano-Pérez and M. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Comm. ACM*, 22(10):pp. 560–570, 1979.

[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.

[Ove92] M. H. Overmars. Point location in fat subdivisions. *Inform. Proc. Lett.*, 44:261–265, 1992.

[Pen90] A. Pentland. Computational complexity versus simulated environment. *Computer Graphics*, 22(2):185–192, 1990.

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

[PW90] A. Pentland and J. Williams. Good vibrations: Modal dynamics for graphics and animation. *Computer Graphics*, 23(3):185–192, 1990.

[SML90] A. Schmitt, H.M. Muüller, and W. Leister Ray tracing algorithms: theory and practice Theoretical Foundations of Computer Graphics and CAD vol. F40, pages 997-1030, Springer-Verlag 1988.

[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.

[SH76] M. I. Shamos and D. Hoey. Geometric intersection problems. In *Proc. 17th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 208–215, 1976.

[Sha87] M. Sharir. Efficient algorithms for planning purely translational collision-free motion in two and three dimensions. In *Proc. IEEE Internat. Conf. Robot. Autom.*, pages 1326–1331, 1987.

[SW82] H. W. Six and D. Wood. Counting and reporting intersections of d-ranges. *IEEE Trans. on Computers*, C-31(No. 3), March 1982.

[SWF+93] J. Snyder, A. Woodbury, K. Fleischer, B. Currin, and A. Bar. Interval methods for multi-point collisions between time dependent curved surfaces. In *Proceedings of ACM Siggraph*, pages 321–334, 1993.

[TN87] W. Thibault and B. Naylor. Set operations on polyhedra using binary space partitioning trees. *ACM Computer Graphics*, (4), 1987.

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

[Zel92] D. Zeltzer. Autonomy, interaction and presence. *Presence*, 1(1):127, 1992.