I-COLLIDE: An Interactive and Exact Collision Detection System for Large-Scale Environments

Jonathan D. Cohen Ming C. Lin * Dinesh Manocha Madhav K. Ponamgi
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175
{cohenj,lin,manocha,ponamgi}@cs.unc.edu

ABSTRACT:
We present an exact and interactive collision detection system, I-COLLIDE, for large-scale environments. Such environments are characterized by the number of objects undergoing rigid motion and the complexity of the models. The algorithm does not assume the objects' motions can be expressed as a closed form function of time. The collision detection system is general and can be easily interfaced with a variety of applications. The algorithm uses a two-level approach based on pruning multiple-object pairs using bounding boxes and performing exact collision detection between selected pairs of polyhedral models. We demonstrate the performance of the system in walkthrough and simulation environments consisting of a large number of moving objects. In particular, the system takes less than 1/20 of a second to determine all the collisions and contacts in an environment consisting of more than a 1000 moving polytopes, each consisting of more than 50 faces on an HP-9000/750.

1 INTRODUCTION

Collision detection is a fundamental problem in computer animation, physically-based modeling, computer simulated environments and robotics. In these applications, an object's motion is constrained by collisions with other objects and by other dynamic constraints. The problem has been well studied in the literature. However, no good general collision detection algorithms and systems are known for interactive large-scale environments.

A large-scale virtual environment, like a walkthrough, creates a computer-generated world, filled with real and 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. However, there may be hundreds, even thousands of objects in the virtual world, so a brute-force approach that tests all possible pairs for collisions is not acceptable. Efficiency is critical in a virtual environment, otherwise its interactive nature is lost [24]. A fast and interactive collision detection algorithm is a fundamental component of a complex virtual environment.

The objective of collision detection is to report all 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. Due to these unconstrained variables, collision detection is currently considered to be one of the major bottlenecks in building interactive simulated environments [20].

Main Contribution: We present a collision detection algorithm and system for interactive and exact collision detection in complex environments. In contrast to the previous work, we show that accurate, interactive performance can be attained in most environments if we use coherence to speed up pairwise interference tests and to reduce the actual number of these tests we perform. We are able to successfully trim the $O(n^2)$ possible interactions of n simultaneously moving objects to $O(n+m)$ where m is the number of objects *very close* to each other. In particular, two objects are very close, if their axis-aligned bounding boxes overlap. Our approach is flexible enough to handle dense environments without making assumptions about object velocity or acceleration. The system has been successfully applied to architectural walkthroughs and simulated environments and works well in practice.

The rest of the paper is organized as follows. In Section 2, we review 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. We describe our algorithm for collision detection between multiple objects in Section 4 and discuss its implementation in Sections 5 and 6. Section 7 presents our experimental results on walkthrough environments and simulations.

*Currently at NC A & T State University, Greensboro. Approved by ARPA for public release; distribution unlimited

2 PREVIOUS WORK

The problem of collision detection has been extensively studied in robotics, computational geometry, and computer graphics. The goal in robotics has been the planning of collision-free paths between obstacles [15]. 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 [1, 3, 11, 18, 20, 21].

At the same time, the emphasis in the computational geometry has been on theoretically efficient intersection detection algorithms [22]. Most of them are restricted to a static instance of the problem and are non-trivial to implement. For convex 3-polytopes [1] linear time algorithms based on linear programming and tracking closest points [10] 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 [3, 17]. Alonso et al.[1] use bounding boxes and spatial partitioning to test all $O(n^2)$ pairs of arbitrary polyhedral objects.

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. 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. Another approach operates directly on four-dimensional volumes swept out by object motion over time [4, 14].

None of these algorithms adequately address the issue of collision detection in a virtual environment which requires performance at interactive rates for thousands of pairwise tests. Hubbard has proposed a solution to address this problem by trading accuracy for speed [14]. In an early extension of their work, Lin and Canny [16] proposed a scheduling scheme to handle multiple moving objects. Dworkin and Zeltzer extended this work for a sparse model [7].

3 BACKGROUND

In this section, we highlight the importance of coherence in dynamic environments. We briefly review the algorithm for exact pairwise collision detection and present our multi-body collision detection scheme, both of which exploit coherence to achieve efficiency.

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.

[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.

3.2 Pairwise Collision Detection for Convex Polytopes

We briefly review the Lin-Canny collision detection algorithm which tracks closest points between pairs of convex polytopes [16, 17]. This algorithm is used at the lowest 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 running time is proportional to the number of feature pairs traversed, which is a function of the relative motion the polytopes undergo). The method for finding closest feature pairs is based on Voronoi regions. The algorithm starts with a candidate pair of features, one from each polytope, and checks whether the closest points lie on these features. Since the polytopes and their faces are convex, this is a local test involving only the neighboring features of the current candidate features. If either feature fails the test, the algorithm steps to a neighboring feature of one or both candidates, and tries again. With some simple preprocessing, the algorithm can guarantee that every feature has a constant number of neighboring 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. 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 at times lead to cycling of feature pairs. To circumvent this problem, we partition the *interior space* of the convex polytopes. The partitioning does not have to form the exact internal Voronoi regions, because we are not interested in knowing the closest features between two interpenetrating polytopes, but only detecting such a case. So instead we use pseudo-Voronoi regions, obtained by joining each vertex of the polytope with the centroid of the polytope [21].

Given a partition of the exterior and the interior of the polytope, we walk from the external Voronoi regions into the pseudo-internal 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-internal Voronoi regions requires special case analysis and will be omitted here.

3.4 Extension to Non-Convex Objects

We 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 or features [21].

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. For instance, a hand may have individual joints in the leaves, fingers in the internal nodes, and the entire hand in the root.

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 returns the closest feature pair of their bounding volumes. If there is a collision, the algorithm expands their children and recursively proceeds down the tree to determine if a collision actually occurs. More details are given in [21].

4 MULTIPLE-OBJECT COLLISION DETECTION

Large-scale environments consist of stationary as well as moving objects. Let there be 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 and M get 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. 1.

Sorting is the key to our pruning approach. Each object is surrounded by a 3-dimensional bounding volume. We sort these bounding volumes in 3-space to determine which pairs are overlapping. We only need to perform exact pairwise collision tests on these 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. Based on this observation, we choose axis-aligned bounding boxes as our bounding volumes. We 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 small or large.

4.1 Bounding Volumes

Many collision detection algorithms have used bounding boxes, spheres, ellipses, etc. to rule out collisions between objects which are far apart. We use bounding box overlaps to trigger the *exact collision detection* algorithm.

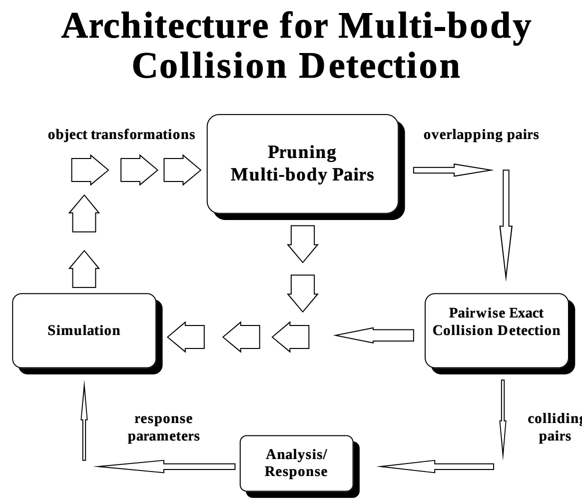


Figure 1: Architecture for Multiple Body Collision Detection Algorithm

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 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 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

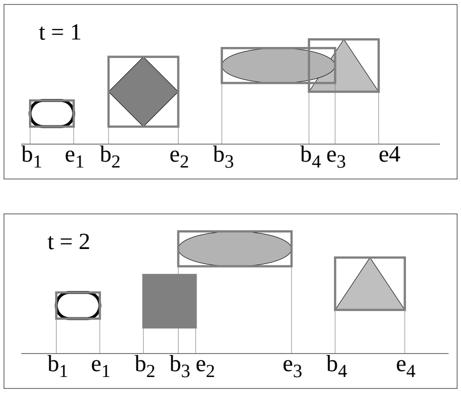


Figure 2: Bounding Box Behavior

walkthrough environment. In such an environment, the savings gained by the reduced number of pairwise collision detection tests outweigh the cost of computing the dynamically-resized boxes.

As a precomputation, we compute each object's initial minima and maxima along each axis. It is assumed that the objects are convex. For non-convex polyhedral models, the following algorithm is applied to their convex hulls. As an object moves, we recompute its minima and maxima at each time step as follows:

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 do not transform all the vertices as the objects undergo motion. As we are updating the bounding boxes new positions are computed for current vertices using matrix-vector multiplications. 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 than coordinates in order to compare neighboring vertices. This reduces the number of arithmetic operations by two-thirds.

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. 2). 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 $O(n \log n)$ time, where n is the number of objects. We can reduce this time bound by keeping the sorted lists from the previous frame, changing only the 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, as shown in [19, 3]. *Insertion sort* works 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 in the worst case $e_x, e_y,$ 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 3-D bounding box overlaps in $O(n \log^2 n + s)$ time, where s is the number of pairwise overlaps [12, 13]. We have reduced this to $O(n + s)$ by using coherence.

4.3 Two-Dimensional Intersection Tests

The two-dimensional intersection algorithm begins by projecting each three-dimensional axis-aligned bounding box onto any two of 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. In situations where the projections onto one-dimension result in densely clustered intervals, the two-dimensional technique is more efficient. The interval tree is a common data structure for performing such two-dimensional range queries [22].

Each query of an interval intersection takes $O(\log n + k)$ 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 $O(n \log n + K)$ where K is the total number of intersecting rectangles [8].

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 [23], octrees, etc.

Several practical and efficient algorithms are based on uniform space division. Divide space into unit cells (or volumes) and place each object in some cell(s). To check for collisions, examine the cell(s) occupied by each object to verify if the cell(s) is(are) shared by other objects. Choosing a near-optimal cell size is difficult, and failing to do so results in large memory usage and computational inefficiency.

5 IMPLEMENTATION

In this section we describe the implementation details of I-COLLIDE based on the Sweep and Prune algorithm, the exact collision detection algorithm, the multi-body simulation, and their applications to walkthrough and simulations.

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.

Each 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 of coordinate values using insertion sort, while maintaining an overlap status for each bounding box pair. 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 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 [7] suggest precomputing a lookup table for each polytope to help find better starting guesses.

5.3 Multi-body Simulation

The multi-body simulation is an application we developed to test the I-COLLIDE system. 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 range from oblong to fat. The simulation parameters of the polytopes were their number, their complexity measured as the number of faces, their rotational velocity, their translational velocity, the density of their environment measured as the ratio of polytope volume to environment volume, and the bounding volume method used for the Sweep and Prune (fixed-size 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 I-COLLIDE system and receives a list of colliding polytope pairs. It exchanges the translational velocities of these pairs to simulate an elastic reaction. Objects also rebound off the walls of the constraining volume.

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 Frame 1 at the end for an example of the simulation.

5.4 Walkthrough

The walkthrough is a head-mounted display application that involves a large number of polytopes depicting a realistic scene. 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 (our video shows a kitchen and 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 2-4 show a sequence of shots from a kitchen walkthrough environment. The pictures show images as seen by the left eye. Frames 5-6 show the user in a porch walkthrough.

6 SYSTEM ISSUES

To use I-COLLIDE, the application first loads a library of polytopes. The file format we use is fairly simple. It is straightforward to convert polytope data from some other format (perhaps the output of some 3D modelling package) to this minimal format for I-COLLIDE. After loading the polytopes, the application then chooses some polytope pairs to activate for collision detection. This set of active pairs is fully configurable between collision passes. Inside the application loop, the application informs I-COLLIDE of the world transformation for each polytope as it moves around. At any point, the application may call the collision test routine. I-COLLIDE returns a list of all the colliding pairs, including a pair of colliding features for each. The application then responds to these collisions in some appropriate way.

6.1 Space Issues

For each pair of objects, I-COLLIDE maintains a structure that contains the bounding box overlap status and the closest feature pair between the objects. These structures conceptually form an upper-triangular $O(n^2)$ matrix. We access an entry in O(1) time by using the object id numbers as (row, column) entries. If only a few pairs of objects are interacting, then the $O(n^2)$ can be reduced at the expense of slightly larger access time. For example, we can traverse a sparse matrix list to access an entry.

6.2 Geometric Robustness

In practice there are several types of degeneracies or errors that can occur in the convex polytope models: duplicate vertices, extraneous vertices, backfacing polygons, tracking error, non-planar faces, non-convex faces, non-convex polytopes, disconnected faces, etc. We have written a pre-processor to scan for common degeneracies and correct them when possible.

6.3 Numerical Issues

Numerical robustness is an important issue in the exact collision detection code. There are many special case geometrical tests in this module, and it is difficult to ensure that the algorithm will not get into a cycle due to degenerate overlap. We deal with this by performing all of our feature tests to some tolerance. Without such a tolerance, floating point errors might allow some of the feature tests to cycle infinitely. We have not observed this in practice so far, and have been careful to make the tests stable in the presence of small errors.

The multi-body sweep and prune code is also designed to resist small numerical errors. The bounding box of each polytope is extended by a small epsilon [2] in each direction. In addition to insulating the overlap tests from errors, this precaution also helps give the exact collision detection test a chance of being activated before the objects are actually penetrating.

[2] This quantity is a function of velocity between the object pairs.

6.4 Generality

While the multi-body pruning code works well with the exact collision detection routine, it functions independently of the underlying collision detection routine. This second level collision routine might or might not be exact, and it certainly need not be limited to handling convex polytopes.

7 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 we use two different types of bounding boxes. Using fixed cubes as bounding boxes resulted in low collision time for the parameter ranges we tested.

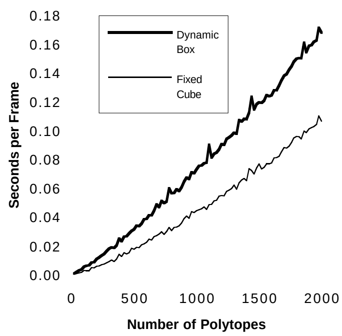
In each of the first four graphs, we plot two lines. The bold line displays the performance of using dynamically-resized bounding boxes whereas the other line shows the performance of using fixed-size cubes. All five graphs refer to "seconds per frame", where a frame is one step of the simulation, involving one iteration of collision detection without rendering time. 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%.

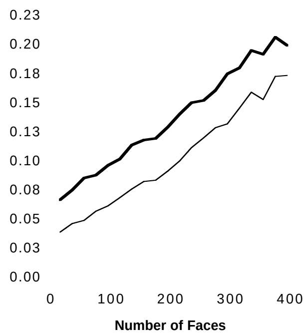
In the graphs, the timing results do not include computing each polytope's transformation matrix, rendering times, and of course any minor initialization cost. We ignored these costs, because we wanted to measure the cost of collision detection alone.

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

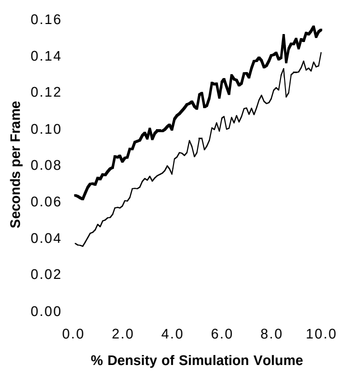
Graph 1



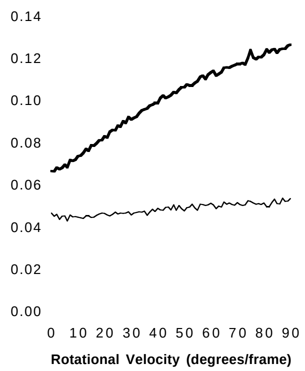
Graph 2



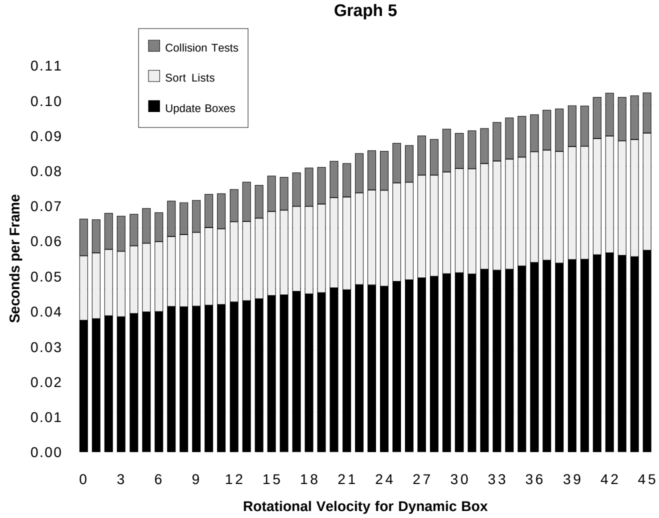
Graph 3



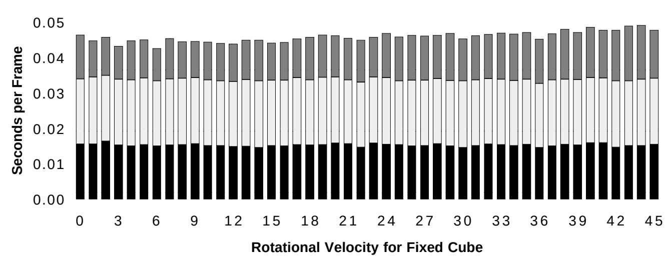
Graph 4



Graph 5



Graph 6



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 [7] as well as those of Hubbard [14]. For a 1000 polytopes in our simulation, our collision time results in **23 frames per second** using the fixed bounding cubes.

Graph 2 shows how the number faces affects the collision 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. 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.

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 overall collision time scales well with the increases in density.

Graphs 4 through 6 show the effect of rotational velocity on the overall collision 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 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 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 *exact* collision detection in demanding environments can be achieved without incurring expensive time penalties. The architectural walkthrough models showed no perceptible performance degradation when collision detection was added (as in Frame 2 to 5).

8 CONCLUSION

Collision detection has been considered a major bottleneck in computer-simulated environments. By making use of geometric and temporal coherence, our algorithm and system detects collisions more efficiently and effectively than earlier algorithms. Under many circumstances our system produces collision frame rates over 20 hertz for environments with over a 1000 moving complex polytopes. Our walkthrough experiments showed no degradation of frame rates when collision detection was added. We are currently working on incorporating general polyhedral and spline models into our system and extending these algorithms to deformable models.

9 ACKNOWLEDGEMENTS

We are grateful to John Canny and David Baraff for productive discussions and to Brian Mirtich for his help in implementation of the convex polytope pair algorithm. The kitchen and porch models used in the walkthrough applications were designed by the UNC Walkthrough group, headed by Fred Brooks. This work was supported in part by DARPA ISTO order A410, NSF grant MIP-9306208, NSF grant CCR-9319957, ARPA contract DABT63-93-C-0048, ONR contract N00014-94-1-0738 and NSF/ARPA Science and Technology Center for Computer Graphics and Scientific Visualization, NSF Prime Contract 8920219.

References

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

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

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

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

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

[6] J. Cohen, M. Lin, D. Manocha, and K. Ponamgi. Interactive and exact collision detection for large-scaled environments. Technical Report TR94-005, Department of Computer Science, University of North Carolina, 1994.

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

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

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

[10] 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.

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

[12] 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.

[13] H.Six and D.Wood. Counting and reporting intersections of D-ranges. *IEEE Transactions on Computers*, pages 46–55, 1982.

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

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

[16] M. Lin and J. Canny. Efficient collision detection for animation. In *Proceedings of the Third Eurographics Workshop on Animation and Simulation*, Cambridge, England, 1991.

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

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

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

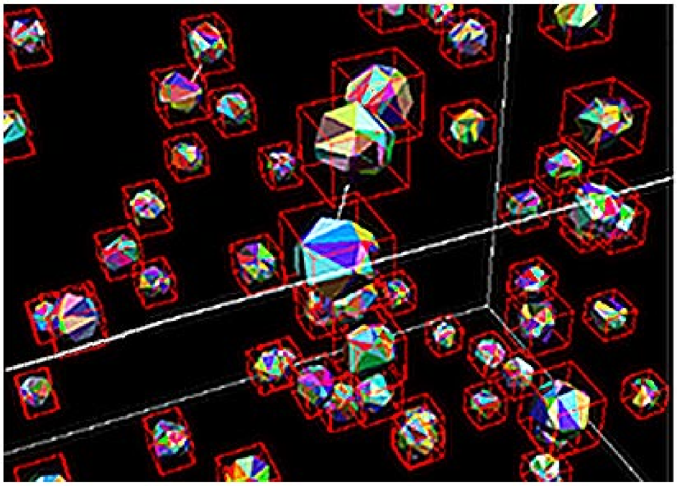
[20] A. Pentland. Computational complexity versus simulated environment. *Computer Graphics*, 22(2):185–192, 1990.

[21] M. Ponamgi, D. Manocha, and M. Lin. Incremental algorithms for collision detection between solid models. Technical Report TR94-061, Department of Computer Science, University of North Carolina, Chapel Hill, 1994.

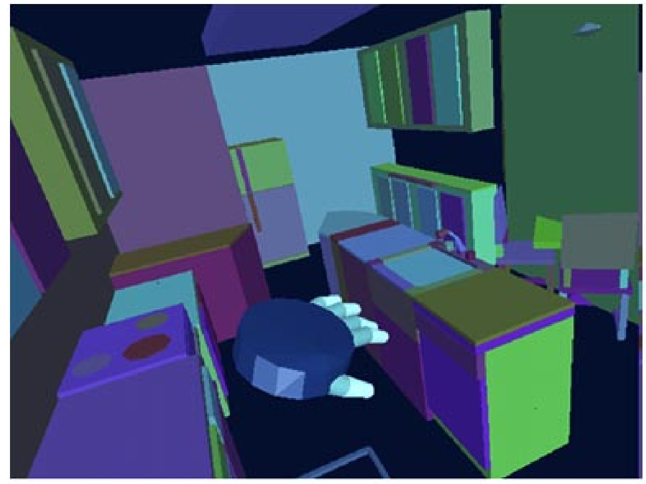
[22] F.P. Preparata and M. I. Shamos. *Computational Geometry.* Springer-Verlag, New York, 1985.

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

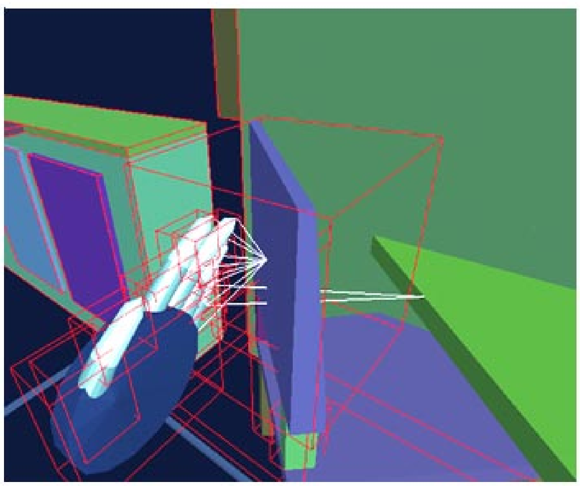
[24] D. Zeltzer. Autonomy, interaction and presence. *Presence*, 1(1):127, 1992.



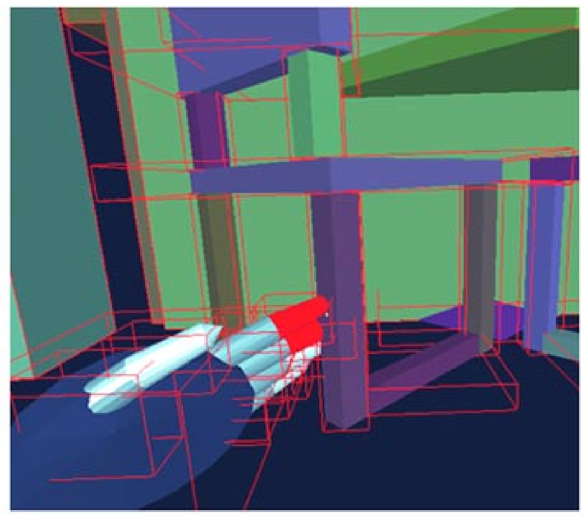
Frame 1: 100 polytopes, 1% density, 56 faces.
Pair of bounding boxes overlapping.



Frame 2: A multi-polytope hand moves through a kitchen walkthrough environment.



Frame 3: When bounding boxes overlap, closest feature pairs appear.



Frame 4: Red polytopes indicate collisions.

Frames 5 and 6: The hand touches a swing in a porch walkthrough.