MERL – A MITSUBISHI ELECTRIC RESEARCH LABORATORY
http://www.merl.com

# Efficient Algorithms for Two-Phase Collision Detection

Brian Mirtich

TR-97-23 December 1997

**Abstract**

This article describes practical collision detection algorithms for robot motion planning. Attention is restricted to algorithms that handle rigid, polyhedral geometries. Both broad phase and narrow phase detection strategies are discussed. For the broad phase, an algorithm using axes-aligned bounding boxes and a hierarchical spatial hash table is described. For the narrow-phase, the Lin-Canny algorithm is presented. Alternatives to these algorithms are also discussed. Finally, the article describes a scheduling paradigm for managing collision checks that can further reduce computation time. Pointers to downloadable software are included.

*To appear in* Practical Motion Planning in Robotics: Current Approaches and Future Directions, *K. Gupta and A.P. del Pobil, editors.*

1. First printing, TR97-23, December 1997

# 1 Introduction

Collision detection algorithms are fundamental in robot motion planning. They serve different purposes depending on the application. In fine motion planning, collision detection algorithms determine the time and location of contacts, a necessary step in computing contact forces. In gross motion planning, collision detection algorithms are used to report the distances between objects, with the aim of constructing feasible motion paths, or keeping the robot maximally clear from obstacles, or building maps of the environment. In any case, collision detection is a low level process in a motion planning system, which must execute before higher level processes can proceed. For this reason, collision detection algorithms must be fast. In the context of rigid-body simulation, Hahn found collision detection often required over 95% of the computation time [12]. Algorithms have improved since then, however, they remain a bottleneck in many situations.

Collision detection algorithms have roots in computational geometry, where the basic problem is to report intersections among a group of static objects. In motion planning, the problem is more difficult because the objects are moving. Their trajectories might be closed form functions of time, as in the case of a kinematically controlled manipulator. Alternatively, the trajectories might be described through the differential equations of Newtonian dynamics.

This article describes some practical algorithms for collision detection, presenting both theoretical and empirical results. To limit the scope, we restrict attention to situations with rigid, polyhedral objects. These restrictions allow significant optimizations and are tolerable for many motion planning applications. In particular, the polyhedral restriction is a mild one since more general shapes, such as parametric surfaces or CSG-style solids, can be approximated to arbitrary closeness with polyhedral models. The algorithms discussed here are quite insensitive to the complexity of the polyhedra. We also restrict attention to *exact* algorithms: those that ultimately use the underlying polyhedral models in their computations. The alternatives are *approximate* methods, which are less concerned with the true geometry and more concerned with real-time performance.[1] A good reference list of work in collision detection is in [14].

## 1.1 Coherence and locality

Early collision detection algorithms for moving polyhedral objects solved problem instances from scratch at every time step. These algorithms required running time that was quadratic in the complexity of the polyhedra to determine if the objects were disjoint [21, 12]; they are too slow for many motion planning applications. Some early linear time algorithms were also proposed [10].

The biggest improvement over these early algorithms came from the use of *coherence*: the fact that the collision detection system solves a series of related problems, each one only slightly different than the one before. This is the usual

[1] See, for example, Hubbard's work on collision detection for time-critical applications using bounding sphere hierarchies [15].

case in motion planning as well as dynamic simulation. Caching results from previous invocations greatly reduces the amortized cost of collision detection. Gilbert, *et. al.* describe an adaptation of their algorithm to take advantage of coherence when it exists [10]. Coherence is also used in Baraff's witness plane algorithm [3], Cameron's enhancement of the Gilbert algorithm [4], the Lin-Canny closest features algorithm [18, 17], and the *V-Clip* algorithm [20]. Use of *locality* often goes hand-in-hand with coherence in many algorithms. This property allows an algorithm to verify that the cached result is still valid using only local geometric tests. Locality usually relies on convexity of the objects. Coherence and locality combine to form *almost constant time* algorithms for verifying disjointness or computing the distance between objects.

### 1.2 Two-phase collision detection

When there are $n$ distinct objects in a motion planning problem, collision detection must be performed between $O(n^2)$ different pairs of objects. Most efficient collision detection systems use a two-phase approach to reduce the computational costs of making these checks. The *broad phase* culls away most of the pairs using a trivial rejection test based on bounding boxes, bounding spheres, octrees, or the like. The culled pairs fall below a relevance threshold for the higher level planning processes. The pairs that survive this culling are passed to the *narrow phase*, which uses a more refined algorithm to check for collisions or to compute distances. Since the broad phase basically acts as a filter for the narrow phase, choices for the two algorithms can usually be made independently.

This article describes collision detection from a two-phase perspective, addressing the broad phase in Section 2 and the narrow phase in Section 3. In both cases, the article focuses on one particular approach, but briefly mentions alternatives and their relative merits. Section 4 describes a novel scheduling paradigm for invoking the collision detection system that has advantages over the more traditional scheme. Many of the algorithms discussed have implementations in the public domain. Section 5 contains pointers to downloadable code.

It is impossible to recommend collision detection algorithms that are best for all applications. The ones emphasized here were effectively used in the *Impulse* simulation system, described in detail in [19]. Impulse places demanding requirements on the collision detection system that are similar to the requirements of many motion planning systems.

## 2 Broad phase collision detection

The function of the broad phase collision detection algorithm is to quickly remove most of the pairs of objects from consideration. Axes-aligned bounding boxes provide a simple but effective approach for many applications. Boxes can be used to bound the region of space occupied by an object at a single instant in time, but when the objects are moving, it is more useful to bound the region of

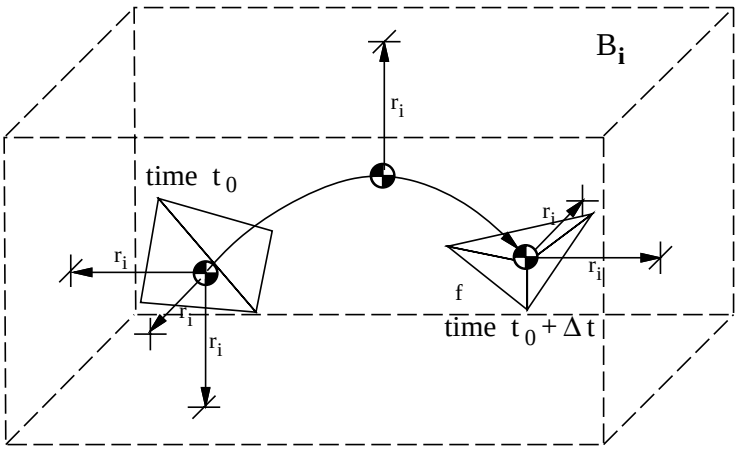


Figure 1: The bounding box for object $i$'s swept volume during a segment of a parabolic trajectory.

space occupied by the object over a time interval. This *swept volume* approach is easiest when objects follow simple trajectories. Consider finding a bounding box $B_i$ that encloses object $i$ over the time interval $[t_0, t_0 + \Delta t]$. Let $r_i$ be the maximum distance of any point on the object from its center of mass, and suppose the center of mass follows a parabolic trajectory over the interval in question. $B_i$ is found by noting the position of the center of mass at the time $t_0$, at the time $t_0 + \Delta t$, and possibly at the apex of its parabolic trajectory, should this occur during the interval $[t_0, t_0 + \Delta t]$. The box which bounds these two or three points is grown by $r_i$ to obtain $B_i$ (Figure 1). In this case, computing $B_i$ is simple and takes constant time.

Other situations may not afford such a simple approach to bounding the swept volume. One example is bounding the swept volume occupied by the end effector of a serial chain manipulator with several links. In these situations, one can use discretized points along the trajectory to compute a bounding box; these points may be available from an integration routine at no extra cost. Larger safety margins can be used to compute the box when there are larger gaps between intermediate points. In motion planning applications, many of the objects are often fixed in space. For these objects, the tightest axes-aligned enclosing box can be computed once for all time.

If boxes $B_i$ and $B_j$ do not intersect, then the corresponding objects $i$ and $j$ do not penetrate over the time interval $[t_0, t_0 + \Delta t]$. Object pairs with intersecting bounding boxes are passed to the narrow phase collision detector for further analysis. There are different ways to find intersections among the set of swept-volume bounding boxes. We first describe an algorithm based on a hashing scheme for finding intersections among a set of static boxes in space. Next, we modify the algorithm to exploit coherence among moving boxes. We also

discuss alternative methods based on coordinate sorting.

## 2.1 Finding static box intersections

Point location problems occur frequently in computational geometry. One variant is expressed as follows:

**Problem 1 (Point Location)** *Given a number of non-intersecting cells in space, store the arrangement such that for a given query point p, the cell containing p (if any) can be determined efficiently.*

Here, a *cell* is a connected region of space. Overmars presents two solutions to this problem under certain restrictions in the cell shape [23]. The more efficient solution involves surrounding each cell by an axes-aligned bounding box, and storing the location of these boxes in a hash table. The technique can be extended to solve a more useful problem for collision detection:

**Problem 2 (Static box intersections)** *Given n axes-aligned rectangular boxes $B_1, \ldots, B_n$, fixed in space, store this arrangement such that all pairs of boxes that intersect can be determined efficiently.*

To attack this problem, consider partitioning space into a cubical tiling with resolution $\rho$. Any point $(x, y, z)$ in space belongs to a unique tile, specified by integer coordinates, under the tiling map $\tau$:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \xrightarrow{\tau} \begin{bmatrix} \lfloor x/\rho \rfloor \\ \lfloor y/\rho \rfloor \\ \lfloor z/\rho \rfloor \end{bmatrix} \tag{1}$$

The tiles that the box $B_i$ intersects are found by computing the images under $\tau$ of two of $B_i$'s corners: the one of minimum $x$, $y$, and $z$ coordinates, and the one of maximum $x$, $y$, and $z$ coordinates. The two tiles containing these corners, and the other tiles "between" them, are the tiles that intersect $B_i$. A tile with coordinates $(a, b, c)$ is between the two tiles with coordinates $(a^-, b^-, c^-)$ and $(a^+, b^+, c^+)$ if and only if $a^- \leq a \leq a^+$, $b^- \leq b \leq b^+$, and $c^- \leq c \leq c^+$. There are an infinite number of tiles in unbounded space, but only a finite number that are intersected by at least one box. For each tile that a box intersects, the box's label is stored in the hash table, hashed under the tile's integer coordinates. When two different labels are stored in the same hash bucket, the corresponding boxes are marked as possibly intersecting.

How to choose the tiling resolution $\rho$ is not obvious, and in fact this can be problematic when the sizes of the boxes vary widely. If $\rho$ is small, the larger boxes may intersect a huge number of tiles, and thus require a large amount of storage in the hash table. In addition, when the static assumption is relaxed, updating the positions of these large, moving boxes would be inefficient. On the other hand, if $\rho$ is large, the tiling will have poor resolution power for the smaller boxes. Many small boxes may hash to the same tile, so the more expensive box intersection test will be performed on many pairs. Overmars solves this problem

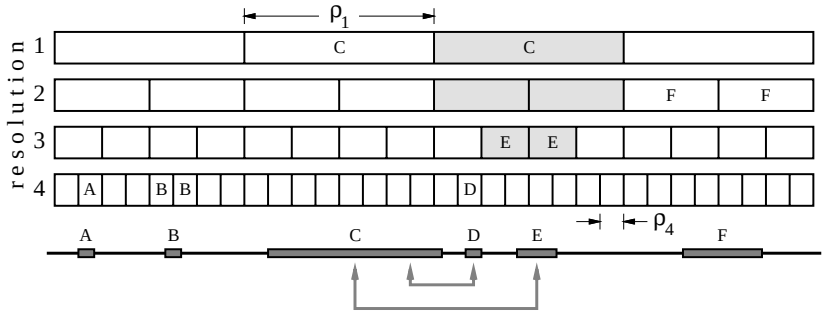


Figure 2: A one-dimensional example of a hierarchical spatial hash table, with four resolutions. The one-dimensional boxes are shown at the bottom, labeled $A$-$F$. The value $\rho_i$ is the size of the tiles at resolution $i$. Boxes are stored in the hash table in order of increasing resolution. The cells which must be checked when box $E$ is stored in the table are shaded. This hash table verifies that all boxes are disjoint except for the pairs $(C, D)$ and $(C, E)$.

by partitioning the set of boxes into groups of similar size, and creating one hash table for each group. For collision detection, however, each of the boxes must be checked for intersection with all others—this is not true in the point location problem—and so partitioning boxes into disjoint groups is not helpful. The solution is to build a hierarchical hash table, comprising several resolutions, and checking for intersections among boxes at different resolutions.

To understand the method, consider the one dimensional example, shown in Figure 2. Here, the six "boxes" (that is, line segments) populating space are labeled $A$ through $F$. Let $X$ denote an arbitrary box from this set, and define $\mathrm{sz}(X)$ as the size (in this case, length) of $X$. As a preprocessing step, one chooses constants $\alpha$ and $\beta$, and a minimal sequence of tiling resolutions, $\rho_1, \ldots, \rho_n$, such that

$$\begin{aligned} 0 < \alpha < 1 \\ \beta \geq 1 \\ \rho_1 > \rho_2 > \ldots > \rho_n > 0, \end{aligned}$$

and so that for each box $X$ there exists an integer $1 \leq k \leq n$ with

$$\alpha \leq \frac{\mathrm{sz}(X)}{\rho_k} \leq \beta. \tag{2}$$

The minimum integer $k$ satisfying (2) is called the *resolution* of box $X$, abbreviated $\mathrm{res}(X)$. In Figure 2, $\alpha = 0.5$ and $\beta = 1.0$. This means that each box $X$ must have a length that is from 0.5 to 1.0 times the width of the cells at resolution $\mathrm{res}(X)$. The constraints (2) are met by choosing four tiling resolutions, as shown in the figure, with $\mathrm{res}(A) = \mathrm{res}(B) = \mathrm{res}(D) = 4$, $\mathrm{res}(E) = 3$,

$\text{res}(F) = 2$, and $\text{res}(C) = 1$. The location of box $X$ is hashed at tiling resolution $\text{res}(X)$. In two (or three) dimensions, the idea is the same. The cells are squares (or cubes) of side length $\rho_i$, and the boxes to be stored are rectangles (or rectangular prisms). For box $X$, $\text{sz}(X)$ is the maximum distance between two opposite edges (or faces) of the box. In what follows, $d$ denotes the dimension of the boxes and ambient space.

When a box is stored in the hash table, overlap with other boxes must be checked, some of which may be stored at other resolutions. Assume the boxes are hashed in order of increasing resolution. When box $X$ is hashed, all enclosing cells at resolutions less than or equal to $\text{res}(X)$ must be checked for other boxes. In Figure 2, the cells that must be checked when box $E$ is stored are shaded. Since box $C$ overlaps one of these cells, the boxes $E$ and $C$ are reported as *close*, meaning the hash table is unable to verify that the boxes do not overlap. Formally,

**Definition 1** *Boxes $X$ and $Y$ are* **close** *if and only if they overlap a common cell at resolution* $\min(\text{res}(X), \text{res}(Y))$.

When box $D$ is stored in the hash table, it is found to be close to box $C$ since they overlap a common cell at resolution 1, but it is not close to box $E$ since they do not overlap a common cell at resolution 3. When boxes are close, the corresponding objects can be passed directly to the narrow phase collision detection algorithm, or a box-intersection test may be used first to check that the boxes actually do overlap, since it is possible that non-intersecting boxes share a hash bucket. We now analyze the storage requirements and relevant time complexities for a hierarchical spatial hash table scheme, assuming perfect hashing.

**Lemma 1** *Let $\text{sz}_{\min}$ and $\text{sz}_{\max}$ be the sizes of the smallest and largest boxes to be stored in the hierarchical hash table. If $n$ is the the number of resolutions required,*

$$n \leq \left\lceil \log_{\frac{\beta}{\alpha}} \frac{\text{sz}_{\max}}{\text{sz}_{\min}} \right\rceil .$$

*Proof:* Choose $\rho_1 = \text{sz}_{\min}/\alpha$ and subsequent tile resolutions such that the constraint

$$\beta \rho_{i-1} = \alpha \rho_i \tag{3}$$

is satisfied for $2 \leq i \leq n$. In this way, an appropriate $k$ can be found to satisfy (2) for any box dimension in the interval $[\text{sz}_{\min}, \beta\rho_n]$; an $n$ is needed such that $\rho_n \geq \text{sz}_{\max}/\beta$. From (3),

$$\rho_n = \left(\frac{\beta}{\alpha}\right)^{n-1} \rho_1 .$$

Substituting $\text{sz}_{\min}/\alpha$ for $\rho_1$, and using $\rho_n \geq \text{sz}_{\max}/\beta$, yields

$$\left(\frac{\beta}{\alpha}\right)^{n-1} \frac{\text{sz}_{\min}}{\alpha} \geq \frac{\text{sz}_{\max}}{\beta}.$$

The lemma follows. □

Lemma 1 gives an important theoretical bound, but it is not always tight. For instance, if all boxes are one of three sizes (small, medium, or large), than at most three resolutions are required for the hierarchical spatial hash table, regardless of the ratio of the dimensions of the largest to smallest box.

**Theorem 1** *For a set of boxes to be stored in a hierarchical spatial hash table, let $R$ be the ratio of the largest to smallest box dimension. Then the total number of hash buckets which must be checked for other boxes when storing a box in the hash table is $O(\beta^d \log R)$.*

*Proof:* When box $X$ is stored, cells at resolutions $i \leq k$, where $k = \text{res}(X)$, must be checked for other boxes. By (2), $\text{sz}(X) \leq \beta\rho_k$, and so box $X$ overlaps at most $(\beta + 1)^d$ cells at resolution $k$. The number of cells overlapped at a given resolution $i < k$ can not be more than this, and since there are $O(\log R)$ resolutions by Lemma 1, the number of overlapped cells is $O(\beta^d \log R)$. Each cell check corresponds to one bucket check in the hash table, and the theorem follows. □

**Theorem 2** *Treating $\alpha, \beta$, and $R$ as constants, a hierarchical hash table can report all pairs of close boxes among $n$ boxes in $O(n + c)$ time, where $c$ is the number of close pairs.*

*Proof:* By Theorem 1, a constant number of hash buckets must be examined upon storing each box; the total number of buckets checked is $O(n)$. The time spent reporting closest pairs among these buckets is $O(c)$. □

A final theorem relates the resolution power of the hierarchical hash table to the parameter $\alpha$.

**Theorem 3** *The hierarchical spatial hash table can guarantee that two boxes $X$ and $Y$ do not intersect if the distance between them exceeds*

$$\frac{1}{\alpha}\sqrt{d}\max[\text{sz}(X), \text{sz}(Y)]. \tag{4}$$

*Proof:* Without loss of generality, assume $\text{sz}(X) \geq \text{sz}(Y)$, so that $\text{res}(X) \leq \text{res}(Y)$. $X$ and $Y$ are reported as close if and only if they overlap a common cell at resolution $\text{res}(X)$. The maximum distance between any two points in this cell is $\sqrt{d}\rho_{\text{res}X}$, and so the distance $D$ between boxes $X$ and $Y$ satisfies

$$D < \frac{\sqrt{d}\rho_{\text{res}(X)}}{\text{sz}(X)}\ \text{sz}(X). \tag{5}$$

From (2),

$$\frac{\rho_{\text{res}(X)}}{\text{sz}(X)} \leq \frac{1}{\alpha}, \tag{6}$$

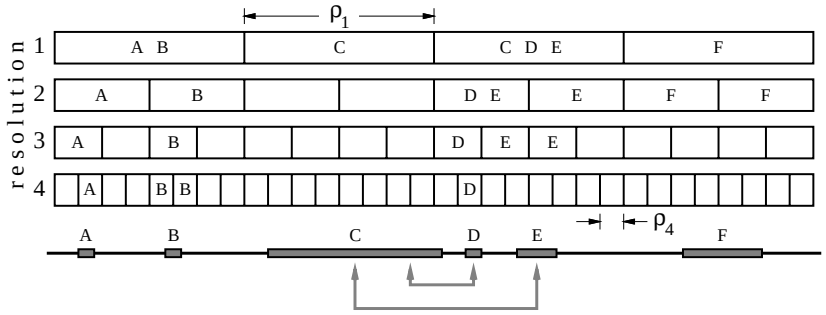


Figure 3: In a variant of the hierarchical hash table scheme, the labels of each box are stored in every bucket that is checked for other boxes.

and the theorem follows. □

The tradeoffs involved in choosing the parameters $\alpha$ and $\beta$ are now apparent. Recall that $0 < \alpha < 1$ and $\beta \geq 1$. By Theorem 1, the closer $\beta$ is to the minimum value of 1, the fewer cells must be checked when storing boxes. By Theorem 3, the closer $\alpha$ is to the upper bound 1, the better the resolving power of the hash table. Finally, Lemma 1 implies that the larger the ratio $\beta/\alpha$, the fewer resolutions are required to store all of the boxes.

## 2.2 Coherence and the tiling scheme

In motion planning applications, a slight variant of the hierarchical hash table scheme can improve performance by taking advantage of coherence between problem instances. Instead of storing box $X$'s label in buckets at $\text{res}(X)$, and only checking the appropriate buckets at lower resolutions, the label is stored in all buckets that are checked. Using the example of Figure 2, the labels are stored as shown in Figure 3. Boxes $X$ and $Y$ are reported as close if and only if their labels appear in a common hash bucket at resolution $\min(\text{res}(X), \text{res}(Y))$; the close pairs returned by the algorithm are exactly the same as with the original version. A two-dimensional array of *close counters* tracks which boxes are close. Each time $X$ is stored into a hash bucket at resolution $i$ that already contains $Y$, if $i = \min(\text{res}(X), \text{res}(Y))$, then the counter corresponding to the pair $(X, Y)$ is incremented. When $X$ is removed from such a bucket, the counter is decremented. Pairs of boxes for which the corresponding counter is zero are not close, and narrow phase collision checking is not performed between the corresponding objects. When a counter is incremented from zero to one, the pair enters the set of pairs on which narrow phase collision checking is performed.

This scheme retains the state of the boxes between invocations. If a box intersects the same cells that it did on the last call, no additional work needs to be done to store that box on the current call. If the box has moved into new

cells or left old ones, only buckets corresponding to these cells must be changed. Boxes corresponding to fixed objects need only be stored into the hash table once. The disadvantage of this scheme is that more processing is sometimes required to store a box's label into a bucket. In Figure 3, even though $D$ and $E$ are verified as not close at resolution 3, they must be stored in common buckets at resolutions 2 and 1, requiring extra processing and adjustment of close counters. For this reason, the claim of Theorem 2 is not valid (or at least not readily apparent) for this variant on the algorithm. However this variant is quite efficient in practice, as shown in Table 1.

Table 1: Comparison of hashing schemes. The table shows the number of cycles spent on broad phase collision detection for four example rigid-body simulations, using both the standard hashing algorithm and the coherence hashing algorithm. The simulations themselves are described in detail in [19]. The coherence hashing algorithm is significantly better, consistently running six or more times faster than the standard hashing algorithm.

| | millions of cycles | | |
|---|---|---|---|
| | standard | coherence | |
| example | hashing | hashing | ratio |
| coins | 361 | 58 | 6.2 |
| bowling | 2188 | 314 | 7.0 |
| rattleback top | 641 | 107 | 6.0 |
| part feeder chute | 409 | 57 | 7.2 |

The bounding boxes are chosen to enclose the swept volumes of moving objects, thus their sizes depend on the current velocities of the objects. Often these are not known ahead of time, and so the tiling resolutions can not be chosen as described in Lemma 1. In practice, this is not such a problem. The tiling resolutions can be based simply upon the maximum radii of the objects. Unless the objects are moving at extremely fast speeds, the number of tiles intersected by the various boxes remains small. Figure 4 shows the reduction in narrow phase collision detection due to the hierarchical hashing scheme.

## 2.3 Coordinate sorting

There are alternatives to hierarchical spatial hashing for finding intersections of axes-aligned bounding boxes. One such algorithm is based on sorting the coordinates of the boundaries of the bounding boxes along each of the three coordinate axes; it is used in the *I-Collide* system [7] and also in [3]. The algorithm works as follows. The minimum and maximum $x$-coordinates of each axes-aligned box are maintained in a sorted list. The same is done for the $y$ and $z$ coordinates. Two boxes overlap if and only if their coordinates overlap in each of the three coordinate directions. A two-dimensional example is shown in Figure 5. For example, since $x'_1$, the maximum $x$-coordinate of box $B_1$ is less than $x_2$, the minimum $x$-coordinate of box $B_2$, these boxes can not overlap.

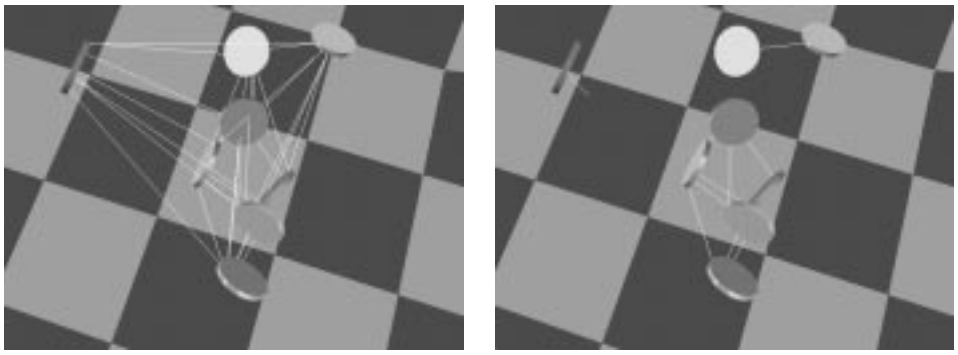

Figure 4: Snapshots taken during the simulation of eight coins tossed onto a flat surface. The lines indicate the tracking of closest points between objects (including the floor). Each line corresponds to a pair of objects subject to narrow phase collision detection. The left and right figures show the situation with and without broad phase culling, respectively.

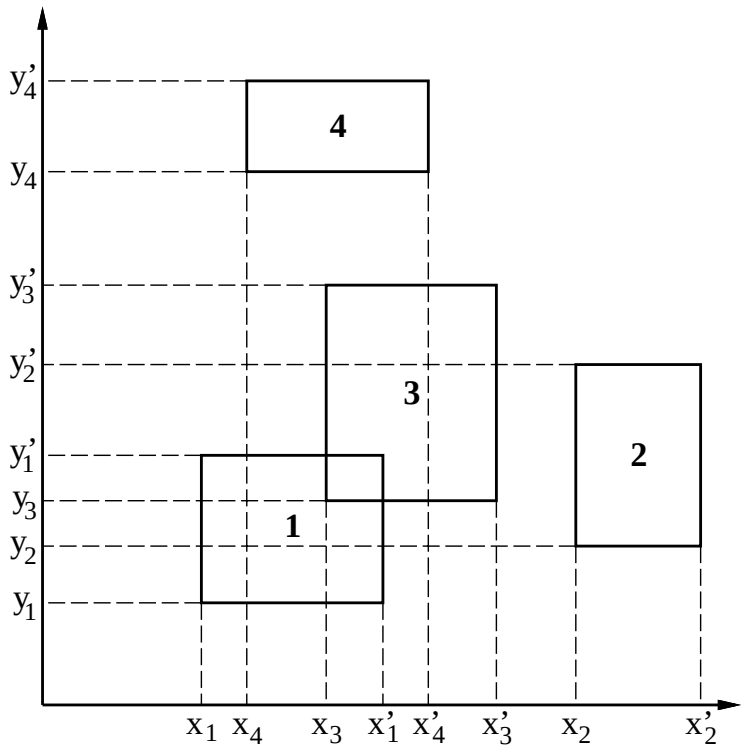


Figure 5: Finding intersections of two-dimensional boxes using coordinate sorting. Two boxes overlap if and only if their projections onto the $x$- and $y$-axes overlap.

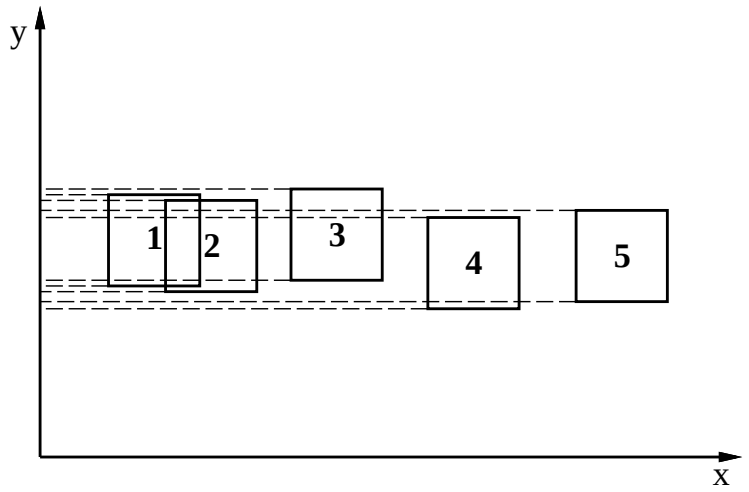


Figure 6: A bad case for coordinate sorting. The dense clustering of box extrema along the $y$-axis results in $O(n^2)$ exchanges for each new sort of the coordinates.

On the other hand, $x_3 < x_1' < x_3'$, and $y_1 < y_3 < y_1'$. Thus, boxes $B_1$ and $B_3$ overlap in both the $x$ and $y$ coordinates, and therefore the boxes themselves overlap. Cohen, *et. al.* discuss the relative merits of using a fixed size, cubical box that can accommodate an object at any orientation versus tighter fitting boxes that change in shape as the object rotates [7].

Coherence is exploited by updating previously sorted lists to obtain new sorted lists. In this way, the number of exchanges needed to obtain the new sorted list is expected to be $O(n)$. It can, however, be $O(n^2)$. Consider the situation depicted in Figure 6. The maximum and minimum $y$-coordinates of all the boxes are clustered closely together. Even with very small motions from one time step to the next, $O(n^2)$ exchanges result in resorting the $y$-coordinates; coherence breaks down. This example is not contrived. Imagine modeling the dropping of a group of small parts onto a flat horizontal surface in order to singulate them. As they come to rest, their bounding boxes will cluster along the vertical coordinate. Since coordinate sorting is based on dimension reduction, the coordinates may be clustered even when the original boxes are not; the clustering becomes worse in higher dimensions. One way of handling the clustering problem is to perform a less drastic dimension reduction, projecting the three-dimensional boxes first into two-dimensional rectangles in the plane, and reporting intersections among the rectangles in $O(n \log n + k)$ time, where $k$ is the number of intersections [9]. Another alternative is to skew the axes to which the bounding boxes are aligned, so that all three coordinates vary along surfaces like table tops. This may cause the boxes to be much larger than in the unskewed system. Hashing schemes do not suffer from the clustering problem. Coherence always results in efficient updating of the hash table, unless the

number of box overlaps in *three* dimensions is large.

Coordinate sorting does have one advantage over hashing: no hashing scheme culls as many object pairs as coordinate sorting. Cohen, *et. al.* claim that choosing a near-optimal cell size is difficult, and failing to do so results in large memory usage and computational inefficiency. These claims are largely mitigated with a hierarchical hash table based on multiple cell sizes. In passing, we note that *Rapid* is a very efficient collision detection algorithm based on oriented bounding boxes, that is, boxes boxes that are not axes aligned. See [11] for details.

# 3 Narrow phase collision detection

Narrow phase collision detectors more carefully analyze the pairs of objects that are not culled by the broad phase detector. They employ more refined but slower algorithms to determine with certainty whether objects are penetrating or disjoint, or to determine the distance between them. Some narrow phase detectors return only a boolean value indicating if penetration is occurring or not. These are sometimes called *interference detectors*. This boolean function alone is enough to isolate the time of collision. When penetration is detected, a binary search method can be used to localize the time of the transition from the disjoint state to the penetration state, within a desired tolerance. Narrow phase detectors that return the distance between disjoint objects are more useful. The distance information can be used to more quickly localize the time of collision, or to determine how soon the collision check should be repeated. Some detectors also return the distance of penetration when objects penetrate, which can be used to compute contact forces using a penalty method.

This section focuses primarily on the *Lin-Canny closest features algorithm* as a basis for narrow phase collision detection. The algorithm operates on rigid, convex polyhedra, specified by a boundary representation. It is among the fastest algorithms known for tracking the closest features between such objects in a setting where coherence can be exploited. The algorithm's output can easily be used to compute the distance between the polyhedra, which serves as a basis for collision detection. The I-Collide collision detection package for large environments uses the Lin-Canny algorithm for its narrow phase detection [7]. Lin-Canny has some drawbacks. The two most limiting are that it does not terminate when presented with penetrating polyhedra,[2] and that it sometimes exhibits poor convergence in degenerate situations. Despite these drawbacks, it can be a useful tool for motion planning. Some alternatives to this algorithm are discussed in Section 3.3.

[2] In practice, one can detect when the algorithm is caught in an infinite cycle, force termination, and report that the polyhedra are penetrating. Unfortunately, this is not an efficient solution.

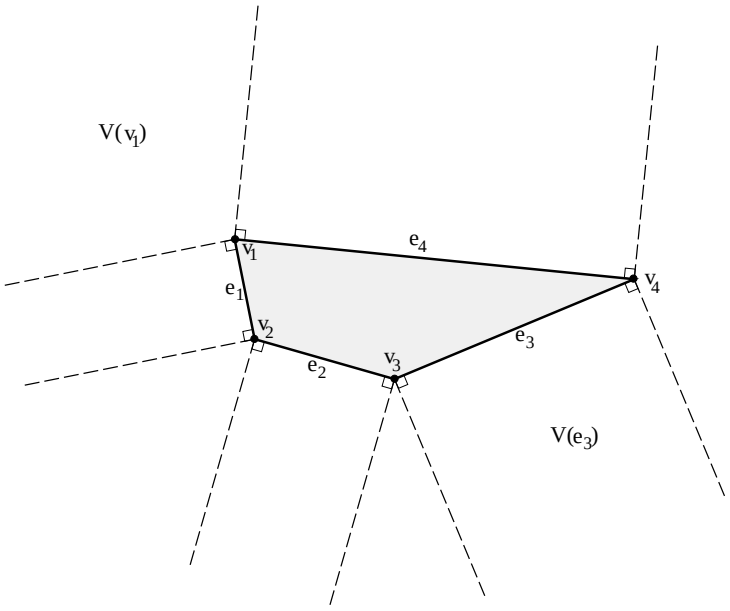


Figure 7: A polygon and its Voronoi regions.

### 3.1 The Lin-Canny closest features algorithm

The Lin-Canny closest features algorithm [17, 18] is an extremely fast method for tracking the closest features (faces, edges, or vertices) between a pair of convex polyhedra moving through space. The principle behind the algorithm is best described with a two-dimensional example. A fundamental concept in the Lin-Canny algorithm is that of a Voronoi region. Consider the polygon shown in Figure 7. The polygon has eight features: four vertices and four edges. For each feature $F$, the set of points closer to $F$ than to any other feature of the polygon is called the *Voronoi region* of $F$, and denoted $V(F)$. The shapes of the Voronoi regions are easily deduced for polygonal objects. From each vertex, extend two rays outward from the polygon, each perpendicular to one of the edges incident to the vertex. These rays form boundaries between the Voronoi regions. The Voronoi region of a vertex is the infinite cone lying between the two rays emanating from that vertex. The Voronoi region of an edge is the semi-infinite rectangle lying between two parallel rays passing through the edge's endpoints. Collectively, the Voronoi regions partition the space outside the polygon.

**Theorem 4** *Given non-intersecting polygons $A$ and $B$, let* **a** *and* **b** *be the closest points between feature $F_a$ of $A$, and feature $F_b$ of $B$, respectively. If* **a** *and*

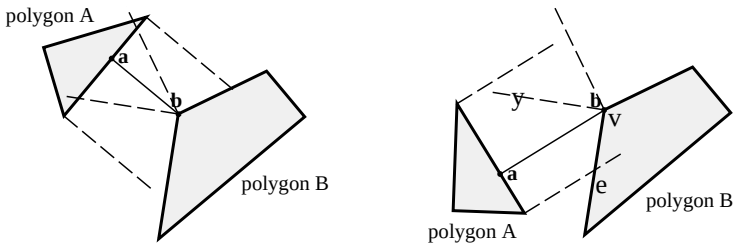


Figure 8: *Left:* Theorem 5 implies **a** and **b** are the closest points between $A$ and $B$. *Right:* $\mathbf{a} \notin V(F_b)$, and so **a** and **b** are no longer closest points. The closest feature on $B$ will be updated from vertex $v$ to edge $e$.

**b** *are the closest points between* $A$ *and* $B$*, then* $\mathbf{a} \in V(F_b)$ *and* $\mathbf{b} \in V(F_a)$.[3]

*Proof:* Suppose $\mathbf{a} \notin V(F_b)$. Then **a** is in some other Voronoi region, say $V(F_c)$, and **a** is closer to $F_c$ than to any other feature on $B$. Since $\mathbf{b} \in F_b$, $\mathbf{b} \notin F_c$, and so **a** and **b** can not be closest points. A similar results holds if $\mathbf{b} \notin V(F_a)$. □

The fundamental basis of the Lin-Canny algorithm is the converse of Theorem 5, which is true for convex objects (see [17] for the proof).

**Theorem 5** *Given non-intersecting* convex *polygons* $A$ *and* $B$*, let* **a** *and* **b** *be the closest points between feature* $F_a$ *of* $A$*, and feature* $F_b$ *of* $B$*, respectively. If* $\mathbf{a} \in V(F_b)$ *and* $\mathbf{b} \in V(F_a)$*, then* **a** *and* **b** *are the closest points between* $A$ *and* $B$*.*

Theorem 5 suggests an algorithm for finding the closest points between convex polygons. The steps are:

1. Compute the closest points between the current pair of features.
2. If each point lies within the Voronoi region of the other feature, return the current features as the closest points between the polyhedra. (The points computed in Step 1 are the closest ones between the polyhedra.)
3. Update one or both of the current features, go to Step 1.

This process can continue for many iterations, however, it is guaranteed to eventually terminate. The heart of the algorithm is in selecting the proper feature(s) to update to in Step 3. Consider the example on the left of Figure 8. Here, closest point candidates **a** and **b** each lie in the Voronoi region of the other's containing feature. By Theorem 5, they are the closest points. Suppose polygon $A$ moves so that the situation is as depicted on the right of Figure 8, and Lin-Canny is called again. Now **b** is still in the Voronoi region of $F_a$, however,

[3] For simplicity, degenerate cases where the points are on the boundary of Voronoi regions are ignored here. See [17] for more details.

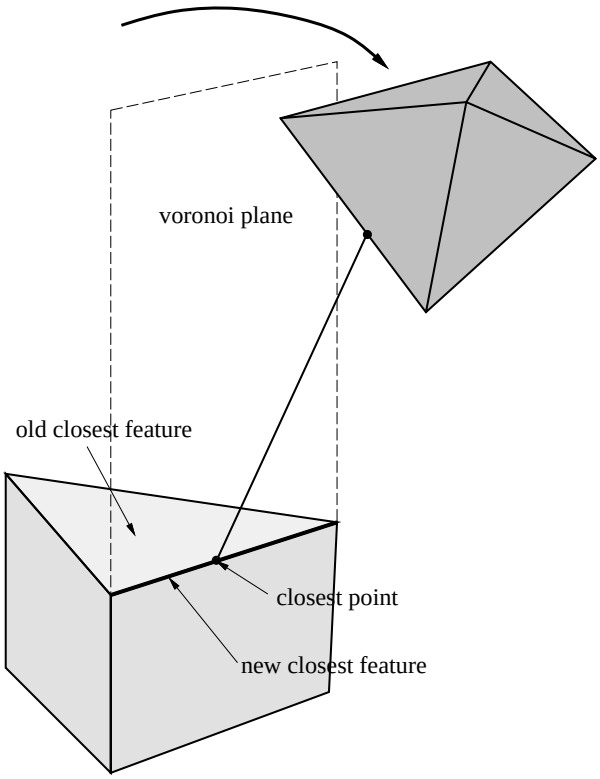


Figure 9: Tracking closest features of polyhedra with the Lin-Canny algorithm.

**a** is no longer in the Voronoi region of $F_b$. Specifically, **a** lies on the wrong side of ray $y$. In this case, the Lin-Canny algorithm specifies that feature $F_b$ should be updated to the feature on the other side of ray $y$, namely the edge $e$. After the update, the closest points between the new features are computed, and the Voronoi check is made again. The three-dimensional version of the algorithm is a natural extension of the two-dimensional case. The objects in question are polyhedra, the features are vertices, edges, and faces, the Voronoi regions are infinite regions of space bounded by constraint planes rather than rays (Figure 9). The basic algorithm remains the same. For details, see [17, 18].

Although designed to track the closest features, the Lin-Canny algorithm is easily extended to a collision detection algorithm. The distance between two polyhedra is computable from simple geometric formulae, given the closest features. The closest points are obtained as a by-product of these calculations. With finite precision arithmetic, a *collision epsilon* $\varepsilon_c$ must be used. The collision detection system reports a collision when the inter-polyhedral distance falls below $\varepsilon_c$. The particular value of $\varepsilon_c$ is not critical; a value is chosen based on how large a gap is tolerable.

## 3.2 Lin-Canny and coherence

For efficient collision detection, coherence must be exploited. In Lin-Canny, coherence means that the closest features between a given pair of objects usually change infrequently. Even if the features are changing upon every invocation of the algorithm, due to highly discretized polyhedral models or large motions between invocations, the pair of closest features from the last invocation of the algorithm is a good starting point for the search for the current pair of closest features. By caching these features from one invocation to the next, significant speedup is obtained. Figure 10 illustrates the effect of coherence on tracking the closest features.

The Lin-Canny algorithm and others like it have been described as taking "expected constant" time to report a pair of closest features. This claim stems from coherence; often the closest features do not change between successive calls, and the algorithm verifies this fact in constant time. This is a bit misleading. Consider tracking closest features between a small satellite orbiting the Earth, over its equator. If the Earth is modeled as a tessellated sphere with $N$ facets, then during one orbit of the satellite, tracking on the Earth must progress through $O(\sqrt{N})$ features. As the resolution of the Earth model increases, more work is clearly being done to track the closest feature as it circumnavigates the planet, even if the satellite speed remains constant. In this case, the Lin-Canny algorithm is $O(\sqrt{N})$. Figure 10 and Graph 2 in Cohen, *et. al.* [7] also illustrate that the running time of the Lin-Canny algorithm depends on the number of features. One difficulty of assigning a complexity to the algorithm is that it is very dependent on how the objects are moving. If the satellite mentioned above falls straight down toward Earth, the algorithm is again constant time. The claim of "expected constant" time raises more questions than it answers, however, and perhaps *almost constant* time is a better description. In the satellite example, the coefficient on $\sqrt{N}$ is probably extremely small compared to the constant term. In experiments with the Impulse simulation system, slowdown is negligible when when polyhedral models of spheres with a few hundred facets are replaced with polyhedral models with over 20,000 facets (over 60,000 features).

## 3.3 Alternative narrow-phase collision detection algorithms

The extension of the basic Lin-Canny algorithm to curved objects has been studied by Lin and Manocha [16]. Curved objects are approximated with a polyhedral mesh, and closest points are tracked between these meshes. The closest points on the meshes are projected onto the actual curved surfaces, and a numerical root finding method uses these points as a starting point to locate the true closest points. A general form of this algorithm has not been implemented.

There are several alternatives to Lin-Canny that use coherence to obtain almost constant time performance when the objects move continuously through space. For dynamic simulation applications, Baraff uses an algorithm based

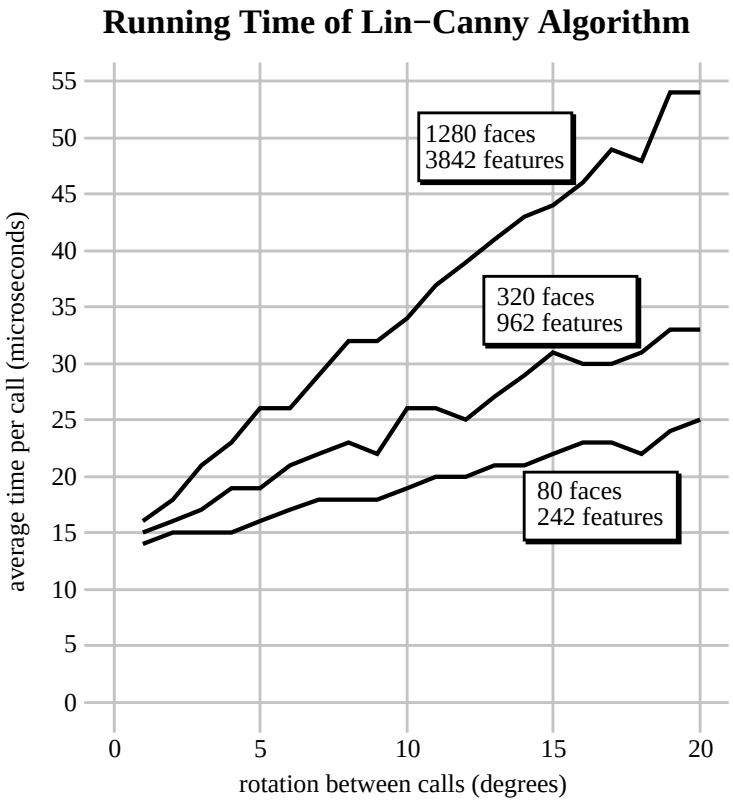


Figure 10: This graph shows the effect of coherence on the performance of the Lin-Canny algorithm. The algorithm was used to track the closest features between a fixed cube and a polyhedral model of a sphere as the sphere rotated on an axis parallel to the nearest surface of the cube. The amount of sphere rotation between successive calls to the algorithm was varied from one to 20 degrees, in one degree steps. This experiment was performed for three different discretization resolutions for the sphere, as indicated above. Note the difference in execution times between the left side of the graph, where coherence is high, and the right side of the graph, where coherence is low. Also note the insensitivity of the algorithm to polyhedron complexity, when coherence is high. At a rotational speed of one degree between calls, a 16-fold increase in complexity results in only a 15% increase in execution time. (Execution times from an R4400 SGI Indigo II.)

on the following observation. Two convex polyhedra are disjoint if and only if there exists a separating plane that either embeds one of the faces of one of the polyhedra, or that embeds an edge from each polyhedra. This separating plane is cached along with points from each polyhedron that are closest to it. Given the plane and the closest point to it, one can verify disjointness in constant time [2]. This algorithm is simpler in principle than Lin-Canny, and might be suitable in some applications. Baraff does not describe the update step that must occur when the current pair of witnesses fails to verify disjointness.

Rather than focusing on polyhedral features, some algorithms treat a polyhedron as the convex hull of a point set, and perform operations on simplices defined by subsets of these points. An algorithm designed by Gilbert, Johnson and Keerthi (GJK) was one of the earliest examples of this type [10]. Given two polyhedra, GJK searches for a simplex, defined by vertices of the Minkowski difference polyhedron, that either encloses or is nearest to the origin. If the origin is not enclosed, the distance between the origin and the nearest simplex of the difference polyhedron is equal to the distance between the original polyhedra. If the origin is enclosed, the polyhedra are penetrating, and a measure of the penetration is available.

Rabbitz advanced the original GJK algorithm by making better use of coherence [25]. *Q-Collide* is a collision detection library spawned from I-Collide, which replaces Lin-Canny with Rabbitz's algorithm for the low-level distance computation [6]. It shares I-Collide's broad phase detection scheme. Cameron has recently developed the fastest descendent of GJK: it includes mechanisms to exploit coherence, and also uses topological vertex information to more carefully choose new simplices when the current simplices fail to satisfy the termination criteria. With these improvements, the algorithm attains the same almost constant time complexity as Lin-Canny [4].

The *Voronoi-clip*, or V-Clip, algorithm is a polyhedral collision detection algorithm, derived from similar principles as the Lin-Canny algorithm. It overcomes the principle limitations of Lin-Canny. In particular,

1. V-Clip handles the penetration case.
2. V-Clip is robust in the presence of degenerate configurations.
3. The code for V-Clip is significantly simpler than that of Lin-Canny.

Enhanced GJK also exhibits many of these advantages but usually at the cost of 50–100% more floating point operations [20].

## 3.4 Nonconvex objects

Algorithms for convex polyhedra can be adapted to nonconvex polyhedra by convex decomposition. If object $A$ is decomposed into $m$ convex pieces, $A_1, \ldots, A_m$, and object $B$ is decomposed into $n$ convex pieces, $B_1, \ldots, B_n$, then the distance

between $A$ and $B$ can be computed as

$$d(A,B) = \min_{\substack{1\le i\le m \\ 1\le j\le n}} d(A_i, B_j) \tag{7}$$

Significant improvement over this naive scheme is possible by computing the convex hulls of each of the nonconvex objects, and approximating the distance between the objects by the distance between their hulls. In cases where the hulls are disjoint, the $nm$ invocations in (7) are replaced with a single invocation that gives a lower bound on the true distance between objects. If the hulls are not disjoint, the objects are "unwrapped" and treated as collections of convex pieces. For complex objects, this scheme can be applied recursively to obtain an entire convex hierarchy for a nonconvex object. The inner nodes of the tree correspond to convex hulls of various subsets of the entire object. The leaves of the tree correspond to the the underlying convex pieces in the object's decomposition. The root corresponds to the convex hull of the entire object. The collision detection algorithm proceeds only as far down the tree as necessary to verify disjointness: if the distance to a hull polyhedron is positive, checking stops at the corresponding node, otherwise it proceeds to the node's children. Details of this scheme may be found in Ponamgi, *et. al.* [24]. Facilities for performing collision detection between these convex hierarchies are provided in the V-Clip package [20].

The strategy described above works well when a convex decomposition is available with a moderate number of pieces, and a hierarchy not more than a few levels deep; it breaks down for utterly nonconvex objects. Other types of collision detection algorithms are then more suitable, such as those based on octrees [1], binary space partitioning trees [22], sphere hierarchies [15], or oriented bounding boxes (OBBs). These algorithms also often provide robustness in the presence of modeling errors, such as improperly oriented or missing facets. For example, the Rapid collision detection library, based on the OBB algorithm by Gottschalk, *et. al.*, adeptly handles "polygon soup" [11]. Algorithms like Rapid are the best choice in applications such as complex walk through environments. The convex polyhedra algorithms are faster and preferable when the models are well-behaved, of moderate size, and not exceedingly nonconvex.

# 4 Scheduling collision checks

The typical paradigm for performing collision detection is at fixed-length time intervals. That is, the system state is advanced forward in time by some amount $\Delta t$, and then collision detection is performed among all pairs of objects. This process continues until a collision is detected, initiating collision processing. In computer graphics, the value of $\Delta t$ is often the screen refresh period, 1/30 second.

One problem with this paradigm is that the system might miss collisions. A pathological example is a bullet speeding toward a thin wall; no matter what

the minimum sampling period of the collision detection system (the *minimum temporal resolution* [15]), one can choose a bullet speed and wall thickness such that the bullet passes completely through the wall between collision checks. One solution is to apply detection algorithms to the four-dimensional hyper-polyhedra swept out in space-time [5]. This method is slow and has not seen wide practical use. Most simulation systems simply ignore the problem [8, 21, 12, 2].

There is an alternative paradigm for performing collision checks that can reduce the number of collision checks performed, and also prevent missed collisions. It relies on computing a lower bound on the time of impact (TOI) between a pair of objects. If the objects obey the laws of Newtonian dynamics, one can derive lower bounds on the TOI, based on the current positions and velocities of the objects, and some assumptions about the types of forces acting on them [19]. Bounds on robot joint accelerations are also useful for computing lower bounds on the TOI.

The basic idea is to schedule collision checks in a priority queue. Each element in the queue corresponds to a particular pair of objects currently subject to narrow phase collision detection. Pairs that are culled by the broad phase are not represented in the queue. The elements are sorted on the estimated TOI that is computed for each pair, so that the pair with the earliest TOI is first. After advancing the state of the system to the TOI at the front of the collision check queue, narrow phase collision checking is invoked only on the pair at the front of the queue. If penetration is detected, collision processing begins. Otherwise, a new TOI is computed for the pair, and its position in the priority queue is adjusted. It may drop back in the queue, or it may remain at the front, depending on how close to collision the objects are. As this process is repeated, the system is advanced in time while all necessary collision checks are performed in a timely manner. The Impulse simulation system used this scheme to determine the integration step size to use between successive collision checks, as shown in Figure 11. Since each computed TOI for a pair of objects is a lower bound on the true TOI for the pair, no pair can collide any sooner than the TOI at the front of the queue. The conservative bounds force the algorithm to detect collisions in the order they occur, and prevent the system from missing a collision.

Other systems have also adopted the scheduling paradigm for collision checking. Von Herzen, *et. al.* [13] present an algorithm that uses Lipschitz bounds to derive limits on how far parts of a parametric surfaces can move over a time interval; their system is guaranteed to catch all collisions. The bounds must be supplied by the user when the surface is defined. The algorithm of Snyder, *et. al.* uses an interval version of Newton's root finding method to achieve the same goal: guaranteeing that the very next collision will be detected [26]. Here, Lipschitz bounds are not needed since the exact trajectories of the surfaces over time are input data for the problem.

The scheduling scheme can still be used if the computed TOI is not a lower bound on the true TOI but simply an estimate of it. In this case, however, the system might detect a collision at time $t$ *after* the system state has been advanced to some time later than $t$. This necessitates a *rollback* to bring the

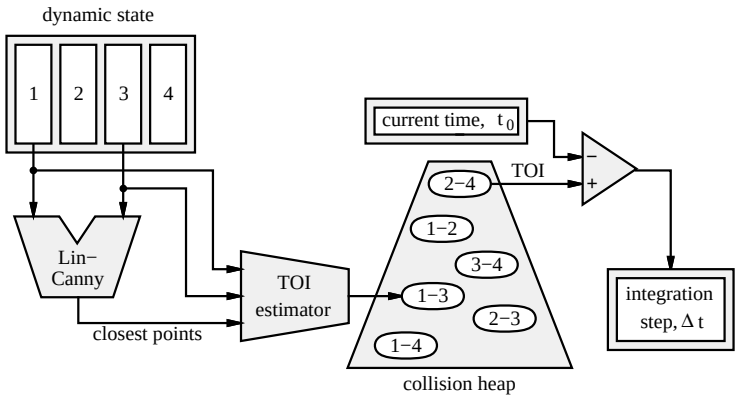


Figure 11: Collision checks are scheduled in a priority queue, implemented as a heap with modifiable keys, based on estimated times of impact. The TOI at the front of the queue determines the size of the next integration step. Numbers designate the indices of the different objects.

state of the system back to the point of collision, and of course some of the previous computation is wasted.

In the fixed interval approach to collision detection, collision checks occur between all pairs of objects at the same rate. Often this rate must be quite high to avoid missing too many collisions. The scheduling scheme reduces the number of collision checks by allowing their rate to vary, both in time and among different pairs of objects. When two objects are far apart or moving slowly, collision checks between them occur less often; at the same time, checks between other pairs in danger of colliding can occur frequently.

# 5 Summary

Efficient collision detection is crucial to many applications, including robot motion planning. While no algorithm is the best solution for all problems, there are important guiding principles that should be used wherever possible. Exploiting coherence results in much faster algorithms. Using a broad phase to reduce the number of pairs processed by the narrow phase is beneficial. A broad phase method that has proven effective is one based on axes-aligned bounding boxes, stored coherently in a hierarchical hash table. For the narrow phase, many options are available. The Lin-Canny closest features algorithm is one of the fastest ways to compute the distance between disjoint objects. When possible, it is better to independently schedule collision checks between pairs of objects instead of checking all pairs at fixed intervals. This further reduces the computational cost of collision detection.

### 5.1 Downloadable software

This article discusses several collision detection algorithms that have publicly available implementations. Web addresses for downloading this software are given below (as of November 1997). All packages are written in C or C++.

- **Narrow phase collision detectors**

  1. *Lin-Canny closest features algorithm.* A feature-based algorithm for tracking closest features between convex polyhedra. Available from: www.cs.berkeley.edu/~mirtich/collDet.html.
  2. *Enhanced Gilbert-Johnson-Keerthi algorithm.* An enhanced version of the GJK simplex-based algorithm for computing distance between convex polyhedra. Available from: www.comlab.ox.ac.uk/oucl/users/stephen.cameron/distances.html.
  3. *V-Clip algorithm.* A Voronoi-style closest feature algorithm, with facilities for nonconvex polyhedral hierarchies. Available from: www.merl.com/people/mirtich/vclip.html.
  4. *Rapid.* A robust interference detector based on oriented bounding boxes. Good for highly nonconvex objects and polygon soup. Available from: www.cs.unc.edu/~geom/OBB/OBBT.html.

- **Full collision detection packages (narrow & broad phases)**

  1. *I-Collide,* a package using the Lin-Canny closest features algorithm for narrow phase detection. Available from: www.cs.unc.edu/~geom/I_COLLIDE.html.
  2. *V-Collide,* a package using the Rapid algorithm for narrow phase detection. Available from: www.cs.unc.edu/~geom/V_COLLIDE.
  3. *Q-Collide,* a derivative of I-Collide, with the Lin-Canny algorithm replaced by a simplex-based algorithm for narrow phase detection. Available from: www.cs.hku.hk/~tlchung/collision_library.html.

## Acknowledgments

The author thanks Sarah Gibson for many helpful comments on this article.

## References

[1] D. Ayala, P. Brunet, R. Juan, and I. Navazo. Object representation by means of nonminimal division quadtrees and octrees. *ACM Transactions on Graphics*, 4(1):41–59, January 1985.

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

[3] David Baraff. *Dynamic Simulation of Non-Penetrating Rigid Bodies*. PhD thesis, Department of Computer Science, Cornell University, March 1992.

[4] Stephen Cameron. Enhancing GJK: Computing minimum penetration distances between convex polyhedra. In *Proceedings of International Conference on Robotics and Automation*. IEEE, April 1997.

[5] John Canny. Collision detection for moving polyhedra. Technical Report MIT A.I. Lab Memo 806, Massachusetts Institute of Technology, October 1984.

[6] Kelvin Chung. An efficient collision detection algorithm for polytopes in virtual environments. Master's thesis, University of Hong Kong, September 1996.

[7] Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, and Madhav K. Ponamgi. I-collide: An interactive and exact collision detection system for large-scaled environments. In *Symposium on Interactive 3D Graphics*, pages 189–196. ACM Siggraph, ACM Siggraph, April 1995.

[8] James F. Cremer and A. James Stewart. The architecture of newton, a general-purpose dynamics simulator. In *Proceedings of International Conference on Robotics and Automation*, pages 1806–1811. IEEE, May 1989.

[9] H. Edelsbrunner. A new approach to rectangle intersections, part i. *International Journal of Computational Mathematics*, 13:209–219, 1983.

[10] Elmer G. Gilbert, Daniel W. Johnson, and S. Sathiya Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal of Robotics and Automation*, 4(2):193–203, April 1988.

[11] S. Gottschalk, M. C. Lin, and D. Manocha. Obb-tree: A hierarchical structure for rapid interference detection. In *Computer Graphics Proceedings, Annual Conference Series*, Proceedings of SIGGRAPH 96. ACM SIGGRAPH, 1996.

[12] James K. Hahn. Realistic animation of rigid bodies. *Computer Graphics*, 22(4):299–308, August 1988.

[13] Brian Von Herzen, Alan H. Barr, and Harold R. Zatz. Geometric collisions for time-dependent parametric surfaces. In *Computer Graphics Proceedings, Annual Conference Series*, Proceedings of SIGGRAPH 90, pages 39–48. ACM SIGGRAPH, 1990.

[14] Philip M. Hubbard. *Collision Detection for Interactive Graphics Applications*. PhD thesis, Department of Computer Science, Brown University, October 1994.

[15] Philip M. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics*, 15(3), July 1996.

[16] M.C. Lin and Dinesh Manocha. Interference detection between curved objects for computer animation. In *Models and Techniques in Computer Animation*, pages 431–57. Springer-Verlag, 1993.

[17] Ming C. Lin. *Efficient Collision Detection for Animation and Robotics*. PhD thesis, University of California, Berkeley, December 1993.

[18] Ming C. Lin and John F. Canny. A fast algorithm for incremental distance calculation. In *Proceedings of International Conference on Robotics and Automation*, pages 1008–1014. IEEE, May 1991.

[19] Brian Mirtich. *Impulse-based Dynamic Simulation of Rigid Body Systems*. PhD thesis, University of California, Berkeley, December 1996.

[20] Brian Mirtich. *V-Clip:* fast and robust polyhedral collision detection. Technical Report TR97-05, Mitsubishi Electric Research Lab, Cambridge, MA, July 1997.

[21] Matthew Moore and Jane Wilhems. Collision detection and response for computer animation. *Computer Graphics*, 22(4):289–298, August 1988.

[22] Bruce F. Naylor. Interactive solid modeling via partitioning trees. In *Graphics Interface*, pages 11–18, May 1992.

[23] M. Overmars. Point location in fat subdivisions. *Information Processing Letters*, 44:261–265, 1992.

[24] Madhav K. Ponamgi, Dinesh Manocha, and Ming C. Lin. Incremental algorithms for collision detection between solid models. In *Proceedings of Third ACM Symposium on Solid Modeling and Applications*, pages 293–304, New York, May 1995. ACM Press.

[25] Rich Rabbitz. Fast collision detection of moving convex polyhedra. In Paul S. Heckbert, editor, *Graphics Gems IV*, pages 83–109, Cambridge, 1994. Academic Press, Inc.

[26] John M. Snyder, Adam R. Woodbury, Kurt Fleischer, Bena Currin, and Alan H. Barr. Interval methods for multi-point collisions between time-dependent curved surfaces. In *Computer Graphics Proceedings, Annual Conference Series*, Proceedings of SIGGRAPH 93, pages 321–333. ACM SIGGRAPH, 1993.