# Efficient Collision Detection Using Bounding Volume Hierarchies of *k*-DOPs

James T. Klosowski, Martin Held, Joseph S.B. Mitchell, Henry Sowizral, and Karel Zikan

**Abstract**—Collision detection is of paramount importance for many applications in computer graphics and visualization. Typically, the input to a collision detection algorithm is a large number of geometric objects comprising an environment, together with a set of objects moving within the environment. In addition to determining accurately the contacts that occur between pairs of objects, one needs also to do so at *real-time* rates. Applications such as haptic force-feedback can require over 1,000 collision queries per second.

In this paper, we develop and analyze a method, based on bounding-volume hierarchies, for efficient collision detection for objects moving within highly complex environments. Our choice of bounding volume is to use a "discrete orientation polytope" ("*k*-dop"), a convex polytope whose facets are determined by halfspaces whose outward normals come from a small *fixed* set of *k* orientations. We compare a variety of methods for constructing hierarchies ("BV-trees") of bounding *k*-dops. Further, we propose algorithms for maintaining an effective BV-tree of *k*-dops for moving objects, as they rotate, and for performing fast collision detection using BV-trees of the moving objects and of the environment.

Our algorithms have been implemented and tested. We provide experimental evidence showing that our approach yields substantially faster collision detection than previous methods.

**Index Terms**—Collision detection, intersection searching, bounding volume hierarchies, discrete orientation polytopes, bounding boxes, virtual reality, virtual environments.

————————————————— ❖ —————————————————

## 1 INTRODUCTION

THE collision detection (CD) problem takes as input a geometric model of a scene or environment (e.g., a large collection of complex CAD models), together with a set of one or more moving ("flying") objects, possibly articulated, and asks that we determine all instants in time at which there exists a nonempty intersection between some pair of flying objects, or between a flying object and an environment model. Usually, we are given some information about how the flying objects are moving, at least at the current instant in time; however, the motions may change rapidly, depending on the evolution of a simulation (e.g., modeling some physics of the system), or due to input devices under control of the user. In some applications, it is important to make computations based on the geometry of the region of intersection between pairs of colliding objects; in these cases, we must not only detect that a collision occurs, but also report all pairs of primitive geometric elements (e.g., triangles) that are intersecting at that instant. Thus, we can distinguish between the CD problem of *pure detection* and the CD problem of *detect and report*.

Real-time collision detection is of critical importance in computer graphics, visualization, simulations of physical systems, robotics, solid modeling, manufacturing, and molecular modeling. The requirement for speed in interactive use of virtual environments is particularly challenging; e.g., haptic force-feedback can require on the order of 1,000 intersection queries per second. One may, for example, wish to interact with a virtual world that models a cluttered mechanical workspace, and ask how easily one can assemble, access, or replace component parts within the workspace: Can a particular subassembly be removed without collisions with other parts, and while not requiring undue awkwardness for the mechanic? When using haptic force-feedback, the mechanic is not only alerted (e.g., audibly or visually) about a collision, but actually *feels* a reactionary force, exerted on his body by a haptic device.

A simple-minded approach to CD involves comparing all pairs of primitive geometric elements. This method quickly becomes infeasible as the model complexity rises to realistic sizes. Thus, many approaches have recently been proposed to address the issue of efficiency; we discuss these below.

### 1.1 Our Contribution

In this paper, we present a new approach to CD, based on a form of bounding volume hierarchy ("BV-tree"). Our main contributions include:

1) A careful study of effective methods of constructing BV-trees, using "discrete orientation polytopes" ("*k*-dops");

2) An effective method for applying BV-trees of *k*-dops for moving (rotating) objects, as well as an efficient algorithm, using BV-trees, for detecting collisions,[1] as

- *J.T. Klosowski , M. Held, and J.S.B. Mitchell are with the Department of Applied Mathematics and Statistics, State University of New York, Stony Brook, NY 11794-3600.*
  *E-mail: {jklosow, held, jsbm}@ams.sunysb.edu.*
- *H. Sowizral is with Sun Microsystems, 2550 Garcia Avenue, UMPK14-202, Mountain View, CA 94043-1100.*
  *E-mail: henry.sowizral@eng.sun.com.*
- *K. Zikan is with the Faculty of Informatics, Masaryk University, Botanicka 68a, Brno, Czech Republic, and is a consultant for Sun Microsystems.*
  *E-mail: zikan@fi.muni.cz.*

---

1. Strictly speaking, we check for intersections among surfaces rather than volumes. Thus, if one object contains another object but their surfaces do not intersect, then "no collision" is reported by our algorithm.

objects move within a complex static environment; and

3) Experimental results, with real and simulated data, to study design issues of BV-trees that are most relevant to collision detection.

We have paid careful attention to the generation of particularly challenging and diverse datasets for algorithm design and for comparative studies. Our tests provide experimental evidence that our methods compare quite favorably with the best previous methods.

This paper is accompanied by supplementary material on the WWW, including additional color photos, sample datasets, and the (soon to be released) source code; refer to the authors' web pages.

The remainder of the paper is organized as follows: Prior and related work is reviewed in Section 2. Section 3 provides an introduction to BV-trees, discrete orientation polytopes, and design choices in constructing effective BV-trees. Section 4 highlights our collision detection algorithm and several key issues related to it. Implementation details and experimental results are reported in Section 5. The conclusion, Section 6, includes a discussion of extensions and future work.

## 2 PREVIOUS WORK

Due to its widespread importance, there has been an abundance of work on the problem of collision detection. Many of the approaches have used hierarchies of bounding volumes or spatial decompositions to address the problem. The idea behind these approaches is to approximate the objects (with bounding volumes) or to decompose the space they occupy (using decompositions), to reduce the number of pairs of objects or primitives that need to be checked for contact.

Octrees [33], [35], $k$-d trees [24], BSP-trees [34], brepindices [8], [42], tetrahedral meshes [24], and (regular) grids [19], [24] are all examples of spatial decomposition techniques. By dividing the space occupied by the objects, one needs to check for contact between only those pairs of objects (or parts of objects) that are in the same or nearby cells of the decomposition. Using such decompositions in a hierarchical manner (as in octrees, BSP-trees, etc.) can further speed up the collision detection process.

Hierarchies of bounding volumes have also been a very popular technique for collision detection algorithms. (They have also been widely used in other areas, e.g., ray tracing [1], [20], [29], [44].) In building hierarchies on objects, one can obtain increasingly more accurate approximations of the objects, until the exact geometry of the object is reached. The choice of bounding volume has often been to use spheres [27], [28], [37] or axis-aligned bounding boxes (AABBs) [7], [24], due to the simplicity in checking two such volumes for overlap (intersection). In addition, it is simple to transform these volumes as an object rotates and translates.

Another bounding volume that has become popular recently is the oriented bounding box (OBB), which surrounds an object with a bounding box (hexahedron with rectangular facets) whose orientation is arbitrary with respect to the coordinate axes; cf. Fig. 1b. This volume has the advantage that it can, in general, yield a better (tighter) outer approximation of an object, as its orientation can be chosen in order to make the volume as small as possible. In 1981, Ballard [2] created a two-dimensional hierarchical structure, known as a "strip tree," for approximating curves, based on oriented bounding boxes in the plane. Barequet et al. [6] have recently generalized this work to three dimensions (resulting in a hierarchy of OBBs known as a "BOXTREE"), for applications of oriented bounding boxes for fast ray tracing and collision detection. Zachmann and Felger [45], [46] have used a similar term, "BoxTree," for their hierarchies of oriented boxes, which are also used for collision detection, but are differently constructed from the "BOXTREE" of Barequet et al.

One leading system publicly available for performing collision detection among arbitrary polygonal models is the "RAPID" system, which is also based on a hierarchy of oriented bounding boxes, called "OBBTrees," implemented by Gottschalk et al. [21]. The efficiency of this method is due, in part, to an algorithm for determining whether two oriented bounding boxes overlap. This algorithm is based on examining projections along a small set of "separating axes" and is claimed to be an order of magnitude faster than previous algorithms. (We note that Greene [22] previously published a similar algorithm; however, we are not aware of any empirical comparisons between the two algorithms.)

Other approaches to collision detection have included using space-time bounds [27] and four-dimensional geometry [9], [10] to bound the positions of objects within the near future. By using a fourth dimension to represent the simulation time, contacts can be pinpointed exactly; however, these methods are restrictive in that they require the motion to be prespecified as a closed-form function of time. Hubbard's space-time bounds [27] do not have such a requirement; by assuming a bound on the acceleration of objects, he is able to avoid missing collisions between fast-moving objects.

There has been a collection of innovative work which utilizes Voronoi diagrams [11], [30], [31], [32] to keep track of the closest features between pairs of objects. One popular system, I-COLLIDE [11], uses spatial and temporal coherence in addition to a "sweep-and-prune" technique to reduce the pairs of objects that need to be considered for collision. Although this software works well for many simultaneously moving objects, the objects are restricted to be convex. More recently, Ponamgi et al. have generalized this work to include nonconvex objects [38].

In addition to the "practical" work highlighted above, there have also been a considerable number of "theoretical" results on the problem of collision detection in the field of computational geometry. In particular, the distance (and thus intersection) between two convex polytopes can be determined in $O(\log^2 n)$, where $n$ is the total number of vertices of the polytopes, by using the Dobkin-Kirkpatrick hierarchy [15], [16], [17], which takes $O(n)$ time and space to construct. In the case of one convex polytope and one *non*-convex polytope, the intersection *detection* time increases to $O(n \log n)$ [14], [40], while actually *computing* the intersection [18] takes $O(K \log K)$ time, where $K$ is the size of the

input plus output. Schömer [40] detects the intersection between two translating "$c$-iso-oriented" polyhedra (nonconvex, having normals among $c$ directions, where $c$ is a fixed constant) in time $O(n^{5/3+\epsilon})$, for any fixed positive constant $\epsilon > 0$. Schömer and Thiel [41] have recently provided the first provably (worst case) subquadratic time algorithm for a general collection of polyhedra in motion along fixed trajectories. However, the result is purely of theoretical interest, as the methods are based on several sophisticated (unimplemented) techniques.

Recently, Suri et al. [43] have given theoretical results that may help to explain the practicality of bounding volume methods, such as our own. In particular, they show that in a collection of objects that have bounded aspect ratio and scale factor,[2] the number of pairs of objects whose bounding volumes intersect is roughly proportional, asymptotically, to the number of pairs of objects that actually intersect, plus the number of objects. Suri et al. use this result to obtain an output-sensitive algorithm for detecting all intersections among a set of convex polyhedra, having bounded aspect ratio and scale factor; their time bound is $O((n + k)\log^2 n)$, for $n$ polyhedra, where $k$ is the number of pairs of polyhedra that actually intersect.

## 3   BV-TREES

We assume as input a set $S$ of $n$ geometric "objects," which, for our purposes, are generally expected to be triangles in 3D that specify the boundary of some polygonal models. Much of our discussion, though, applies also to more general objects.

A *BV-tree* is a tree, BVT($S$), that specifies a bounding volume hierarchy on $S$. Each node, $v$, of BVT($S$) corresponds to a subset, $S_v \subseteq S$, with the root node being associated with the full set $S$. Each internal (nonleaf) node of BVT($S$) has two or more children; the maximum number of children for any internal node of BVT($S$) is called the *degree* of BVT($S$), denoted by $\delta$. The subsets of $S$ that correspond to the children of node $v$ form a partition of the set $S_v$ of objects associated with $v$. In a *complete* BV-tree of $S$, the leaf nodes are associated with singleton subsets of $S$. The total number of nodes in BVT($S$) is at most $2n - 1$; the height of a complete tree is at least $\lceil \log_\delta n \rceil$, which is achieved if the BV-tree is *balanced*.

Also associated with each node $v$ of BVT($S$) is a *bounding volume*, $b(S_v)$, that is an (outer) approximation to the set $S_v$ using a smallest instance of some specified class of shapes (e.g., boxes, spheres, polytopes of a given class, etc.).

In most of this paper, we will be focusing on the case of a single (rigid) object, specified by a set $F$ of boundary primitives (triangles), given in one particular position and orientation, that is moving ("flying") within an environment, specified by a set $E$ of "obstacle" primitives (triangles). We refer to BVT($F$) as the *flying hierarchy* and BVT($E$) as the *environment hierarchy*.

## 3.1  Design Criteria

The choice of which class (or classes) of shapes to use as bounding volumes in a BV-tree is usually dependent upon the application domain and the different constraints inherent to it. In ray tracing, for example, the bounding volumes chosen should tightly fit the primitive objects but also allow for efficient intersection tests between a ray and the bounding volumes [29]. Weghorst et al. [44] discussed making this choice for ray tracing, and they provided a cost function to help analyze hierarchical structures of bounding volumes. Gottschalk et al. [21] looked at this same cost function in the context of collision detection. Given two large input models and hierarchies built to approximate them, the total cost to check the models for intersection was quantified as

$$T = N_v \times C_v + N_p \times C_p, \qquad (1)$$

where $T$ is the total cost function for collision detection, $N_v$ is the number of pairs of bounding volumes tested for overlap, $C_v$ is the cost of testing a pair of bounding volumes for overlap, $N_p$ is the number of pairs of primitives tested for contact, and $C_p$ is the cost of testing a pair of primitives for contact.

While (1) is a reasonable measure of the cost associated with performing a single intersection detection check, it does not take into account the cost of *updating* the flying hierarchy as the flying object rotates. While, for some choices of bounding volumes (e.g., spheres), there is little or no cost associated with updating the flying hierarchy, in general, there will be such a cost, and, in particular, we experience an update cost for our choice ($k$-dops). Thus, we propose that for collision detection in motion simulation that the cost is best written as the sum of *three* component terms:

$$T = N_v \times C_v + N_p \times C_p + N_u \times C_u, \qquad (2)$$

where $T$, $N_v$, $C_v$, $N_p$, and $C_p$ are defined as in (1), while $N_u$ is the number of nodes of the flying hierarchy that must be updated, and $C_u$ is the cost of updating each such node.

Based upon this cost function, we would like our bounding volumes to:

1) Approximate tightly the input primitives (to lower $N_v$, $N_p$, and $N_u$),
2) Permit rapid intersection tests to determine if two bounding volumes overlap (to lower $C_v$), and
3) Be updated quickly when the primitives (and consequently the bounding volumes) are rotated and translated in the scene (to lower $C_u$).

Unfortunately, these objectives usually conflict, so a balance among them must be reached.

## 3.2  Discrete Orientation Polytopes

Here, we concentrate on our experience with bounding volumes that are convex polytopes whose facets are determined by halfspaces whose outward normals come from a small *fixed* set of $k$ orientations. For such polytopes, we have coined the term *discrete orientation polytopes*, or "$k$-dops," for short.[3] See Fig. 1c for an illustration in two dimensions of

---

2. The aspect ratio of an object is defined here to be the ratio between the volume of a smallest enclosing sphere and a largest enclosed sphere. The scale factor for the collection of objects is the ratio between the volume of the largest enclosing sphere and the smallest enclosing sphere.

3. An alternative name for what we call a "dop" is the term *fixed-directions hull* [47] (FDH)—perhaps a slightly more precise term, but a harder to pronounce abbreviation.

(a)                                                  (b)                                                  (c)
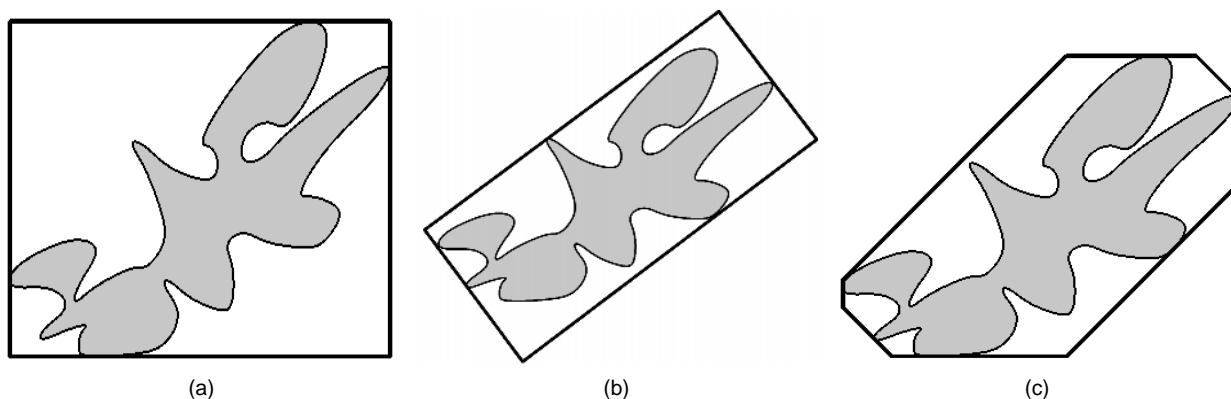
Fig. 1. Approximations of an object by three bounding volumes: an axis-aligned bounding box (AABB), an oriented bounding box (OBB), and a $k$-dop (where $k = 8$).

an eight-dop, whose eight fixed normals are determined by the orientations at ±45, ±90, ±135, and ±180 degrees. Axis-aligned bounding boxes (in 3D) are 6-dops, with orientation vectors determined by the positive and negative coordinate axes. In this paper, we concentrate on 6-dops, 14-dops, 18-dops, and 26-dops, defined by orientations that are particularly natural; see Section 3.3.3 for more detail.

Researchers at IBM have used the same 18-dops (which they call "triboxes" or "T-boxes") for visual approximation purposes within 3DIX [12], [13]. This idea of using planes of fixed orientations to approximate a set of primitive objects was first introduced in the ray tracing work of Kay and Kajiya [29].

Axis-aligned bounding boxes (AABBs) are often used in hierarchies because they are simple to compute and they allow for very efficient overlap queries. But AABBs can also be particularly poor approximations of the set that they bound, leaving large "empty corners"; consider, for example, a needle-like object that lies at a 45-degree orientation to the axes. Using $k$-dops, for larger values of $k$, allows the bounding volume to approximate the convex hull more closely. Of course, the improved approximation (which tends to lower $N_v$, $N_p$, and $N_u$) comes at the cost of increasing the cost, $C_v$, of testing a pair of $k$-dops for intersection (since $C_v = O(k)$) and the cost, $C_u$, of updating $k$-dops in the flying hierarchy (since $C_u = O(k^2)$).

To keep the associated costs as small as possible, we have been using only $k$-dops whose discrete orientation normals come as pairs of collinear, but oppositely oriented, vectors. Kay and Kajiya referred to such pairs as bounding slabs [29]. Thus, as an AABB bounds (i.e., finds the minimum and maximum values of) the primitives in the $x$, $y$, and $z$ directions, our $k$-dops will also bound the primitives but in $k/2$ directions. This has the advantage in that our (conservative) disjointness test for two $k$-dops is essentially as trivial as checking two AABBs for overlap: we simply perform $k/2$ interval overlap tests. This test is far simpler than checking for intersection between OBBs or between convex hulls. Further, since the $k/2$ defining directions are fixed, the memory required to store each $k$-dop is only $k$ values (one value per plane), since the orientations of the planes are known in advance.

Bounding spheres are another natural choice to approximate an object, since it is particularly simple to test pairs for overlap, and the update for a moving object is trivial. However, spheres are similar to AABBs in that they can be very poor approximations to the convex hull of the contained object. Hence, bounding spheres yield low costs $C_v$ and $C_u$, but may result in a large number, $N_p$, of pairs of primitives to test. Oriented bounding boxes (OBBs) can yield much tighter approximations than spheres and AABBs, in some cases. Also, it is relatively simple to update an OBB, by multiplying two transformation matrices. However, the cost $C_v$ for determining if two OBBs overlap is roughly an order of magnitude larger than for AABBs [21]. At the extreme, convex hulls provide the tightest possible convex bounding volume; however, both the test for overlap and the update costs are relatively high.

In comparison, our choice of $k$-dops for bounding volumes is made in hopes of striking a compromise between the relatively poor tightness of bounding spheres and AABBs, and the relatively high costs of overlap tests and updates associated with OBBs and convex hulls. The parameter $k$ allows us some flexibility too in striking a balance between these competing objectives. For moderate values of $k$, the cost $C_v$ of our conservative disjointness test is an order of magnitude faster than testing two OBBs. Also, while updating a $k$-dop for a rotating object is more complex than updating some other bounding volumes, we have developed a simple approximation approach, discussed in Section 4.1, that works well in practice.

Fig. 1 highlights the differences in some of the typical bounding volumes. Here, we provide a simple two-dimensional illustration of an object and its corresponding approximations by an axis-aligned bounding box (AABB), an oriented bounding box (OBB), and a $k$-dop (where $k = 8$).

## 3.3 Design Choices

Our study has included a comparison of various design choices in constructing BV-trees, including:

1) the degree, $\delta$, of the tree (binary, ternary, etc.);
2) top-down versus bottom-up construction;
3) the choice of the $k$-dops; and
4) splitting rules.

### 3.3.1 Degree of the Tree

Minimizing the height of the tree is usually a desirable quality when building a hierarchy, so that, when searches are performed, we can traverse the tree, from the root to a leaf, in a small number of steps. The degree, $\delta$, specifies the maximum number of children any node can have. Typically, the higher the degree, the smaller the height of the tree. There is, of course, a trade-off between trees of high and low degree. A tree with a high degree will tend to be shorter, but more work will be expended per node of the search. On the other hand, a low-degree tree will have greater height, but less work will be expended per node of the search.

We have chosen to use *binary* ($\delta = 2$) trees for all of the experiments reported herein, for two reasons. First, they are simpler and faster to compute, since there are fewer options in how one splits a set in two than how one partitions a set into three or more subsets. Second, analytical evidence suggests that binary trees are better than $\delta$-ary trees, for $\delta > 2$. In particular, if one considers balanced trees (with $n$ leaves) whose internal nodes have degree $\delta \geq 2$, then the amount of work expended in searching a single path from root to leaf is proportional to $f(\delta) = \delta \log_\delta n$. Simple calculus shows that the function $f(\delta)$ is minimized over $\delta \in (1, \infty)$ by setting $\delta = e$ (the base of the natural logarithm). Restricting $\delta$ to integer values greater than one, we see that $f(\delta)$ is minimized by $\delta = 2$. Of course, this analysis does not address the fact that a typical search of a BV-tree will not consist of a single root-to-leaf path. However, from our limited investigation of some typical searches, we have found that our choice of $\delta = 2$ is justified. We leave for future work the thorough experimental investigation of the trade-offs between different values of $\delta$.

### 3.3.2 Top-Down Versus Bottom-Up

In constructing a BV-tree on a set, $S$, of input primitives, we can do so in either a *top-down* or a *bottom-up* manner. A bottom-up approach begins with the input primitives as the leaves of the tree and attempts to group them together recursively (taking advantage of any local information), until we reach a single root node which approximates the entire set $S$. One example of this approach is the "BOXTREE," by Barequet et al. [6].

A top-down approach starts with one node which approximates $S$, and uses information based upon the entire set to recursively divide the nodes until we reach the leaves. OBBTrees [21] are one example of this approach.

In all of our tests reported here, we also construct our BV-trees in a top-down approach. While we have some limited experience with one bottom-up method of tree construction, we do not have enough experience yet in comparing alternatives to be able to make definitive conclusions about which is better; thus, we leave this issue for future investigation.

### 3.3.3 Choice of k-DOPs

Our investigations use 6-dops (AABBs), 14-dops, 18-dops, and 26-dops. More specifically, for our choice of 14-dop, we find the minimum and maximum coordinate values of the vertices of the primitives along each of seven axes, defined by the vectors $(1, 0, 0)$, $(0, 1, 0)$, $(0, 0, 1)$, $(1, 1, 1)$, $(1, -1, 1)$, $(1, 1, -1)$, and $(1, -1, -1)$. Thus, this particular $k$-dop uses the six halfspaces that define the facets of an AABB, together with eight additional diagonal halfspaces that serve to "cut off" as much of the eight corners of an AABB as possible. Our choice of 18-dop also derives six of its halfspaces from those of an AABB, but augments them with 12 additional diagonal halfspaces that serve to cut off the 12 edges of an AABB; these 12 halfspaces are determined by six axes, defined by the direction vectors $(1, 1, 0)$, $(1, 0, 1)$, $(0, 1, 1)$, $(1, -1, 0)$, $(1, 0, -1)$, and $(0, 1, -1)$. Finally, our choice of 26-dop is simply determined by the union of the defining halfspaces for the 14-dops and 18-dops, utilizing the six halfspaces of an AABB, plus the eight diagonal halfspaces that cut off corners, plus the 12 halfspaces that cut off edges of an AABB.

We emphasize that our choice of $k$-dops is strongly influenced by the ease with which each of these bounding volumes can be computed. In particular, the normal vectors are chosen to have integer coordinates in the set $\{-1, 0, 1\}$, implying that no multiplications are required for computing them. We leave to future work the investigation of other (larger) values of $k$, e.g., $k$-dops determined by normal vectors having integer coordinates in the set $\{0, \pm1, \pm2\}$.

Fig. 4a provides an example of each of our $k$-dops. In the center of the picture is the input model: a "spitfire" aircraft. The four other images of Fig. 4a show, from left to right, and top to bottom, the corresponding six-, 14-, 18-, and 26-dop which approximates the spitfire. In a BV-tree of this model, the bounding volumes shown would represent the bounding volume, $b(S)$, associated with the root node (Level 0), for each choice of $k$. Similarly, Figs. 4b, c, and d depict Levels 1, 2, and 5 of the corresponding BV-trees of the spitfire.

### 3.3.4 Splitting Rules for Building the Hierarchies

Each node $v$ in a BV-tree corresponds to a set $S_v$ of primitive objects, together with a bounding volume (BV), $b(S_v)$. In constructing effective BV-trees, our goal is to assign subsets of objects to each child, $v'$, of a node $v$, in such a way as to minimize some function of the "sizes" of the children, where the *size* is typically the volume or the surface area of $b(S_{v'})$. For ray tracing applications, the objective is usually to minimize the surface area, since the probability that a ray will intersect a BV is proportional to its surface area. For collision detection, though, we minimize the volume, expecting that it is proportional to the probability that it intersects another object.

Since we are using binary trees, the assignment of objects to children reduces to the problem of partitioning $S_v$ in two. There are $\frac{1}{2}\left(2^{|S_v|} - 2\right)$ different ways to do this; thus, we cannot afford to consider all partitions. Instead, we associate each triangle of $S_v$ with a single "representative" point (we use the centroid), and we split $S_v$ in two by picking a plane orthogonal to one of the three coordinate axes, and assigning a triangle to the side of the plane where the centroid lies. This results in at most $3 \cdot (|S_v| - 1)$ different nontrivial splits, since there are three choices of axis and, for each axis, there are $|S_v| - 1$ different splits of the centroid points.

**Choice of Axis.** We choose a plane orthogonal to the $x$-, $y$-, or $z$-axis based upon one of the following objective functions:

*Min Sum:* Choose the axis that minimizes the sum of the volumes of the two resulting children.

*Min Max:* Choose the axis that minimizes the larger of the volumes of the two resulting children.

*Splatter:* Project the centroids of the triangles onto each of the three coordinate axes and calculate the variance of each of the resulting distributions. Choose the axis yielding the largest variance.

*Longest Side:* Choose the axis along which the $k$-dop, $b(S_v)$, is longest.

The amount of time required to evaluate each of the above objective functions varies greatly, and this leads to corresponding variation in the preprocessing time to build a BV-tree. The "longest side" method is the fastest, requiring only three subtractions and two comparisons to determine which axis to choose. The next fastest is the "splatter" method, which runs in linear time, $O(|S_v|)$. The slowest methods are "min sum" and "min max," which both require that we calculate the volumes occupied by each of the three pairs of possible children; this requires time $O(k|S_v|)$ to compute the six $k$-dops for the candidate children, plus $O(k \log k)$ to compute the volumes of these $k$-dops.[4]

In Section 5, we report on the results of experiments comparing these four methods of selecting the axis. (See Tables 4 and 5.) The default method in the current software is the "splatter" method, which, while giving slightly worse collision detection times than the "min sum" method, gives a preprocessing time that is an order of magnitude less than "min sum."

An interesting question for future work is to investigate the effect of allowing the axis to be chosen from a larger set, e.g., it may be beneficial to permit the axis to be in any of the $k/2$ directions that define the $k$-dops that we are using in the BV-tree. Of course, any such potential improvement in collision detection time must be weighed against the increased cost of preprocessing.

**Choice of Split Point.** Once we have chosen the axis that will be orthogonal to the splitting plane, we must determine the position of the splitting plane, from among the $|S_v| - 1$ possibilities.

We have investigated in depth two natural choices for the splitting point: the *mean* of the centroid coordinates (along the chosen axis), or the *median* of the centroid coordinates.

In the prior work of Held et al. [24], the median was always used for splitting, with the rationale that one wants to obtain the most balanced possible BV-tree.

However, here we investigated also the option of splitting at the mean, in case this results in a tighter fitting bounding volume approximation, while not harming the balance of the tree too severely. In fact, in the earlier work of Held et al. [25], experiments showed that the total volume of

the BVs in the tree was *less* in the case of splitting at the mean versus the median. Since the total volume associated with the tree may be considered to be a good indicator of the quality of approximation, this previous work suggested that we should investigate the impact of this choice (median versus mean) on the efficiency of collision detection.

For the datasets reported in Section 5.2, we compared the number of operations required for collision detection ($N_v$, $N_p$, and $N_u$) when the hierarchies were built using each of the two choices. In *every* test run, there were more operations performed when using the median than when using the mean. Thus, even though the hierarchies were usually deeper when using the mean, the overall amount of work done during the collision detection checks was less due to the better approximations. In addition, the average collision detection time was also *greater* in every case when using the median: the smallest increase being 1 percent, and the largest increase being 35 percent. It thus became clear that the tighter approximations provided by using the mean outweighed the better balanced trees produced by using the median. For more details on these experiments, please refer to Tables 4 and 5 in Section 5.2.

Our implementation selects between only these two possibilities (the mean or the median). We can, however, propose some alternatives for future investigation in the optimizing of the splitting decision, depending on how much preprocessing time is available for constructing the hierarchy: We could optimize over

1) *all* $|S_v| - 1$ different centroid coordinates, or
2) a random subset of these coordinates.

## 4 COLLISION DETECTION USING BV-TREES

We turn now to the problem of how best to use the flying hierarchy, BVT($F$), and the environment hierarchy, BVT($E$), to perform collision detection (CD) queries. In processing these CD queries, we consider choices of:

1) the method of updating the $k$-dops in the flying hierarchy as the flying object rotates, so that they continue to approximate the same subset of primitive objects;
2) the algorithm for comparing the two BV-trees to determine if there is a collision;
3) the depth of the flying hierarchy; and
4) the order in which to perform the $k/2$ interval overlap tests when testing two $k$-dops for intersection.

### 4.1 Tumbling the BV-Trees

For each position of the flying object in the scene, we will need to have a BV-tree representing the flying hierarchy, in order to be able to perform CD queries efficiently. If the flying object were only to *translate*, then the BV-tree that we construct for its initial position and orientation would remain valid, modulo a translation vector, in any other position. However, the flying object also *rotates*. This means that if we were to transform (translate and rotate) each bounding $k$-dop, $b(S_v)$, represented at each node of the flying hierarchy, we would have a new set of bounding $k$-dops, forming a valid BV-tree for the transformed object, *but* the normal vectors defining them would be a different set of

---

4. The volume of a $k$-dop can be computed by first finding the B-rep, to identify the vertices, and, then, summing the volumes of the $O(k)$ tetrahedra in a tetrahedralization of the $k$-dop, e.g., obtained simply from the vertex information. The B-rep of the $k$-dop can be found in time $O(k \log k)$, as explained in Section 4.1.
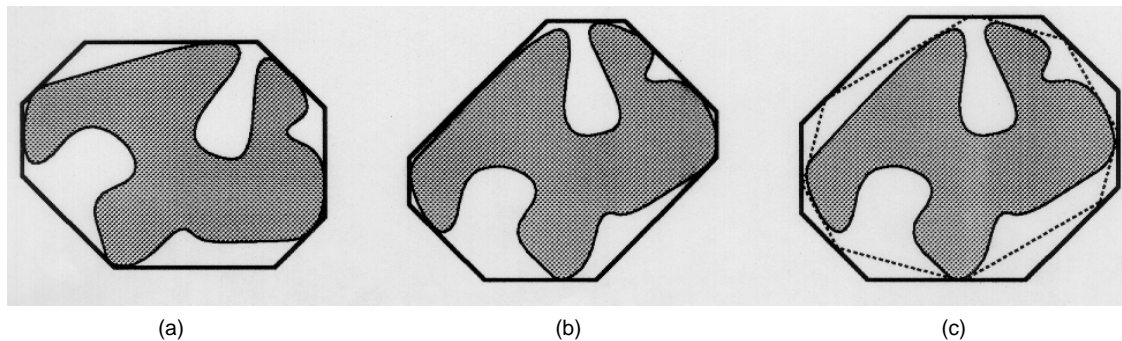
Fig. 2. Illustration of the approximation method of handling a rotating object.

$k$ vectors than those defining the $k$-dops in the environment hierarchy (which did not rotate). This would defeat the purpose of having $k$-dops as bounding volumes, since the overlap test between two $k$-dops having *different* defining normal vectors is far more costly than the conservative disjointness test used for aligned $k$-dops. Thus, it is important to address the issue of "tumbling" the bounding $k$-dops in the flying hierarchy. The cost of each such updating operation has been denoted by $C_u$ in (2).

One "brute force" approach to this issue is to recompute the entire flying hierarchy at each step of the flight. This is clearly too slow for consideration. A somewhat less brute force approach is to preserve the structure of the flying hierarchy, with no changes to the sets $S_v$, but to update the bounding $k$-dops for each node of the flying hierarchy, at each step of the flight. This involves finding the new maximum and minimum values of the primitive vertex coordinates along each of the $k/2$ axes defining the $k$-dops. This is still much too costly, both in terms of time and in terms of storage, since we would have to store with each node the coordinates of all primitive vertices (or at least those that are on the convex hull of the set of vertices in $S_v$).

So, we considered two other methods to tumble the nodes $v$, while preserving the structure of the hierarchy:

1) A "hill climbing" algorithm that stores the B-rep (boundary representation) of the convex hull of $S_v$, and uses it to perform *local* updates to obtain the new (exact) bounding $k$-dop from the bounding $k$-dop of $S_v$ in the previous position and orientation. The local updates involve checking a vertex that previously was extremal (say, maximal) in one of the $k/2$ directions, to see if it is still maximal; this is done by examining the neighbors of the vertex. If a vertex is no longer maximal, then we "climb the hill" by going to a neighboring vertex whose corresponding coordinate value increases the most. By its very nature, this algorithm exploits step-to-step coherence, requiring less time for updates corresponding to smaller rotations. The worst-case complexity, though, is $O(k^2)$, and this upper bound is tight, since each of the $k$ extremal vertices may require $\Omega(k)$ local moves on the B-rep to update.

2) An "approximation method" that attempts only to find an *approximation* (an outer approximation) to the true bounding $k$-dop for the transformed $S_v$. This method stores only the vertices, $V(S_v)$, of the $k$-dop $b(S_v)$, com-

puted once, in the model's *initial orientation*.[5] Then, as $S_v$ tumbles, we use the "brute force" method to compute the exact bounding $k$-dop of the transformed set $V(S_v)$; this bounding $k$-dop still contains the transformed $S_v$, but it need not be the smallest $k$-dop bounding it.

Fig. 2 shows a two-dimensional example of method 2. In this example, $k = 8$, and the original object and eight-dop are shown in Fig. 2a. Fig. 2b depicts the object rotated 30 degrees (counterclockwise) and the corresponding eight-dop. The result of tumbling the original $k$-dop and recomputing the new $k$-dop is shown in Fig. 2c. The dashed lines represent the rotated (original) eight-dop, and the solid lines show the new eight-dop that we use to approximate the object. Ideally, we want our approximate eight-dop to be very close to the exact eight-dop shown in Fig. 2b. Note that a tumbled $k$-dop need not be strictly larger than the exact $k$-dop of a rotated object (although this is typically the case). For instance, for the eight-dop depicted in the figure, rotating the object by 45 degrees causes the tumbled $k$-dop to coincide with the exact $k$-dop.

Both methods 1 and 2 rely on a preprocessing step in which we compute a B-rep. In method 1, we precompute the convex hull of the vertices of $S_v$, and store the result in a simple B-rep. In method 2, we must compute the vertices in the B-rep of the $k$-dop $b(S_v)$, for the original orientation of $S_v$. This means that we must compute the intersection of $k$ halfspaces. This is done by appealing to the following fact (see, e.g., [36]): The intersection of a set of halfspaces can be determined by computing the convex hull of a set of points (in 3D), each of which is *dual*[6] to one of the planes defining the halfspaces, and then converting the convex hull back to primal space; a vertex, edge, facet of the convex hull corresponds to a facet, edge, vertex of the intersection of halfspaces. We compute the convex hull of the dual points in 3D using a simple incremental insertion algorithm (see [36]).[7]

We have considered some of the trade-offs between methods 1 and 2. The nodes closest to the root of the flying hierarchy are the most frequently visited during searching. Thus, it is important that the bounding $k$-dops for these nodes be as tightly fitting as possible, so that we can hopefully prune off a branch of the tree here. This suggests that we apply method 1 at the root node, and at nodes "close" to the root node of the flying hierarchy.

We implemented and tested our algorithm using both methods, and conducted experiments to determine if the extra cost of method 1 was worth it for nodes near the root of the hierarchy. In all cases, it was worthwhile spending the time to compute the exact bounding $k$-dop at the root node; the time saved due to pruning greatly outweighed the additional time spent doing the hill-climbing. We also performed experiments in which we applied method 1 to nodes on levels of the tree close to the root. However, we found that this additional overhead was not justified; the time saved due to additional pruning did not outweigh the extra time required to perform the hill-climbing. In fact, the total running time *increased* when using method 1 for any nodes other than the root node. Consequently, we are currently using the approximation method 2 for all nodes in the hierarchy, except the root node, where we perform hill-climbing 1.

An interesting future research question is also suggested here. The flying hierarchy is constructed according to the object's initial position and orientation, as it is given to us. An alternative to this is to try finding an "optimal" orientation for the flying object, where "optimal" could possibly be interpreted as the orientation that minimizes the total volume of the hierarchy or that allows for the most efficient collision checks.

## 4.2 Tree Traversal Algorithm

Given the environment hierarchy, BVT($E$), and the flying hierarchy, BVT($F$) (after tumbling), we must traverse the two trees efficiently to determine if any part of the flying object collides with some part of the environment. The algorithm we use is outlined in Algorithm 1. It consists of a recursive call to *TraverseTrees*($v_F$, $v_E$), where $v_F$ is the current node of the flying hierarchy and $v_E$ is the current node of the environment hierarchy. Initially, we set $v_F$ and $v_E$ to be the root nodes of the hierarchies.

> **Algorithm** *TraverseTrees*($v_F$, $v_E$)
> **Input:** A node $v_F$ of the flying hierarchy, a node $v_E$ of the environment hierarchy
> 1. **if** $b(v_F) \cap b(v_E) \neq \varnothing$ **then**
> 2.    **if** $v_E$ is a leaf **then**
> 3.      **if** $v_F$ is a leaf **then**
> 4.        **for** each triangle $t_E$ of $S_{v_E}$
> 5.          **for** each triangle $t_F$ of $S_{v_F}$
> 6.           check test triangles $t_E$ and $t_F$ for intersection
> 7.      **else**
> 8.        **for** each child $v_f$ of $v_F$
> 9.          TraverseTrees($v_f$, $v_E$)
> 10.    **else**
> 11.      **for** each child $v_e$ of $v_E$
> 12.        TraverseTrees($v_F$, $v_e$)
> 13. **return**

**Algorithm 1:** Pseudocode of the tree traversal algorithm.

At a general stage of the traversal algorithm, we test for overlap between the bounding volume $b(S_{v_F})$ and the bounding volume $b(S_{v_E})$. If they are disjoint, then we are done with this call to the function. Otherwise, if $v_E$ is *not* a leaf, we step down one level in the environment hierarchy, recursively calling *TraverseTrees*($v_F$, $v_e$) for each of the children $v_e$ of $v_E$. If $v_E$ *is* a leaf, then we check if $v_F$ is a leaf: If it is, we do triangle-triangle intersection tests between each triangle of $v_E$ and each triangle of $v_F$; otherwise, we step down one level in the flying hierarchy, recursively calling *TraverseTrees*($v_f$, $v_E$) for each of the children $v_f$ of $v_F$.

For comparison purposes, we have also implemented a variant of this traversal algorithm in which line 9 of the algorithm is replaced by *TraverseTrees*($v_f$, root of the environment hierarchy). The rationale for this variant is that it may be that the bounding volume at a node $v_F$ of the flying hierarchy intersects a large number of leaves in the environment hierarchy, BVT($E$), while the children of $v_F$ form a much tighter approximation and intersect far fewer leaves of BVT($E$). (This is especially true for nodes of the flying hierarchy, since our approximation method of tumbling $k$-dops results in "looser" fitting bounding volumes.) Thus, by restarting the search at the root of BVT($E$), for each child of $v_F$, we may actually end up with fewer overlap checks in total. We have found experimentally, though, that this variant does not perform as well in practice as what we describe in Algorithm 1. While there are cases in which this variant is better, yielding a slightly lower (by about 5 percent) average CD time, overall it usually is inferior. In particular, for the suite of experiments reported in this paper, the variant is almost always slower, in some cases by as much as 10-20 percent.

## 4.3 Depth of the Flying Hierarchy

The depth of the flying hierarchy has a significant impact upon the total cost, $T$, associated with performing a collision detection query, since it can affect the values $N_v$, $N_p$, and $N_u$ in (2). A deeper hierarchy will tend to increase the number of bounding volume overlap tests ($N_v$) and the number of nodes that have to be updated ($N_u$), but to decrease the number of pairs of primitives (triangles) which will be checked for contact ($N_p$). A shallower tree will tend to have the opposite effect.

The problem of selecting the optimal depth is difficult to address in general because it is highly data-dependent, as well as dependent upon the costs $C_v$, $C_p$, and $C_u$. At the moment, we have "hard-coded" a threshold $\tau$; once the number of triangles associated with a node falls below $\tau$, we consider this node to be a leaf of the tree. For all of the experiments reported in this paper, we used a threshold of $\tau = 1$ for the environment hierarchy, and a threshold of $\tau = 40$ for the flying hierarchy. These values were determined to work well on a large variety of datasets. We leave it as an open problem to determine effective methods of automatically determining good thresholds, or of allowing variable thresholds at different nodes within a hierarchy.

## 4.4 Overlap and Intersection Tests

While processing a CD query, the most frequently called function is usually that of testing whether or not two $k$-dops overlap. The cost of this operation has been denoted by $C_v$ in (2). Recall that all our $k$-dops are defined by the same fixed set of directions for any particular $k$. Thus, a $k$-dop is completely defined by $k/2$ intervals describing the extents along those directions. Two $k$-dops $D_1$, $D_2$ do *not* overlap if at least one of the $k/2$ intervals of $D_1$ does not overlap the corresponding interval of $D_2$. If the $k$-dops overlap along *all* $k/2$ directions, then we conclude that they *may* overlap. They may be disjoint, separated by a plane parallel to one edge from each $k$-dop; however, for efficiency, we use a *conservative* disjointness test based on only the $k/2$ directions. Thus, we need at most $k$ floating-point comparisons, and no arithmetic operations, in our overlap test.

In performing this overlap test, the order in which we check the $k/2$ intervals may have an effect upon the efficiency of the primitive. For example, it seems likely that, if the intervals defined by one direction overlap, then the intervals defined by another direction, which is fairly "close" to the first one, will also result in an overlap. Thus, we would like to order the interval tests so that we test intervals with largely different directions (one after the other). In doing so, we hope quickly to find a direction (if one exists) along which the given intervals do not overlap, and, thus, exit the routine. This is an interesting question for further study.

Finally, at the lowest level of our CD query algorithm, we ultimately must be able to test whether or not two primitives (triangles) intersect. The cost of this operation has been denoted $C_p$; it involves arithmetic operations on floating-point numbers. We have developed a collection of efficient intersection tests for pairs of primitive geometric elements; see Held [23] for details on the triangle-triangle intersection test that we use.

## 5 IMPLEMENTATION AND EXPERIMENTATION

Our algorithms have been implemented in C and tested upon a variety of platforms (SGI, Sun, PC). They run on general polygonal models (often called "polygon soup"), and can easily handle cracks,[8] (self-)intersections, and other deficiencies of the input data. We assume that the input consists simply of a list of vertices and a list of triangles without any adjacency information.[9]

Our BV-tree construction and collision detection algorithms are robust and relatively simple; they do not make any decisions based upon the topology of the data, so cannot run into inconsistency problems (due to floating-point errors) when searching (or building) the BV-trees. To avoid missing collisions between objects, we use an epsilon threshold, $\epsilon$, which can be specified by the user.

In order to maintain efficiency in the implementation of $k$-dops, we have "hardcoded" the logic for each of the four

choices of $k$. Therefore, we choose the value of $k$ (and the appropriate code) at compile time by means of compiler switches.

Throughout this section, we report on some comparisons with the system called "RAPID" (Rapid and Accurate Polygon Interference Detection), which has been made publicly available by the University of North Carolina at Chapel Hill.[10] This library utilizes oriented bounding box trees (OBBTrees) [21].

**Memory Requirements.** For an environment dataset of $n$ input triangles, we store in one array ($72n$ bytes) the vertices of the triangles (whose coordinates are eight-byte floating point numbers), and in another array ($12n$ bytes) the integer indices into the vertex array, indicating for each of the $n$ triangles which three vertices comprise it.[11]

For each node of the environment hierarchy, we need to store the $k$ numbers that define the bounding $k$-dop ($8k$ bytes), two pointers to the children of the node (eight bytes), and an integer index to indicate which triangle is stored in each leaf (four bytes). Thus, we need $8k + 12$ bytes per node. There are approximately $2n$ ($2n - 1$, to be exact) nodes in the hierarchy, since it is a complete binary tree, with each leaf containing just one triangle.

In total, we will, therefore, need $(16k + 108)n$ bytes to store all $n$ triangles of the environment, together with the hierarchy. Substituting $k = 6, 14, 18$, and 26, we see that we need 204, 332, 396, and 524 bytes per input triangle, respectively. For comparison, it has been reported in [21] that the RAPID implementation requires 412 bytes per input triangle.

The memory used to store a flying object of $m$ triangles is identical to that of storing the environment ($84m$ bytes). However, the memory needed for the flying hierarchy is more difficult to put into a closed-form expression, since it is highly data dependent. In particular, our threshold, $\tau$, for stopping the construction of the hierarchy (as discussed in Section 4.3), is $\tau = 40$, which means that, instead of having $2m - 1$ nodes in the hierarchy, we will have $2m' - 1$, where $m'$ denotes the number of leaves, which can vary between one and $m$. Also, we need to store the original B-rep vertices of the initial $k$-dops (Section 4.1) with each node, and these numbers vary for each choice of flying object. We can, however, compute worst-case upper bounds on the number of vertices in the B-rep of each of our $k$-dops: for $k = 6, 14, 18, 26$, the maximum possible number of vertices in a $k$-dop is 8, 24, 32, and 48, respectively.

For each node of the flying hierarchy, we store the $k$ numbers that define a $k$-dop ($8k$ bytes), two pointers to the children (eight bytes), the number of triangles bounded by the $k$-dop (four bytes)—since the threshold is not one in this case, the list of triangle indices bounded by this node, the number of original B-rep vertices (four bytes), a list of the B-rep vertices, and an integer to indicate when the node was last "tumbled" (four bytes)—to avoid retumbling the node if it is accessed more than once during the CD query for one step of the flight.

---

8. "Cracks" are gaps on the surface of a polygonal model caused by an edge having only one incident face.
9. For this reason, we can only report surface intersections, rather than volumetric intersections.

10. The library can be found on the web at http://www.cs.unc.edu/~geom/OBB/OBBT.html.
11. These indices are not absolutely necessary: however, since most triangles do share vertices, it is more memory efficient to do so, at the expense of appearing wasteful in our memory calculations here.

In addition, we also need to store the B-rep for the convex hull associated with the root node of the flying hierarchy (Section 4.1). In the experiments reported here, the flying "Pipes" dataset required the most memory, almost 1.65 megabytes, to store its convex hull.

## 5.1 Experimental Set-Up

Our experiments have used real and simulated datasets of various complexities, ranging from tens of triangles to a few hundred thousand triangles. We made a special effort to devise datasets that were particularly difficult for our method and others. For instance, we considered "swept volume" datasets, in which a moving object is swept through space on a random motion, then numerous obstacles are randomly placed close to, but not penetrating, the swept volume; finally, we fly the object on the original path, causing it to come very close to collision with thousands of nearby obstacles, without it actually hitting any of them. While these "challenging" datasets are unlikely to arise in practice, a goal of our study was a systematic comparison of alternative methods and alternative choices of parameters within our own methods.

For all of the results reported here, we used a Silicon Graphics Indigo$^2$, with a single 195 MHz IP28/R10000 processor, 192 Mbytes of main memory, and a Maximum Impact Graphics board. The code was compiled with GNU gcc (respectively, g++ for RAPID). All timings were obtained by adding the system and user times reported by the C library function "times." In order to smooth out minor variations in the timings, all tests have been run repeatedly, and we report average times.

Although we ran RAPID on the same machine and with the same timing command, we appreciate the difficulty that exists in making comparisons between different algorithms implemented by different people. Many issues, such as tolerances (for overlaps) and what geometric primitives to use and how they are tested for intersection, can play a crucial role in an algorithm's performance. Also, we do not know to which extent RAPID has been optimized to achieve efficiency. (However, RAPID does use assembler code in order to speed up computations, which serves as an indication that it has certainly been optimized to some extent.)

## 5.2 Experimental Results

### 5.2.1 Average Costs of $C_p$, $C_v$, and $C_u$

We begin by reporting results of an experiment to determine the average cost of testing two *primitives* (triangles) for intersection, using our code. For 100,000 triangle-triangle intersection queries, all of which had their bounding boxes overlap, in order to avoid simple rejections, the average query time per test, $C_p$, was 0.0035 milliseconds (ms).

Next, we investigate how the costs $C_v$ and $C_u$ vary with choice of $k$. In Table 1, we show experimental results comparing the average cost, $C_v$, of testing two $k$-dops for overlap. The table also shows the average time, $C_u$, required to perform updates on the $k$-dops, using the approximating $k$-dops method described in Section 4.1. The $k$-dops used in these tests were taken directly from the experiments (flights) described later in this section.

TABLE 1
AVERAGE COSTS OF $C_v$ AND $C_u$ (IN MS), FOR DIFFERENT CHOICES OF $k$

|       | 6-dop  | 14-dop | 18-dop | 26-dop |
|-------|--------|--------|--------|--------|
| $C_v$ | 0.0008 | 0.0016 | 0.0020 | 0.0028 |
| $C_u$ | 0.0045 | 0.0174 | 0.0235 | 0.0509 |

TABLE 2
AVERAGE CD TIME (IN MS), USING OUR "SPLATTER" SPLITTING RULE

|                                    | Pipes   | Torus  | 747     | Swept  | Interior |
|------------------------------------|---------|--------|---------|--------|----------|
| Env. Size (no. tri.)               | 143,690 | 98,114 | 100,000 | 40,000 | 169,944  |
| Object Size (no. tri.)             | 143,690 | 20,000 | 14,646  | 36     | 404      |
| No. of Steps                       | 2,000   | 2,000  | 10,000  | 1,000  | 2,528    |
| No. of Contacts                    | 2,657   | 1,472  | 7,906   | 0      | 84,931   |
| Hier. Method (ms per check)        |         |        |         |        |          |
| 6-dop                              | 0.487   | 0.294  | 1.639   | 0.582  | 4.375    |
| 14-dop                             | 0.392   | 0.191  | 0.760   | 0.153  | 2.701    |
| 18-dop                             | 0.366   | 0.184  | 0.356   | 0.109  | 2.754    |
| 26-dop                             | 0.525   | 0.210  | 0.415   | 0.076  | 2.639    |
| RAPID                              | 0.934   | 0.242  | 0.494   | 0.556  | 4.375    |

### 5.2.2 Average Collision Detection Query Times

Table 2 shows timing data on four typical datasets:

1) *Pipes*: an interweaving pipeline flying among a larger copy of the same system of pipes;
2) *Torus*: a deformed torus flying in the presence of stalagmites;[12]
3) *747*: a Boeing 747 model flying among 25,000 random disjoint tetrahedral obstacles; and
4) *Swept*: an "axis-shaped" polyhedron flying through a swept volume surrounded by 10,000 random tetrahedral obstacles.

In order to simulate motion of these "flying" objects, we implemented a form of "billiard paths": A flying object is moved along a random path, "bouncing" off of obstacles that it hits in the environment. We do not attempt to simulate a real "bounce"; rather, we simply reverse the trajectory when a collision occurs. For a more detailed look at accurately handling collision response, please refer to the work by Moore and Wilhelms [33], Bouma and Vaněček [8], and the large collection of work by Baraff [3], [4], [5].

Timing results for a fifth dataset, *Interior*, are also listed in Table 2. Images of this particular flight are shown in Figs. 5a and 5b. This industrial dataset was provided to us by The Boeing Company and models a small section of the interior of an airplane. The flying object in this case is a model of a "hand," whose path was generated by an engineer at Boeing, using a data-glove, as an example of how one would like to use collision detection when immersed in a virtual environment. Our collision detection algorithms were applied to this flight in order to detect all of the contacts,

12. Datasets 1 and 2 were graciously provided by the University of North Carolina at Chapel Hill.

i.e., all pairs of triangles that are in intersection, during the flight. As seen in Table 2, there were many such contacts for this flight, with an average of 33 contacts per step over the 2,500 steps; it was the intention of the engineer generating the data to provide a "rigorous workout" for CD algorithms.

For comparison, we have recorded the results obtained by using the collision detection library RAPID.

All of the timings reported here give the average CPU-consumption per check for collision, exclusive of rendering and of motion simulation.

Based solely upon these times, our 14-, 18-, and 26-dop methods perform well in comparison with RAPID's OBB method, running faster on all five of the datasets; the only exception being the 14-dop method during the 747's flight on our own generated data. As expected, the six-dop method (i.e., axis-aligned bounding boxes) did not perform as well as these other methods, nor as well as when using OBBs in the RAPID implementation. Out of all of our methods, using an 18-dop for our bounding volume in the BV-trees, appears to be the best. In addition, most of the collision detection times are below two milliseconds (many are even below one millisecond), which allows us to perform these queries at real-time rates.

For the results in Table 2, all of our hierarchies were built using one of our fastest construction algorithms, based upon the "splatter" splitting rule discussed in Section 3.3.4. We chose this algorithm because of its speed, and because of the fast CD query times which were obtained. As our 18-dop method appears to be the best, we have provided the following tables which highlight the amount of preprocessing time required for all of the construction methods (longest side, min sum, min max, and splatter), as well as the CD query times which each method generated.

Table 3 highlights the amount of time (in minutes) it takes to preprocess (build) the environment hierarchy for our 18-dop method, for each of the four construction rules: longest side, min sum, min max, and splatter.[13] Our fastest methods are clearly the "longest side" and "splatter" algorithms, which are essentially equal for all of the datasets. Likewise, the "min sum" and "min max" methods both require about the same amount of work; however, these two methods are typically an order of magnitude slower than the others. The fastest method, in terms of preprocessing time, is RAPID, which requires only about 30-40 percent of the time required by the "splatter" method. The longest preprocessing time that we have witnessed (45 minutes) occurred when using the "min max" method on the *Interior* dataset for the 26-dop method. In order to avoid a lengthy wait each time the code is run on a standard dataset, our software has the option to store the environment hierarchy to a binary file. For this dataset, having 169,944 input triangles, the binary file to store the 26-dop hierarchy is roughly 69 megabytes in size and takes just under 10 seconds to load.

In conjunction with Table 3, Table 4 highlights the corresponding CD query times for each of the construction methods. From this table, it becomes clear that the "min sum" method is typically the best; however, unless one can

13. In each of these cases, we split at the mean rather than the median.

TABLE 3
PREPROCESSING TIME (IN MINUTES),
USING OUR 18-DOP METHOD

| Construction Method | Pipes | Torus | 747 | Swept | Interior |
|---|---|---|---|---|---|
| Longest Side | 3.61 | 1.61 | 1.69 | 0.31 | 5.78 |
| Min Sum | 26.75 | 19.12 | 20.98 | 7.17 | 31.03 |
| Min Max | 28.03 | 19.16 | 20.87 | 7.19 | 31.13 |
| Splatter | 3.63 | 1.62 | 1.71 | 0.32 | 5.71 |
| RAPID | 1.05 | 0.69 | 0.71 | 0.26 | 1.31 |

TABLE 4
AVERAGE CD TIME (IN MS), USING OUR 18-DOP METHOD,
DIVIDING AT THE MEAN

| Construction Method | Pipes | Torus | 747 | Swept | Interior |
|---|---|---|---|---|---|
| Longest Side | 0.384 | 0.192 | 0.366 | 0.111 | 3.036 |
| Min Sum | 0.356 | 0.185 | 0.330 | 0.108 | 2.667 |
| Min Max | 0.391 | 0.191 | 0.439 | 0.111 | 2.783 |
| Splatter | 0.366 | 0.184 | 0.356 | 0.109 | 2.754 |

TABLE 5
AVERAGE CD TIME (IN MS), USING OUR 18-DOP METHOD,
DIVIDING AT THE MEDIAN

| Construction Method | Pipes | Torus | 747 | Swept | Interior |
|---|---|---|---|---|---|
| Longest Side | 0.476 | 0.193 | 0.412 | 0.116 | 3.164 |
| Min Sum | 0.450 | 0.192 | 0.359 | 0.111 | 2.822 |
| Min Max | 0.530 | 0.196 | 0.450 | 0.114 | 3.080 |
| Splatter | 0.481 | 0.194 | 0.396 | 0.113 | 2.774 |

afford to spend a great deal of additional time preprocessing the environments, the best choice appears to be the "splatter" method, as it takes considerably less time to preprocess and provides CD query times that are nearly as good.

In addition to the four construction methods that we have been mentioning, we also discussed, in Section 3.3.4, the option of splitting based on the mean versus the median of the centroid coordinates along the selected axis. In the preceding tables, we have always used the mean. To provide some justification for our using the mean by default, we have included Table 5, which shows the average CD query time for the 18-dop method for each of the four splitting rules when we use the median instead of the mean.

In comparing Tables 4 and 5, we see that using the median never results in faster query times. In quite a few cases, the median method is at least 5 percent slower than the mean method, and, in the "Pipes" dataset, the median method is between 24 percent and 35 percent slower for all of the entries. The preprocessing times required for the median method are almost identical to those of the mean. In some cases, it is slightly faster, in others, slightly slower.

As Tables 2, 3, 4, and 5 report on (random) flight paths which we ourselves have generated (with the exception of the *Interior* flight), we have also tried to design experiments in which other methods will perform well, in order to make this a fair comparison. In particular, the OBBTrees in RAPID are reported to perform especially well in situations in which there exists "parallel close proximity" between the models [21]. This situation occurs when many points on the flying object come close to several points in the environment, and a large number of the nodes of the hierarchies

TABLE 6
NUMBERS OF OVERLAP QUERIES AMONG $k$-DOPS
OF THE 2,000-FACETED NESTED SPHERES,
FOR DIFFERENT VALUES OF ALPHA AND $k$

| Hier. | alpha | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Method | 0.55 | 0.1 | 0.055 | 0.01 | 0.0055 | 0.001 | 0.00055 | 0.0001 |
| 6-dop | 388 | 49,494 | 76,506 | 109,086 | 113,200 | 116,340 | 116,710 | 116,948 |
| 14-dop | 32 | 16,888 | 41,782 | 85,656 | 90,896 | 95,150 | 95,564 | 96,056 |
| 18-dop | 46 | 11,236 | 34,744 | 79,968 | 86,036 | 91,482 | 92,124 | 92,684 |
| 26-dop | 22 | 4,652 | 23,774 | 74,052 | 81,160 | 87,622 | 88,322 | 88,968 |
| RAPID | 121 | 3,333 | 7,479 | 41,645 | 60,327 | 91,983 | 95,717 | 100,047 |

TABLE 7
NUMBERS OF OVERLAP QUERIES AMONG $k$-DOPS
OF THE 20,000-FACETED NESTED SPHERES,
FOR DIFFERENT VALUES OF ALPHA AND $k$

| Hier. | alpha | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Method | 0.55 | 0.1 | 0.055 | 0.01 | 0.0055 | 0.001 | 0.00055 | 0.0001 |
| 6-dop | 278 | 289,126 | 494,278 | 1,129,398 | 1,223,900 | 1,320,158 | 1,329,154 | 1,337,116 |
| 14-dop | 14 | 85,012 | 239,884 | 831,528 | 960,952 | 1,102,260 | 1,115,030 | 1,127,908 |
| 18-dop | 46 | 55,390 | 194,414 | 762,668 | 903,056 | 1,063,346 | 1,079,676 | 1,095,868 |
| 26-dop | 14 | 12,218 | 119,556 | 675,152 | 831,104 | 1,019,272 | 1,038,126 | 1,058,072 |
| RAPID | 117 | 2,441 | 5,495 | 43,589 | 87,071 | 428,027 | 609,843 | 932,561 |

will have to be searched in order to resolve all of the conflicts. Examples of this situation are in virtual prototyping and tolerance analysis applications [21]. Therefore, we have run an experiment similar to one run in [21], in order to see if our methods based on $k$-dops are competitive in this situation.

We have generated datasets consisting of polygonal approximations to two concentric spheres, with the outer sphere having radius 1.0, and the inner sphere being a scaled copy of the outer sphere, having radius $1.0 - \alpha$, for small positive values of $\alpha$. In this "parallel close proximity" situation, all of the points of the inner sphere are very close to points on the outer sphere, yet there is no intersection between the inner and the outer surfaces.

Here, as in [21], our objectives is to determine how many bounding volume overlap queries, $N_v$, are required to process the collision detection query: Does the inner surface intersect the outer surface?

Now, as previously discussed, our default implementation uses a threshold of $\tau = 40$ to terminate the construction of the flying hierarchies. However, RAPID uses no such threshold; it always builds a complete binary tree. Thus, in order to make a fair comparison, we modified our code for this particular experiment to be consistent with RAPID, by using a threshold $\tau = 1$ for the flying hierarchy. Then, both methods produce trees having an identical number of internal nodes and leaf nodes. (The structures of the hierarchies, and in particular their heights, can, of course, be different.)

Tables 6 and 7 report our results for spheres of 2,000 triangles each, and spheres of 20,000 triangles each.[14] It came as no surprise that the RAPID implementation of OBBTrees requires fewer bounding volume comparisons than the axis-aligned bounding boxes (6-dops). In fact, for the nested spheres of 20,000 triangles, the OBBs often require over an order of magnitude fewer queries; this is consistent with the conclusion drawn from the similar experiment in [21].

Our goal here, though, was to compare the OBB method to the $k$-dops methods. As the tables show, for both of the datasets, our 14-, 18-, and 26-dop methods performed fewer bounding volume overlap queries for the largest value of $\alpha$, 0.55, when the nested spheres are relatively well separated. For the remaining values of $\alpha$, however, the OBBTrees perform considerably fewer overlap queries in the spheres dataset having 20,000 triangles. Also, OBBTrees perform fewer queries in the smaller dataset, although not by the same magnitude. Once $\alpha$ becomes small enough (0.001), which happens when the nested spheres are *very* close to one another, the $k$-dop methods start to overtake the OBB method.

### 5.2.3 Behavior of CD Time Over Flight

While we have compiled our results primarily using the statistic of *average-case* collision detection time, it is important in some applications to study the *worst-case* collision detection time for the flight of a moving object. On a typical flight (that of the "Pipes" being flown within the larger system of "Pipes"), we show a plot in Fig. 3 of how the CD time varies with position along the flight, over the 2,000 steps in the simulation. One can see that the CD time increases substantially at various positions along the flight; these correspond to when the flying object comes in very close proximity to the environment. In this particular example, the maximum CD query time is roughly 18 milliseconds.

Putting an upper bound on worst-case CD time is especially important in VR applications, where one needs to perform time-critical collision detection [28]. In such situations, our algorithms can be applied, and terminated early (according to the time budgeted for each CD test), resulting in an answer of "maybe": The flying object *might* be intersecting the environment at this instant. The goal, then, in using the BV-tree is to use the information present in the search of the BV-tree, at the time of early termination, to obtain bounds on how much penetration there can be (if at all) between the flying object and the environment. (See, e.g., [47].) This problem is left for future investigations.
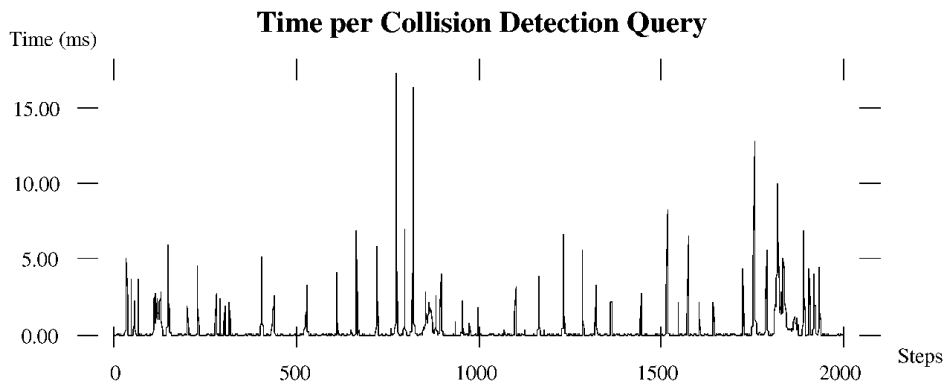
---

14. For these runs, we used one of our fastest construction algorithms, based on the "splatter" splitting rule.

**Time per Collision Detection Query**



Fig. 3. Individual collision detection query times for the "Pipes" dataset.

## 6 CONCLUSION

We have proposed a method for efficient collision detection among polygonal models, based on a bounding volume hierarchy (BV-tree) whose bounding volumes are $k$-dops (discrete orientation polytopes). Our $k$-dops form a natural generalization of axis-aligned bounding boxes, providing flexibility through the choice of the parameter $k$. We have studied the problem of updating an approximate bounding $k$-dop for moving (rotating) objects, and we have studied the application of BV-trees to the collision detection problem.

Our methods have been implemented and tested, for a variety of datasets and various choices of the design parameters (e.g., $k$). Our results show that our methods compare favorably with a leading system ("RAPID," presented at ACM SIGGRAPH '96 [21]), whose hierarchy is based on oriented bounding boxes. Further, our algorithms are robust, relatively simple to implement, and are applicable to general sets of polygonal models. Experiments have shown that our algorithms can perform at interactive rates on real and simulated data consisting of hundreds of thousands of polygons.

### 6.1 Extensions and Future Work

Throughout the paper, we have mentioned several possible extensions of our work, including some alternative methods for constructing BV-trees, such as

- using values of $k$ larger than 26 for our $k$-dops (Section 3.3.3),
- using alternative "splitting rules" (Section 3.3.4), and
- using a bottom-up method to construct BV-trees (Section 3.3.2).

We have also suggested some possible future investigations that could lead to faster collision detection queries, including

- finding an "optimal" orientation of the initial flying hierarchy (Section 4.1),
- avoiding a hard-coded threshold to control the depth of the hierarchies (Section 4.3), and
- using a specially designed ordering when performing interval overlap queries (Section 4.4).

In addition to these "design" alternatives, we plan to investigate further extensions of our BV-tree methods, including:

**Use of temporal coherence.** From one time step to the next, the flying object will occupy roughly the same area of our workspace and, thus, overlap roughly the same set of nodes of the environment hierarchy. It should be possible to give our search algorithm a "hot start" at each step, thereby (potentially) greatly reducing the number of bounding volume overlap calls. The use of coherence may also help address the problem raised at the end of the last section—that of bounding the worst-case query time, and providing an estimate of depth of possible penetration, should the query be terminated before completion.

**Multiple flying objects.** Currently, our collision detection software is programmed to handle only one flying object. Incorporating multiple objects is particularly trivial if we use a brute-force approach, quadratic in the number of flying objects: Check each flying hierarchy against the environment hierarchy, and check every pair of flying hierarchies. If the number of flying objects is relatively small, this approach may be acceptable. However, if the number of flying objects is large, one can apply a "sweep and prune" technique, similar to the one used in [11], or, possibly, design effective new strategies.

**Dynamic environments.** Allowing the environment to change via *insertions* and *deletions* of objects is an important extension for work on environments that are constantly being modified, e.g., a CAD model that is under development and is being edited on a daily basis. The interesting research issue is that of efficiently rebalancing the BV-tree hierarchies under a sequence of insertions and deletions.

**Deformable objects.** In addition to allowing dynamic environments, we would also like to extend our hierarchies to handle deformable objects. By "buffering" (enlarging slightly) the $k$-dops in our *BV-trees*, we can continue to approximate the deformed objects over a short period of time (depending on the velocity of deformation). But, rebalancing or rebuilding sections of the hierarchy will also be necessary over the course of time, and it is an interesting topic for future investigation to devise efficient means for doing so.
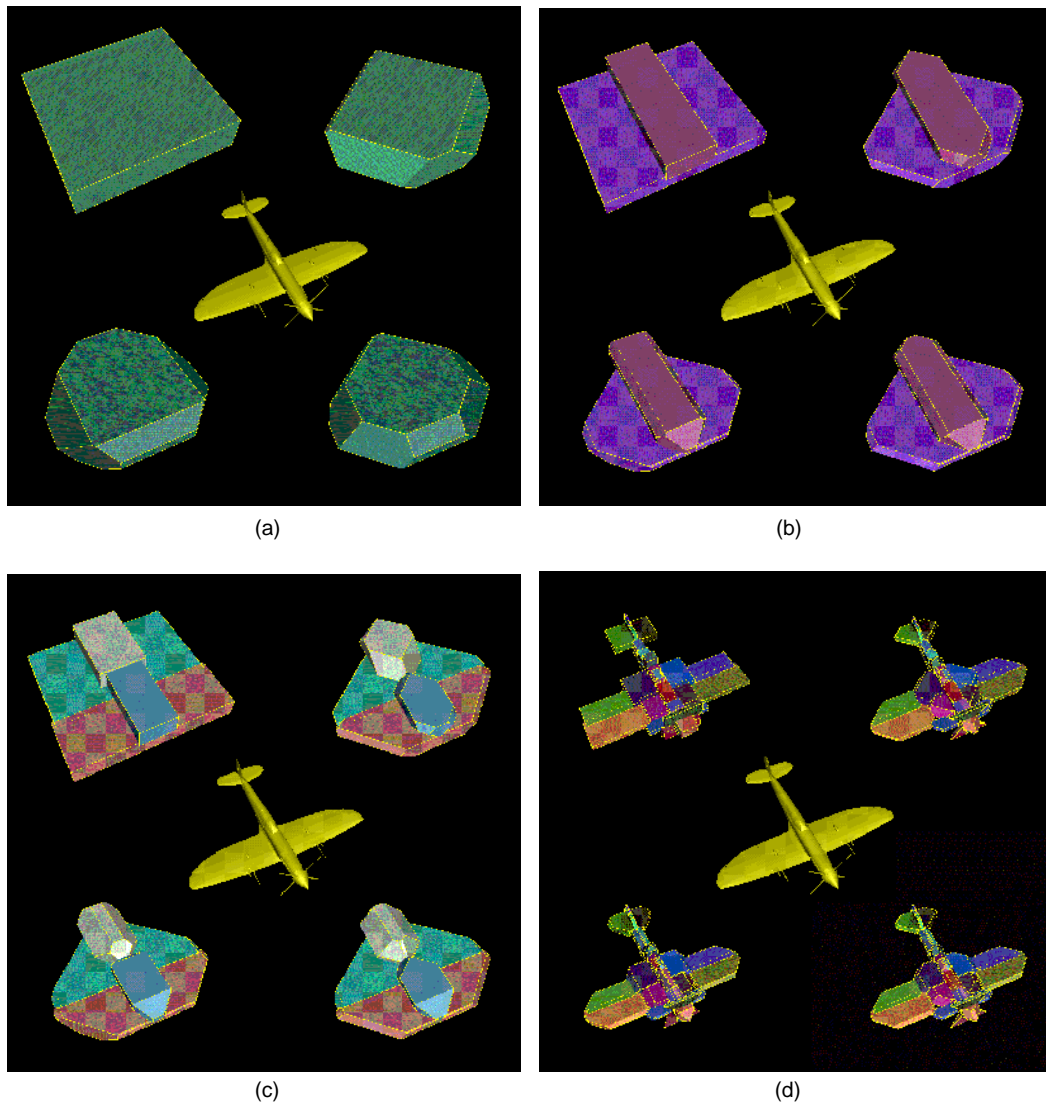
Fig. 4. A spitfire aircraft and the corresponding *k*-dops.
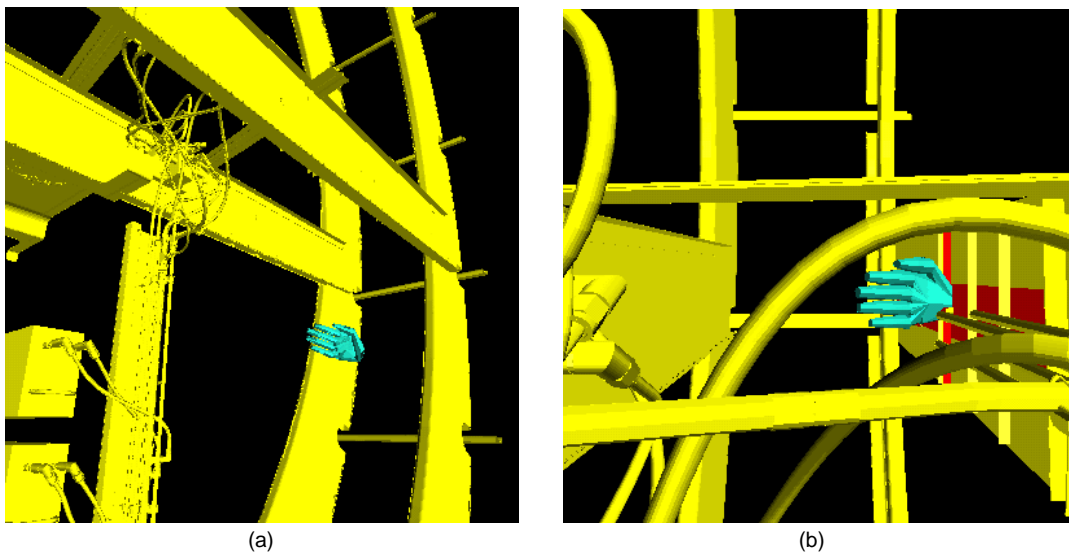


Fig. 5. (a) A hand moving within the "Interior" dataset. (b) A Close-up of the "interior" dataset. The contact region is highlighted in red.

**NC verification.** Our methods may be applied to the task of verifying tool paths in NC (Numerically Controlled) machining, where it is important to check whether a tool penetrates (beyond a specified threshold) the surface of a part to be machined, at any position along the tool's motion. This problem constitutes quite a challenge for a general-purpose CD code since, by the very nature of the tool motion, which is designed to sculpt the part, the tool will be in constant contact with the part's surface. Further, this application requires an extension of our CD code in order to be able to handle spheres and cylinders (without using polyhedral approximations), and (approximate) swept volumes.

## ACKNOWLEDGMENTS

The author's web site addresses are:
http://www.ams.sunysb.edu/~jklosow/jklosow.html
http://www.ams.sunysb.edu/~held/held.html
http://www.ams.sunysb.edu/~jsbm/jsbm.html

## REFERENCES

[1] J. Arvo and D. Kirk, "A Survey of Ray Tracing Acceleration Techniques," *An Introduction to Ray Tracing*, A.S. Glassner, ed., pp. 201-262. Academic Press, 1990.

[2] D.H. Ballard, "Strip Trees: A Hierarchical Representation for Curves," *Comm. ACM*, vol. 24, no. 5, pp. 310-321, May 1981.

[3] D. Baraff, "Curved Surfaces and Coherence for Non-Penetrating Rigid Body Simulation," *Computer Graphics (SIGGRAPH '90 Proc.)*, vol. 24, pp. 19-28, Dallas, Tex., Aug. 1990.

[4] D. Baraff, "Fast Contact Force Computation for Nonpenetrating Rigid Bodies," *Computer Graphics (SIGGRAPH '94 Proc.)*, vol. 28, pp. 23-34, Orlando, Fla., July 1994.

[5] D. Baraff, "Interactive Simulation of Solid Rigid Bodies," *Computer Graphics Applications*, vol. 15, no. 3, pp. 63-75, May 1995.

[6] G. Barequet, B. Chazelle, L.J. Guibas, J.S.B. Mitchell, and A. Tal, "BOXTREE: A Hierarchical Representation for Surfaces in 3D," *EuroGraphics '96*, J. Rossignac and F. Sillion, eds., vol. 15, no. 3, pp. C-387–C-484. Blackwell Publishers, Eurographics Assoc., 1996.

[7] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 322-331, 1990.

[8] W. Bouma and G. Vaněček, Jr., "Collision Detection and Analysis in a Physical Based Simulation," *Eurographics Workshop on Animation and Simulation*, pp. 191-203, Vienna, Sept. 19, 1991.

[9] S. Cameron, "Collision Detection by Four-Dimensional Intersection Testing," *IEEE Trans. Robotics and Automation*, vol. 6, no. 3, pp. 291-302, 1990.

[10] J. Canny, "Collision Detection for Moving Polyhedra," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 8, no. 2, pp. 200-209, Mar. 1986.

[11] J.D. Cohen, M.C. Lin, D. Manocha, and M.K. Ponamgi, "I-COLLIDE: An Interactive and Exact Collision Detection System for Large-Scale Environments," *Proc. ACM Interactive 3D Graphics Conf.*, pp. 189-196, 1995.

[12] International Business Machines Corporation, *User's Guide, IBM 3D Interaction Accelerator*$^{TM}$, version 1, release 2.0. IBM T.J. Watson Research Center, Yorktown Heights, N.Y., Sept. 1995.

[13] A. Crosnier and J. Rossignac, "T-BOX: The Intersection of Three Mini-Max Boxes," internal report, IBM T.J. Watson Research Center, Yorktown Heights, N.Y., 1995.

[14] D. Dobkin, J. Hershberger, D. Kirkpatrick, and S. Suri, "Computing the Intersection-Depth of Polyhedra," *Algorithmica*, vol. 9, pp. 518-533, 1993.

[15] D. Dobkin and D. Kirkpatrick, "Fast Detection of Polyhedral Intersection," *Theoretical Computer Science*, vol. 27, pp. 241-253, 1983.

[16] D. Dobkin and D. Kirkpatrick, "A Linear Algorithm for Determining the Separation of Convex Polyhedra," *J. Algorithms*, vol. 6, pp. 381-392, 1985.

[17] D. Dobkin and D. Kirkpatrick, "Determining the Separation of Preprocessed Polyhedra—A Unified Approach," *Proc. 17th Int'l Colloquium Automata, Languages, and Programming, Lecture Notes in Computer Science*, vol. 443, pp. 400-413. Springer-Verlag, 1990.

[18] K. Dobrindt, K. Mehlhorn, and M. Yvinec, "A Complete and Efficient Algorithm for the Intersection of a General and a Convex Polyhedron," *Proc. Third Workshop Algorithms Data Structures, Lecture Notes in Computer Science*, vol. 709, pp. 314-324. Springer Verlag, 1993.

[19] A. Garcia-Alonso, N. Serrano, and J. Flaquer, "Solving the Collision Detection Problem," *IEEE Computer Graphics and Applications*, vol. 14, pp. 36-43, May 1994.

[20] J. Goldsmith and J. Salmon, "Automatic Creation of Object Hierarchies for Ray Tracing," *IEEE Computer Graphics and Applications*, vol. 7, pp. 14-20, 1987.

[21] S. Gottschalk, M.C. Lin, and D. Manocha, "OBBTree: A Hierarchical Structure for Rapid Interference Detection," *Computer Graphics (SIGGRAPH '96 Proc.)*, vol. 30, pp. 171-180, New Orleans, Aug. 1996.

[22] N. Greene, "Detecting Intersection of a Rectangular Solid and a Convex Polyhedron," *Graphics Gems IV*, P.S. Heckbert, ed., pp. 74-82. Academic Press, 1994.

[23] M. Held, "A Library of Efficient and Reliable Intersection Routines," technical report, Dept. of Applied Mathematics and Statistics, State Univ. of New York at Stony Brook, 1997. http://www.ams.sunysb.edu/~held/

[24] M. Held, J.T. Klosowski, and J.S.B. Mitchell, "Evaluation of Collision Detection Methods for Virtual Reality Fly-Throughs," *Proc. Seventh Canadian Conf. Computer Geometry*, pp. 205-210, 1995.

[25] M. Held, J.T. Klosowski, and J.S.B. Mitchell, "Speed Comparison of Generalized Bounding Box Hierarchies," technical report, Dept. of Applied Math, State Univ. of New York at Stony Brook, 1995.

[26] M. Held, J.T. Klosowski, and J.S.B. Mitchell, "Real-Time Collision Detection for Motion Simulation Within Complex Environments," *SIGGRAPH '96 Visual Proc.*, p. 151, New Orleans, Aug. 1996

[27] P.M. Hubbard, "Collision Detection for Interactive Graphics Applications," *IEEE Trans. Visualization and Computer Graphics*, vol. 1, no. 3, pp. 218-230, Sept. 1995.

[28] P.M. Hubbard, "Approximating Polyhedra With Spheres for Time-Critical Collision Detection," *ACM Trans. Graphics*, vol. 15, no. 3, pp. 179-210, July 1996.

[29] T.L. Kay and J.T. Kajiya, "Ray Tracing Complex Scenes," *Computer Graphics (SIGGRAPH '86 Proc.)*, vol. 20, pp. 269-278, Aug. 1986.

[30] M. Lin, "Efficient Collision Detection for Animation and Robotics," PhD thesis, Dept. of Electrical Eng. and Computer Science, Univ. of California, Berkeley, 1993.

[31] M. Lin and J. Canny, "Efficient Algorithms for Incremental Distance Computation," *Proc. IEEE Int'l Conf. Robotics and Automation*, vol. 2, pp. 1,008-1,014, 1991.

[32] M. Lin and D. Manocha, "Fast Interference Detection Between Geometric Models," *Visual Computing*, vol. 11, no. 10, pp. 542–561, 1995.

[33] M. Moore and J. Wilhelms, "Collision Detection and Response for Computer Animation," *Computer Graphics (SIGGRAPH '88 Proc.)*, vol. 22, pp. 289-298, Aug. 1988.

[34] B. Naylor, J.A. Amatodes, and W. Thibault, "Merging BSP Trees Yields Polyhedral Set Operations," *Computer Graphics (SIGGRAPH '90 Proc.)*, vol. 24, pp. 115-124, Dallas, Tex., Aug. 1990.

[35] H. Noborio, S. Fukuda, and S. Arimoto, "Fast Interference Check Method Using Octree Representation," *Advanced Robotics*, vol. 3, no. 3, pp. 193-212, 1989.

[36] J. O'Rourke. *Computational Geometry in C*. New York: Cambridge Univ. Press, 1994.

[37] I. Palmer and R. Grimsdale, "Collision Detection for Animation Using Sphere-Trees," *Computer Graphics Forum*, vol. 14, no. 2, pp. 105-116, June 1995.

[38] M. Ponamgi, D. Manocha, and M. Lin, "Incremental Algorithms for Collision Detection Between General Solid Models," *Proc. ACM SIGGRAPH Symp. Solid Modeling*, pp. 293-304, 1995.

[39] F.P. Preparata and M.I. Shamos, *Computational Geometry—An Introduction*. New York: Springer-Verlag, 1985.

[40] E. Schömer, "Interaktive Montagesimulation mit Kollisionserkennung," PhD thesis, Universität des Saarlandes, Germany, 1994.

[41] E. Schömer and C. Thiel, "Subquadratic Algorithms for the General Collision Detection Problem," *Abstracts 12th European Workshop Computational Geometry (CG '96)*, pp. 95-101, 1996.

[42] G. Vaneček, Jr., "Brep-Index: A Multidimensional Space Partitioning Tree," *Int'l J. Computational Geometry and Applications*, vol. 1, no. 3, pp. 243-261, 1991.

[43] S. Suri, P.M. Hubbard, and J.F. Hughes, "Collision Detection in Aspect and Scale Bounded Polyhedra," *Proc. Ninth ACM-SIAM Symp. Discrete Algorithms*, to appear, Jan. 1998.

[44] H. Weghorst, G. Hooper, and D.P. Greenberg, "Improved Computational Methods for Ray Tracing," *ACM Trans. Graphics*, vol. 3, no. 1, pp. 52-69, Jan. 1984.

[45] G. Zachmann, "Exact and Fast Collision Detection," Diploma thesis, Fraunhofer Inst. for Computer Graphics, Technische Hochschule Darmstadt, Fachbereich Informatik, Germany, 1994.

[46] G. Zachmann and W. Felger, "The BoxTree: Enabling Real-Time and Exact Collision Detection of Arbitrary Polyhedra," *Proc. SIVE '95*, pp. 104-113, 1995.

[47] K. Zikan and P. Konečny , "Lower Bound of Distance in 3D," *Proc. Winter School of Computer Graphics (WSCG '97)*, vol. 3, pp. 640-649, 1997. Available as Technical Report FIMU-RS-97-01, Faculty of Informatics, Masaryk Univ., Czech Republic, Jan. 1997. (http://www.fi.muni.cz/informatics/reports/)
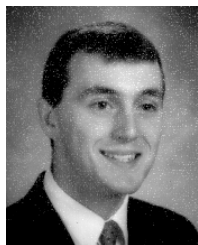
**James T. Klosowski** received a BS in computer science and mathematics from Fairfield University in 1992. He received an MS in applied mathematics from the State University of New York at Stony Brook, where he is currently completing his PhD. In his final year at Stony Brook, Klosowski was awarded the Catacosinos Fellowship for Excellence in Computer Science. His research interests include computer graphics, computational geometry, virtual reality, and the design of algorithms and data structures.

**Martin Held** studied technical mathematics at the University of Linz (Austria), where he received his Dipl.-Ing. degree in 1987. In 1990, he obtained his PhD from the Department of Mathematics and Computer Science at the University of Salzburg (Austria). He serves on the faculty of Computer Science at the University of Salzburg, and is an adjunct associate professor at the State University of New York at Stony Brook. His research interests include algorithms and data structures, computational geometry, and computer graphics. Furthermore, he is actively involved in implementing and testing his research results, and in bridging the gap between theory and practice by applying them to practical problems of CAD/CAM and VR.

**Joseph S.B. Mitchell** received a BS in physics and applied mathematics in 1981, and an MS in mathematics from Carnegie-Mellon University in 1981. He received a PhD in operations research from Stanford University in 1986, while on a Howard Hughes doctoral fellowship and serving on the technical staff at Hughes Research Labs. From 1986 to 1991, Prof. Mitchell served on the faculty of Cornell University. In 1991, he joined the faculty of the State University of New York at Stony Brook, where he is currently a professor of applied mathematics and statistics and a research professor of computer science. Prof. Mitchell has received various research awards (U.S. National Science Foundation Presidential Young Investigator, Fulbright Scholar) and numerous teaching awards, including the President's and Chancellor's Awards for Excellence in Teaching. His primary research area is computational geometry applied to problems in computer graphics, visualization, manufacturing, and geographic information systems.

**Henry A. Sowizral** received a BS in information and computer science from the University of California, Irvine, in 1975. He received an MPhil in 1978 and a PhD in computer science from Yale University in 1982. Through the years, he has been on the research staffs of the Rand Corporation (1980-1985), Schlumberger Palo Alto Research Center (1985-1988), The Research Institute for Advanced Computer Science (RIACS) (1988-1991), and Boeing Research and Technology (1991-1996). He is currently with Sun Microsystems. His primary areas of interest are computer graphics (virtual and augmented reality), parallel and distributed systems, and distributed simulation.

**Karel Zikan** received a BS in mathematics from San Jose State University in 1983. He received an MS in statistics in 1988 and a PhD in operations research in 1989 from Stanford University. He was on the technical staff of the Hughes Research Laboratories from 1985-1989, and on the technical staff of the Research and Technology organization of The Boeing Company from 1990-1996. He served as a Fulbright Visiting Professor on the Faculty of Informatics, Masaryk University, Brno, Czech Republic, from 1995-1996. Currently, he consults for Sun Microsystems. The primary areas of his present research interests are compute graphics, virtual and augmented reality, and wireless communication systems.