EFFICIENT COLLISION DETECTION FOR INTERACTIVE 3D GRAPHICS AND VIRTUAL ENVIRONMENTS

A Dissertation Presented

by

James Thomas Klosowski

to the graduate school in partial fulfillment

of the requirements for the Degree of

Doctor of Philosophy

in

Applied Mathematics and Statistics

State University of New York at Stony Brook

May 1998

© Copyright 1998

by

James Thomas Klosowski

State University of New York at Stony Brook

The Graduate School

James Thomas Klosowski

We, the dissertation committee for the above candidate for the Doctor of Philosophy degree, hereby recommend acceptance of this dissertation.

Joseph S. B. Mitchell, Dissertation Advisor
Professor, Applied Mathematics and Statistics

Steven S. Skiena, Committee Chair
Associate Professor, Computer Science

Esther M. Arkin
Associate Professor, Applied Mathematics and Statistics

Amitabh Varshney
Assistant Professor, Computer Science

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies & Research

Abstract of the Dissertation

Efficient Collision Detection for Interactive 3D Graphics and Virtual Environments

by

James Thomas Klosowski

Doctor of Philosophy

in

Applied Mathematics and Statistics

State University of New York
at Stony Brook

1998

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 1000 collision queries per second.

We initially analyze several methods of performing collision detection which utilize standard box-based data structures. These methods are compared against a more sophisticated technique, based upon recent computational geometry results on ray shooting using meshes of low stabbing number.

Building upon these results, 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 (BV) 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, including a novel technique in which the BV-tree "oscillates" as it is constructed. 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. By maintaining a "front" as the objects move within the environment, we are able to exploit temporal coherence during the collision detection checks and accelerate the tree traversal algorithm considerably.

Our algorithms have been implemented, tested, and are being publicly released within our `QuickCD` library of collision detection routines. We provide experimental evidence showing that our approach yields substantially faster collision detection than previous methods.

To further test the practicality and effectiveness of our method, we also demonstrate its efficiency in performing NC verification, i.e., in testing tool paths for gouging in 3-axis machining.

To Marianne.

Contents

List of Figures

List of Tables

Acknowledgments

There has been an enormous number of people who have helped me during my graduate studies. I would like to take the time to thank them all.

I would like to express the utmost thanks to Joseph S.B. Mitchell, my mentor and my friend. Without his continued support and encouragement, I would not be here today. I have thoroughly enjoyed the many hours which I have spent in his office discussing our work together. Martin Held has been a great friend and colleague, who has served as a second advisor to me. I thank Martin for his integral role in my research. Joe, Martin and I have made a great team together.

I would also like to thank my thesis committee members, who are also my good friends: Esther Arkin, Steven Skiena, and Amitabh Varshney. I appreciate all of the time you have spent with me and all that you have taught me. I would also like to acknowledge the help of Alan Tucker, whose door was always open for me.

The support staff in our department are all wonderful people, whose hard work is often overlooked and under-appreciated. I thank Loretta Bellantoni, Scott Connelly, Claire Dugan, Terry Mills, and Victor Poon.

More than anyone else, I would like to thank my wife Marianne. She has been my constant companion during my time at Stony Brook and has been a continuous source of love, support, and encouragement. I could not have finished this without her.

My parents, Tom and Mary Lu, have always provided me with their love and support. I thank them for the wonderful foundation upon which my life has been built. Everything which I accomplish in my life can be directly attributed to their years of hard work and love.

My sisters, Judy and Joy, and my brother John, I thank for making all of my trips home worthwhile. I also thank them for not holding things against me...like forgetting about their birthdays and anniversaries. I would also like to thank my new family, the Mulrys, for welcoming me so warmly into their home and for all of the Sunday dinners.

Many friends have made my time at Stony Brook very enjoyable. I thank Jane and LeRoy Wenstrom, Rich Single, and Peg Boyle for all of the good times we had in P-134a and at 17 Twisting Drive. Yi-Jen Chiang, Mary Jane Graham, Lorraine Greenwald, Changkil Lee, Cláudio Silva, and Jim Wagner have also been good friends.

Chapter 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. Therefore, a collision detection system must be flexible enough to handle all of these situations. 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, as well as a number of other fields. Each day, researchers and programmers are finding new applications in which solving the collision detection problem is essential. Within all of these applications, the most important factor (typically) when solving the collision detection problem is *speed*.

The requirement for speed in interactive use of virtual environments is particularly challenging; e.g., haptic force-feedback can require on the order of 1000 intersection queries per second [65, 64]. 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. Consider, for example, an 3D environment consisting of 100,000 triangles together with a single flying object consisting of 20,000 triangles. Using this naive approach to CD, for each location of the flying object, we would check each of the triangles in the flying object against all of the triangles in the environment. In total, therefore, we would need to perform two billion of these triangle-triangle checks for each and every position of the flying object to determine if it comes into contact with the (static) environment. Using our current implementation of the triangle-triangle check (as discussed in Sections 4.5 and 5.4), testing each location of the flying object would take almost two hours to perform; a far cry from the 1000 intersection checks per second required by applications utilizing haptic force-feedback. Thus, many approaches have recently been proposed to address the issue of efficiency; we discuss these below.

1.1 Previous Work

Algorithms in Practice 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. As indicated in the previous section, checking all pairs of primitives is an infeasible approach (for realistic sized input).

Octrees [70, 73], k-d trees [15], BSP-trees [72], brep-indices [16, 93], tetrahedral meshes [44], and (regular) grids [35] 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 of computer graphics, e.g., ray tracing [5, 36, 56, 97].) 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 [49, 50, 77] or axis-aligned bounding boxes (AABBs) [14, 91, 44], 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; 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 [9] 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.* [13] 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 [100, 101] 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 which is 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, Lin, and Manocha [38]. 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 [40] previously published a similar algorithm; however, we are not aware of any empirical comparisons between the two algorithms.

The RAPID system has addressed the same CD problem that we will address: moving a single object amongst a large number of unstructured, complex models. For this reason, we have run our experiments back-to-back with RAPID; see Chapter 5.

The purpose of using spatial subdivisions or bounding volume hierarchies was to reduce the number of polygons (or primitives) that need to be tested for contact. Vaněček has tried another approach to this reduction problem by using a modified backface culling technique [94]. His method removes all polygons whose normals face away from the (moving) polygon, P, as these polygons must therefore lie on the far side of the object that has been hit by P (if any). This assumes, however, that polygons on the near side of the object will be hit by P first, and therefore will prevent a deep penetration.

Other approaches to collision detection have included using space-time bounds [49] and four-dimensional geometry [17, 18] to bound the positions of objects within the near future. By using a fourth dimension to represent the simulation time, contacts can be pin-pointed exactly; however, these methods are restrictive in that they require the motion to be pre-specified as a closed-form function of time. Hubbard's space-time bounds [49] 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 [24, 60, 61, 62, 68] to keep track of the closest features between pairs of objects. One popular system, I-COLLIDE [24], 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, Manocha, and Lin have generalized this work to include non-convex objects [79].

More recently, the Q-COLLIDE [23] system has greatly improved upon I-COLLIDE by replacing the "closest pairs of features" algorithm, with a "separating vector" algorithm, which quickly finds a separating plane between two convex polytopes if they do not intersect. Empirical evidence [23] has shown that Q-COLLIDE is significantly faster than I-COLLIDE; in some cases it is even an order of magnitude faster.

I-COLLIDE [24] and Q-COLLIDE [23] are designed to work on environments consisting only of convex objects, all of which can move simultaneously. As we are addressing different problems, we have not run any comparisons versus the I-COLLIDE software, which has been made publicly available, or the Q-COLLIDE software, which has not been made available.

Mirtich has designed V-Clip [68] (or Voronoi Clip) as another improvement of the "closest feature algorithm" within I-COLLIDE. According to [68], the limitations of the original algorithm were that it could not handle penetrations between the objects (nor non-convex objects), it was not particularly robust, and it was fairly complicated to implement. Although V-Clip was developed to address all of these problems, it too is limited in that it is designed for objects that are bounded by closed surfaces, and not highly non-convex.

Another software package has recently been designed in order to perform interactive collision detection within VRML environments. V-COLLIDE [51] combines the "sweep and prune" algorithm from I-COLLIDE with the pairwise collision detection check of RAPID. Unfortunately, V-COLLIDE has not yet been integrated into a VRML 2.0 browser (due to the authors' lack of the browser's source code), so it remains open as to how effective it will be in practice. The system has been tested in stand-alone mode and the preliminary results show promise, although the prototype does use a considerable amount of memory.

All of the methods which we have mentioned up to this point has been polygon-based. There has also been a nice progress recently using voxel-based collision detection [42, 64]. In this dissertation, however, we shall focus only on exact methods of collision detection for polygonal models.

Theoretical Results 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 $O(\log^2 n)$ time, where n is the total number of vertices of the polytopes, by using the Dobkin-Kirkpatrick hierarchy [30, 31, 32], 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 $O(n \log n)$ [29, 84], while actually *computing* the intersection [33] takes $O(K \log K)$ time, where K is the size of the input plus output. Schömer [84] detects the intersection between two translating "c-iso-oriented" polyhedra (non-convex, having normals among c directions, where c is a fixed constant) in time $O(c^2 n \log^2 n)$.

For two general polyhedra moving along fixed (known) trajectories, Schömer and Thiel [85] have obtained an $O(n^{8/5+\epsilon})$ algorithm for the case of pure translations, and an $O(n^{5/3+\epsilon})$ algorithm for rotation of one polyhedron about a fixed axis. (Here, $\epsilon > 0$ is an arbitrarily small constant, affecting the constant in the big-Oh.) They have recently generalized their method to yield the first *provably (worst-case) sub-quadratic* time algorithm for a general collection of polyhedra in motion along fixed trajectories [86]; however, the result is likely of purely theoretical interest, having been based on several sophisticated (unimplemented) techniques.

Recently, Suri, Hubbard, and Hughes [90] 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, 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. Here, the aspect ratio of an object is defined 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 (of one object) and the smallest enclosing sphere (of another object).

1.2 Overview

We present a preliminary investigation into the collision detection problem in Chapter 2. We have implemented several methods which use standard box-based data structures and compare these, through experimentation, to a "mesh-based" method, motivated by the theoretical results recently obtained using meshes of low stabbing number in computational geometry.

The results of our experiments in Chapter 2 indicate that one of our box-based data structures, the R-tree, has considerable potential for efficiently performing collision detection queries. Chapter 3 discusses our generalization of this structure in great detail. In particular, we provide an introduction to BV-trees, discrete orientation polytopes, and design choices in constructing effective BV-trees.

Chapter 4 presents our collision detection algorithm and several key issues related to the efficient use of the BV-trees. Our method has been implemented and tested on a wide variety of data. We discuss implementation details and provide our collection of experimental results in Chapter 5.

Recent improvements and extensions to our method are reported in Chapter 6. We discuss several options for building better BV-trees, including a new idea which lets the tree "oscillate" up and down as it is constructed, to build the tightest possible hierarchy. We also introduce new ideas which make more efficient use of the BV-trees which are provided. The most promising of these includes the use of a "front", which takes advantage of temporal coherence to accelerate the tree traversal algorithm.

Our library of collision detection algorithms and data structures, in its current state, is presented in Chapter 7, as well as comparisons versus several other libraries.

Chapter 8 highlights another application to which we have successfully applied our collision detection algorithms and data structures. NC (numerically controlled) verification is a natural application of our techniques, as one problem within NC milling is to verify that a tool does not *gouge* (i.e., collide) the material which is to be left behind after milling the desired shape. Extensions and conclusions are finally presented in Chapter 9.

We note here that throughout this dissertation, it is inadvisable to compare collision detection timings between the various chapters. In essentially all cases, there have been changes made to the code, the compiler, or the hardware used. Thus, the most important thing to keep in mind are the relative timings between the various software packages. In any one specific location where experimental results are given, we have made every attempt possible to make the comparisons accurate and fair. To see the most up to date experimental results for our collision detection system and others, refer to Chapter 7.

Chapter 2

A Preliminary Investigation

This chapter is largely based upon a paper written in 1995 with Martin Held and Joseph S.B. Mitchell. In [44], we presented a new method for performing collision detection which is based on tracking the motion of a flying object within a tetrahedral mesh. To determine the practicality of this method, we implemented several additional methods for performing collision detection which were based upon standard data structures. Experiments were then conducted to determine which method was the most efficient in performing collision detection queries.

As the mesh-based algorithm was fully implemented by Martin Held, we refer to [44] for the complete coverage of this method including the generation of the meshes, the implementation details, and the relevant properties of the meshes. We have included here a brief overview of the generation of the meshes, however, for completeness. As it is the *use* of the mesh for collision detection which we are particularly interested in, we have included the description of this algorithm here. We also include the details of the other three methods which utilize standard box-based data structures. One of these data structures (the R-tree) has become the foundation upon which the rest of this dissertation is built. The relevant experimental results are reported in Section 2.4; however, for the the full set of experimental results, we again refer the reader to [44].

2.1 Introduction

Recall that the collision detection problem takes as input a geometric environment and a moving (or flying) object and asks that we find all instants in time at which the flying object comes into contact with the environment. We assume that the environment, $\mathcal{E}$, is given to us as the boundary of a set of polyhedral obstacles. This representation typically consists of the facets of the obstacles, which we shall assume are always triangulated. Thus, we often refer to the triangles which make up the environment as "obstacle triangles". The flying object, which we shall refer to as F, is also described by the triangles which comprise its surface.

Within this chapter, we shall occasionally distinguish between several types of collision detection:

- *static* collision detection, where one is to check whether a flying object at one particular position and orientation intersects the environment.
- *pseudo-dynamic* collision detection, where one is to check whether the flying object intersects the environment at any of a set of discrete position/orientation pairs corresponding to the object's motion.

- *dynamic* collision detection, where one is to check whether the volume swept out by the flying object intersects with the environment.

Our study has focused mainly on pseudo-dynamic collision detection, which is consistent with the vast majority of related work on the subject. Exact methods of treating dynamic collision detection have been addressed by [17, 18], by considering the four-dimensional time-space problem or by modeling the configuration space exactly.

2.2 A Mesh-Based Algorithm

The mesh-based method for collision detection is based upon recent computational geometry results on "ray shooting" using meshes (triangulations) of low "stabbing number". In computational geometry, several data structures have been developed to support *ray shooting queries*: Determine the first object hit by a ray that emanates from a query point in a query direction. It is possible to obtain $O(\log n)$ query time, with roughly $O(n^4)$ space, and sublinear query time with $O(n)$ space. An excellent reference on the subject is [27]; see also [2, 3, 22, 28, 37, 71, 78, 87]. From the *practical* point of view, however, the apparently most promising methods are based on the recent "pedestrian" approach to ray shooting: Build a subdivision (mesh) of low "stabbing number", so that query processing becomes simply a walk through the subdivision [21, 47, 69]. The complexity of a query is simply the number of cells (triangles) of the mesh that are met by the query ray before it hits an obstacle. The (line) *stabbing number* of the mesh is the maximum number of cells met by any query ray before it encounters an obstacle. In [1], it is shown that, in the worst case, the best possible triangulation of a set of n points in space has a line stabbing number of $\Theta(n)$ (without Steiner points) or $\Theta(\sqrt{n})$ (with Steiner points).

In [44], a collision detection method was introduced whose preprocessing step was to construct a decomposition of free space (the complement of the obstacles in $\mathcal{E}$) into a tetrahedral mesh, marking the (triangular) facets that correspond to obstacle boundaries. In the rest of this section, we provide a brief overview of the mesh generation method followed by the collision detection algorithm which utilizes the mesh. This algorithm has as its foundation the recent "pedestrian" approach to searching a subdivision.

2.2.1 Mesh Generation

Our current attempt to generate conforming meshes of low stabbing number is based on using a Delaunay triangulation of the obstacle vertices, with Steiner points added in order to make the triangulation (or mesh) conform to $\mathcal{E}$. A mesh, $\mathcal{M}$, *conforms* with $\mathcal{E}$ if no tetrahedron of $\mathcal{M}$ lies partially inside and partially outside of some environment obstacle. In other words, the boundary of the obstacle environment corresponds to the union of some faces of the tetrahedra.

We start with the original vertices of $\mathcal{E}$ and compute their Delaunay triangulation. Afterwards, we check whether any triangular facet of $\mathcal{E}$ is intersected by a facet of some Delaunay tetrahedron. (Of course, we do not compute all pairwise intersections between all facets of $\mathcal{E}$ and all tetrahedra, but rather scan the mesh, thus only checking the "neighborhood" of every obstacle triangle.) If no intersection exists, then the mesh conforms.

Otherwise, for every obstacle triangle, up to one Steiner point is chosen and the Delaunay triangulation is incrementally updated. This scheme is applied repeatedly until the mesh conforms. Suitable Steiner points are obtained by taking the points of intersections of the edges of the obstacle triangles with the facets of the tetrahedra, and vice versa.

2.2.2 Collision Detection within a Mesh

Suppose we are given a tetrahedral mesh $\mathcal{M}$, which conforms to the obstacle environment, $\mathcal{E}$, and a flying object, F, whose facets are assumed to be triangles. We want to determine for a query instance of F, at a given position and orientation, whether F intersects the environment.

Obviously, the flying object does not intersect the environment if none of its facet triangles intersects the environment, unless an obstacle of the environment lies completely inside the flying object, or vice versa. (We will address this problem later in this section.) This simple observation can be refined into a two-phase approach to static collision detection within a mesh: in phase I we compute $BB(F)$, the (axis-aligned) bounding box of F, and enumerate all tetrahedra which lie partially inside the bounding box. For the sake of simplicity, the current implementation enumerates the (possibly larger) set of all tetrahedra whose bounding boxes overlap with $BB(F)$. Provided that one tetrahedron which contains one of the corners of $BB(F)$ is known, these tetrahedra can be determined by a straightforward, and computationally inexpensive, breadth-first search of $\mathcal{M}$.

If none of the faces of these tetrahedra is part of an obstacle boundary then no collision can occur. Otherwise, in phase II a full-resolution static collision check is performed, checking each facet triangle of F for collision with any of the (hopefully few) obstacle faces determined in phase I. In theory, this all-pairs collision check between facets of F and selected obstacle faces could be avoided by again making use of the mesh. However, up to now all attempts to take additional advantage of the mesh structure turned out to be not competitive with this limited all-pairs check in practice.

This static collision check can easily be extended to a pseudo-dynamic collision check. In a preprocessing step, we locate one vertex of $BB(F)$ in the mesh; i.e., we determine that tetrahedron of $\mathcal{M}$ that contains it. Locating a point in $\mathcal{M}$ is done by shooting a ray from the center of gravity of some tetrahedron of $\mathcal{M}$ to this point. As soon as this vertex has been located, the first (static) check for collision can be carried out. After each motion step, the vertices of F and of $BB(F)$ are updated. Furthermore, the new location of a corner of $BB(F)$ within the mesh is obtained by shooting a ray from its previous location, and the algorithm outlined above is applied.

The algorithm for shooting a ray is fairly simple. Basically, one marches from one tetrahedron to one of its neighbors if the face shared by both tetrahedra is intersected by the ray. The algorithm stops as soon as the goal point is contained in the actual tetrahedron (and thus no new intersection of the ray with a face of this tetrahedron exists).

Note that a test whether or not F lies completely in the interior of some obstacle (or vice versa) can easily be carried out, too[1]. Starting at the boundary of the environment, the mesh $\mathcal{M}$ is scanned and all tetrahedra which lie inside some obstacle are marked. Now, a test whether F lies completely in the

[1]This subalgorithm has not yet been implemented, however.

interior of some obstacle (or whether an obstacle lies completely in the interior of F) boils down to checking whether any of the tetrahedra (partially) occupied by F is marked. These tetrahedra are enumerated during the collision check, anyway, and can thus be checked efficiently.

2.3 Some Simple "Box" Methods

The mesh-based method was implemented and tested, but in order to determine its practicality, it is necessary to compare it against alternatives. In this section, we describe three alternative collision detection algorithms, which we have implemented and tested, and which were used for comparison purposes in our experimental results (described in the next section). These methods are all based upon standard data structures and are used in an attempt to "localize" the flying object in the environment so that we can greatly reduce the number of obstacle triangles that need be considered for contact with the flying object F.

2.3.1 A Grid of Boxes

Perhaps the simplest method that one can imagine for doing "spatial indexing" is that of imposing a grid of equal-sized boxes over the workspace (which is assumed to be the unit cube). In our implementation of this method, we use an $N \times N \times N$ grid of boxes (cubes). We ran experiments to optimize over the choice of N, which is typically kept relatively small, on the order of 5–50. For each box ("voxel") in this grid, we store a list of the obstacle triangles that intersect the voxel. A single obstacle triangle may be stored in the lists of many voxels; thus, the size of the resulting data structure could be large in comparison with the size of the input. (This issue was addressed in our experimentation.) This preprocessing can trivially be done within worst-case time $O(n \cdot N^3)$; however, it is possible to do it faster, in time proportional to the sum of the list sizes, over all boxes (e.g., by "scan converting" each obstacle facet within the grid, in time proportional to the number of boxes intersected by the facet).

Processing a collision query is done by means of a two-phase approach: In phase I, we compute the (axis-aligned) bounding box, $BB(F)$, of the flying object, and identify the set of voxels that intersect $BB(F)$, simply by checking the x-, y-, and z-ranges of $BB(F)$. Then, we consider each obstacle triangle, T, associated with each of these voxels. If $BB(T) \cap BB(F) = \emptyset$, then we know that T does not intersect F. If $BB(T) \cap BB(F) \neq \emptyset$, then T is a witness of a potential collision, and we perform a full-resolution check in phase II: For each triangle T_F of the flying object, we identify the voxels intersected by $BB(T_F)$ and check T_F for intersection with the associated obstacle triangles, stopping if we find an intersection. During phase II we make use of the information gathered in phase I; e.g., we do not check a triangle for intersection with facets of F if its bounding box does not intersect $BB(F)$.

An alternative method is, for a given query F, to compute the set of voxels intersected by F (e.g., by 3D scan-conversion), and to test each obstacle triangle in the associated lists for intersection with F. This alternative, however, has not been implemented.

2.3.2 A k-d Tree Method

Another simple approach to representing spatial data is to store it in a 3-d tree, which is a binary space partition (BSP) tree whose cuts are chosen orthogonal to the coordinate axes.

We begin with the root node, which is associated with the workspace (box). Each node of the tree is associated with a hyperrectangle (box), and (implicitly) with the set of obstacle triangles that intersect that box. Each non-leaf node is also associated with a "cut" plane. To define the children of a node, we must make two decisions: (1) Will the cut be orthogonal to the x-, y-, or z-axis? (2) At what value of the chosen coordinate axis will the partition take place? If the number of obstacle triangles intersecting a box falls below a threshold, τ, then the box is not partitioned, and the corresponding node is a leaf in the tree. We explicitly store with each leaf the list of obstacle triangles that intersect its box.

To process a query q (which is a region being tested for intersection with the environment), we start by comparing q with the cut plane of the root. If q lies entirely on one side of the cut plane, then we visit recursively that corresponding child; otherwise, we visit both children. When we visit a leaf node, we check each of the associated obstacle triangles for intersection with q.

In our implementation, we process an intersection query for F by doing a two-phase search of the tree. In the first phase, we search the tree using $q = BB(F)$, checking at the leaves for an intersection of the bounding box of F with the bounding box of an obstacle triangle. If we find such an intersection, then we immediately enter the second, "full resolution," phase, in which we search the tree using $q = T_F$, for each triangular facet T_F of the flying object

F, checking at the leaves for intersection between T_F and each obstacle triangle stored at a leaf.

In our implementation, we make the choice (2) (of where to split a box, given an axis along which to split) by always splitting at the *midpoint* of the corresponding box length. One might consider the option of splitting at a "median" value of the coordinate, but there is an issue of "median" with respect to what discrete values? The problem is that all of the obstacle vertices for the associated triangles may fall outside the range of the box.

We make choice (1) in four different ways and choose the one which works the best on average: (a) minimize $\max\{n_1, n_2\}$; (b) minimize $|n_1 - n_2|$; (c) minimize $n_1 \cdot n_2$; and (d) divide the longest side of the box. Here, n_1 and n_2 are the "sizes" (number of associated obstacle triangles) of the two new problems created at the children. All of the results reported in Section 2.4 used choice (c), which was selected after running several comparisons.

We also set a "no-gain-threshold", to handle the instances (e.g., near a high-degree vertex) in which splitting a node does not decrease the number of obstacle triangles in one or both of the two children. We do allow splitting to occur in such cases, but only a bounded number of times (at most the "no-gain-threshold"). We ran experiments to optimize over the choice of this threshold, and we ended up using five in the runs reported here.

2.3.3 R-trees of Boxes

An alternative tree-based representation of obstacles is based on the simple idea of "R-trees" (rectangle trees) and their variants [41, 82, 88, 14]. The basic idea is to partition the set of obstacles associated with a node, rather than partitioning the bounding box (the "space") associated with a node, as is done in a BSP tree. The child nodes will then correspond to their respective subsets of the obstacles and the associated space occupied by the bounding boxes (which may overlap). But, at any one level of the tree, each obstacle triangle is associated with only a single node. This type of data structure has been extensively studied within the database community and numerous comparisons have been performed to determine which performs the best (e.g., fastest queries, fewest disk accesses). For more information, please refer to [39, 8, 91].

We describe a binary tree version; multi-ary trees are also possible. Each node of the tree corresponds to a rectangular axis-aligned box and a subset of the obstacle triangles. The root is associated with the entire workspace and all of the obstacles. Consider a node having associated set $\mathcal{T}$ of obstacle triangles. If $|\mathcal{T}| \leq \tau$, where τ is some threshold (whose value will be a parameter in our experiments), the node is a leaf, and we store the triangles $\mathcal{T}$ in a list associated with this leaf. Otherwise, we split $\mathcal{T}$ in (roughly) half, by using the median x-, y-, or z-coordinate of the centroids of $\mathcal{T}$, and by assigning triangles to the two children nodes according to how the centroids fall with respect to the median value. We select among the three choices of splits based on either minimizing (a) $\max\{V_1, V_2\}$, or (b) $V_1 + V_2$, where V_1 and V_2 are the volumes of the bounding boxes of the two subsets that result from the choice of split. For the results reported below, we choose to minimize $\max\{V_1, V_2\}$, which was selected after running several comparisons.

2.4 Experimental Results

2.4.1 Environments and Flying Objects

Ideally, for the purpose of experimentation, one would have an algorithm to generate "random" instances of realistic obstacle environments. But, the problem of generating "random" collections of obstacles is a challenging one; even the problem of generating a single "random" simple polygon on a given set of vertices is open. Our approach for this set of experiments was to use a "random" BSP tree to partition the workspace (the unit cube) into a set of disjoint boxes (the leaves), into which we place scaled copies of various obstacles.

We generate a random BSP tree as follows: At each of $N-1$ stages (where N is the desired number of leaf boxes), we select at random a leaf (box) to split, from among a set of "active" leaves (initially, just the single node consisting of the workspace). Among those axes of the corresponding box that are longer than 2ϵ (for an $\epsilon < (1/2)N^{-1/3}$), we select one at random, and we split the box with a plane perpendicular to the axis, at a point uniformly distributed between $\xi_{min} + \epsilon$ and $\xi_{max} - \epsilon$, where ξ_{min} and ξ_{max} are the min/max box coordinates along the chosen axis. The purpose of the ϵ here is to prevent "pancake-like" obstacles from being generated. The leaf is removed from the active list, and two new (children) leaves are created.

We recorded flight statistics for three different groups of obstacle environments called "scenes", "tetras" (for "tetrahedra"), and "terrains". The first two groups were generated by means of the random BSP-tree partition (described above), where a random subset of the leaves were either filled with simple objects (for the environments in the "scenes" group) or with tetrahedra (for the "tetras" group). We used various test objects[2] such as chess pieces, mechanical parts, aircraft, and models of animals. Table 1 summarizes the complexities of these environments, where $\#(V)$ denotes the number of vertices and $\#(T)$ denotes the number of triangles.

The third group of environments was generated from data on real-world terrains. We sampled so-called "1-degree DEM" data[3], thereby converting 1201×1201 elevation arrays into 120×120 elevation arrays. These elevation arrays were afterwards converted into triangulated surfaces by means of a straightforward triangulation of the data points. Table 1 lists the terrains used for our tests.

These three groups of environments were tested with six flying objects of various complexities, as listed in Table 2. All environments were scaled to fit into the unit cube, and all flying objects were scaled to fit into a cube with side length 0.05.

2.4.2 Flight Paths

For a given choice of a flying object F, and for a given obstacle environment $\mathcal{E}$, we wanted to gather statistics on the efficiency of collision detection for rigid motions of F within the free space. One option for doing this is to pick random (feasible) initial placements for F, and then to perform random (rigid) flight paths, stopping when an obstacle is first encountered. In reasonably

[2]Most test objects have been obtained by means of anonymous ftp from the ftp-site "avalon.chinalake.navy.mil" and converted from various data formats to our input format.

[3]We obtained the DEM data by means of anonymous ftp from the ftp-site "edcftp.cr.usgs.gov".

			Grid	R-Tree	3d-Tree	
env. name	$\#(V)$	$\#(T)$	$\#(T_{st})$	*Height*	*Height*	$\#(T_{st})$
buffalo-w	14400	28322	50897	15	33	100383
denver-e	14400	28322	51849	15	33	148528
denver-w	14400	28322	62563	15	33	150360
eagle-e	14400	28322	51817	15	30	145789
g_canyon-e	14400	28322	61940	15	36	150425
jackson-w	14400	28322	51711	15	35	127642
moab-w	14400	28322	58045	15	33	151884
seattle-e	14400	28322	68271	15	32	139390
25t	100	100	12298	7	11	270
50t	200	200	20584	8	12	519
100t	400	400	37237	9	17	1761
250t	1000	1000	9544	10	19	1804
500t	2000	2000	10662	11	24	4220
1000t	4000	4000	21676	12	27	9346
2000t	8000	8000	30191	13	27	20163
4000t	15996	16000	49881	14	30	40374
s9.20rp	5680	11200	26178	14	35	84253
s20.40rp	11360	22400	76884	15	35	181021
s22.150room	10492	13800	17740	14	42	45107
s23.300room	20995	27600	34395	15	40	79816
s16.6easy	11604	22092	71039	15	34	135833
s15.4cows	11612	23216	33422	15	44	146502
s10.2each	748	1376	5507	11	31	8933

Table 1: Characteristics of the test environments: $\#(V)$ and $\#(T)$ are the number of vertices and triangles in each environment, respectively. $\#(T_{st})$ is the number of triangles stored by the Grid and 3d-tree methods. The heights of the R-tree and 3d-tree are also provided.

Flying Object	$\#(V)$	$\#(T)$	$\#(V_{CH})$
tetra	4	4	4
socbal	60	116	60
bishop	250	496	41
balloon	1141	2094	395
spitfire	2479	5152	51
747	7594	14643	499

Table 2: Complexities of the flying objects: $\#(V)$, $\#(T)$, and $\#(V_{CH})$ are the number of vertices, triangles, and convex hull vertices, respectively.

cluttered workspaces, though, this method resulted in very short flight paths before a collision; thus, the experiment had to be repeated over and over in order to obtain extensive data, and each selection of a random start point was relatively costly, since we generated a random placement and then checked for feasibility — an unlikely event in cluttered environments. (Alternatively, one could construct the full configuration space for F relative to $\mathcal{E}$, and then generate a point at random within this space; but we believed this approach to be too costly to implement.)

We devised a simple solution to the path generation problem: We use "billiard paths", allowing F to "bounce off" an obstacle that it hits. We do not attempt to simulate any real "bounce"; rather, we simply reverse the trajectory when a collision occurs. The motion of F is generated using a simple scheme of randomly perturbing the previous motion parameters (displacement vector and angles of rotation) to obtain the new motion parameters.

As our algorithms heavily rely on the availability of the bounding box of F, we also implemented a fast method of updating $BB(F)$ during the flight. In particular, for each step the new bounding box is obtained efficiently by a simple hill-climbing algorithm applied to the (precomputed) convex hull of F.

2.4.3 Collision Detection Results

In the sequel we describe the results of our collision detection experiments. Note that the implementation (in the C programming language) was not fine-tuned in order to achieve optimal speed. However, the tests were fair because all four of the methods implemented rely on the same basic primitives. Thus, a potential speed-up gained by fine-tuning can be expected to benefit all methods uniformly. All the tests reported were run on a Silicon Graphics Indigo^2, with a single 150 MHz R4400 processor and 128M of main memory.

The time consumed by our algorithms includes all computations with the exception of rendering. The random flight paths are generated at run-time and are therefore included in the timings. For each new location of the flying object, we transform all of the vertices of the object so that it is located in its new position in the workspace. The rationale for this is that if we were to visualize the flight of the objects within the environment, we would have to have the current location of the object for rendering. Also, we are currently only interested in *detecting* that a collision occurs between the flying object and the environment at each location, therefore, once we have determined that there is a collision, we terminate the collision detection algorithm. We do not report all pairs of triangles which are in contact, only the first one (if it exists).

Tables 5–7 (found at the end of this Chapter) present the flight statistics gathered for the four methods tested. For each method and each environment/flying object pair, we list the frame-rate, f/s, which is the number of collision detection queries we can perform in one second. The frame-rates listed have been determined as follows: for each environment/flying object pair, all four methods recorded statistics for a flight of the object along the same "billiard path", for 10,000 steps with an average displacement of the flying object by about 0.005 between subsequent steps. Among other values, the elapsed cpu-time and the number of full-resolution collision checks were recorded. Based on the elapsed cpu-time and the number of steps, the average number of frames or steps per second was computed. The average frame-rate highly depends on the number of full-resolution collision checks (since this is where the large majority of the computations are performed), which is identical for all our four methods (for one environment/object pair). Thus, in order to be able to compare frame-rates among different flying objects and different environments, we normalized the frame-rates f/s with respect to the number of actual full-resolution collision checks. In the Tables 5–7, the column f_N/s shows the hypothetical average frame-rate for a flight where 5% of all steps give rise to a full-resolution collision check. This translates to 500 full-resolution checks for 10,000 steps; our flights averaged 532 full-resolution collision checks and 35 actual collision.

Based on these normalized frame-rates, we counted the number of times each method was the fastest among the four methods; see Table 3. In case of close ties, several methods were counted as winners. Summarizing, the R-tree method was the clear winner, with the mesh-based method taking second place, and with the 3d-tree method as the clear loser.

As the previous ranking only counts the number of wins, it may not correctly highlight that method which performs best "on the average". Thus, we used a second method for ranking, as follows. For each environment/object

env. name	R-Tree	Grid	3d-Tree	Mesh
terrains	45	2	4	1
tetras	3	8	7	34
scenes	25	11	7	9
total	73	21	18	44

Table 3: Number of wins for each method.

pair, we assigned a score to the four methods according to their sorted frame-rates: the fastest (first) method scored 1, whereas the slowest (fourth) method scored 4, and the other two methods scored 2 and 3. Again, the scoring was adapted for close ties. After summing over all environment/object pairs, the best method, in some hypothetical average case, is the method with smallest total score; see Table 4. The R-tree again took first place in this comparison, closely followed by the grid-based method, with the 3d-tree taking third place, and the mesh-based algorithm in fourth place.

env. name	R-Tree	Grid	3d-Tree	Mesh
terrains	53	122	116	176
tetras	166	96	128	81
scenes	75	90	108	130
total	294	308	352	387

Table 4: Weighted ranking for each method.

Mesh-Based Method: It is evident from Tables 5–7 that the computational overhead carried by the mesh-based method is too high to pay off for relatively simple flying objects. In all our tests the mesh-based method always was by far the slowest when flying a tetrahedron ("tetra"). However, it became more competitive with more complex flying objects; it had quite a few wins when flying aircraft models (spitfire and 747).

Similarly, we investigated why the mesh-based method performed rather poorly for some environments of the "scenes" group (s22.150room, s23.300room, and s15.4cows). For all these environments, our flights achieved a surprisingly low number of full-resolution collision checks, and the mesh-based method therefore had little to gain by exploiting the structure of the mesh.

Box-Based Methods: Tables 5–7 clearly show that the R-tree method was the fastest for all three of our test environments. We were somewhat surprised by this, although the positive results were encouraging and have since lead us to a much more extensive investigation into data structures of this type.

In optimizing the various parameters for these methods, we chose $N = 40$ for the grid method, and a threshold $\tau = 10$ for the 3-d tree. The R-tree was given a threshold $\tau = 1$; therefore, the number of triangles stored equals the number of original triangles. The numbers of triangles stored for the other two methods are given in the columns "$\#(T_{st})$" in Table 1. The columns labelled as "*Height*" refer to the heights of the various trees.

Full-Resolution Checks per Second The four algorithms which we have used for collision detection were all two-phase methods. The first phase was rather crude, using the bounding box of the flying object F to determine if the more refined second phase need be entered. The second phase was referred to as a "full-resolution" phase, in that the actual triangles (or facets) of the flying object were used when searching the appropriate data structure.

The frame-rates presented in Tables 5–7 were the average number of collision queries that each of the algorithms were able to perform in one second. These averages were taken over flights of 10,000 steps in length, where around only 5% of the collision queries resulted in the expensive full-resolution checks. If a flight required considerably more full-resolution checks, the average frame-rate could drop dramatically. We have therefore, investigated the amount of time required to perform a full-resolution check, so that in the worst case, if all of the queries reached the second phase of our algorithms, we would then know how many queries we could answer per second.

To determine the cpu-time consumed by one static full-resolution collision check, we ran another series of experiments. For two environments from each of the three groups of environments, and for all flying objects, we timed 20 different full-resolution collision checks. The timings were obtained by moving the flying objects along a billiard path, and by timing collision checks for placements of the flying object which resulted in collisions. In order to get accurate timings, every such collision check (for a particular placement of a flying object) was performed repeatedly and the total cpu-time consumed was measured and afterwards divided by the number of collision checks performed. Averaging over all 20 different placements yielded the average cpu-consumption of one full-resolution collision check.

The resulting numbers of full-resolution checks per second are given in Table 8. Roughly, the ranking of our four algorithms according to the number of full-resolution checks matches the rankings given in Tables 3 and 4.

2.5 Conclusions

We have implemented several simple box-based methods for answering collision detection queries and compared these against a mesh-based method which was founded on the principles of low stabbing number meshes, which have been put forth in the computational geometry literature. This chapter has been an experimental study comparing these relatively straight-forward methods to a more complex (theoretical) method to determine which will perform better in practice.

The results of our experiments were somewhat surprising. By simply counting the number of times each method won, the R-tree method was by far the fastest. In a distant second place was the mesh-based method. However, in looking at the average performance of the methods (by ranking the methods one through four based upon how they finished for each environment/flying object pair), we found that the mesh-based method performed the worst out of the four methods. The R-tree again was the best, followed closely by the grid method.

The R-tree method was not only the fastest method, but also the most memory efficient. Recall that the grid and 3d-tree methods often stored (obstacle) triangles in several of the cells or leaves, thus increasing the amount of memory needed. The R-tree method never stores a triangle more than once, which becomes even more important as the sizes of the datasets increase to the hundreds of thousands, and even millions, of triangles.

For these reasons, we begin a thorough investigation into the R-tree method, its variations, and several new extensions. In the following chapters, the notation and terminology have changed considerably when describing the R-tree method. In fact, we shall no longer refer to it as an R-tree, since this implies that we are using "R"ectangles (axis-aligned bounding boxes) as the bounding volumes at each node of the tree. As we soon shall see in Chapter 3, there are other bounding volumes which are much more efficient.

	Mesh		Grid		R-Tree		3d-Tree	
Env./ Object	f/s	f_N/s	f/s	f_N/s	f/s	f_N/s	f/s	f_N/s
s9.20rp / tetra	1122.3	814.8	4444.4	3226.7	5882.4	4270.6	5181.3	3761.7
s9.20rp / socbal	909.1	567.3	1964.6	1225.9	2590.7	1616.6	1934.2	1207.0
s9.20rp / bishop	742.4	218.3	1224.0	359.9	1186.2	348.8	753.0	221.4
s9.20rp / balloon	209.1	97.9	242.6	114.0	271.1	127.4	190.9	89.7
s9.20rp / spitfire	95.5	45.3	96.1	45.7	96.4	45.9	88.7	42.2
s9.20rp / 747	48.1	15.4	37.5	12.0	38.9	12.4	33.9	10.8
s20.40rp / tetra	874.1	2007.0	2702.7	6232.4	2873.6	6626.4	2881.8	6645.5
s20.40rp / socbal	708.2	1301.7	954.2	1757.6	994.0	1831.0	846.0	1558.4
s20.40rp / bishop	565.3	482.8	648.9	555.5	473.3	405.1	485.4	415.5
s20.40rp / balloon	87.5	149.9	97.5	168.2	66.9	115.5	77.6	133.9
s20.40rp / spitfire	57.5	90.4	49.5	77.9	32.2	50.7	30.4	47.8
s20.40rp / 747	22.7	29.7	19.1	25.2	12.9	17.0	13.7	18.0
s22.150rm / tetra	925.9	625.9	3076.9	2080.0	3367.0	2276.1	4201.7	2840.3
s22.150rm / socbal	1340.5	136.7	3861.0	393.8	5291.0	539.7	4878.0	497.6
s22.150rm / bishop	728.3	125.3	1536.1	264.2	1531.4	263.4	1006.0	173.0
s22.150rm / balloon	124.7	19.5	347.0	54.1	452.1	70.5	414.6	64.7
s22.150rm / spitfire	89.0	9.3	140.8	14.6	231.1	24.0	203.6	21.2
s22.150rm / 747	39.2	4.4	54.2	6.1	64.5	7.2	64.2	7.2
s23.300rm / tetra	786.2	445.0	2732.2	1546.4	3311.3	1874.2	3846.2	2176.9
s23.300rm / socbal	581.4	174.4	2681.0	804.3	3048.8	914.6	2331.0	699.3
s23.300rm / bishop	483.1	78.3	1177.9	190.8	1512.9	245.1	1264.2	204.8
s23.300rm / balloon	79.5	32.8	245.2	101.0	258.7	106.6	237.6	97.9
s23.300rm / spitfire	43.5	11.0	99.3	25.0	147.7	37.2	117.4	29.6
s23.300rm / 747	33.4	5.2	56.5	8.8	61.8	9.6	49.4	7.7
s16.6easy / tetra	606.1	1343.0	2941.2	6764.7	2451.0	5637.3	1941.7	4466.0
s16.6easy / socbal	598.8	952.1	1084.6	1828.6	762.8	1286.0	949.7	1601.1
s16.6easy / bishop	377.5	469.6	385.8	557.1	199.4	288.0	265.5	383.3
s16.6easy / balloon	113.4	202.8	84.8	167.7	40.5	80.1	55.5	109.8
s16.6easy / spitfire	74.6	83.2	58.1	68.4	28.8	33.9	38.6	45.4
s16.6easy / 747	25.8	25.3	22.0	23.0	12.3	12.8	16.5	17.2
s15.4cows / tetra	880.3	197.2	4587.2	1027.5	10526.3	2357.9	8695.7	1947.8
s15.4cows / socbal	1221.0	7.3	4132.2	33.1	7874.0	63.0	7751.9	62.0
s15.4cows / bishop	739.1	47.3	1512.9	96.8	2309.5	147.8	1193.3	76.4
s15.4cows / balloon	298.0	20.3	504.5	34.3	561.5	38.2	279.2	19.0
s15.4cows / spitfire	193.7	13.6	247.0	17.8	249.8	18.0	245.5	17.7
s15.4cows / 747	66.6	7.3	64.2	7.6	71.4	8.4	60.7	7.2
s10.2each / tetra	1960.8	278.4	6849.3	986.3	16666.7	2400.0	14705.9	2117.6
s10.2each / socbal	1838.2	55.1	3846.2	115.4	7633.6	229.0	6944.4	208.3
s10.2each / bishop	1215.1	14.6	2178.6	26.1	2666.7	32.0	2666.7	32.0
s10.2each / balloon	466.4	33.6	538.5	38.8	518.4	37.3	504.0	36.3
s10.2each / spitfire	209.7	24.3	227.7	26.4	211.1	24.5	198.3	23.0
s10.2each / 747	79.0	9.9	69.2	8.7	66.5	8.4	60.8	7.7

Table 5: Flight statistics (scenes).

	Mesh		Grid		R-Tree		3d-Tree	
Env./ Object	f/s	f_N/s	f/s	f_N/s	f/s	f_N/s	f/s	f_N/s
buffalo-w / tetra	500.8	440.7	2544.5	2249.4	3891.1	3439.7	275.4	243.5
buffalo-w /socbal	327.3	609.5	650.6	1221.9	1050.4	1972.7	54.6	102.6
buffalo-w /bishop	256.8	211.6	271.3	225.2	770.4	639.4	93.7	77.8
buffalo-w / balloon	94.1	85.8	181.6	166.7	216.0	198.3	93.7	86.0
buffalo-w / spitfire	55.0	46.7	66.4	57.1	72.8	62.6	55.2	47.5
buffalo-w / 747	43.7	16.1	39.4	14.8	53.8	20.2	32.6	12.2
denver-e / tetra	894.5	982.1	2907.0	3267.4	4902.0	5509.8	4184.1	4702.9
denver-e /socbal	1111.1	280.0	3164.6	797.5	6024.1	1518.1	4132.2	1041.3
denver-e /bishop	423.9	250.1	578.7	341.4	1094.1	645.5	736.4	434.5
denver-e / balloon	181.1	103.6	234.0	133.9	319.4	182.7	193.5	110.7
denver-e / spitfire	57.0	48.8	59.3	50.8	79.8	68.3	67.2	57.5
denver-e / 747	25.0	20.1	25.3	20.7	30.9	25.2	23.3	19.0
denver-w / tetra	727.3	1552.0	1923.1	4130.8	2227.2	4784.0	2197.8	4720.9
denver-w /socbal	569.2	960.7	905.8	1554.3	1216.5	2087.6	797.4	1368.4
denver-w /bishop	522.2	334.2	632.9	415.2	777.0	509.7	646.0	423.8
denver-w / balloon	82.0	103.3	108.6	138.2	114.3	145.4	94.9	120.7
denver-w / spitfire	45.6	45.0	55.4	55.6	55.0	55.2	48.0	48.1
denver-w / 747	10.9	17.2	12.7	20.2	13.6	21.7	12.7	20.3
eagle-e / tetra	772.2	665.6	2949.9	2542.8	3787.9	3265.2	4739.3	4085.3
eagle-e /socbal	691.1	716.0	1158.7	1207.4	2375.3	2475.1	1264.2	1317.3
eagle-e /bishop	384.5	259.9	485.2	329.0	860.6	583.5	646.0	438.0
eagle-e / balloon	144.3	94.6	186.8	122.5	297.9	195.4	186.4	122.3
eagle-e / spitfire	64.5	62.1	53.3	51.2	47.6	45.8	65.6	63.1
eagle-e / 747	31.0	24.4	29.0	22.8	37.4	29.4	29.6	23.3
g_canyon-e / tetra	607.5	942.9	1612.9	2506.5	2164.5	3363.6	2207.5	3430.5
g_canyon-e /socbal	661.4	877.0	1183.4	1588.2	1369.9	1838.4	1243.8	1669.2
g_canyon-e /bishop	286.6	244.8	536.8	472.4	666.2	586.3	556.8	490.0
g_canyon-e / balloon	57.5	85.8	50.3	78.8	101.3	158.8	88.3	138.5
g_canyon-e / spitfire	39.9	48.6	51.1	62.4	61.8	75.5	48.0	58.6
g_canyon-e / 747	17.7	17.6	21.3	21.8	22.9	23.5	19.9	20.4
jackson-w / tetra	825.1	493.4	3300.3	2000.0	5405.4	3275.7	5025.1	3045.2
jackson-w /socbal	738.0	646.5	1589.8	1405.4	2907.0	2569.8	2028.4	1793.1
jackson-w /bishop	592.1	311.4	722.5	382.9	1153.4	611.3	839.6	445.0
jackson-w / balloon	113.8	74.9	207.4	137.7	270.4	179.6	189.7	125.9
jackson-w / spitfire	65.5	43.3	92.2	61.0	109.7	72.6	91.1	60.3
jackson-w / 747	34.6	19.2	32.8	18.2	47.5	26.4	39.9	22.2
moab-w / tetra	829.2	1313.4	2369.7	3772.5	3246.8	5168.8	3115.3	4959.5
moab-w /socbal	927.6	712.4	1600.0	1248.0	2475.2	1930.7	1901.1	1482.9
moab-w /bishop	394.8	300.0	453.5	347.4	689.7	528.3	572.7	438.7
moab-w / balloon	92.5	82.0	148.6	134.7	171.5	155.4	130.9	118.6
moab-w / spitfire	30.5	38.0	42.9	54.1	54.0	68.1	39.9	50.3
moab-w / 747	22.4	19.8	24.1	21.4	28.8	25.6	23.7	21.1
seattle-e / tetra	967.1	1431.3	2421.3	3646.5	3802.3	5726.2	3257.3	4905.5
seattle-e /socbal	673.4	940.1	1100.1	1584.2	1730.1	2491.3	1013.2	1459.0
seattle-e /bishop	370.0	421.8	436.7	503.1	730.5	841.5	496.5	572.0
seattle-e / balloon	96.4	61.3	181.0	115.8	266.9	170.8	189.5	121.3
seattle-e / spitfire	17.3	33.3	32.2	63.9	30.7	61.0	28.1	55.7
seattle-e / 747	19.7	13.4	21.5	14.7	35.8	24.6	28.0	19.2

Table 6: Flight statistics (terrains).

	Mesh		Grid		R-Tree		3d-Tree	
Env./ Object	f/s	f_N/s	f/s	f_N/s	f/s	f_N/s	f/s	f_N/s
25t / tetra	3021.1	3939.6	5000.0	7310.0	9259.3	13537.0	9434.0	13792.5
25t /socbal	1736.1	1788.2	1626.0	1775.6	1468.4	1603.5	1510.6	1649.5
25t /bishop	651.0	673.2	390.5	432.6	329.5	365.1	358.8	397.6
25t / balloon	175.1	207.3	109.6	136.1	83.3	103.4	81.2	100.8
25t / spitfire	62.0	99.9	33.7	59.2	23.2	40.7	25.7	45.1
25t / 747	33.4	26.4	19.8	16.7	13.0	11.0	14.5	12.3
50t / tetra	2155.2	7534.5	3663.0	12981.7	4651.2	16483.7	5208.3	18458.3
50t /socbal	998.0	2934.1	694.9	2093.1	593.5	1787.5	657.0	1979.0
50t /bishop	420.9	809.8	292.9	575.9	173.7	341.6	215.0	422.6
50t / balloon	111.0	242.3	69.3	159.0	47.1	108.1	48.7	111.8
50t / spitfire	41.2	102.0	24.8	63.9	14.5	37.2	17.7	45.6
50t / 747	27.4	30.3	16.5	19.3	10.9	12.7	13.4	15.7
100t / tetra	1901.1	8619.8	2832.9	14141.6	3389.8	16922.0	4201.7	20974.8
100t /socbal	739.6	3146.4	433.8	2102.4	263.8	1278.3	344.9	1671.6
100t /bishop	302.8	959.7	149.2	541.8	82.1	298.1	97.0	352.3
100t / balloon	138.4	258.8	72.1	153.8	39.9	85.2	50.6	108.0
100t / spitfire	43.3	116.9	20.5	62.2	11.7	35.5	15.1	45.7
100t / 747	14.6	38.4	7.1	21.3	3.7	11.1	5.0	15.1
250t / tetra	2145.9	4240.3	4524.9	9086.0	6711.4	13476.5	6993.0	14042.0
250t /socbal	1305.5	1885.1	1053.7	1591.1	932.8	1408.6	1191.9	1799.8
250t /bishop	1005.0	496.5	909.9	476.8	691.6	362.4	758.2	397.3
250t / balloon	222.3	195.2	149.1	132.4	115.2	102.3	113.9	101.1
250t / spitfire	90.3	102.7	55.3	71.0	35.8	45.9	37.5	48.2
250t / 747	38.9	35.4	22.6	22.6	14.8	14.8	16.1	16.1
500t / tetra	1647.4	2197.7	5291.0	7079.4	7092.2	9489.4	7462.7	9985.1
500t /socbal	1123.6	2476.4	1206.3	2680.3	907.4	2016.3	988.1	2195.7
500t /bishop	742.4	555.3	687.8	553.0	451.5	363.0	517.9	416.4
500t / balloon	253.4	175.8	194.3	138.3	141.8	100.9	103.4	73.6
500t / spitfire	90.6	102.6	62.2	76.5	39.9	49.1	42.1	51.8
500t / 747	39.4	32.6	27.6	23.1	17.0	14.2	20.0	16.7
1000t / tetra	1222.5	4369.2	3436.4	12343.6	3448.3	12386.2	3831.4	13762.5
1000t /socbal	855.4	2249.8	1050.4	2821.4	675.2	1813.6	723.6	1943.6
1000t /bishop	338.1	726.2	330.1	709.8	172.6	371.1	205.7	442.3
1000t / balloon	137.8	256.5	60.2	112.3	47.3	88.3	65.5	122.3
1000t / spitfire	86.6	103.2	61.3	73.4	40.1	48.1	39.8	47.7
1000t / 747	43.9	26.8	31.2	19.2	20.9	12.9	23.0	14.2
2000t / tetra	712.8	3388.5	2164.5	10350.6	1901.1	9091.3	2100.8	10046.2
2000t /socbal	1040.6	1444.3	1457.7	2134.1	1182.0	1730.5	1302.1	1906.2
2000t /bishop	341.3	824.6	305.8	767.0	113.8	285.4	187.8	471.1
2000t / balloon	104.2	254.2	84.1	211.9	38.7	97.5	51.2	129.1
2000t / spitfire	55.5	114.4	39.3	83.9	20.2	43.0	25.4	54.2
2000t / 747	21.7	42.1	14.7	29.5	7.1	14.2	9.2	18.5
4000t / tetra	362.2	901.1	2631.6	6557.9	3906.2	9734.4	3663.0	9128.2
4000t /socbal	460.4	2060.8	599.9	2827.8	295.1	1391.0	384.8	1813.8
4000t /bishop	217.2	734.7	234.5	800.0	117.2	400.0	144.7	493.7
4000t / balloon	94.5	172.1	100.8	183.5	101.0	183.7	81.8	148.9
4000t / spitfire	32.6	111.7	26.8	92.9	13.5	46.9	16.0	55.5
4000t / 747	22.3	37.4	16.2	28.0	6.6	11.5	10.7	18.5

Table 7: Flight statistics (tetras).

env. name	fly. obj.	R-tree	Grid	3d-Tree	Mesh
eagle-e	tetra	477.3	709.2	743.5	92.3
eagle-e	socbal	293.7	183.3	209.9	66.3
eagle-e	bishop	44.5	28.7	45.5	16.8
eagle-e	balloon	114.0	63.2	67.2	37.1
eagle-e	spitfire	20.6	16.4	19.1	6.7
eagle-e	747	16.3	8.6	15.9	4.8
g_canyon-e	tetra	492.6	809.7	790.5	99.1
g_canyon-e	socbal	379.5	240.1	278.9	88.3
g_canyon-e	bishop	48.3	25.5	47.1	25.4
g_canyon-e	balloon	24.9	33.5	36.3	9.9
g_canyon-e	spitfire	17.1	19.9	21.4	7.4
g_canyon-e	747	6.4	7.6	8.3	3.4
1000t	tetra	1123.6	1342.3	1739.1	481.9
1000t	socbal	309.6	306.7	265.6	282.5
1000t	bishop	69.8	72.1	47.4	109.2
1000t	balloon	9.7	17.9	12.1	27.4
1000t	spitfire	7.1	14.2	6.4	24.2
1000t	747	7.5	20.4	15.0	32.1
4000t	tetra	1342.3	2150.5	2197.8	187.6
4000t	socbal	277.4	189.9	188.7	92.9
4000t	bishop	48.8	63.8	57.1	82.5
4000t	balloon	37.2	32.5	30.9	10.9
4000t	spitfire	10.0	9.3	7.4	3.8
4000t	747	5.6	11.0	8.7	12.9
s20.40rp	tetra	1129.9	1709.4	1818.2	438.6
s20.40rp	socbal	257.7	293.3	295.9	141.7
s20.40rp	bishop	25.4	52.8	35.6	53.4
s20.40rp	balloon	14.1	16.8	15.9	8.3
s20.40rp	spitfire	7.4	11.4	10.5	9.6
s20.40rp	747	10.7	15.3	15.5	18.1
s23.300rm	tetra	196.7	315.0	363.6	47.4
s23.300rm	socbal	189.8	229.4	208.3	68.5
s23.300rm	bishop	48.2	31.8	34.9	11.9
s23.300rm	balloon	14.3	14.5	18.7	5.5
s23.300rm	spitfire	7.2	4.4	8.7	1.2
s23.300rm	747	18.8	11.4	10.3	4.7

Table 8: Number of full resolution checks per second for each method.

Chapter 3

BV-Trees

As presented in Chapter 2, the collision detection method based upon a hierarchy of axis-aligned bounding boxes (i.e., R-trees) was the most promising of all of the methods we considered. It was the success of this method which directed our investigation towards utilizing similar data structures. In this chapter, we provide a much more thorough description of these data structures, including the choice of which bounding volume to use and other design choices when constructing these hierarchies. In our preliminary investigation, we built a hierarchy only on the environment; however, we have found that we can also build a hierarchy on the flying object and dramatically reduce the collision detection query times.

Parts of this and the next two chapters were originally published in the journal paper [59], written with Martin Held, Joseph S.B. Mitchell, Henry Sowizral, and Karel Zikan.

3.1 Introduction

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, ν, of BVT(S) corresponds to a subset, $S_\nu \subseteq S$, with the root node being associated with the full set S. Each internal (non-leaf) 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 δ. The subsets of S that correspond to the children of node ν form a partition of the set S_ν of objects associated with ν. 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 ν of BVT(S) is a *bounding volume*, $b(S_\nu)$, that is an (outer) approximation to the set S_ν using a smallest instance of some specified class of shapes (e.g., boxes, spheres, convex hulls, polytopes of a given class, etc.).

We emphasize that our BV-trees are a hierarchy on the *geometric objects*, not a hierarchy on the partitioning of space, such as octrees, BSP trees, k-d trees, etc. Spatial subdivisions and naive geometric hashing tend to blow up the memory consumption, as objects that overlap the boundary between cells of these subdivisions will be stored multiple times. Since our goal is to handle complex environments, potentially consisting of millions of polygons, we cannot afford this memory blowup. As we discuss our algorithms and data structures in the following chapters, we indicate the memory requirements of our methods and in Section 5.2, we provide a summary of the total memory usage.

We will be focusing primarily 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*.

When designing and constructing BV-trees, there are several important decisions that need to be made. The first, and perhaps most important, is which class of bounding volumes will be used to approximate the set, S_ν, of objects associated with each node, ν, in the hierarchy. This issue is addressed in the next two sections.

3.2 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 [56]. Weghorst, Hooper, and Greenberg [97] have discussed making this choice for ray tracing, and they provided a cost function to help analyze hierarchical structures of bounding volumes. Gottschalk, Lin, and Manocha [38] 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

$$T = N_v \times C_v + N_p \times C_p, \tag{1}$$

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 Equation (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 the cost is best written as the sum of *three* component terms:

$$T = N_v \times C_v + N_p \times C_p + N_u \times C_u, \tag{2}$$

where T, N_v, C_v, N_p, and C_p are defined as in Equation (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 (a) approximate tightly the input primitives (to lower N_v, N_p, and N_u), (b) permit rapid intersection tests to determine if two bounding volumes overlap (to lower C_v), and (c) 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.

3.3 Discrete Orientation Polytopes

We concentrate here 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.[1] See Figure 2(d) (page 61) for an illustration in two dimensions of an 8-dop, whose 8 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. We concentrate on 6-dops, 14-dops, 18-dops, and 26-dops, defined by orientations that are particularly natural; see Section 3.4.5 for more detail.

Researchers at IBM have used the same 18-dops (which they call "tri-boxes" or "T-boxes") for visual approximation purposes within 3DIX [25, 26]. We have also recently learned that researchers at GE have begun using 26-dops (which they have called "gems") for speeding up visualization of large datasets [7]. 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

[1]An alternative name for what we call a "dop" is the term *fixed-directions hull* [102] (FDH)—perhaps a slightly more precise term, but a harder to pronounce abbreviation.

and Kajiya [56].

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. Figure 2(a) highlights such a situation. 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" [56]. 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. In other words, we can think of k-dops as capturing the dimensions of objects along $k/2$ lines: project the objects onto a set of lines and then compute the *extents* along each line. Computing a k-dop can thus be carried out by computing simple dot products among the vertices and the k vectors. Consequently, computing the k-dop of a set of polyhedral objects with a total of n vertices can be done using this straight-forward (brute force) approach in $O(kn)$ time. More importantly, it is not only a simple but also a robust computation.

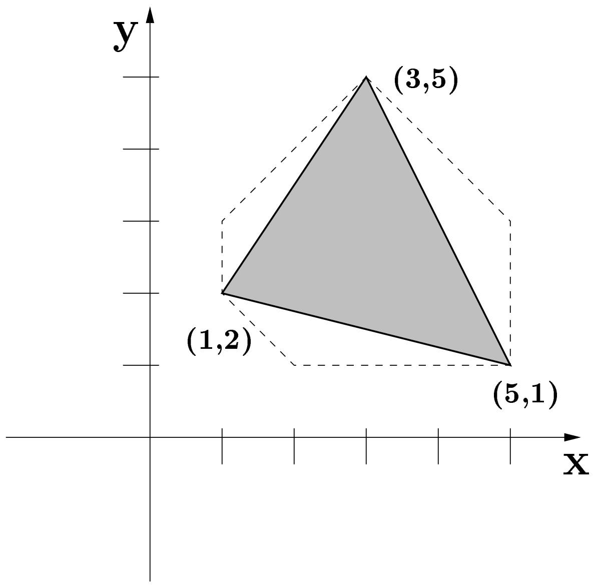


Figure 1: A simple 2D example of how we compute our k-dops. The 8-dop of the triangle is shown by the dashed lines.

To illustrate how we compute k-dops, consider the simple two-dimensional triangle shown in Figure 1. In this example, we will approximate the triangle with an 8-dop, as shown in Figure 2(d), and described above. We compute the 8-dop for the triangle by taking the dot-product of the three vertices of the triangle with each of the four vectors ($(1,0)$, $(0,1)$, $(1,1)$, and $(1,-1)$) which define this particular 8-dop. For example, to find the minimum and maximum extents along the direction $(1,1)$, we perform the three dot-products:

$$(1,1) \circ (1,2) = 3$$
$$(1,1) \circ (5,1) = 6$$
$$(1,1) \circ (3,5) = 8$$

and determine the minimum and maximum to be 3 and 8 respectively. Performing the other dot-products, we find that the minimum and maximum extents along the other three directions $(1,0)$, $(0,1)$, and $(1,-1)$ are $[1,5]$, $[1,5]$, and $[-2,4]$ respectively. This procedure is easily modified to compute the k-dops for more complicated objects.

Using tools from computational geometry, we can prove a better bound on the complexity of computing a k-dop when k is large. While this result is interesting in its own right, typically the construction of k-dops will be done using the straight-forward approach due to the ease with which it can be implemented.

Proposition 1 *The bounding k-dop of a set $\mathcal{S}$ of n triangles can be computed in time $O(\min\{kn, n\log n + k\log h\})$, where $h = |ch(\mathcal{S})|$ is the complexity of the convex hull of $\mathcal{S}$.*

Proof. We compute $ch(\mathcal{S})$ in time $O(n\log n)$ (e.g., using [80]). We then compute the Dobkin-Kirkpatrick hierarchy ([30]) for $ch(\mathcal{S})$, in time $O(h)$, after which "extremal queries" in each of the k directions can be answered in time $O(\log h)$ per query. □

In thinking of k-dops as intervals along a set of $k/2$ directions, our (conservative) test for intersection between two k-dops is essentially as trivial as checking two AABBs for overlap: we simply perform $k/2$ interval overlap tests. (For a complete discussion of our overlap test between two k-dops, please refer to Section 4.5). This test is far simpler than checking for intersection between

OBBs or between convex hulls. Even with the elegant results of [38], comparing two OBBs takes around 100 operations on average (200 operations in the worst case) and is significantly harder than comparing k-dops. 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. (See Figure 2(b).) Hence, bounding spheres yield low costs C_v and C_u, but may result in a large number, N_v, of bounding volume overlap tests, as well as a large number, N_p, of pairs of primitives to test. As a result of spheres being, in general, poor approximations to objects, a considerable amount of (preprocessing) time is required to construct a good hierarchical approximation using spheres, which was illustrated in Hubbard's work [49, 50].

Oriented bounding boxes (OBBs) can yield much tighter approximations than spheres and AABBs, as illustrated in Figure 2(d). In particular, when an OBB is used to approximate a single three dimensional triangle, the oriented box will tend to be "flat" along one of its axes, which is parallel to the normal vector of the triangle. 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 [38]. 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 overlap test of two k-dops 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 4.2, that works well in practice.

Figure 2 on page 61 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), a sphere, an oriented bounding box (OBB), and a k-dop (where $k = 8$).

3.4 Design Choices for Constructing BV-Trees

One of the most important decisions to make when building BV-trees is which class of bounding volume to use to approximate the input models. Having discussed this issue in the previous sections, and deciding to use k-dops, we now highlight several additional decisions that are also extremely relevant. In the following sections, we consider a comparison of various design choices in constructing BV-trees, including: (1) the degree, δ, of the tree (binary, ternary, etc.); (2) top-down versus bottom-up construction; (3) static versus dynamic data structures; (4) which k fixed directions to use for our k-dops; and (5) splitting rules. Before this, however, we mention a few words about building BV-trees in general.

3.4.1 The Preprocessing Step

In general, the construction of the BV-tree, BVT(S), is a preprocessing step, which does not have to be done in real time. This permits some optimizations to be done, which may result in greater speed during the real-time usage of the tree in collision detection. However, preprocessing effort is not free. While in some applications it may be acceptable to spend hours preprocessing a large environment in order to optimize the BV-tree for speed, in other interactive applications, the user will not be willing to wait more than a few seconds or, at most, a few minutes to complete the preprocessing. Thus, we feel it is important to allow the user to select options that control the degree of optimization used, and therefore the speed of preprocessing. Another option is to allow the user to input and output pre-computed BV-trees, which makes it possible to perform extensive optimizations on complex environments during off-peak times, and to have them handy in a matter of seconds when they are needed.

In the following sections of this chapter, we shall describe several choices that need to be made when constructing our BV-trees. Where appropriate, we shall highlight the amount of preprocessing time needed for the various options. The more complex the option (i.e., the more optimization that is performed), the faster we hope the queries will be. We may not, for instance, want to spend a week building a BV-tree to get a speed-up of 1% in query time as compared to a BV-tree that was built in one hour.

3.4.2 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, δ, 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 δ-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 $f(\delta) = (\delta - 1) \cdot \log_\delta n$, since at most $\delta - 1$ of the δ children need to be tested before we know how to descend. Simple calculus shows that the function $f(\delta)$ is monotonically increasing over the interval $\delta \in (1, \infty)$ (and $f'(1) = 0$). Restricting δ 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 δ.

3.4.3 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.* [13].

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 [38] 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.

3.4.4 Static versus Dynamic Structures

When constructing BV-trees, we also distinguish between *static* and *dynamic* data structures. A "static" structure is one which, once constructed, cannot be easily modified to accommodate insertions into or deletions from, the set S of geometric objects. Whenever a change in the set of primitive objects is made, the entire structure needs to be rebuilt from scratch. A "dynamic" data structure is the just the opposite: it *can* be updated as new objects are added or removed from the set S.

When using static structures, we must specify all of the objects in the set S initially, and then begin building our hierarchy. This has an advantage over dynamic structures in that having full knowledge of all of the objects which are to be approximated can allow us to perform more optimizations and thus get (potentially) better query times. Of course, at the same time, we are also restricted in that we cannot change the input set S without rebuilding BVT(S). Top-down and bottom-up approaches can be used for constructing static structures, although the top-down approach is more common.

Dynamic structures are used due to the flexibility they provide. When dealing with very large datasets, the amount of time required to build a BV-tree can be on the order of hours (or even days). Thus, if we will need to change some of the primitive objects in our set S, then we will not want to rebuild the entire hierarchy from scratch. The speed of the hierarchy update procedure (after a change in S has been made) is typically of great concern, as it makes little sense to spend as much time updating the hierarchy as it would take to rebuild it entirely. The update operations must therefore be done in real-time. In general, dynamic structures are built in a bottom-up fashion; the application of data retrieval below is one such example.

We will typically assume that the input set of primitives, S, is completely provided before any of the CD queries are to be performed and thus, we will build static data structures. This approach is also taken by most of the other CD packages available (e.g., RAPID). There are arguments as to whether this assumption is a good one, as different applications which make use of CD queries have different views on it. However, for almost all of the applications in which we have been working, this approach is very reasonable. For instance, once engineers have finished designing a large model (e.g., the new Boeing 777), it is unlikely to change. Therefore, making our assumption makes sense. This approach is also deemed reasonable for ray tracing. Once an environment has been designed (e.g., the model of an office building or house) it is not going to change and the one-time preprocessing step is worth the effort.

We have previously mentioned that R-trees are used for performing collision detection. These structures are also frequently used for searching and retrieving information from large (multidimensional) databases [41, 82, 88, 14, 55, 98, 57]. In this application, the databases are often subject to many insertions and deletions, so being able to efficiently maintain a dynamic data structure is very important. From this viewpoint, assuming full knowledge of the set of primitive objects is not realistic and therefore, not often done. Typically, the hierarchies for this application are built by inserting (or deleting) each primitive object one at a time and then updating the hierarchy accordingly. While in general, dynamic structures will not permit queries which are as fast as those performed on static structures, it is interesting to note that in comparing hierarchical data structures for data retrieval [57], the SR-tree, which is dynamic, performs almost as well as the VAM-Split R-tree, which is a static structure.

3.4.5 Choice of k-DOPs

We have discussed our rationale for using k-dops in Sections 3.2 and 3.3. Having done this, we still need to decide which $k/2$ fixed directions to use. Our investigations have consisted of four choices of k-dops: 6-dops (which are 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 directions, 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. Recall that one of the problems we mentioned about AABBs was their tendency to leave "empty corners". Thus, we have tried to avoid this problem by cutting off the corners.

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 $\{0, \pm1, \pm2\}$.

Figure 3(a) (page 62) 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 Figure 3(a) show, from left to right, and top to bottom, the corresponding 6-dop, 14-dop, 18-dop, 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, Figure 3(b) and Figure 4 depict our approximations of the spitfire using two k-dops (Level 1), four k-dops (Level 2), and 32 k-dops (Level 5).

3.4.6 Splitting Rules for Building the Hierarchies

Each node ν in a BV-tree corresponds to a set S_ν of primitive objects, together with a bounding volume (BV), $b(S_\nu)$. In constructing effective BV-trees, our goal is to assign subsets of objects to each child, ν', of a node ν, 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 $b(S_{\nu'})$. 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 [5]. 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_ν in two. There are $\frac{1}{2}(2^{|S_\nu|}-2)$ different ways to do this; thus, we cannot afford to consider all partitions. Instead, we associate each triangle of S_ν with a single "representative" point (we use the centroid), and we split S_ν 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 $3 \cdot (|S_\nu|-1)$ different nontrivial splits, since there are three choices of axis and, for each axis, there are $|S_\nu|-1$ 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, $b(S_\nu)$, is longest.

The amount of time required to evaluate each of the above objective functions varies greatly, and this leads to a 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, $O(|S_\nu|)$. 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 $O(k|S_\nu|)$ to compute the six k-dops for the candidate children, plus $O(k \log k)$ to compute the volumes of these k-dops.[2]

In Chapter 5, we report on the results of experiments comparing these four methods of selecting the axis. (See Tables 20 and 21.) The default method in the current software is the "splatter" method, which, while giving

[2]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 4.2.

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 variation 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. We have recently investigated this option and report upon its effectiveness in Section 6.3.

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 $|S_\nu| - 1$ 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 Chapter 2, for our preliminary investigation of bounding volume hierarchies, the median was always used for splitting, with the rationale that one wants to obtain the most balanced possible BV-tree. However, here we also investigate 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.

Table 9 shows the total volume of the environment hierarchies (using our 18-dop method) for the five datasets (Pipes, Torus, 747, Swept, Interior) which we commonly use for testing our collision detection algorithms. (See Chapter 5 for more details on these datasets.) For each of the datasets and for each of the splitting rules (as described earlier in this section), we computed the total volume occupied by the environment BV-tree for each of the two splitting points: the mean and the median. For all of these computations, using the mean *always* produced a hierarchy with smaller total volume than when using the median. In some cases, the reduction in the volume is as great as 27%, while in most cases the reduction was between five and 10%. It was also discovered that using the "Min Sum" splitting rule always provided the hierarchy with the least total volume, and the "Longest Side" rule produced the hierarchies with the greatest volumes.

As we have previously mentioned, the trade-off in using the mean as opposed to the median is that we no longer generate the most balanced BV-trees and thus, cannot guarantee the height of the hierarchy will be logarithmic. However, we have found in our experience that the height does not increase that dramatically by using the mean. In Table 10, we present the height of the environment hierarchies (again using the 18-dop method) for each of the splitting rules and splitting points. By using the median as a split point, all of the split rules will generate BV-trees of the same height, therefore, we only report using the median as a single row in the table.

For the datasets reported in Section 5.4, we also 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 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

Split Rule	Mean	Median
Pipes		
Longest Side	5.888139	7.974877
Min Sum	5.007636	6.444470
Min Max	5.272285	7.002653
Splatter	5.652836	7.797155
Torus		
Longest Side	0.651512	0.686250
Min Sum	0.591982	0.607904
Min Max	0.611783	0.635005
Splatter	0.617350	0.630568
747		
Longest Side	13.252572	14.748006
Min Sum	12.560419	13.627911
Min Max	13.587699	16.574298
Splatter	13.262544	14.819896
Swept		
Longest Side	30.241402	31.866640
Min Sum	26.312114	27.895800
Min Max	27.089401	29.058312
Splatter	27.722404	29.710274
Interior		
Longest Side	73,821,800	89,149,376
Min Sum	38,152,800	38,314,718
Min Max	38,599,044	43,419,997
Splatter	43,193,251	45,476,371

Table 9: For our five standard data sets, we have recorded the volumes of the environment hierarchies for our four splitting rules (longest side, min sum, min max, splatter) and two split points (mean, median).

Split Point	Splitting Rule	Pipes	Torus	747	Swept	Interior
Median	All	18	17	17	16	18
Mean	Longest Side	21	20	21	19	26
Mean	Min Sum	21	19	20	18	24
Mean	Min Max	22	19	21	18	24
Mean	Splatter	21	19	20	18	24

Table 10: For our five standard data sets, we report the height of the environment hierarchies for our four splitting rules and two split points.

time was also *greater* in every case when using the median: the smallest increase being 1%, and the largest increase being 35%. 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 20 and 21 in Section 5.4.

As we have previously mentioned, using the median does have its advantages: we can guarantee that the height of the BV-tree will be logarithmic. Using this fact, we present the following proposition which highlights the amount of time it takes to build a BV-tree on a set $\mathcal{S}$ of n input triangles.

Proposition 2 *A balanced BV-tree (built upon $\mathcal{S}$) of (full) height $O(\log n)$, can be constructed in time $O(\min\{kn \log n, (n+k)\log^2 n\})$.*

Proof. We construct the bounding k-dop of the full set of primitives and store it at the root. Then, to determine how to split, we split at the median coordinate of one of the centroids; this takes time $O(n)$ (by standard median-finding), assuming we select one from only a constant number of splitting axes.

Using Proposition 1, we compute in time $O(\min\{kn, (n+k)\log n\})$ the new k-dops of the resulting children, and recurse. This divide and conquer algorithm requires time $f(n)$, where

$$f(n) = O(\min\{kn, (n+k)\log n\}) + 2f(\frac{n}{2}),$$

which solves to $f(n) = O(\min\{kn\log n, (n+k)\log^2 n\})$. $\square$

Our implementation selects between only these two possibilities (the mean or the median) for our split point. 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 (a) *all* $|S_\nu| - 1$ different centroid coordinates, or (b) a random subset of these coordinates.

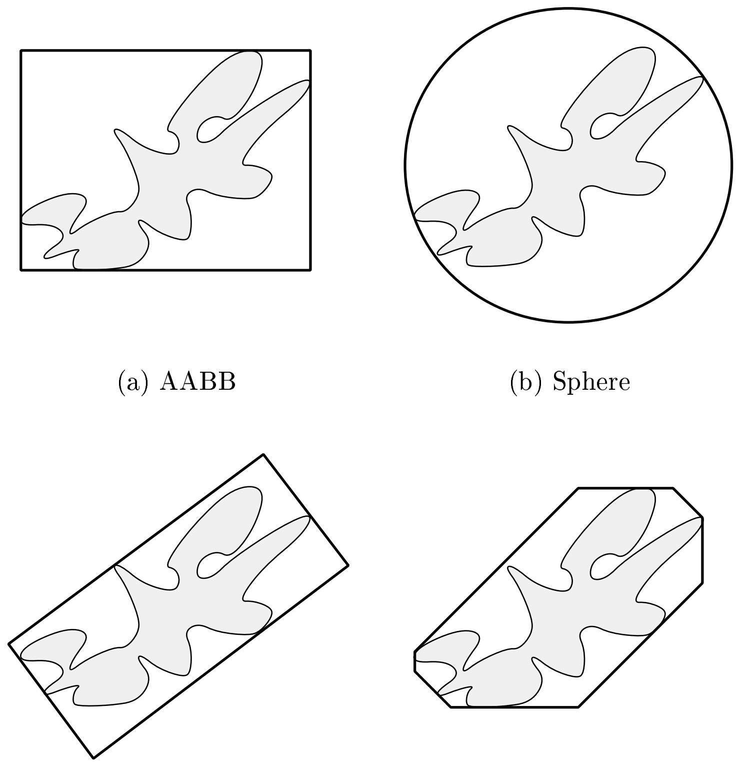

(a) AABB (b) Sphere (c) OBB (d) 8-dop

Figure 2: Approximations of an object by four bounding volumes: an axis-aligned bounding box (AABB), a sphere, an oriented bounding box (OBB), and a k-dop (where $k = 8$).

Chapter 4

Collision Detection Using BV-Trees

4.1 Introduction

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: (a) 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; (b) the algorithm for comparing the two BV-trees to determine if there is a collision; (c) the depth of the flying hierarchy; and (d) the overlap test between two k-dops.

4.2 Tumbling the BV-Trees

For each position of the flying object, 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, $b(S_\nu)$, 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 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 required 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 Equation (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 as it can easily take a few seconds each time the hierarchy is constructed. A somewhat less naive approach is to preserve the structure of the flying hierarchy, with no changes to the sets S_ν, 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_ν).

So, we considered two other methods to tumble the nodes ν, while preserving the structure of the (original) hierarchy:

(I) A "hill climbing" algorithm that stores the B-rep (boundary representation) of the convex hull of S_ν, and uses it to perform *local* updates to obtain the new (exact) bounding k-dop from the bounding k-dop of S_ν 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.

(II) An "approximation method" that attempts only to find an *approximation* (an outer approximation) to the true bounding k-dop for the transformed S_ν. This method stores only the vertices, $V(S_\nu)$, of the k-dop $b(S_\nu)$, computed once, in the model's *initial orientation.*[1] Then, as S_ν tumbles,

[1]It is important that we transform the original B-rep vertices, rather than those of the

we use the "brute force" method to compute the exact bounding k-dop of the transformed set $V(S_\nu)$; this bounding k-dop still contains the transformed S_ν, but it need not be the smallest k-dop bounding it.

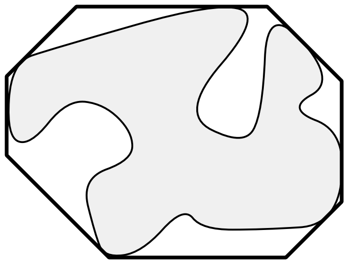
Figure 5 shows a two-dimensional example of method (II). In this example, $k = 8$, and the original object and 8-dop are shown in Figure 5(a). Figure 5(b) depicts the object rotated 30 degrees (counterclockwise) and the corresponding 8-dop. The result of tumbling the original k-dop and recomputing the new k-dop is shown in Figure 5(c). The dashed lines represent the rotated (original) 8-dop, and the solid lines show the new 8-dop that we use to approximate the object. Ideally, we want our approximate 8-dop to be very close to the exact 8-dop shown in Figure 5(b). 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 8-dop depicted in the figure, rotating the object by 45 degrees causes the tumbled k-dop to coincide with the exact k-dop.

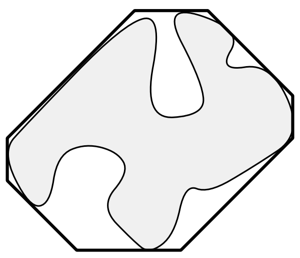
Both methods (I) and (II) rely on a preprocessing step in which we compute a B-rep. In method (I), we precompute the convex hull of the vertices of S_ν, and store the result in a simple B-rep. In method (II), we must compute the vertices in the B-rep of the k-dop $b(S_\nu)$, for the original orientation of S_ν. This means that we must compute the intersection of k halfspaces. This is done by appealing to the following fact (see, e.g., [76]): 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*[2] to one of the planes defining the halfspaces,

bounding k-dop at each step. Indeed, if we were to transform the bounding k-dop, compute a new bounding k-dop, transform it, etc., the bounding volume would grow increasingly larger with each step.

[2]In one standard definition of duality, the *dual point* associated with the plane whose

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 [76]).[3]


(a) 8-dop


(b) 8-dop of rotated object

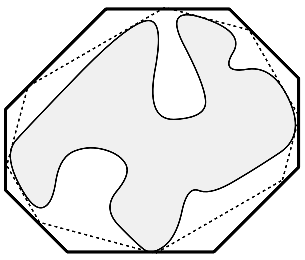

(c) 8-dop of rotated 8-dop

Figure 5: Illustration of the approximation method (II) of updating the k-dop which approximates a rotating object.

equation is $ax + by + cz = 1$ is the point (a, b, c). See [76].

[3]Although this algorithm has worst-case quadratic ($O(k^2)$) running time, it works well in practice, is only used during preprocessing, and k is small. Worst-case optimal $O(k \log k)$-time algorithms are known for this problem; see [81].

We have considered some of the trade-offs between methods (I) and (II). 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 (I) 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 (I) 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 (I) 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 (I) for any nodes other than the root node. Consequently, we are currently using the approximation method (II) for all nodes in the hierarchy, except the root node, where we perform hill-climbing (I).

Regardless of which method is used to update the nodes in the flying hierarchy, it is important to understand that it is not necessarily required that we update *all* nodes of the hierarchy at each location of the flying object. In most of the cases, we will be able to determine which triangles in the flying object (if any) are in contact with the environment without updating every node. As we shall see in the following section, we begin searching the environment hierarchy with the root node of the flying BV-tree. If we determine that no environment triangles overlap with this node, the crudest approximation to the flying object, then have determined that the flying object and the environment are disjoint and we do not have to update (or tumble) any of the other nodes in the flying BV-tree. If the root node of the flying BV-tree does overlap some of the environment triangles, then we may have to tumble additional nodes, however, it is a highly pathological case in which all nodes of the flying BV-tree will need to be updated. In addition, a node, ν, that is tumbled at a particular location of the flying object will not be tumbled again due to a flag which is maintained for each node in the flying hierarchy. Once the flying object has been moved to a new location, ν may need to be updated again (as determined by the search algorithm), but it will only be updated once for each location of the flying object.

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.

4.3 Tree Traversal Algorithm

Given the environment hierarchy, BVT(E), and the flying hierarchy, BVT(F), 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 currently use is outlined in Algorithm 1. It consists of a recursive call to *TraverseTrees*(ν_F, ν_E), where ν_F is the current node of the flying hierarchy and ν_E is the current node of the environment hierarchy. Initially, we set ν_F and ν_E to be the root nodes of the hierarchies. Prior to calling the search algorithm, however, we must first update ν_F (using Method I as described in Section 4.2) so that it is approximating the flying object in the current location. Other nodes in the flying hierarchy will be updated prior to the recursive call in Step 9 (see the description below).

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

Algorithm 1: Pseudo-code of the tree traversal algorithm.

At a general stage of the traversal algorithm, we test for overlap between the bounding volume $b(S_{\nu_F})$ and the bounding volume $b(S_{\nu_E})$. If they are disjoint, then we are done with this call to the function. Otherwise, if ν_E is *not* a leaf, we step down one level in the environment hierarchy, recursively calling *TraverseTrees*(ν_F, ν_e) for each of the children ν_e of ν_E. If ν_E *is* a leaf, then we check if ν_F is a leaf: if it is, we do triangle-triangle intersection tests between each triangle of ν_E and each triangle of ν_F; otherwise, we step down one level in the flying hierarchy, recursively calling *TraverseTrees*(ν_f, ν_E) for each of the children ν_f of ν_F. It is at this stage that we update (or tumble) the nodes in the flying hierarchy if they have not already been updated for this particular location of the flying object.

For comparison purposes, we have also implemented a variant of this traversal algorithm in which Step 9 of the algorithm is replaced by *TraverseTrees*$(\nu_f$, root of the environment hierarchy). The rationale for this variant is that it may be that the bounding volume at a node ν_F of the flying hierarchy intersects a large number of leaves in the environment hierarchy, BVT(E), while the children of ν_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 ν_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. In general, this approach performs more overlap checks and updates more nodes than our default search algorithm. While there are cases in which this variant is better, yielding a slightly lower (by about 5%) 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%. Table 22, in

Section 5.4, compares these two search algorithms for our 18-dop method.

We should also emphasize that no search algorithm will always be the most efficient under every circumstance. Having some information *a priori* about the interaction of the moving object and the environment can greatly improve the collision detection query time since we could better tailor the search algorithm for these specific conditions.

Time-Critical Computing. Bounding volume hierarchies are useful for time-critical CD checks (cf. Hubbard [49, 50]), in which it is essential to maintain a given frame rate (e.g., for haptics), even at the expense of interrupting the CD algorithm before it completes a query with a definitive answer. Of course, once a CD query discovers disjointness, it stops and reports it, so if the CD query algorithm is interrupted, it must be because we do not yet know that the flying object is disjoint from the environment. (Another possibility is that the algorithm has found at least one instance of contact between the flying object and the environment, but has not yet completed the search for *all* contacts.) Thus, we could run our algorithm in time-critical mode and report a *conservative* answer – "collide." The effectiveness of our bounding volume hierarchies for time-critical computing depends, then, on how well we can bound the degree of overlap (depth of possible penetration) in those cases in which we terminate the search prematurely. (See, e.g., [102])

In particular, in order to terminate with a small degree of possible overlap, we must consider carefully our choice of tree traversal algorithm: By descending the trees of the environment and the flying object(s) in *lock-step* (e.g., alternately descending one level of each hierarchy), we are able to decrease the average overlap between the bounding volumes that cover the flying object(s) and those that cover the environment obstacles, at any given state of the search. This is in contrast with the searching strategy that we otherwise employ (as described above), when our goal is to traverse the trees in such a way as to minimize the total time needed to report *all* collisions exactly.

4.4 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 Equation (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. This is because as the depth of the hierarchy increases, the quality of the approximation of the input models also increases. Referring again to Figure 3 (page 62) and Figure 4 (page 63), we can see that the quality of the approximation at Level 1 of the BV-tree is better, or tighter, than the Level 0 approximation, for each of the k-dop methods. Similarly, if we compare the Level 2 approximation (which uses $2^2 = 4$ k-dops) to Level 1, we again have a more tightly fitting set of bounding volumes. Finally, it is plain to see that the Level 5 approximation, which consists of $2^5 = 32$ k-dops, is the best.

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 τ; once the number of triangles associated with a node falls below τ, 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. We have recently, however, taken a closer look at the selection of our "hard-coded" thresholds, and in Section 6.3.3, we suggest a simple-minded scheme which has lead to improved CD query times.

4.5 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 Equation (2). Recall that all of 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. As in the case of an AABB (which is defined by three intervals along the x, y and z-axis), two k-dops, D_1 and 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 we find that the k-dops overlap along all of the directions, then we conclude that the k-dops *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 comparisons of floating-point numbers, and no arithmetic operations, in our overlap test.

4.5.1 k-dop Overlap Tests

As we indicated above, two k-dops *can* overlap along all $k/2$ intervals which define the k-dops, but still be disjoint. This claim is made based upon the following proposition and corollary [4].

Proposition 3 *Given convex polytopes P and Q (in $\Re^d$), with disjoint interiors, there exists a separating hyperplane h that is spanned by some set of $(d-1)$ vectors that are parallel to edges of P or Q.*

Proof. $P \cap Q = \emptyset$ if and only if $0 \notin P - Q$ if and only if a hyperplane supporting a facet of $P - Q$ separates 0 from $P - Q$.

This completes the proof, since it is well known that facets of $P - Q$ are of the form $F - G$ (for faces F and G of P and Q), so such a facet is spanned by a set of $d-1$ edges of $F - G$; these edges are parallel to edges of F or G (and hence of P or Q) [5]. □

More specifically, for three dimensional polytopes P and Q, we (immediately) have the following corollary, where $\pi(P, \ell)$ denotes the projection of P onto the line ℓ.

[4]These results are closely related to what has recently been called the "separating axis theorem" by Gottschalk, *et al.* [38].

[5]I thank Lou Billera (Cornell University) for discussions about these facts of polyhedral theory. His proof is included here.

Corollary 4 *$P \cap Q = \emptyset$ if and only if $\pi(P, \ell) \cap \pi(Q, \ell) = \emptyset$, for all $\ell \in L$, where L is a set of lines parallel to the set of directions determined by cross products between all pairs of edges of P and of Q.*

These results raise the interesting question as to when is it *sufficient* to check only the intervals defined by the $k/2$ directions to determine if two k-dops overlap. It is well-known that when testing two axis-aligned bounding boxes (our 6-dops), we can simply check the three intervals defined by the x, y, and z directions. However, we have found that when checking our 14-dops, it is not sufficient to test the only the seven intervals defined by the seven normal vectors. By checking only the seven intervals, we can conclude that the two k-dops overlap (if all of the intervals overlap) when, in fact, the k-dops do not overlap. Our approach is conservative as we have pointed out and we cannot, therefore, miss any collisions between objects.

Let us define the *closure* of a set of k normal vectors (these are the normals which define a k-dop) to be a the set of direction vectors obtained by taking the cross product of any pair of edges of the k-dops. Since the facets of the k-dops have *fixed* directions, all of the edges of a k-dop must also be from a certain set of fixed directions. We can find the directions of the edges of a k-dop by simply taking the cross product of a pair of normal vectors which define the k facets of our k-dop. A naive bound, therefore, on the number of directions in the closure of a set N of k normal vectors is k^4.

We can then conclude from the proposition and corollary above that if the closure of N, $\mathcal{C}(N)$, equals N, (i.e., if N is *closed*) then it is sufficient to check only the intervals defined by these normals to determine if two k-dops actually overlap.

Proposition 5 *Let N_6, N_{14}, N_{18}, and N_{26} to be the sets of k vectors which define our 6-dop, 14-dop, 18-dop, and 26-dop, respectively. Then,*

$\mathcal{C}(N_6) = N_6$, *and*

$\mathcal{C}(N_{14}) = N_{26}$.

Proof. The proof of the proposition is by means of an exhaustive enumeration of the k^4 directions in the closure for these particular k-dops. $\square$

For our 18-dop and 26-dop, we suspect the closure of these sets is a superset of N_{18} and N_{26}; that is, they are NOT closed under this operation. But we note that, since we are taking 2-by-2 determinants of matrices of 0, 1, -1, each of these closures is a set of vectors having integer coordinates in $\{0, \pm 1, \pm 2\}$, and thus there are at most $(5^3 - 1)/2 = 62$ nonzero elements defining distinct directions. We do not enumerate the sets here.

For each of our k-dops, we could easily perform an *exact* overlap test by using all of the candidate separating directions that are within the closure of N_k. As mentioned above however, we have taken a conservative approach to this problem, using only the $k/2$ directions which define the k-dop. We note again, that our "conservative" approach is exact for our 6-dops, as $N_6 = \mathcal{C}(N_6)$.

4.5.2 Order of the Interval Tests

In performing our 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 to quickly find a direction (if one exists) along which the given intervals do not overlap, and thus exit the routine.

Figure 6 highlights this situation (in 2D). If we project the two boxes[6], B_1 and B_2, onto the x-axis (denoted d_1), we see that their projections clearly overlap. Similarly, the boxes' projections onto a second direction, d_2, which is oriented fairly closely to d_1, also results in an overlap. In fact, all directions in between d_1 and d_2 will also result in an overlap of the projections. However, if we consider a direction which varies greatly from d_1, such as d_3, we may be able to determine that the boxes do not overlap, as in this case. Of course, as we have already pointed out, if the two boxes do overlap along all $k/2$ directions, then no matter how we order the intervals to be tested, they will all result in overlaps. The purpose of this approach is to terminate the routine as quickly as possible by determining that the k-dops are disjoint (if they actually are).

Our current implementation uses this philosophy for ordering the intervals in our overlap test. However, we are also considering other orderings which may also speed up (on average) the overlap queries. These details are discussed in Section 6.4.1

4.5.3 Triangle-Triangle Intersection Tests

Finally, at the lowest level of our CD query algorithm (when a leaf of the environment hierarchy overlaps a leaf of the flying hierarchy), we ultimately must be able to test whether or not two primitives (triangles) intersect. The

[6]We would refer to these two-dimensional boxes as 4-dops.

cost of this operation has been denoted C_p; it involves arithmetic operations on floating-point numbers. Martin Held has developed a collection of efficient intersection tests for pairs of primitive geometric elements; see Held [43] for details on the triangle-triangle intersection test that we use.

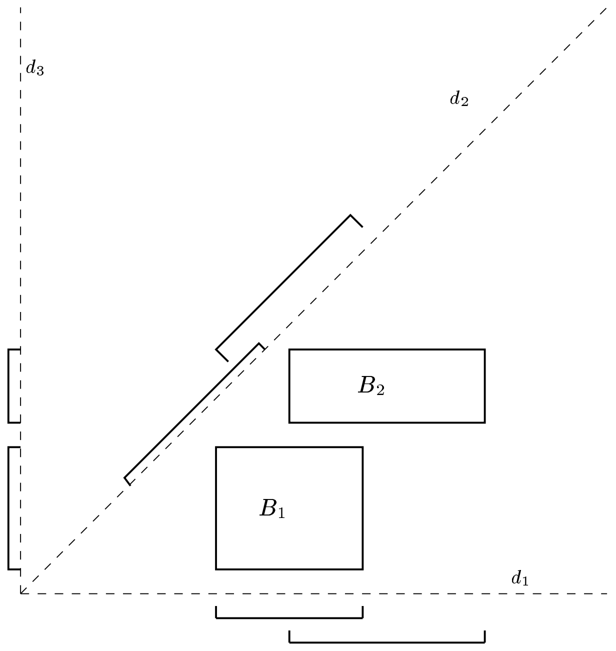


Figure 6: An example to illustrate our philosophy in ordering the intervals in the overlap test. Directions d_1 and d_2 are oriented relatively closely, and both result in an overlap of the projections of the two boxes. Direction d_3 is oriented at a 90 degree angle to d_1, and disjointness is discovered.

Chapter 5

Implementation and Experimentation

5.1 Introduction

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[1], (self-)intersections, and other deficiencies of the input data.

For many applications, one is forced to handle such "bad data", which may be CAD surface models having no topologically consistent boundary representation, or having unintentional penetrations between some pairs of primitives. Thus, we have assumed no topological structure on the input. We simply

[1] "Cracks" are gaps on the surface of a polygonal model caused by an edge having only one incident face.

assume that the input consists of a list of vertices and a list of triangles without any adjacency information. For this reason, we can only report surface intersections, rather than volumetric intersections.

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, ϵ, which can be specified by the user. Rather than using ϵ during the search of our BV-trees, we have opted to "buffer" the k-dops in the environment hierarchy by ϵ when they are computed. This ensures that we will not miss collisions and that our search will not be slowed down by any arithmetic operations. Of course, our routines which test two triangles for intersection *do* perform arithmetic operations and use ϵ.

In order to maintain efficiency in the implementation of k-dops, we have "hard-coded" 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 and future chapters, we report on some comparisons with the system called "RAPID" (Rapid and Accurate Polygon Interference Detection), which has been made publicly available[2] by the University of North Carolina at Chapel Hill, and utilizes oriented bounding box trees (OBBTrees).

[2]http://www.cs.unc.edu/~geom/OBB/OBBT.html

RAPID, which was developed in parallel with our system (both were first reported at SIGGRAPH'96; see [38, 45]), is one of the leading CD systems available. Our results here (and reported in [45]) show that our system, `QuickCD`, compares favorably with RAPID, in both speed and memory usage, being substantially faster in many cases. We conclude that in most cases the use of our methods based on bounding k-dops are potentially a better choice than methods based on oriented bounding boxes.

5.2 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 8-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. 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.

For each node of the environment hierarchy, we need to store the k numbers that define the bounding k-dop (eight 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,

	Pipes	Torus	747	Swept	Interior
No. of Triangles	143,690	98,114	100,000	40,000	169,944
6-dop	28.0	19.1	19.5	7.8	33.1
14-dop	45.5	31.1	31.7	12.7	53.8
18-dop	54.3	37.1	37.8	15.1	64.2
26-dop	71.8	49.0	50.0	20.0	84.9
RAPID	56.5	38.6	39.3	15.7	66.8

Table 11: Total memory (in megabytes) required to store the environment and its hierarchy for the five datasets.

and 26, we see that we need 204, 332, 396, and 524 bytes per input triangle, respectively. For comparison, it has been reported in [38] that the RAPID implementation requires 412 bytes per input triangle. Table 11 highlights the total memory required to store the environment and its BV-tree, for the five datasets which we typically use for testing our collision detection algorithms. These datasets are to be introduced shortly.

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, τ, for stopping the construction of the hierarchy (as discussed in Section 4.4) 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 4.2) 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

	Pipes	Torus	747	Swept	Interior
6-dop	16.4 MB	2.3 MB	1.5 MB	7.3 KB	47.5 KB
14-dop	20.8 MB	2.9 MB	1.9 MB	7.3 KB	60.7 KB
18-dop	22.1 MB	3.1 MB	2.1 MB	7.4 KB	72.2 KB
26-dop	25.9 MB	3.6 MB	2.4 MB	7.4 KB	79.5 KB
RAPID	56.5 MB	7.9 MB	5.8 MB	14.5 KB	162.5 KB

Table 12: Total memory required to store the flying object and hierarchy for our five datasets. Megabytes and kilobytes are abbreviated MB and KB, respectively.

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 eight, 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 re-tumbling the node if it is accessed more than once during a single CD query (Section 4.2). In addition, we also need to store the B-rep for the convex hull associated with the root node of the flying hierarchy (Section 4.2). In the experiments reported here, the flying "Pipes" dataset required the most memory, almost 1.65 megabytes, to store its convex hull.

Table 12 reports the *actual* total memory used[3] to store the flying object and hierarchy for each of our datasets. In the following discussion, we further

[3]This does not include the memory required by the run-time environment.

	Pipes	Torus	747	Swept	Interior
6-dop	12.9 MB	1.8 MB	1.3 MB	3.2 KB	37.0 KB
14-dop	13.5 MB	1.9 MB	1.3 MB	3.2 KB	38.9 KB
18-dop	13.8 MB	1.9 MB	1.4 MB	3.3 KB	45.1 KB
26-dop	14.5 MB	2.0 MB	1.4 MB	3.3 KB	41.8 KB

Table 13: Number of bytes required to store the flying object and hierarchy, excluding the B-rep vertices and the convex hull.

	Pipes	Torus	747	Swept	Interior
6-dop	1.9 MB	262.1 KB	203.3 KB	0 KB	5.6 KB
14-dop	5.7 MB	778.8 KB	591.1 KB	0 KB	16.9 KB
18-dop	6.7 MB	997.0 KB	706.5 KB	0 KB	22.1 KB
26-dop	9.8 MB	1.4 MB	1.0 MB	0 KB	32.7 KB

Table 14: Number of bytes required to store the B-reps of the original flying hierarchy.

break down this memory usage by components.

As indicated above, the number of leaves, m', in the flying hierarchy helps to determine the amount of memory required by our BV-tree. The number of leaves in each of the flying BV-trees was respectively 5235, 700, 543, 1, 16, for the “Pipes”, “Spikes”, “747”, “Swept” and “Interior” datasets. Using this information, we have tabulated in Table 13 the memory needed to store the flying object and the hierarchy, excluding the B-reps and convex hull. The remaining two components which contribute to the memory needed by the flying object cannot be computed with a simple formula. Therefore, we have tabulated the actual memory needed for each of the five datasets for these components in Tables 14 and 15.

	Pipes	Torus	747	Swept	Interior
6-dop	1.6 MB	251.8 KB	47.3 KB	4.1 KB	4.9 KB
14-dop	1.6 MB	251.9 KB	47.3 KB	4.1 KB	4.9 KB
18-dop	1.6 MB	251.9 KB	47.3 KB	4.1 KB	5.0 KB
26-dop	1.6 MB	251.9 KB	47.3 KB	4.1 KB	5.0 KB

Table 15: Number of bytes required to store the convex hull of the flying object.

	Pipes	Torus	747	Swept	Interior
6-dop	44.4	21.4	21.0	7.8	33.1
14-dop	66.3	34.0	33.6	12.7	53.9
18-dop	76.4	40.2	39.9	15.1	64.3
26-dop	97.7	52.6	52.4	20.0	85.0
RAPID	113.0	46.5	45.1	15.7	67.0

Table 16: Total memory (in megabytes) required to store the environment, the flying object, and their BV-trees.

The *total* memory required by our methods to process each of the datasets is listed in Table 16. We see that our three smallest k-dops all require less memory than the RAPID implementation. Although our 26-dop requires less memory on the "Pipes" dataset, in general, using this bounding volume will require more memory than when using the OBB within RAPID.

5.3 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 megabytes 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.

5.4 Experimental Results

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 17, 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 4.2. The k-dops used in these tests were taken directly from the experiments (flights) described later in this section.

	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 17: Average costs of C_v and C_u (in ms), for different choices of k.

Average Collision Detection Query Times

Table 18 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[4]; (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. Figures 8 and 9, which show a portion of these flights, can be found at the end of this chapter on pages 102–103.

	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
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
RAPID / 18-dop	2.552	1.315	1.388	5.101	1.589

Table 18: Average CD time (in ms), using our "Splatter" splitting rule.

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. This is the identical method of path generation as described in Chapter 2. For a more detailed look at accurately handling collision response, please refer to the work by Moore and Wilhelms [70], Bouma and Vaněček [16], and the large collection of work by Baraff [10, 11, 12].

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

Timing results for a fifth dataset, *Interior*, are also listed in Table 18. Images of this particular flight are shown on page 104 in Figures 10(a) and 10(b). 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, i.e., all pairs of triangles that are in intersection, during the flight. As seen in Table 18, there were many such contacts for this flight, with an average of 33 contacts per step, over the 2500 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. In each of our tests, we report all of the contacts/collisions that occurred. Naturally, we can expect the tests to be much faster in the case that we stop at the first detection of a collision. We chose to report all contacts, since this is what is done in the RAPID system, and we wanted to make direct comparisons.

Based solely upon these times, our 14-dop, 18-dop, 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 6-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.

The last row of Table 18 highlights the improvement of using our 18-dop method over using RAPID's implementation of OBBs. That is, we have divided the time required when using OBBs by the time required when using 18-dops, to see the relative speed-up. In the best case (Swept), our 18-dop method is five times faster. In the worst case, we are 1.3 times faster.

Splitting Rules and Split Points

For the results in Table 18, all of our hierarchies were built using one of our fastest construction algorithms, based upon the "splatter" splitting rule discussed in Section 3.4.6. 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 19 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[5]. Our fastest methods are clearly the "longest side" and "splatter" algorithms, which are

[5]In each of these cases, we split at the mean rather than the median.

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 19: Preprocessing time (in minutes), using our 18-dop method.

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% 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 19, Table 20 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 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

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 20: Average CD time (in ms), using our 18-dop method, dividing at the mean.

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 3.4.6 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 21, 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.

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

Table 21: Average CD time (in ms), using our 18-dop method, dividing at the median.

In comparing Tables 20 and 21, we see that using the median never results in faster query times. In quite a few cases, the median method is at least 5%

slower than the mean method, and in the "Pipes" dataset, the median method is between 24% and 35% 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.

Tree Traversal Algorithms

Recall that in Section 4.3, we discussed our current algorithm for searching the two BV-trees to determine if the flying object comes into contact with the environment. We also highlighted an alternative search algorithm which, on average, did not perform as well. In Table 22, we report on the average collision detection times for each of the methods for our 18-dop method. We refer to our current algorithm as "current" and the alternative algorithm as "modified". As we mentioned earlier, the alternative method is typically always slower, often by as much as 10–20%.

	Pipes	Torus	747	Swept	Interior
Current	0.366	0.184	0.356	0.109	2.754
Modified	0.434	0.220	0.430	0.109	2.934

Table 22: Average CD Time (in ms), using our 18-dop method, for our current tree-traversal algorithm and the modified algorithm. For all of these comparisons, the splitting rule was the Splatter method and the splitting point was the mean.

Parallel Close Proximity

As Tables 18–21 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 [38]. 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 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 [38]. Therefore, we have run an experiment similar to one run in [38], 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 α. 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 [38], our objectives are 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.

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

Table 23: Numbers of overlap queries among k-dops of the 2,000-faceted nested spheres, for different values of alpha and k.

Tables 23 and 24 report our results for spheres of 2,000 triangles each, and spheres of 20,000 triangles each[6]. 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 [38].

Our goal here, though, was to compare the OBB method to the k-dops

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

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

Table 24: Numbers of overlap queries among k-dops of the 20,000-faceted nested spheres, for different values of alpha and k.

methods. As the tables show, for both of the datasets, our 14-dop, 18-dop, and 26-dop methods performed fewer bounding volume overlap queries for the largest value of α, 0.55, when the nested spheres are relatively well separated. For the remaining values of α, 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 α 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.

We emphasize that in modifying our methods for this experiment, we are considerably handicapping them. By their very nature, OBBs tend to approximate a single triangle by a box with one axis aligned with the normal vector of the triangle. Thus, the OBB will tend to be a “flattened” three dimensional box along one of its directions. Our k-dops are not as proficient in approximating a triangle, since the k-dops directions are predetermined. It is therefore, not very surprising that in this experiment (where we force the triangles on the inner sphere to come very close to the triangles on the outer sphere) that the RAPID implementation will perform fewer bounding volume overlap tests. Having said that, it is still an interesting experiment to get a better feel for how the bounding volumes will perform.

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. Putting an upper bound on worst-case CD time is especially important in virtual reality applications, where one often needs to perform time-critical collision detection (see Section 4.3).

On a typical flight (that of the “Pipes” being flown within the larger system of “Pipes”), we show a plot in Figure 7 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.

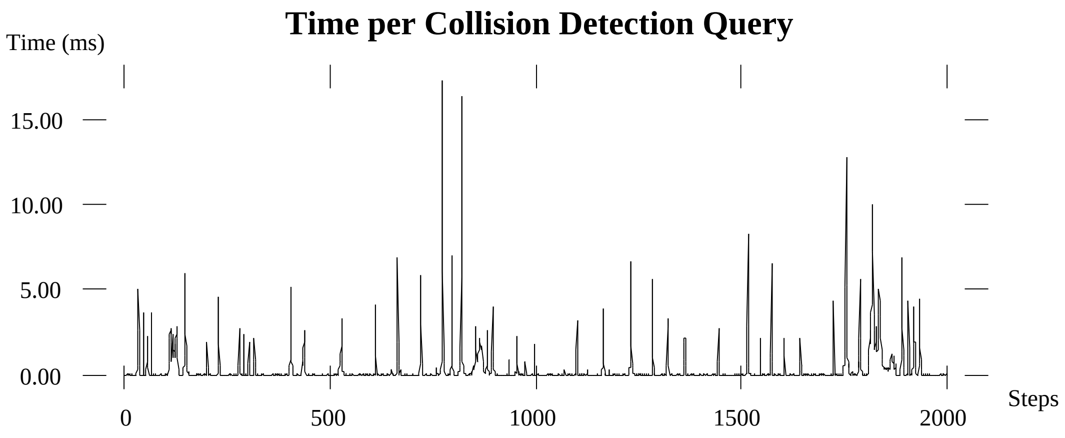


Figure 7: Individual collision detection query times for the “Pipes” dataset.

Chapter 6

Improvements and Extensions

6.1 Introduction

Throughout the descriptions of our data structure and algorithm for collision detection (Chapters 3 and 4), we have pointed out several possible improvements. In the following sections, we shall discuss some of these as well as some additional modifications that have recently been tested. Some of the changes lead to significant improvements in the average collision detection (CD) query time and have become part of our current implementation (see Chapter 7). Others, while interesting in their own right, cannot be included, as the benefits (if any) they provide, do not outweigh the additional cost (e.g., longer preprocessing, additional memory) associated with them.

The motivation for all of the improvements (with the exception discussed in Section 6.2) is to speed up the average CD query time; however, we have classified the modifications based upon their underlying method, either improving the design of the BV-trees or making better use of the BV-trees provided.

The former involves constructing better (tighter) approximations of the environment and flying object, usually through a more complicated preprocessing step. The latter consists of some new ideas and clever ways of improving some of the current ideas.

Prior to presenting these modifications, we discuss a dramatic improvement in the overall construction of our BV-trees. In particular, we can construct the same trees as we had before, but the amount of preprocessing time required has been greatly reduced.

6.2 Faster Preprocessing

One of the most disappointing statistics within Chapter 5 was that the OBB implementation known as RAPID was performing its preprocessing step considerably faster than all of our methods. Thus, even though our methods can perform collision detection queries faster on average, some users may still choose to use the OBB method due to the faster hierarchy construction. We have therefore spent some time improving our preprocessing step by reorganizing the code and by making better use of some of the code that is already available to us. In particular, by eliminating some of the options that have been shown to be inferior for answering collision detection queries (e.g., using the median as the split point of a node in a BV-tree), we have been able to simplify the code considerably and also achieve dramatic speed-ups. In addition to the cleaning that was done, we have also implemented many of the basic operations as “macros” to achieve greater efficiency. One additional item which may have helped these timings is that the machine on which we have run our experiments has recently been upgraded so that it now has 320 megabytes (MB) of main memory, as opposed to the 192 MB, which it originally had when we ran the experiments reported in Section 5.4.

In comparison to Table 19, we have included in Table 25 the current preprocessing times for our 18-dop method. In each case, the mean was used as the splitting point. Note that the units in this new table are *seconds*, as opposed to minutes. The speed-ups are quite dramatic. The Interior dataset, for example, previously required over five minutes to preprocess, but now we can do so in only a matter of 11 seconds.

As we have previously said, a large majority of the speed-up is due to reorganizing the code and simplifying things due to a greatly-reduced number of options which had cluttered the original code. However, the improvements have been so great that we are deserving of some criticism for not implementing the preprocessing step more efficiently in the first place. There were several instances where macros were used very inefficiently, and thus the original code was fairly slow for the larger datasets. In addition, to be completely fair in our comparisons with the RAPID implementation, the preprocessing times reported in Tables 19 and 25 are not for the most recent release (version 2.01) of the RAPID system. However, these times are reported in Chapter 7, which describes our current implementation of `QuickCD` and compares it to the latest versions of two collision detection systems (including RAPID).

Construction Method	Pipes	Torus	747	Swept	Interior
Longest Side	9.10	5.76	6.88	2.34	11.02
Splatter	10.05	6.39	7.85	2.56	12.18
RAPID	62.97	41.28	42.86	15.62	78.48

Table 25: Preprocessing Time (in seconds), using our 18-dop method.

6.3 Improving the Design of the BV-Trees

6.3.1 Choice of k-DOPs

Chapter 3 explained our rationale behind choosing k-dops for our bounding volumes in our BV-trees. In Section 3.4.5, we discussed the four specific k-dops which we have been investigating: the 6-dop, 14-dop, 18-dop, and 26-dop. At that time, we also discussed using other k-dops as well. Here, we have tried using three additional bounding volumes, not with values of k greater than 26, but instead which are slightly modified versions of some of our existing ones. In particular, we have tried removing the six planes which define the axis-aligned bounding box from the 14-dop, 18-dop, and 26-dop. This has produced 8-dops, 12-dops, and 20-dops. We felt that these k-dops are also quite natural to experiment with in order to see if the remaining planes (which shave off the corners and edges of the bounding box) approximate the primitives well enough that we do not increase our query time that significantly. However, in exchange for the loss of accuracy of our approximations (and consequently the increase in the number of bounding volume overlap tests), we can potentially save up to six comparisons (or three interval overlap tests) during *every* overlap test. These new k-dops also maintain the nice properties of the original ones, in particular, we can easily construct them without using multiplications (see Section 3.4.5). The results when using these new k-dops shall be reported in the following section.

6.3.2 Splitting Rules for Building the Hierarchies

Recall from Section 3.4.6 that each node, ν, of a BV-tree, corresponds to a subset, S_ν, of the original set of input primitives S. For each internal node of the tree, we need to divide the primitives (triangles) into two subsets so as to minimize some function of the sizes of the children. Based upon one of four splitting rules, a plane orthogonal to one of the three coordinate axes was chosen to divide the triangles. Dividing along one of the coordinate axes seems very natural when the bounding volumes we are using are axis-aligned bounding boxes (AABBs), as the boxes can be thought of as bounding extents (or slabs) along each of the x, y, and z directions.

As we have highlighted in Section 3.3, k-dops can be considered a simple generalization of AABBs and thus can be thought of as extents along the $k/2$ directions which define our k-dops. Therefore, we proposed earlier that it seemed natural to consider using planes which are orthogonal to each of the $k/2$ directions when splitting the triangles into two subsets. By doing so, we hope to compute BV-trees which are tighter approximations to the set of primitives, and consequently, perform CD queries more quickly.

As the new choices of k-dops (discussed in the previous section) do not use the x, y, or z axes as part of their $k/2$ directions, they have been implemented to always split along any of the $k/2$ directions. However, for the 14-dop, 18-dop, and 26-dop, we can split along the three coordinate axes as before, or split along all seven, nine, or 13 directions, respectively. Of course, this generalization will require more time for preprocessing.

In the following table, we have re-run the five datasets on our new set of 10 k-dop methods. These methods include using: 6-dops, 8-dops, 12-dops, 14-dops (splitting along three directions), 14-dops (splitting along seven directions), 18-dops (splitting along three directions), 18-dops (splitting along nine directions), 20-dops, 26-dops (splitting along three directions), and 26-dops (splitting along 13 directions). In the parentheses in the first column of the table is the number of splitting directions that the method is using. This number will either be three, for the original splitting method along the x, y, or z-axis, or $k/2$, for the newer methods.

	Pipes	Torus	747	Swept	Interior	Avg. Rank
6-dop (3)	0.357	0.263	1.415	0.798	3.627	8.4
8-dop (4)	13.938	0.711	1.002	0.644	13.311	9.6
12-dop (6)	0.556	0.190	0.348	0.234	3.639	7.0
14-dop (3)	0.283	0.174	0.633	0.143	2.289	4.8
14-dop (7)	0.334	0.173	0.621	0.144	2.416	5.0
18-dop (3)	0.282	0.178	0.308	0.105	2.245	2.6
18-dop (9)	0.353	0.172	0.291	0.107	2.295	2.8
20-dop (10)	0.754	0.259	0.370	0.142	3.271	7.0
26-dop (3)	0.365	0.198	0.342	0.082	2.128	3.6
26-dop (13)	0.426	0.215	0.318	0.087	2.214	4.2

Table 26: Average CD query times (in ms) for our k-dop methods. Next to each method in parentheses is the number of splitting directions used for constructing the BV-trees. For all of the methods, we used the Splatter splitting rule and the mean was the splitting point.

If we compare the original 14-dop, 18-dop, and 26-dop methods versus the new modified versions (which use $k/2$ splitting planes), we see that the added flexibility pays off only occasionally. In particular, the new method is faster than the original in just five of the 15 runs. Moreover, when this method is faster, it is not that much of an improvement. The five speed-ups consist of an improvement of 1, 2, 3, 6, and 7 percentage points, on the 14-dop Torus, 14-dop 747, 18-dop Torus, 18-dop 747, and 26-dop 747 runs, respectively. As mentioned earlier, this modification requires additional preprocessing time. As it turns out, the occasional savings we get from using this new idea comes at the expense of increasing the preprocessing time by a minimum of 17%, a maximum of 60%, and an average of 30%.

In generalizing our splitting rule to consider splitting along any of the $k/2$ directions, we were hoping to compute BV-trees which were better (tighter) than the previous ones. However, we have found that, in general, this is not the case. Recall that in Table 9, we tabulated the total volume occupied by our BV-trees (using the 18-dop method) for all of the splitting rules and splitting points. Table 27 highlights the total volume occupied by the BV-trees generated by our 18-dop method, when the splitting rule (Splatter) was using the original three directions and when using the nine directions, as described above. We can see that for all of the datasets, the new method produced hierarchies which occupied a greater amount of total volume than the original methods. Thus, we are not terribly surprised that the new methods also resulted in slower CD query times due to the poorer approximations that are resulting.

Together with the average collision detection query times, the average rank of each method is also provided in Table 26. For each dataset, the fastest

	Pipes	Torus	747	Swept	Interior
18-dop (3)	5.652836	0.617350	13.262544	27.722404	43,193,251
18-dop (9)	6.771913	0.625194	14.022255	29.392082	45,441,778

Table 27: Total volume occupied by our BV-trees when constructed using the 18-dop method. Next to each method in parentheses is the number of splitting directions used for constructing the BV-trees. For both of the methods, we used the Splatter splitting rule and the mean was the splitting point.

method was ranked 1 and the slowest was ranked 10. There were no ties in the runs, so we have simply ranked the methods in the order in which they have finished without worrying about methods that are very close together. After summing up the rankings for all of the datasets, we determined the average rank of each method and included it in the final column of the table. The 18-dop method continues to be our best method, with the original approach of using only the three coordinate axes for splitting the nodes being the fastest on average. The 26-dop methods were next, followed by the 14-dop methods. The 12-dop and 20-dop methods were the next in line, while the 6-dop and 8-dop pulled up the rear.

On average, when comparing the three versus $k/2$ splitting directions, the three direction methods were always faster. We have, therefore, concluded that, in general, this option is not worthwhile due to the significant increase in preprocessing time, which provided occasional, at best, improvement in query times. In addition, we find that new k-dops (8-dop, 12-dop, and 20-dop) which were presented in the previous section, are also not of great interest in performing CD queries. As the results have shown, these three new bounding volumes performed very poorly, managing only to outperform the axis-aligned bounding box (6-dop). Moreover, the 8-dop appears to be the worst possible choice of k-dop, as it finished last in nearly all of the runs, and in some cases, by a huge margin. The poor performance of the 8-dop method is easily explained as it uses bounding volumes that are essentially worst possible with respect to objects that are nearly axis-aligned, as in the “Pipes” and “Interior” datasets. This also points to the sensitivity to orientation of the model, when k is small.

6.3.3 Improved Selection of the Threshold τ

In Section 4.4, we discussed the importance of the threshold τ which is used to terminate the construction of our flying BV-trees. Until recently, we had used a threshold $\tau = 40$ to determine when a node was to be distinguished as a leaf of the BV-tree. That is, once the number of primitives (triangles) associated with a node falls below this level, we consider this node a leaf of the tree. This threshold was chosen because it worked well on average for a large variety of datasets. This choice, however, was made fairly early on in the design of our data structures and algorithms. Therefore, we now revisit this choice in the hopes of increasing the average CD query time for a large percentage of our datasets.

For each of the five datasets presented in Chapter 5, we have varied the threshold τ for constructing the BV-trees of the flying objects. The values of τ include one, and then all multiples of five up to and including 150. The results are presented in Tables 29–32. We should point out that some of the entries of the tables are empty; there are a few reasons for this. The first is that some of the runs were unnecessary due to the size of the flying objects. For example, the Swept dataset has a flying object which has only 36 triangles, thus for all values of $\tau \geq 36$, the same flying BV-tree will be constructed and the same query time should result. Other entries in the tables are absent simply because we grew impatient. In particular, for the Pipes dataset, as the value of k increased and the value of τ decreased, the average CD query time increased dramatically. Thus, we ended up terminating a small number of these runs (typically for $\tau = 1$) for the Pipes dataset. We should also point out that as the threshold is decreased, the amount of memory required also increases considerably. Thus, this was also a consideration for the Pipes dataset since it was, by far, the largest flying object. Lastly, some of the entries for the Interior dataset are empty because we have already discovered the values of τ for which the minimum query times are achieved. Thus, as we increased the threshold, the times continued to increase and so we did not bother running this dataset for the values of τ between 105 and 150.

As a reminder, the number of triangles in the flying objects of the five datasets are shown in Table 28.

Dataset	Pipes	Torus	747	Swept	Interior
Object Size (no. tri.)	143,690	20,000	14,646	36	404

Table 28: Number of triangles in the flying objects of the five datasets.

Using Tables 29–32, we can determine for each particular dataset (input models *and* flight path) and for each value of k, which threshold τ will provide the fastest CD query times. Unfortunately, there are very few instances in which we have the luxury of performing all of these tests to determine the best values of our parameters. One instance of this situation might be a

Threshold	Pipes	Torus	747	Swept	Interior
150	0.512	0.349	1.196		
145	0.504	0.345	1.190		
140	0.505	0.334	1.191		
135	0.497	0.335	1.193		
130	0.477	0.320	1.191		
125	0.470	0.317	1.153		
120	0.474	0.311	1.142		
115	0.465	0.307	1.119		
110	0.451	0.311	1.121		
105	0.444	0.309	1.107		
100	0.435	0.309	1.083		3.710
95	0.420	0.302	1.083		3.547
90	0.416	0.300	1.073		3.590
85	0.403	0.308	1.070		3.555
80	0.406	0.297	1.068		3.577
75	0.403	0.274	1.069		3.541
70	0.398	0.266	1.069		3.553
65	0.384	0.254	1.070		3.535
60	0.390	0.264	1.073		3.536
55	0.376	0.255	1.092		3.496
50	0.367	0.255	1.082		3.119
45	0.355	0.244	1.088		3.003
40	0.354	0.242	1.101	0.572	2.957
35	0.360	0.234	1.109	0.298	2.942
30	0.356	0.236	1.115	0.298	2.915
25	0.346	0.234	1.127	0.225	2.669
20	0.355	0.223	1.173	0.225	2.495
15	0.370	0.235	1.248	0.219	2.436
10	0.383	0.244	1.393	0.219	2.446
5	0.428	0.251	1.798	0.261	2.599
1	0.681	0.351	3.861	0.396	3.135

Table 29: Average CD time (in ms), for our 6-dop method, for various threshold values.

Threshold	Pipes	Torus	747	Swept	Interior
150	0.344	0.205	0.524		
145	0.340	0.206	0.534		
140	0.336	0.192	0.527		
135	0.340	0.193	0.525		
130	0.330	0.194	0.531		
125	0.326	0.188	0.517		
120	0.326	0.188	0.514		
115	0.323	0.184	0.504		
110	0.330	0.192	0.501		
105	0.330	0.189	0.501		
100	0.309	0.196	0.493		2.360
95	0.297	0.179	0.497		2.297
90	0.307	0.185	0.502		2.307
85	0.304	0.183	0.500		2.292
80	0.312	0.180	0.495		2.317
75	0.306	0.177	0.498		2.303
70	0.308	0.178	0.500		2.303
65	0.296	0.172	0.508		2.337
60	0.306	0.178	0.506		2.315
55	0.303	0.175	0.504		2.309
50	0.314	0.170	0.513		2.059
45	0.310	0.176	0.522		2.001
40	0.292	0.169	0.524	0.112	1.986
35	0.300	0.170	0.541	0.079	1.990
30	0.308	0.171	0.561	0.079	1.962
25	0.322	0.176	0.583	0.068	1.851
20	0.323	0.175	0.622	0.069	1.759
15	0.336	0.176	0.691	0.068	1.744
10	0.354	0.191	0.802	0.069	1.776
5	0.431	0.226	1.144	0.083	1.933
1		0.364	2.795	0.135	2.359

Table 30: Average CD time (in ms), for our 14-dop method for various threshold values.

Threshold	Pipes	Torus	747	Swept	Interior
150	0.333	0.191	0.292		
145	0.335	0.192	0.292		
140	0.331	0.193	0.293		
135	0.319	0.188	0.291		
130	0.317	0.192	0.299		
125	0.329	0.203	0.283		
120	0.322	0.186	0.284		
115	0.307	0.190	0.278		
110	0.321	0.188	0.274		
105	0.315	0.190	0.272		
100	0.314	0.182	0.272		2.288
95	0.318	0.178	0.278		2.229
90	0.309	0.185	0.277		2.225
85	0.320	0.196	0.266		2.238
80	0.304	0.185	0.269		2.212
75	0.314	0.186	0.268		2.245
70	0.309	0.181	0.270		2.231
65	0.298	0.179	0.268		2.236
60	0.300	0.175	0.268		2.216
55	0.296	0.180	0.274		2.210
50	0.309	0.176	0.271		2.027
45	0.314	0.178	0.275		1.973
40	0.320	0.178	0.278	0.083	1.945
35	0.310	0.180	0.282	0.061	1.961
30	0.308	0.180	0.291	0.061	1.933
25	0.322	0.185	0.305	0.056	1.814
20	0.331	0.195	0.318	0.054	1.736
15	0.342	0.205	0.346	0.056	1.731
10	0.370	0.206	0.391	0.056	1.773
5	0.439	0.253	0.532	0.068	1.961
1		0.428	1.252	0.103	2.542

Table 31: Average CD time (in ms), for our 18-dop method for various threshold values.

Threshold	Pipes	Torus	747	Swept	Interior
150	0.443	0.225	0.344		
145	0.457	0.229	0.343		
140	0.450	0.235	0.343		
135	0.446	0.228	0.339		
130	0.446	0.233	0.346		
125	0.448	0.225	0.340		
120	0.435	0.220	0.337		
115	0.446	0.232	0.340		
110	0.437	0.214	0.343		
105	0.453	0.234	0.344		
100	0.461	0.219	0.335		2.158
95	0.435	0.224	0.335		2.121
90	0.434	0.221	0.339		2.123
85	0.441	0.227	0.338		2.123
80	0.455	0.232	0.341		2.126
75	0.465	0.223	0.343		2.149
70	0.456	0.235	0.341		2.117
65	0.460	0.236	0.339		2.121
60	0.464	0.238	0.353		2.120
55	0.453	0.232	0.355		2.128
50	0.479	0.241	0.354		1.994
45	0.469	0.243	0.367		1.955
40	0.468	0.239	0.372	0.072	1.971
35	0.479	0.250	0.386	0.066	1.954
30	0.481	0.248	0.397	0.066	1.958
25	0.509	0.256	0.415	0.065	1.895
20	0.506	0.257	0.442	0.064	1.860
15	0.537	0.285	0.485	0.066	1.919
10	0.584	0.316	0.567	0.066	2.005
5		0.381	0.796	0.074	2.347
1		0.638	1.804	0.100	3.443

Table 32: Average CD time (in ms), for our 26-dop method for various threshold values.

demonstration which is often given using the same set of input primitives. If the environment, the flying object, and the positions of the flying object are not going to change for the particular demo from one run to the next, then spending a lot of time to initially adjust the parameters for optimal performance may be reasonable. Typically, however, a user will want to move a flying object within an environment, but he (or she) will often change the objects used as input. In addition, the time required to perform a large batch of experiments is usually not available. Therefore, we need to have the threshold value predetermined, using only the available information at run-time (e.g., the number of primitives in the input).

If we consider the query times in these tables, we notice that for the datasets with the smaller flying objects (Swept and Interior) we see that a smaller threshold, between 15 or 20, seems to be the best choice, regardless of our value of k. However, for the datasets with the larger flying objects, the threshold should be quite a bit larger. Unfortunately, there is not a clear choice as to which value we should use for the larger datasets. Part of the problem is that since the flying objects are larger, we necessarily will have larger (deeper) flying hierarchies, and consequently, we will perform more bounding volume overlap tests (N_v) and update more nodes (N_u) in the flying BV-tree. However, as we pointed out in Section 3.3, the cost of these operations is directly related to our choice of k. The cost to perform an overlap test is $C_v = O(k)$, and to update a node requires (asymptotically) more time as $C_u = O(k^2)$.

We can see the effect of our choice of k clearly, if we consider the average query times for the Torus dataset in the four tables. For our 6-dop method, the ideal threshold value would probably fall within the range of 15–35. For the

14-dop method, this range increases from 15–75. As our value of k increases to 18 and 26, the best choice for τ also increases and would likely be between 25–100 and 75–150, respectively. These ranges were determined by taking the fastest CD query time overall for each value of k, and then determining the range of values of τ such that the query times were within five percent of the fastest CD time. From these ranges, we see that for larger values of k, the thresholds which perform the best, are also larger. What is happening here is that the BV-tree built for the 26-dop method cannot be constructed as deeply as, say, the 6-dop method, because the operations which we are performing are much more expensive for the 26-dop method. Thus, it pays off to terminate the construction earlier and then begin to perform the primitive-primitive tests.

To answer the question of what value should τ be, we are considering the following simple, and perhaps natural, idea: Let the threshold τ be determined by the complexity (or size) of the flying object. For flying objects which are relatively small, say less than 1,000 triangles, let $\tau = 20$, regardless of the value of k. For all objects larger than 1,000 triangles, we will vary τ with the value of k. From the four tables, we conclude that on average, the value of τ should be 40, 40, 60, and 100 for the 6-dop, 14-dop, 18-dop, and 26-dop methods respectively. Now, these thresholds certainly will *not* provide the best query times in all cases, but on average they seem to work pretty well. In addition, we would also like to emphasize that of the five datasets used here, only two of them were produced by ourselves. Two datasets were provided by another group of researchers which also works on collision detection problems, and the last was produced by an independent company. Thus, even though the datasets have come from several different sources, the same threshold values seem to work fairly well (on average).

	Pipes	Torus	747	Swept	Interior
6-dop	0.0	0.0	0.0	60.7	15.6
14-dop	0.0	0.0	0.0	38.4	11.4
18-dop	6.3	1.7	3.6	34.9	10.7
26-dop	1.5	8.4	9.9	11.1	5.6

Table 33: Percentage improvement for our four k-dop methods, when using the newer criterion for selecting the threshold τ.

Table 33 highlights the percentage gained for our four k-dop methods when we utilize these newer values of τ. The percentage change is compared to using our original threshold value of 40 (as reported in these tables). For example, in Table 31, using the original threshold value of 40, we would have achieved average CD query times of 0.320, 0.178, 0.278, 0.083, and 1.945 milliseconds, for each of the five datasets. By using our new approach, we would get 0.300, 0.175, 0.268, 0.054, and 1.736 milliseconds.

A Potential Weakness

This section highlights a potential weakness in our overall approach. It seems possible that our value of τ may work very well on some datasets, but work very poorly on others. Thus, some people may worry about having to determine which threshold values will work best for their particular application. This is an understandable concern, but one which we are not overly troubled by due to the results in Tables 29–32.

Table 34 highlights for each our of bounding volumes the value of τ which

	Pipes	Torus	747	Swept	Interior
6-dop	25	20	80	10,15	15
14-dop	40	40	100	15,25	15
18-dop	65	60	85	20	15
26-dop	90	110	95,100	20	20

Table 34: The value of τ which produced the fastest average collision detection times in Tables 29–32.

produced the fastest average CD query times for each of our five standard datasets. Ideally, we would have known beforehand which values of τ to choose for each run, however, as we have previously mentioned, this is usually not the case. Therefore, it seems that we must hope that the value of τ that we have chosen to use is close to the ideal value (as reported in Table 34) and produces query times close to those generated by the ideal value. However, after a closer examination of Tables 29–32, we see that we have a great deal more flexibility in making our choice of τ than we would have at first believed.

	Pipes	Torus	747	Swept	Interior
6-dop	20–45	15–35	30–115	10–25	10–20
14-dop	30–95	15–75	50–125	10–25	10–20
18-dop	30–115	25–100	40–115	10–25	10–25
26-dop	55–150	75–150	65–150	10–35	15–35

Table 35: The range of values of τ within five percent of the best CD query time.

Beginning with the fastest query times (as provided by the values of τ in Table 34) we have generated the ranges of τ which produced query times within five percent of the best query time. These ranges are reported in Table 35. We can see from this table, that we do in fact have a great deal of flexibility in choosing the value of τ. For instance, we note that for our 18-dop method, any threshold value between 25 and 100 (for the Torus dataset) produces query times that are within five percent of the fastest query time. Our choice is just as large for the Pipes and 747 datasets, as the range of τ is between 30–115 and 40–115, respectively. Similar results can be seen for wide ranges of threshold values for the other methods on the large datasets too. It is not that surprising that the range of threshold values is not as large for the smaller flying objects, but we can see that we again have a reasonable range of values within which our chosen value of τ can fall.

6.3.4 An Oscillating BV-tree

When constructing our BV-trees, we need to split a node of the hierarchy into two (or more) children. As described in Section 3.4.6, there were two choices we need to make when performing such a split:

- Along which of the three coordinate axes would we split?
- At which point along this axis would we split?

We described in detail several of our methods for making these choices, and then concluded which of these worked the best by seeing which of them performed the best on a wide variety of datasets. Having concluded to use the “mean” centroid value as the splitting point and the “splatter” splitting rule to determine which axis to split along, we could have ended our investigation into constructing effective hierarchies there. However, one worry that we had was that we might be making a bad decision when splitting the node into its two children, which would then effect the quality of the rest of the tree. That is, we wondered how often our default methods worked poorly when a much better split, yielding tighter fitting bounding volumes on the children, could have been done. We knew that on average our methods worked well, but perhaps given a little more preprocessing time, the methods could be improved. This fear about making a poor decision for one particular split has lead to what we refer to as the "oscillating BV-tree" (OBVT).

To explain how our OBVT works, let us assume that we are splitting a node ν of a BV-tree. Now, the goal is to split the set S of primitives (i.e., triangles) associated with ν into two subsets which will determine the children ν_L and ν_R of ν. The idea behind the OBVT is to temporarily split the node as we would normally have done into two children. These two children would then be split into the four grandchildren of ν. The grandchildren would then be split into the eight great-grandchildren of node ν, and so on. At some point along these splits (as determined by a parameter), we would stop the splitting process and then "regroup" or "merge" the resulting descendants of ν into the two actual children of node ν.

To clarify exactly what we mean, let us assume for the moment, that we stopped the splitting process after the four grandchildren nodes of ν were created (using our original splitting methods). We shall refer to the grandchildren by number: 1, 2, 3 and 4. Now, associated with each of these descendants is a set of primitives and a k-dop (which is used to approximate the primitives). The union of all of the primitives associated with the grandchildren, of course, will exactly equal S, the set of primitives associated with ν. Given these four nodes, we want to merge them together so that we wind up with exactly two sets of primitives which will then determine the actual children of node ν. Given four items, we can pair them up in exactly seven ways:

1. (1) (2, 3, 4)
2. (2) (1, 3, 4)
3. (3) (1, 2, 4)
4. (4) (1, 2, 3)
5. (1, 2) (3, 4)
6. (1, 3) (2, 4)
7. (1, 4) (2, 3)

To determine which of these pairings is the best, we perform the mergings for all of them, and then apply one of our splitting rules (longest side, splatter, min sum, min max) as the objective function. In general, we have found that the "min sum" rule works the best here, as it did in Table 20.

The rationale behind this approach is that by going down several levels in the hierarchy, and then regrouping the descendants into the (actual) two children of a node, we can essentially erase the poor decision that was made (if one was made at all) and come up with a good split of the original node. It was interesting to note however, that in most cases, the regrouping resulted in exactly the same split as we would have originally made using our current splitting techniques.

In general, if we were to split a node until we stopped with m descendants, there would be $2^{m-1} - 1$ possible pairings, since each of the two resulting sets must have at least one primitive, and the order of the sets is irrelevant. To clarify, for the above situation in which we stopped splitting after the four grandchildren were generated, we do not consider (2, 3) (1, 4) to be different from the last item listed.

Preliminary Results Our results when using the OBVT are encouraging, but *not* because this new method helps us produce better quality (e.g., reduced total volume) hierarchies and faster CD query times. In fact, using this method does not help us at all in these regards. We have actually *increased* our query times when using the OBVT as opposed to our original splitting methods. What is encouraging about this method is that our original splitting techniques must be working very well, if they are able to produce hierarchies that generate faster query times than this more complicated preprocessing step. At this point in time, we have not found a single instance in which using the OBVT has generated faster query times.

To be honest, we found these result somewhat surprising. After all, by using a great deal more preprocessing time (often two to ten times more than the original methods), one would expect that the hierarchies would be better, as would the resulting query times. We began looking into why this was the case and we soon discovered that our splitting rules were not complicated enough. What was happening was that occasionally the best merging of the descendants of a node would isolate a single descendant from all of the rest, and we would get a very unbalanced split in terms of the number of primitives in each of the resulting children. This type of split tends to lead to fairly deep hierarchies, an increase in the number of bounding volume overlap tests, and an increase in the average CD query times.

Having performed these experiments with our OBVT, we feel better about the quality of the hierarchies that our current construction methods produce. Having said that, we now feel that a more complicated splitting rule, which will take into account all of the relevant factors (such as the number of primitives and the sizes of the resulting children), is an interesting area of future research.

6.3.5 Working with Problem Data: Skinny Triangles.

A k-dop certainly yields a much better approximation for long skinny objects than a sphere (or an AABB). However, long and skinny triangles still tend to cause troubles as they will necessarily result in long and skinny k-dops. Most likely, they will also cause significant overlaps between k-dops corresponding to sets of “neighboring” skinny triangles. Unfortunately, skinny triangles do arise in practice: polyhedral approximations of pipes (in particular the Pipes dataset) are well-known for lots of skinny triangles. The situation gets particularly troublesome when our splitting rule (split according to mean centroid) fails to separate “small” triangles from skinny triangles, as this will cause many spurious intersection tests, which could be avoided if the skinny triangles were separated from the small triangles.

One could attempt to address this problem by building several individual hierarchies, taking into account the size of the triangles. For skinny triangles one might also want to take into account the predominant orientation of those triangles. Potentially, at the cost of drastically increased bookkeeping for all the diverse trees, this approach might provide better bounding volumes.

We decided to take a less ambitious and more pragmatic approach. Statistics recorded by our algorithms convinced us that long and skinny triangles are no issue for NC machining applications (see Section 8.1), and that mechanical parts tend to contain only a small number of skinny triangles, if at all. Thus, it seems to be natural to get rid of those few triangles by simply subdividing them appropriately along their long sides. By allowing the overall number of triangles to increase by, say, 10% we have been able to abolish the longest triangles and achieve better BV-trees. For instance, for the "Pipes" example, replacing the longest triangles and adding a total of about 7% more triangles provided us with a speed-up of slightly more than 4% for the average CD check.

6.4 Improving the Usage of the BV-Trees

6.4.1 Ordered Overlap Tests

As we have previously mentioned, the most frequently called routine during a collision detection query will test whether two k-dops overlap. This overlap test, whose cost has been denoted C_v in Equation (2), consists of testing each of the $k/2$ intervals of one k-dop versus the corresponding intervals of the other k-dop. If there exists a direction along which the two intervals (or extents) do not overlap, then we know the k-dops do not overlap; otherwise, we (conservatively) conclude that they do (see Section 4.5). This test is the simple generalization of testing two axis-aligned bounding boxes for overlap.

In Section 4.5, we discussed how the order in which we test these intervals for overlap could have an effect upon the efficiency of this routine. Our approach up to this point has been to order the intervals so that intervals with largely different directions are tested one after the other. Here, we investigate an alternative ordering scheme which may decrease the average time spent in the overlap routine.

The idea which we are presently investigating involves ordering the intervals, or bounding slabs as they are also called, from thinnest to thickest. The rationale behind this is that the thinnest slab will tend to prune away more k-dops than the thicker ones, and thus the overlap routine will be terminated earlier. The question which immediately arises is whether to order the intervals with respect to the node of the flying object hierarchy, or with respect to the node of the environment hierarchy. Now, as the flying object rotates, the k-dops in the flying BV-tree are also rotated. Thus, there is not a *fixed* order of intervals that we can use: the widths of the slabs are constantly changing for these nodes. Thus, we would have to recompute this ordering for each node of the flying BV-tree prior to its traversing the environment hierarchy. However, since the environment is static, we can precompute this ordering once (as part of the preprocessing) and then use this ordering for the overlap tests. This ordering (k bytes) will need to be stored within each node of the environment hierarchy, thus an additional $2k$ bytes would be required for each environment triangle.

We have implemented this option and tested its effect upon the five datasets discussed in Chapter 5. The results (for our 26-dop method) are shown in Table 36. In the first three rows of the table, we have recorded the number of

	Pipes	Torus	747	Swept	Interior
Overlap Calls	170,439	78,259	308,386	47,389	993,427
Num. Overlaps	57,402	27,090	98,122	6,654	445,428
Percentage	33.7	34.6	31.8	14.0	44.8
Original Overlap					
Total Comparisons	2,123,816	1,027,002	3,723,089	439,804	14,060,828
Comp. On Misses	631,364	322,662	1,171,917	266,800	2,479,700
Avg. Comp./Miss	5.6	6.3	5.6	6.6	4.5
Ordered Overlap					
Total Comparisons	2,052,217	931,279	3,369,363	350,605	13,535,330
Comp. On Misses	559,765	226,939	818,191	177,601	1,954,202
Avg. Comp./Miss	5.0	4.4	3.9	4.4	3.6
Percent Reduction					
Total Comparisons	3.4	9.3	9.5	20.3	3.7
Avg. Comp./Miss	11.4	29.6	30.2	33.4	21.2

Table 36: Number of comparisons required when testing two 26-dops for overlap. By ordering the intervals of the environment k-dop from thinnest to thickest, we are able to reduce the average number of comparisons needed.

pairs of bounding volumes, N_v, tested for overlap, the number of these tests that ended in an "overlap" result, and the corresponding percentage of tests that were overlaps. The "Interior" dataset resulted in the largest percent of overlap results, while the "Swept" dataset had the fewest. The reason we point out these numbers is that (as previously discussed in Section 4.5) if the two k-dops do overlap, then any extra work we do (in ordering the intervals) is "wasted" in that the order is irrelevant, since all intervals overlap.

In the following six rows of the table we show, for our original overlap routine and the new ordered overlap routine, the total number of comparisons that are performed for all of the overlap tests, the number of comparisons for the tests that result in disjoint k-dops (or "misses"), and the average number of comparisons per overlap call which results in a miss. We use the term "comparison" to mean an inequality test, either a "greater than" or "less than" comparison between two floating-point numbers. Each interval test consists of two such comparisons, thus, an overlap test between two k-dops can result in at most k comparisons.

The final two rows of the table highlight the effect of our modified overlap routine. Here, we see that the total number of comparisons was reduced by a minimum of 3.4% and a maximum of 20.3%. The average number of comparisons per "miss" was also reduced by an average of 25%.

These savings are quite reasonable and seem to suggest that this ordering scheme should be included within our current implementation. However, in using a special ordering of intervals for each node of the environment hierarchy, we need to perform some additional work (i.e., indexing an array) to determine this order. Thus, we must weigh the savings in the number of interval comparisons versus this additional overhead in time, as well as space. Ignoring the added space component, we calculated the average CD query time when using this modified routine and found that the query times were typically *slower*, by a few percentage points. This was very disappointing; however, although the current implementation of the new routine does not seem to pay off overall, we still maintain some hope that with more time and effort (and perhaps some new clever ideas) this idea may be worth while in the end.

6.4.2 Truncating the Overlap Test

One additional modification which we have also looked into was *truncating* the overlap test after a fixed number, M, of intervals tests were made. This assumes, of course, that we are using the more sophisticated ordering scheme just described. The rationale behind this approach is that if two k-dops overlap, then all of the $k/2$ intervals of the k-dops will also overlap. Thus, the extra work to determine the order of the intervals for testing is not gaining us anything. However, since the intervals are tested from thinnest to thickest, we are hoping that a large majority of the disjoint k-dops will have been discovered after just a few interval tests. In fact, if we consider the average number of comparisons needed when testing two disjoint k-dops, we see in Table 36 that we typically need only five or six comparisons, or the equivalent of two or three interval overlap test since each of these takes two comparisons. This seems to suggest that if we let $M = 6$, this technique may work fairly well.

In making this change, we are reducing the number of interval overlap tests from $k/2$ to M for all of the pairs of bounding volumes tested for overlap that actually do overlap. In exchange for this reduction, we are going to perform more bounding volume overlap tests since nodes that were originally found to be disjoint *after* the M^{th} interval test, are now reported as overlapping, since we terminate the routine here and have not discovered that the k-dops are disjoint. Unfortunately, this modification had a similar effect as the previous one. The average query time actually increased for all of the datasets. In the end, the number of additional bounding volume overlap tests, N_v, outweighed the savings we received for the overlapping nodes. Perhaps with more effort and new ideas, this approach could still be salvaged. It seems likely that if we knew that a large number of the bounding volume overlap tests actually resulted in overlaps, then a technique similar to this may actually reduce the average CD query time.

6.4.3 Front Tracking for Temporal Coherence

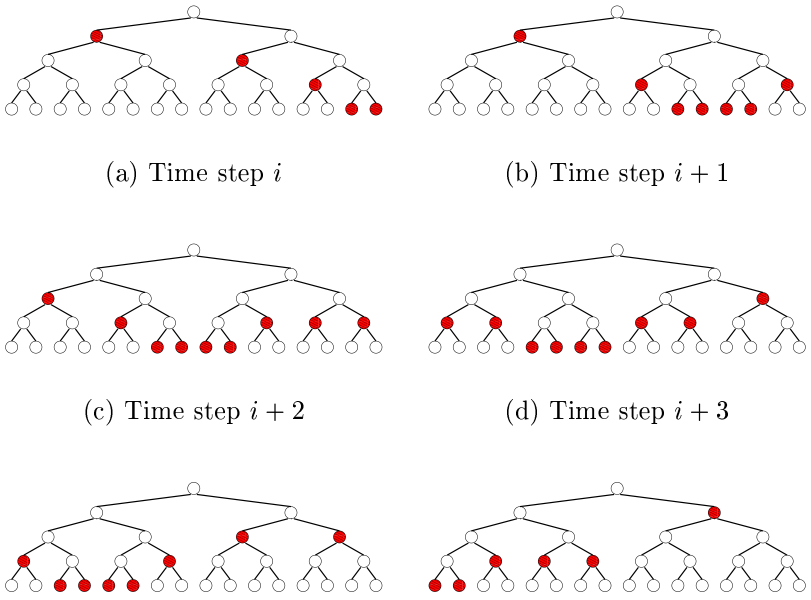
Given BV-trees to approximate the flying object and the environment, we can traverse the two hierarchies to determine if the flying object, in its current position, collides with any of the models in the environment. Our algorithm for traversing these hierarchies, originally discussed in Section 4.3, was to traverse the environment hierarchy with the root of the flying BV-tree. For each leaf that is reached, we take the geometric primitive associated with that leaf and traverse the flying BV-tree. If a leaf is reached in this traversal, we need to test whether the actual geometric primitives (e.g., triangles) intersect.

We originally applied our tree-traversal algorithm naively: For each position of the flying object we checked the two BV-trees to see if there were any collisions. Although this approach will correctly determine all contacts, it was wasteful in that we began "from scratch" at each step, often traversing the environment hierarchy to find that the nodes which were disjoint with the root of the flying BV-tree at the previous position, were again disjoint with the root at the current position.

The flying object can be expected not to change its position very drastically, from one position to the next. In particular, if our tree traversal determines that the root of the flying BV-tree overlaps with a leaf of the environment BV-tree at some given instant of time, then it is quite likely that the root of the flying BV-tree will also overlap this leaf or some leaf representing a geometric primitive close by within a short period of time after this instant. Similarly, a node of the environment BV-tree that did not overlap with the root node of the flying BV-tree has a fair chance not to overlap with it for the next few instants of time too.

In order to exploit this temporal coherence and thus avoiding scanning from the root down to the same nodes of the environment BV-tree repeatedly, we maintain a "front" by keeping track of the root node of the flying BV-tree within the environment BV-tree: Assume that the root ν of the flying BV-tree gets pushed down (traverses) the environment BV-tree. The root ν may get pushed down to leaves in some parts of the environment BV-tree, whereas it may stop at some higher nodes in other parts of the environment BV-tree. The front maintains a list, ordered from left to right, of those nodes of the environment BV-tree where a disjointness with ν was first determined together with a list of the leaves that overlapped with ν. Thus, the next CD query can be started at the nodes of the front, rather than at the root of the environment BV-tree.

Figure 11(a) provides an illustration of such a front. In this particular example, we are looking at a tree which represents the environment BV-tree. There are 16 leaves in the tree, each of which contains one triangle. The five colored nodes represent our front for the current position of the flying object. We can see that the root ν of the flying object overlaps two of the leaves of the environment, while it does not overlap the left child of the root of the environment BV-tree. This node's disjointness from ν is a witness to our not needing to explore this branch of the environment BV-tree any further.



(a) Time step i

(b) Time step $i+1$

(c) Time step $i+2$

(d) Time step $i+3$

(e) Time step $i+4$

(f) Time step $i+5$

Figure 11: Front tracking: As the flying object moves in the environment, we maintain a "front" of (colored) nodes in the environment BV-tree. Colored internal nodes are known to be disjoint from the root of the flying BV-tree.

Of course, we need to update the front accordingly as a flying object keeps moving. If a node that was previously disjoint from ν now overlaps with ν, then we "drop the front" by replacing this node in the front with its two children, and by recursively applying this scheme to these children. An example of this is the left child of the root of the environment BV-tree. In Figures 11(a) and 11(b), this node does not intersect the root of the flying BV-tree, and is therefore an element of the front. However, at time step $i + 2$, the two nodes now overlap, so we replace this node in the front with the new nodes which determine disjointness. In this particular case, the node is replaced by four of its descendants, two of which are leaves.

In addition to dropping the front, we must also provide a mechanism to "raise" the front, so that it closely monitors the overlap relationship of ν within the environment BV-tree. In particular, when a node μ that previously overlapped ν turns out to be disjoint from it now, we then "raise the front" at μ if the successor of μ in the front is its sibling (μ'), and if μ' also does not overlap ν. In our current implementation, pushing the front up is not done recursively; thus, the front gets raised by only one level at a time.[1] This operation can be seen in Figure 11 as the flying object moves from right to left in the environment. At time step i, the two leaves in the right of the hierarchy are overlapped by the root of the flying BV-tree. At the next time step, however, we find that the nodes no longer overlap with the root, and thus we replace these two front nodes with their parent. As the flying object continues to move, we can see other examples of how the front is propagated upwards in this manner.

[1]Since raising the front requires additional overlap tests, it is not exactly clear, anyway, whether raising the front by more than one level would yield a speed-up.

The benefit of using our front has been in reducing the number of bounding volume overlap tests that are required when testing for contact between the (hierarchies of the) flying object and the environment. The results of our implementation have been quite encouraging. In comparing the total number of bounding volume overlap tests for our "naive" search algorithm (Section 4.3) versus our current search algorithm (which exploits coherence), the number of overlap tests for the experiments reported in Table 18 was reduced by a minimum of 10%, an average of 19%, and a maximum of 31%.

The reduction in the number of overlap tests performed does not come for free, however. There is the overhead of maintaining the front which is now included in the query times. In some instances, where the number of overlap tests was not reduced by very much, the average query times are roughly the same for both search algorithms. There are even cases were using the front is a few percentage points slower than without using it. However, in the majority of the experiments, using the front does pay off, and has lowered the average CD query time by as much as 15%. More often than not, however, the reduction in query time has been on the order of 3–5%. This does not surprise us tremendously, as profiling our code has shown that the percentage of time spent performing the overlap tests is roughly 20–25% of the total query time. Thus, by reducing this 20–25% by 19% (which was our average as indicated above), we should achieve a speed-up of roughly 4%.

In addition to reducing the average CD query time, we are also hopeful that by maintaining a front, we can also reduce the *maximum* query time required for all of the CD queries. As pointed out earlier in Section 4.3 and Section 5.4, bounding the worst-case query time is of great importance for many applications which require time-critical collision detection. We have recently determined the maximum query times for our standard datasets both with and without the front. Please refer to Section 7.2.2 for these results.

The front was implemented as an array of cells, each of which contains a pointer to the corresponding node in the environment BV-tree, together with pointers to the preceding and succeeding cells in the front. Thus, to maintain the front requires 12 bytes for each cell in the array. In the worst case, the size of the front can equal the number of leaves in the environment BV-tree, which in our case equals the number of triangles in the environment. This situation actually occurred in our "concentric sphere" experiments, discussed in Section 5.4. However, in practice, the front never consists of this many nodes. For the experiments reported in this section (and Chapter 5), the maximum size of the front never exceeded 1% of the total number of environment triangles.

Chapter 7

QuickCD

7.1 Introduction

We have presented 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). Chapter 3 introduced our bounding volumes as well as a number of design choices for constructing BV-trees. Our collision detection algorithm was presented in Chapter 4, and we highlighted several important issues for performing efficient queries. A vast collection of experimental results highlighted the success of our method.

There have been numerous options as we described our algorithm and data structure, and we tried to address the practicality of each of them. In Chapter 6, we continued testing these options as well as several new ones. Over the course of our investigation into the collision detection problem, the implementation of our method has evolved into `QuickCD`, our library of collision detection routines. At this point in time, we shall summarize the current state of our implementation.

7.2 Current Implementation Status

The bounding volumes which we use are, of course, k-dops. The values of k which are included in the library are 6, 14, 18, and 26; however, the 18-dop has proven to be the most efficient and is the default method. The k directions which define each of our bounding volumes were defined in Section 3.4.5. Our investigation into additional k-dops (Section 6.3.1) was unsuccessful, and so these are not contained in the current version of our code. However, for completeness, we have included the additional k-dops within our latest round of experiments (Section 7.2.1) to provide further justification of our conclusion.

Our default method of constructing the BV-trees is to use the "splatter" splitting rule, although the other three rules (longest side, min sum, and min max) are also available (Section 3.4.6). Currently, the splitting rule will split each internal node of the hierarchy along either the x, y, or z-axis, as the generalization (Section 6.3.2) to the $k/2$ directions does not appear to be worthwhile at this time. Consequently, this option is not included in the current release of `QuickCD`. As the "median" split point was shown not to produce quality hierarchies, this option is also unavailable in our current implementation. Thus, only the "mean" split point is included. All of our hierarchies are binary trees and they are built in a top-down fashion (Sections 3.4.2 and 3.4.3).

Once the BV-trees have been constructed, we utilize the TraverseTrees algorithm in Section 4.3 to process the collision detection queries. To accelerate the tree traversal algorithm, we have included our "front" implementation

(Section 6.4.3), which takes advantage of temporal coherence as the flying object moves. `QuickCD` provides the user with the option of using the front by passing a parameter in the collision detection routine.

The threshold τ, which determines when a node becomes a leaf of the flying BV-tree, is defined as in Section 6.3.3. For relatively small flying objects (those with fewer than 1,000 triangles), the threshold is 20, regardless of the value of k. For larger objects, the value of τ becomes 40, 40, 60, and 100 respectively, for values of $k = 6$, 14, 18, 26.

Our current attempts to reduce the average time necessary to check two bounding volumes for overlap was unsuccessful (Sections 6.4.1 and 6.4.2). Therefore, we are continuing to use the approach described in Section 4.5.

In Section 4.2, we described two techniques for updating our k-dops as the flying object rotates. We originally chose to use the convex hull and a hill-climbing algorithm (Method I) to update the root node of the flying BV-tree. However, due to recent speed-ups in the alternative method (i.e., tumbling the nodes and recomputing the approximate k-dops), we have decided *not* to use this method, but tumble the root node instead.

Although the hill-climbing technique itself is a more expensive operation than tumbling the node and recomputing the approximate k-dop, it was the tightest possible k-dop for the set of primitives which it was trying to approximate, unlike the "tumbled" node, which was a cruder approximation. Thus, overall, the hill-climbing algorithm was faster due to the fewer overlap tests which were required when traversing the environment BV-tree. By switching to the approximate method, we are slowing down the CD query, but not by very much. In addition, by sacrificing this speed, we are able to free the user from having to precompute the convex hull of each flying object which they wish to use. Overall, the code has been cleaned considerably by removing this option, so we feel that the loss of speed has been a small price to pay for the cleaner, friendlier code.

7.2.1 Recent Experiments

In the following, we describe a recent series of experiments which we have performed to compare our collision detection library to other leading collision detection packages. In addition to the RAPID implementation which utilized OBBs, we also compare `QuickCD` to another CD library which has recently been made publicly available. This new library, SOLID, also builds bounding volume hierarchies to approximate the input objects, however, it uses axis-aligned bounding boxes (AABBs) as its choice of bounding volume.

Software/Hardware. Our latest tests compare `QuickCD` to two leading publicly available codes: RAPID (version 2.01,[38]) and SOLID (version 1.01, [89]). All code was compiled using the GNU C/C++ compiler, with the optimization option "-O2" turned on. In order to ensure validity and portability across different hardware, experiments were conducted on multiple platforms: a Pentium Pro PC (180 MHz, 128 MB), running Red Hat Linux 5.0; an SGI Indigo^2 (195 MHz R10000, 320 MB), running IRIX 6.2; and a Sun Ultra 1 (167 MHz, 320 MB), running Solaris 5.5.1. (Unfortunately, SOLID compiled but crashed with STL-related errors on the SGI.)

Gathering Data. We used the same five standard datasets (see Section 5.4) for this series of experiments. Each flight was executed 10 times (except for some runs of SOLID). The mean CPU times (in ms) per collision detection check are reported, being careful to use the same timing routine and to time the same work being done in all three systems. Times include motion simulation, but exclude rendering. Also, for compatibility with RAPID and SOLID, we ran `QuickCD` in the "report all" mode, detecting *all* intersecting pairs at each location, rather than truncating the test with the first primitive pair found to intersect.

We have tried to conduct fair and meaningful comparisons among the three systems. Thus, we checked carefully that all three codes end up reporting (essentially) the same set of contacts for each dataset/flight. We remind the reader that comparing collision detection timings between sections or chapters is meaningless, due to changes in the code, compilers, and hardware that often occur. The relevance of the timings lies in the relative speed-ups between collision detection systems during a particular experiment and not between experiments. In fact, the timings reported in the following section (for the SGI) are considerably slower than those previously reported. The software has changed slightly from the previous chapters, however, the vast majority of the change in timings is due to a different version of the compiler being used. We can see from timings that both `QuickCD` and RAPID have slowed down considerably due to the new compiler, but again, it is the relative timings that are of greatest importance.

7.2.2 Experimental Results

Preprocessing Time. Our construction algorithms for BV-trees are designed to optimize for fast query processing; thus, some effort must be expended during preprocessing. For the five standard datasets, all preprocessing times were less than 20 seconds (on both the SGI and the Sun), using the default choice of 18-dops.

Comparing to RAPID, our 18-dop method was 10-15% slower (on the Sun) and 15-20% faster (on the SGI) than was RAPID. Comparing to SOLID (which uses simpler hierarchies of AABBs), the 18-dop method was 30-35% slower on the Sun; SOLID it did not execute correctly on the SGI.

We also tested different choices of k-dops. A typical example is the "Pipes" dataset, for which our method (on the Sun) took 10.8, 13.3, 15.8, 14.3, 15.6, 23.1, and 19.1 seconds, for $k = 6, 8, 12, 14, 18, 20, 26$, respectively. RAPID took 14.0 sec; SOLID took 11.7 sec. Our 6-dop time is slightly faster than SOLID, even though 6-dops are AABBs, just as SOLID uses. (On the SGI, our method compares more favorably with RAPID: 15.6 sec for RAPID, vs. 7.98, 11.2, 13.0, 12.2, 13.0, 20.2, 17.1 sec.)

Tables 37 and 38 report all of the preprocessing times for our datasets on the Sun Ultra 1 and the SGI Indigo^2.

CD Query Times. For our five datasets, we ran `QuickCD` with various choices of k-dops; the results are shown in Tables 39 and 40. While the choice of 18-dop is not optimal in every case, it does best on average; extensive experience with other datasets also convinced us to make 18-dops the default for `QuickCD`. We note that the 8-dops, 12-dops, and 20-dops are again very

	Pipes	Torus	747	Swept	Interior
6-dop	10.80	7.07	8.18	2.86	13.27
8-dop	13.35	8.69	10.10	3.53	16.23
12-dop	15.77	10.32	12.19	4.21	19.34
14-dop	14.33	9.26	10.64	3.74	17.59
18-dop	15.62	10.03	11.53	4.08	19.17
20-dop	23.11	15.14	18.32	6.27	28.38
26-dop	19.11	12.25	14.10	4.94	23.57
RAPID	14.00	9.28	9.86	3.66	17.89
SOLID	11.75	7.72	8.55	3.03	13.94

Table 37: Preprocessing times (in seconds) on a Sun Ultra 1

	Pipes	Torus	747	Swept	Interior
6-dop	7.98	5.15	6.38	2.06	9.75
8-dop	11.21	7.25	8.77	2.92	13.63
12-dop	13.05	8.44	10.51	3.41	15.76
14-dop	12.25	7.80	9.30	3.10	14.86
18-dop	12.97	8.27	9.84	3.30	15.86
20-dop	20.26	13.25	16.77	5.38	24.53
26-dop	17.11	10.91	12.73	4.34	20.88
RAPID	15.62	10.06	10.51	3.91	19.65

Table 38: Preprocessing times (in seconds) on an SGI Indigo2.

poor choices for our bounding volume. The 8-dop remains, by far, the worst possible choice. Please refer back to Section 6.3.2 for a discussion about the poor performance of these k-dops, and the 8-dop in particular.

Direct comparisons with RAPID and SOLID appear in Table 41. We see that, with the exception of Pipes, RAPID is always faster than SOLID. (SOLID uses AABBs, which perform well on Pipes, since most pipes in the dataset are axis aligned.) We see that `QuickCD` essentially ties with RAPID on

	Pipes	Torus	747	Swept	Interior
6-dop	0.652	0.495	2.563	0.520	5.761
8-dop	23.687	1.422	1.835	0.823	26.413
12-dop	1.047	0.369	0.652	0.331	6.886
14-dop	0.526	0.306	1.094	0.362	3.667
18-dop	0.495	0.315	0.536	0.169	3.682
20-dop	1.485	0.515	0.680	0.192	6.249
26-dop	0.647	0.363	0.559	0.293	3.767

Table 39: Average CD query times (ms) on a Sun Ultra 1

	Pipes	Torus	747	Swept	Interior
6-dop	0.668	0.495	2.394	0.491	5.524
8-dop	27.045	1.520	1.936	0.872	28.284
12-dop	1.226	0.408	0.713	0.369	7.067
14-dop	0.559	0.326	1.097	0.383	3.568
18-dop	0.599	0.364	0.598	0.196	3.596
20-dop	1.906	0.637	0.835	0.225	6.693
26-dop	0.761	0.425	0.639	0.327	3.660

Table 40: Average CD query times (ms) on an SGI Indigo2.

Torus and 747, and is 1.1 to 9.9 times faster on other datasets (on the Sun). (Note, too, that for Torus and Interior, 14-dops lead to slightly better query times for `QuickCD` on all platforms.) `QuickCD` performs a bit better on the SGI than on the Sun, making the speed differences more visible.

We also performed experiments to see how our method scales over a range of similar datasets, as the complexity n (the number of obstacle triangles) increases. We chose the “747” dataset, since we could control the scene complexity by varying the number of randomly generated tetrahedra. The results, for varying the number of tetrahedra from 10K to 100K (number of triangles 40,000-400,000), are given in Figure 12. We see that RAPID takes up to about

CD Package	Pipes	Torus	747	Swept	Interior
Ultra 1					
QuickCD (18-dop)	0.495	0.315	0.536	0.169	3.682
RAPID (v2_01)	4.887	0.317	0.578	0.728	4.102
SOLID (v1_01)	1.697	2.117	7.587	1.665	1845.763
SGI Indigo2					
QuickCD (18-dop)	0.599	0.364	0.598	0.196	3.596
RAPID (v2_01)	6.548	0.400	0.770	0.969	5.385
PC Pentium Pro					
QuickCD (18-dop)	—	0.435	0.724	0.215	4.837
RAPID (v2_01)	—	0.399	0.759	0.958	5.584
SOLID (v1_01)	—	2.454	9.628	2.239	2171.286

Table 41: Average CD query times (ms) across packages.

a factor of two longer.

Effectiveness of Front Tracking. For the experiments reported in Table 39, the result of using our front tracking was a reduction in the number of BV-BV tests by 8% to 28%, with an average decrease of 18%. After accounting for the small overhead of maintaining the front, the net speed-ups in query times (on the Sun) were 3–20%, with an average of 7.7%. This relatively modest improvement is easily understood: Profiling our code shows that roughly 25% of the CD query time is spent on BV-BV tests, so a 20% decrease in BV-BV tests results in about a 5% speed-up. The maximum improvement came on the Swept dataset, where the obstacles are tightly clustered near the trajectory of the flying object, so exploiting coherence is more effective.

As discussed in Section 6.4.3, another goal of using the front was to reduce the maximum query times for our flights. For the experiments run on the Sun Ultra, we determined (for each of our datasets) the maximum query time for

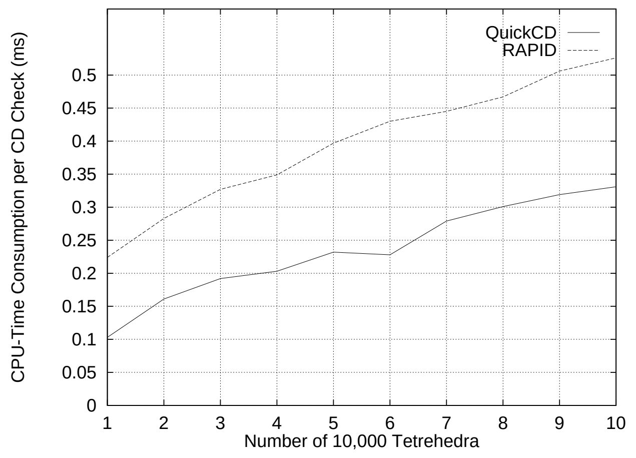


Figure 12: CD query time vs. number of tetrahedra for `QuickCD` and for RAPID. (SGI Indigo2)

any one particular step both with and without the front. By using the front, the maximum query time was reduced for all flights, ranging between 1% and 22%, with an average of 7.9%.

Profiling. For our 18-dop method, we broke down the CD query time into three component times (as described in Section 3.2), for performing: (a) bounding volume overlap tests ($N_v \times C_v$); (b) updates and tumbling of nodes of flying hierarchy ($N_u \times C_u$); and (c) primitive comparisons (triangle-triangle tests) ($N_p \times C_p$). The results are shown in Figure 13. Note that for Interior most of the time is spent on triangle-triangle tests, due to the large number of actual intersections.

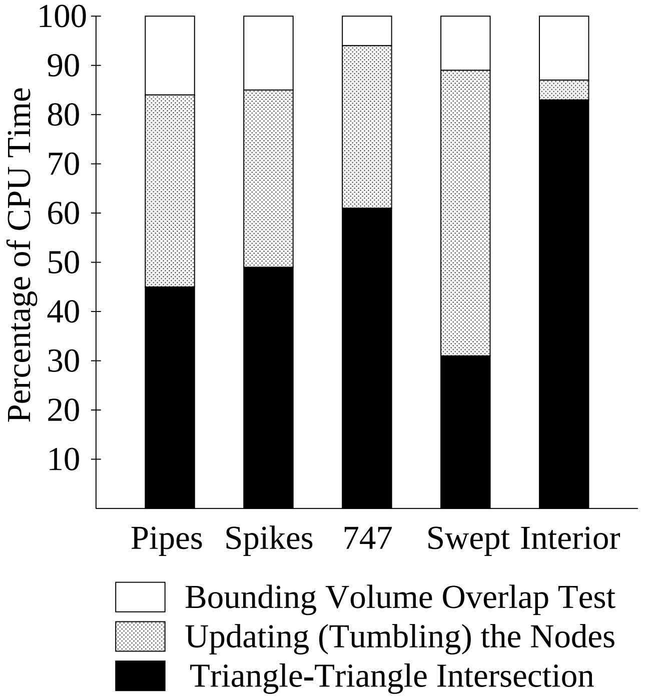


Figure 13: Breakdown of CD query time.

7.3 Newest Results

In the last section of this chapter, we present the most up-to-date experiments of our collision detection library `QuickCD`. In comparing the numbers from Table 42 to those in Table 41, we see a slight decrease in our CD query times for most of the datasets. These improvements have resulted from improving the implementation of our previous ideas, as opposed to inserting new ideas. For the “Swept” dataset, you will notice a much larger decrease which was due to a “bug” in our code. This bug (which effected only the Swept dataset) did not result in the code running incorrectly, but only in running less efficiently

CD Package	Pipes	Torus	747	Swept	Interior
Ultra 1					
QuickCD (18-dop)	0.497	0.305	0.477	0.111	3.240
RAPID (v2_01)	4.887	0.317	0.578	0.728	4.102
SOLID (v1_01)	1.697	2.117	7.587	1.665	1845.763
SGI Indigo^2					
QuickCD (18-dop)	0.594	0.358	0.566	0.121	3.321
RAPID (v2_01)	6.548	0.400	0.770	0.969	5.385

Table 42: Average CD query times (ms) across packages.

than it should have been running. The times for the RAPID and SOLID libraries are exactly the same as in Table 41 because no new releases of these libraries have been made.

We have mentioned several times (see Sections 5.4, 6.4.3, and 7.2.2) throughout this dissertation that the maximum CD query times are also an important factor to consider. Below, as in Section 5.4, we have tabulated the CD query times for each step of the "Pipes" flight, using both our 18-dop method and the RAPID method (since this is our best competition). In looking at Figures 14 and 15, there does not *initially* appear to be much of a difference, as the timings tend to spike at the same moments of the flight, when the flying object comes into contact (or very close proximity) with the environment. However, if we look carefully at the *magnitude* of the CD query times along the y-axis, we can see that the 18-dop method takes considerably less time than does RAPID. The maximum time that our method takes for this particular flight is almost 23 milliseconds, whereas the RAPID system took almost 183 milliseconds for that same CD query. In fact, the RAPID system takes more time on every step of this flight than does our 18-dop method.

To get better idea of how the two systems compare, we have overlayed sections of these two Figures in Figure 16. For clarity, we have only included the first 650 steps of the flight, although if we were to overlay the remaining sections of the graphs, we would see the same relationship. The solid (dotted) line in this Figure represents the 18-dop (RAPID) timings. We can now see clearly from this zoomed-in image, the actually difference in the timings of the two systems for this particular flight.

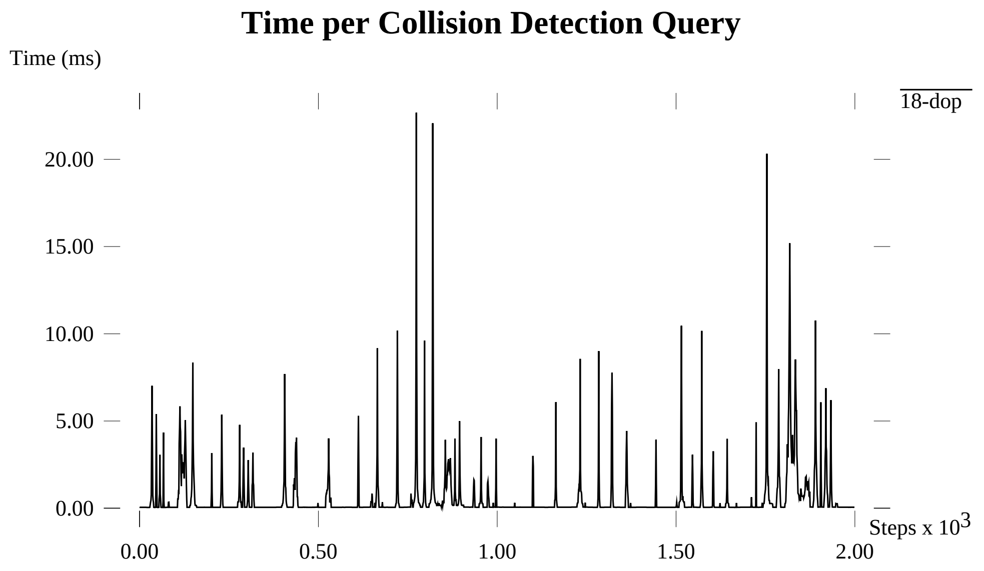


Figure 14: Individual collision detection query times for the "Pipes" dataset using our 18-dops. The maximum query time is roughly 23 milliseconds.

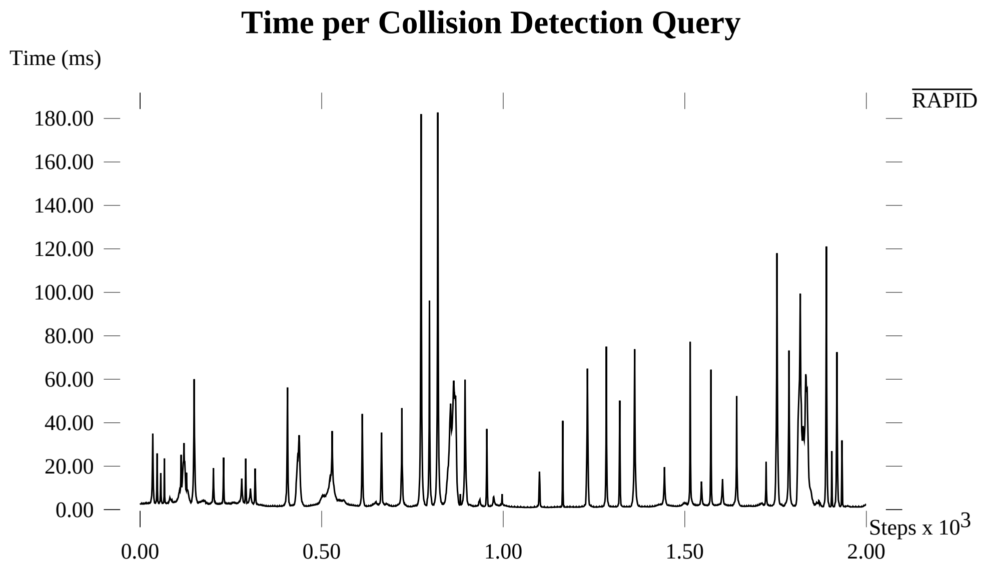


Figure 15: Individual collision detection query times for the "Pipes" dataset using RAPID. The maximum query time is roughly 183 milliseconds.

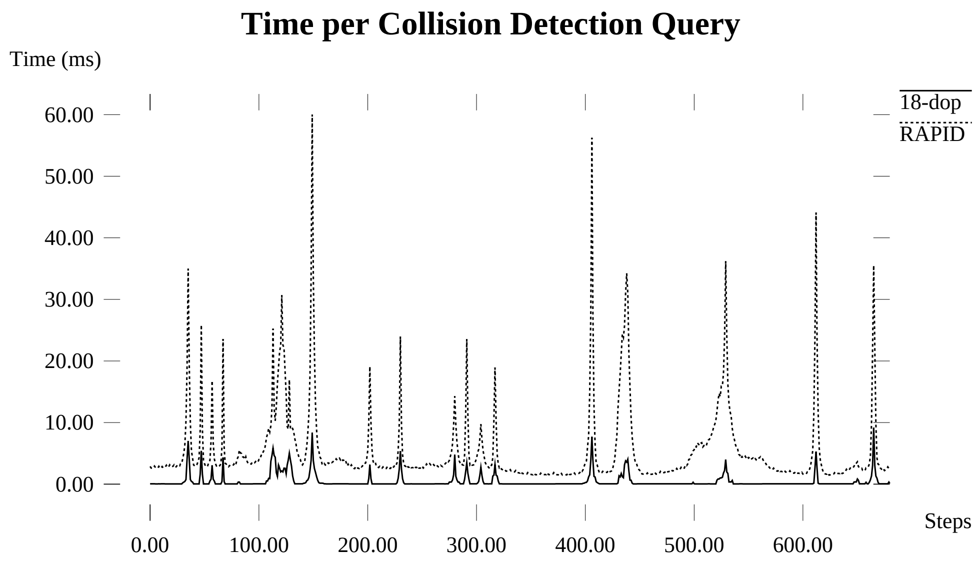


Figure 16: We have overlayed Figure 14 and Figure 15 to highlight the difference in the individual collision detection query times for the "Pipes" dataset using both our 18-dops and RAPID. For clarity, only the first 650 steps are shown.

Chapter 8

Another Application of BV-trees

We have successfully used our collision detection algorithms and data structures in another application . NC (numerically controlled) verification, is a natural application of our techniques, as one of the common problems within NC milling is to verify that a tool does not *gouge* (i.e., collide) the material which is to be left behind after milling the desired shape. The work from this section originally appeared in a manuscript written with Martin Held and Joseph S.B. Mitchell [46]. In our implementation, Martin Held is responsible for coding the routines which test two primitives (e.g., triangles, spheres, cylinders, tori) for contact. He has also generated the terrain datasets and some of the tool paths used in our experiments.

We note that the work in Chapter 8 was performed prior to incorporating many of the improvements and extensions of Chapter 6 into the current version of our CD software. Thus, the software used to produce the results here most closely resembles the software from Chapter 5, which reported our original collision detection results. One improvement which was included in this version of the software was the usage of the "front tracking" technique described in Section 6.4.3. This addition was quite natural to our NC verification work, as the tool paths do not have the tool moving very far from one discrete location to the next. Thus, by taking advantage of this temporal coherence, we were able to speed up the gouge checking queries.

8.1 NC Verification

8.1.1 Introduction

In addition to fundamental advances to collision detection, we give applications of the foundational method of our general-purpose CD algorithm to the problem of NC verification. For given geometric models of input stock, of goal shape, and of a cutting tool, the problem of determining if a given tool path results in removing the appropriate material from the input stock, while avoiding *gouging* (over-cutting) is of paramount importance in verifying a tool path prior to production machining. Whether the tool path was generated manually, automatically, or semi-automatically, it is essential to check the path for deficiencies prior to making a potentially costly and wasteful error. The `GougeChek` system has been developed to solve this problem, based on the underlying technology provided by `QuickCD`. Currently, `GougeChek` supports 3-axis machining. The methodology supports 4-axis and 5-axis machining, however, integrating this into the existing system remains future work.

Related Work In theory, NC verification is a solved problem, as the geometry of the part actually machined can be derived from the geometry of the billet, the cutter tool, and the tool path, by using a sequence of (regularized) Boolean operations, which are standard for CSG-based modelers. This approach is not practical, however, since a tool path for a part of even modest complexity can still contain several thousands of path segments, each of which contributes a swept volume element that must be subtracted from the billet. Several ray-casting techniques have been proposed in order to speed up and/or avoid the CSG (constructive solid geometry) evaluation process, see for instance the work by Wang and Wang [95], and Menon *et al.* [66, 67], who use a ray representation.

Many approaches to NC verification were based on discretizing the part surface, and on using computer graphics techniques (such as a Z-buffer and variants and extensions thereof) for modeling the material removal, cf. [4, 6, 74, 92, 96]. With many of those early systems, the emphasis was placed on visualizing the material removal for a specified direction of view. Image-space based techniques kept playing an important role in NC verification, though. See, for instance, the recent image-space sweeping by Hui [52].

Chappel [19] used verification posts erected on the surface of the part to be machined for simulating the material removal: akin to the mowing of a lawn by a cutting blade, the tool clips these vectors and thus simulates material removal via a sequence of line clipping processes. In [20], he extended this idea to modeling the part itself as a family of discrete vectors, and to use this information for tool path generation.

Drysdale *et al.* [34] and Jerard *et al.* [53, 54] discretize the part surface with respect to a specified chordal tolerance, and then work with a triangular mesh that approximates the part surface. Bounds on the amount of over-cutting/under-cutting are obtained by computing distances between approximations of the swept volume of the tool and between those points on the part surface. See also the similar approach of Oliver and Goodman [75]. Another discrete representation is the R-map employed by Yang and Lee [99].

Space subdivisions for performing NC verification have been popular with many researchers, and they have been used in several variants, including work on quadtrees [63], octrees [58], G-buffers [83], and dexels [92, 48]. Their unifying idea is to decompose the part into simpler cells, thus reducing the computational complexity of the Boolean operations.

8.1.2 NC Verification Experiments

To further demonstrate the applicability of our CD system to real-world problems, we have designed `GougeChek`, which will test tool paths for gouging in 3-axis machining. `GougeChek` was developed from the underlying structure of `QuickCD`, however, several enhancements have been added. In addition to being able to move polygonal models within environments, we are now able to handle actual spheres and cylinders as, or part of, the flying object. As many tools in NC verification are ball-shaped, we could easily use a polyhedral approximation of the tool as our moving object, however, in doing so we introduce an approximation factor into the problem. By using an actual sphere or cylinder, we can avoid this. Similarly, we can use an actual cylinder to model the tool arm and also as the volume swept out by our ball-shaped tool. This allows us to avoid the problem of checking the tool versus the environment (in this case the goal shape) at a fixed number of discrete locations. By checking the swept volume of the tool, we can find all instances of gouging that would occur.

Another enhancement within `GougeChek` which we have made is the ability to take advantage of *translation-only* motion. Since we have only incorporated 3-axis machining within our NC verification system, the tools will not rotate at all, but rather will only translate from one location to the next. Thus, by designing our collision detection methods to be flexible enough to handle this option, we can take advantage of these situations in which the user knows that a translation-only assumption is appropriate.

Datasets. For the tests of `GougeChek` we used data on five molds, provided by Bridgeport Machines, which resemble ducts and bottles: Part_0, Part_1a, Part_1b, Part_2, Part_3. In addition, we generated nine synthetic “terrain molds”: we sampled real-world elevation data, so-called “1-degree DEM” data, thereby converting 1201×1201 elevation arrays into 300×300 elevation arrays. These elevation arrays were afterwards converted into triangulated surfaces by means of a straightforward triangulation of the data points. Each of the terrain molds contains 178,802 triangles. The DEM data was obtained by mean of anonymous ftp from the ftp-site “edcftp.cr.usgs.gov”.

Tool paths. For the four industrial molds we were able to use real-world paths, namely NC tool paths computed by means of *EZ-MILL*, a CNC package developed and distributed by Bridgeport Machines. All tool paths were computed for 3-axis machining with a ball-ended tool. Some of the tool paths were afterwards distorted slightly in order to ensure that `GougeChek` would encounter collisions.

For our synthetic terrain molds we created tool paths of our own. These are pure zigzag paths, created by locally offsetting two-dimensional polygons. The polygons correspond to rows in the elevation array, that is, to polygons in (y, z)-space. (We selected every tenth row/polygon and computed a locally correct offset for it.) Thus, the tip of a ball-ended tool always stays in contact with the terrain, but few collisions should occur while the tool is moving above a polygon that forms a ridge of a terrain. Collisions can and do occur, however, when the tool is moving in a valley of a terrain. As a matter of fact, some runs of `GougeChek` were slowed down by quite a few cases of gouging being reported.

In addition to checking discrete positions of the ball-ended tool (sphere) for gouging, we also ran a second set of experiments in which we modeled the true swept volumes by means of cylinders. Although all of our NC paths were generated only for ball-shaped tools, we also tested tools of different shapes along the same paths. In addition to a sphere, we tested an up-right cylinder, a polyhedral approximation of a sphere, a polyhedral approximation of the cylinder, and two polyhedral approximations of other tools. One tool was a flat-ended cylindrical tool with rounded corners, and the other tool was nearly conical, with a rounded tip. All tools where scaled to have approximately the same size as the original sphere used for the tool path computation, and were moved such that their tips would touch the same point as did the tip of the sphere. Nevertheless, the swept volumes of the additional tools differ significantly from the swept volume of a sphere. Except for the conical tool, significantly more incidents of gouging were reported.

Results All of the runs reported in this section were conducted on a PC running Linux, with a Pentium Pro 180 processor and 128 megabytes of main memory.

Table 43 shows timings on our terrain molds. As expected, a CD check for a sphere is by far the fastest query. When considering the sphere plus its swept volume, the results strongly depend on the number of contacts encountered. For the two Denver terrains and for Moab-W, checking the entire swept volume of the sphere for gouging is still considerably faster than checking any of the polyhedral tools at discrete tool positions only. (We simply used the endpoints of each tool path segment as a tool position to be checked for gouging.) However, in the case of few contacts, the overhead of dealing with a general cylinder becomes dominant; due to increased costs for overlap tests between the cylinder and the k-dop, the average time consumed by a check for gouging goes up as k is increased. This observation is also confirmed by tests with a cylindrical tool (used instead of the sphere). (The corresponding entries have been omitted from the table due to lack of space.)

For the polyhedral tools we could also compare our timings to the timings obtained by using RAPID. Here, as in Chapter 5, we used the current version of RAPID at the time the experiments were performed. It is apparent that using OBBs is particularly wasteful in the case of Buffalo-W, which includes portions of Lake Erie (which, of course, are level with the (x, y)-plane). RAPID does much better on more rugged terrain, such as Moab-W, where `GougeChek` is faster by a factor of only about 4.5.

For this reason we selected Moab-W for another experiment. Using our polyhedral tools, we approximated their swept-volumes in a brute-force way by sampling the tool path (between tool path points) such that every sample was at most ϵ units apart from its predecessor and its successor, for $\epsilon = 1.0, 0.5, 0.1$. (The parts had a size of 300×300 units, and the true sphere used for calculating the paths had a radius of 10 units.) These additional results, which are omitted here, confirmed the statistics of Table 43.

Timings for the industrial molds, reported in Table 44, again give evidence to our hypothesis that using elaborate k-dops hardly pays off for queries against spheres and cylinders, unless a lot of intersections are encountered. The only exception to this hypothesis is Part_0, which exhibited a speed-up when increasing k, in spite of a small overall number of intersections. Part_0, being our least complex dataset, also is exceptional for another reason: it is the only environment where RAPID won over `GougeChek`, scoring three victories. We examined the dataset and the test runs but cannot offer any explanation why RAPID performed particularly well on this part in comparison with `GougeChek`. In particular, Part_0 has a striking similarity with Part_1a/Part_1b, where RAPID did not even get close to winning.

The success of our `GougeChek` system in these experiments was very exciting; however, we feel that our greatest achievement with `GougeChek` is that it has recently been incorporated into the Release 10 version of *EZ-MILL*, the CNC package sold by Bridgeport Machines. It is one thing to perform pseudo-realistic experiments and determine that your method works the best. It is entirely another matter for your implementation to be incorporated into a commercial product and successfully used by many customers in their every day work. We are especially proud of this fact. This is also gratifying in that although we have not tested our `GougeChek` system versus any software packages specifically designed for NC milling and verification (RAPID is a more general collision detection system), our system is still considered valuable by one of this industry's leading companies.

	Buffalo-W	Denver-E	Denver-W	Eagle-E	Moab-W	Seattle-E
Env. Size: #(tri.)	178,802	178,802	178,802	178,802	178,802	178,802
#(Steps)	5,232	6,672	6,134	6,785	6,237	6,129
	Ball-Shaped Tool					
#(Contacts)	316	9,604	65,266	0	49,345	21,757
6-dop	0.007	0.022	0.144	0.004	0.097	0.043
14-dop	0.006	0.023	0.146	0.004	0.102	0.047
18-dop	0.006	0.024	0.153	0.004	0.108	0.051
26-dop	0.006	0.024	0.153	0.004	0.108	0.051
	Ball-Shaped Tool with Cylindrical Swept Volumes					
#(Contacts)	520	14,947	90,612	167	69,959	37,765
6-dop	0.030	0.069	0.409	0.020	0.274	0.134
14-dop	0.038	0.076	0.324	0.034	0.241	0.143
18-dop	0.045	0.083	0.344	0.041	0.246	0.148
26-dop	0.055	0.096	0.339	0.051	0.261	0.166
	Polyhedral Ball-Shaped Tool (1,680 tri.)					
#(Contacts)	608	17,183	10,3123	0	77,385	40,248
6-dop	0.051	0.332	2.498	0.009	1.713	0.760
14-dop	0.020	0.266	1.782	0.005	1.297	0.639
18-dop	0.022	0.256	1.769	0.005	1.266	0.619
26-dop	0.020	0.253	1.696	0.007	1.236	0.620
RAPID	0.220	1.300	8.240	0.080	5.940	3.040
RAPID / 18-dop	10.000	5.078	4.658	16.000	4.692	4.911
	Polyhedral Tool with Rounded Corners (1,148 tri.)					
#(Contacts)	2,472	21,908	166,024	102	112,254	46,265
6-dop	0.108	0.439	3.642	0.017	2.369	0.912
14-dop	0.056	0.353	2.868	0.008	1.916	0.800
18-dop	0.057	0.328	2.753	0.008	1.834	0.742
26-dop	0.054	0.327	2.689	0.008	1.797	0.753
RAPID	0.380	1.650	12.130	0.100	8.000	3.590
RAPID / 18-dop	6.667	5.030	4.406	12.500	4.362	4.838
	Polyhedral Conical Tool with Rounded Tip (1,230 tri)					
#(Contacts)	185	11,765	49,189	0	41,819	29,440
6-dop	0.029	0.278	1.633	0.006	1.209	0.673
14-dop	0.010	0.211	0.901	0.005	0.742	0.533
18-dop	0.011	0.203	0.934	0.005	0.749	0.514
26-dop	0.009	0.200	0.846	0.006	0.704	0.507
RAPID	0.160	0.950	4.450	0.070	3.510	2.350
RAPID / 18-dop	14.545	4.680	4.764	14.000	4.686	3.880

Table 43: Statistics for NC verification (`GougeChek`): terrain data.

	Part_0	Part_1a	Part_1b	Part_2	Part_3
Env. Size: #(tri.)	1,634	11,010	11,010	12,880	80,840
#(Steps)	9,230	9,230	9,230	2,896	24,144
	Ball-Shaped Tool				
#(Contacts)	2,262	0	0	739,393	1,458,162
6-dop	0.156	0.074	0.074	2.193	1.422
14-dop	0.154	0.079	0.080	2.807	1.596
18-dop	0.154	0.079	0.080	2.807	1.626
26-dop	0.143	0.084	0.084	3.069	1.636
	Ball-Shaped Tool with Cylindrical Swept Volumes				
#(Contacts)	2,292	0	0	793,882	1,710,061
6-dop	0.325	0.188	0.188	4.443	3.947
14-dop	0.274	0.185	0.184	3.980	2.762
18-dop	0.257	0.193	0.193	3.980	2.713
26-dop	0.260	0.227	0.226	4.270	2.642
	Polyhedral Ball-Shaped Tool (1,680 tri.)				
#(Contacts)	19,437	0	0	346,250	742,197
6-dop	8.013	1.025	1.024	36.306	26.527
14-dop	7.244	0.700	0.699	29.618	17.113
18-dop	5.928	0.442	0.442	28.431	15.591
26-dop	5.957	0.436	0.437	28.302	14.571
RAPID	2.900	1.520	1.520	48.780	21.500
RAPID / 18-dop	0.489	3.439	3.439	1.716	1.379
	Polyhedral Tool with Rounded Corners (1,148 tri.)				
#(Contacts)	155,508	46,040	46,040	517,788	3,596,799
6-dop	11.083	2.431	2.432	37.478	39.475
14-dop	10.313	1.623	1.624	31.795	31.689
18-dop	9.010	1.186	1.188	30.283	28.834
26-dop	9.043	1.165	1.163	30.729	28.658
RAPID	8.600	1.820	1.820	62.970	58.840
RAPID / 18-dop	0.955	1.535	1.532	2.079	2.041
	Polyhedral Conical Tool with Rounded Tip (1,230 tri)				
#(Contacts)	15,465	0	0	210,991	413,156
6-dop	6.814	0.501	0.498	26.046	16.680
14-dop	6.317	0.305	0.305	14.527	6.171
18-dop	4.684	0.162	0.160	13.242	5.357
26-dop	4.729	0.156	0.156	11.840	4.991
RAPID	1.680	1.160	1.160	27.340	11.080
RAPID / 18-dop	0.359	7.160	7.250	2.065	2.068

Table 44: Statistics for NC verification (`GougeChek`): industrial mold data.

Chapter 9

Conclusion

We began our investigation into collision detection algorithms by utilizing three standard box-based data structures. Our initial motivation was to validate the practicality of another method, based upon building a tetrahedral mesh on the environment and tracking the motion of moving objects within the mesh. Somewhat to our surprise, one of the "naive" data structures, the R-tree, was found to be quite efficient, both in its usage of memory and in its ability to answer collision detection queries accurately and rapidly.

Building upon this preliminary examination, we 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 [38]), whose hierarchy is based on oriented bounding boxes. Our system also compares very well to another publicly available collision detection system ("SOLID" [89]), which utilized axis-aligned bounding boxes for approximations. 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.

To further demonstrate its applicability to real-world problems, we have extended our `QuickCD` system to produce the `GougeChek` system, which can be used efficiently for NC verification. Experimental results aside, the incorporation of `GougeChek` into the latest commercial release of *EZ-MILL* is ample evidence of this fact.

9.1 Extensions and Future Work

Throughout this dissertation, we have mentioned several possible extensions of our work, including alternative methods for constructing BV-trees, such as:

using values of k larger than 26 for our k-dops (Section 3.4.5),

using alternative "splitting rules" (Section 3.4.6),

using a bottom-up method to construct BV-trees (Section 3.4.3), and

using BV-trees with degree greater than two (Section 3.4.2).

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 4.2),
>
> avoiding a hard-coded threshold to control the depth of the hierarchies (Section 4.4), and
>
> using a specially designed ordering when performing interval overlap queries (Section 4.5).

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

Multiple Flying Objects. `QuickCD` could easily support multiple flying objects in the naive manner: by checking each pair of hierarchies for collision (as well as checking each flying object against the environment as is done now for one moving object). However, a more efficient method is to use a "sweep-and-prune" technique, as in I_COLLIDE [24]. It is natural, however, to generalize the method somewhat, based on our overall strategy of discretizing orientations: We project the flying objects onto the $k/2$ lines corresponding to the k-dops, and maintain these projection intervals over time. Note that these projection intervals "come for free" as they are trivially derived from the k-dop of the root of each flying hierarchy. Obviously, we need to check only those flying objects for pairwise collision that have overlapping intervals in all projections.

Dynamic Environments. Insertions or deletions of objects pose a challenge for any method that relies on preprocessing in order to achieve high-speed queries. Within `QuickCD`, we could handle insertions of a new obstacle model in much the same way as we perform collision queries — by traversing the environment tree, we could identify a good placement of the new obstacle primitives within the tree. Deletions could also be done. (This is analogous to the insertion/deletion of data in an R-tree.) Of course, repeated insertions or deletions may cause the BV-tree to become highly unbalanced. Thus, every so often, we would need to do re-balancing and local re-optimizations. Section 3.4.4 provided a brief introduction into using dynamic data structures in these situations.

Additional Applications Our data structures have recently been used to perform approximate nearest neighbor queries for points in arbitrary dimensions. By building a BV-tree upon the points, we are able to quickly answer these types of queries in the L_2 and L_∞ metrics. A more thorough investigation into this application is warranted.

Bibliography

[1] P. K. Agarwal, B. Aronov, and S. Suri. Line stabbing bounds in three dimensions. Manuscript, Dec., 1993.

[2] P. K. Agarwal and J. Matoušek. Ray shooting and parametric search. In *Proc. 24th Annu. ACM Sympos. Theory Comput.*, pp. 517–526, 1992.

[3] P. K. Agarwal and M. Sharir. Ray shooting amidst convex polytopes in three dimensions. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pp. 260–270, 1993.

[4] R.O. Anderson. Detecting and Eliminating Collisions in NC Machining. *Comput. Aided Design*, 10(4):231–237, July 1978.

[5] J. Arvo and D. Kirk. A Survey of Ray Tracing Acceleration Techniques. In A.S. Glassner, editor, *An Introduction to Ray Tracing*, pp. 201–262. Academic Press, 1990. ISBN 0-12-286160-4; 3rd printing.

[6] P. Atherton, C. Earl, and C. Fred. A Graphical Simulation System for Dynamic Five-Axis NC Verification. In *AUTOFACT Show of SME*, pp. 2.1–2.12, Detroit, MI, USA, Nov. 1987. Society of Manufacturing Engineers (SME).

[7] L.S. Avila and W. Schroeder. Interactive Visualization of Aircraft and Power Generation Engines. In *Proc. IEEE Visualization '97*, pp. 483–486, Phoenix, AZ, USA, Oct. 1997.

[8] L. Bachmann, B.-U. Pagel, H.-W. Six. Optimizing Spatial Data Structures for Static Data In *Proc. IGIS'94: Geographic Information Systems*, vol. 884 of *Lecture Notes Comput. Sci.*, pp. 247–258, 1994.

[9] D. H. Ballard. Strip trees: A hierarchical representation for curves. *Commun. ACM*, 24(5):310–321, 1981.

[10] D. Baraff. Curved Surfaces and Coherence for Non-penetrating Rigid Body Simulation. In *Comput. Graphics (SIGGRAPH '90 Proc.)*, pp. 19–28, Dallas, TX, USA, Aug. 1990.

[11] D. Baraff. Fast contact force computation for nonpenetrating rigid bodies. In *Comput. Graphics (SIGGRAPH '94 Proc.)*, pp. 23–34, Orlando, FL, USA, Jul. 1994.

[12] D. Baraff. Interactive Simulation of Solid Rigid Bodies. *IEEE Comput. Graph. Appl.*, 15(3):63–75, May 1995.

[13] G. Barequet, B. Chazelle, L.J. Guibas, J.S.B. Mitchell, and A. Tal. BOX-TREE: A Hierarchical Representation for Surfaces in 3D. *Comput. Graph. Forum*, 15(3):C387–C484, 1996. Proc. Eurographics'96.

[14] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In

Proc. ACM SIGMOD Int. Conf. Management Data, Atlantic City, NJ, USA, pp. 322–331, May 1990.

[15] J.L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Comm. ACM*, 18(9):509–517, 1975.

[16] W. Bouma and G. Vaněček, Jr. Collision Detection and Analysis in a Physical Based Simulation. In *Eurographics Workshop on Animation and Simulation*, pp. 191–203, Vienna, Austria, Sep. 1991.

[17] S. Cameron. Collision Detection by Four-Dimensional Intersection Testing. *IEEE Trans. Robot. Autom.*, 6(3):291–302, 1990.

[18] J. Canny. Collision Detection for Moving Polyhedra. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8(2):200–209, 1986.

[19] I.T. Chappel. The Use of Vectors to Simulate Material Removed by Numerically Controlled Milling. *Comput. Aided Design*, 15(3):156–158, May 1983.

[20] I.T. Chappel. A New Approach to Automatic Tool Path Generation For Numerically Controlled Milling Machines. In *Proc. 4^{th} Int. Conf. on Manufacturing Engineering*, pp. 29–32, Brisbane, Australia, May 1988.

[21] B. Chazelle, H. Edelsbrunner, M. Grigni, L. Guibas, J. Hershberger, M. Sharir, and J. Snoeyink. Ray shooting in polygons using geodesic triangulations. In *Proc. 18^{th} Int. Colloq. Automata Lang. Program.*, vol. 510 of *Lecture Notes Comput. Sci.*, pp. 661–673. Springer-Verlag, 1991.

[22] S. W. Cheng and R. Janardan. Algorithms for ray-shooting and intersection searching. *J. Algorithms*, 13:670–692, 1992.

[23] K. Chung. *An Efficient Collision Detection Algorithm for Polytopes in Virtual Environments.* Master of philosophy thesis, CS Dept., U. Hong Kong, 1996.

[24] 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. In *Proc. ACM Interactive 3D Graphics Conf.*, pp. 189–196, 1995.

[25] International Business Machines Corporation. User's Guide, IBM 3D Interaction Accelerator^{TM}. Version 1 release 2.0, IBM T.J. Watson Res. Center, Yorktown Heights, NY 10598, USA, Sep 1995.

[26] A. Crosnier and J. Rossignac. T-BOX: The Intersection of Three Minimax Boxes. Internal report, IBM T.J. Watson Res. Center, Yorktown Heights, NY 10598, USA, 1995.

[27] M. de Berg. *Efficient algorithms for ray shooting and hidden surface removal.* Ph.D. dissertation, Dept. Comput. Sci., Utrecht Univ., Utrecht, Netherlands, 1992.

[28] M. de Berg, D. Halperin, M. Overmars, J. Snoeyink, and M. van Kreveld. Efficient ray shooting and hidden surface removal. In *Proc.* 7th *Annu. ACM Sympos. Comput. Geom.*, pp. 21–30, 1991.

[29] D.P. Dobkin, J. Hershberger, D.G. Kirkpatrick, and S. Suri. Computing the Intersection-Depth of Polyhedra. *Algorithmica*, 9:518–533, 1993.

[30] D.P. Dobkin and D.G. Kirkpatrick. Fast Detection of Polyhedral Intersection. *Theoret. Comput. Sci.*, 27:241–253, 1983.

[31] D.P. Dobkin and D.G. Kirkpatrick. A Linear Algorithm for Determining the Separation of Convex Polyhedra. *J. Algorithms*, 6:381–392, 1985.

[32] D.P. Dobkin and D.G. Kirkpatrick. Determining the Separation of Preprocessed Polyhedra – A Unified Approach. In *Proc.* 17th *Int. Colloq. Automata Lang. Program.*, vol. 443 in *Lecture Notes Comput. Sci.*, pp. 400–413. Springer-Verlag, 1990.

[33] K. Dobrindt, K. Mehlhorn, and M. Yvinec. A Complete and Efficient Algorithm for the Intersection of a General and a Convex Polyhedron. In *Proc.* 3rd *Workshop Algorithms Data Struct.*, vol. 709 in *Lecture Notes Comput. Sci.*, pp. 314–324. Springer-Verlag, 1993.

[34] R.L. Drysdale, R.B. Jerard, B. Schaudt, and K. Hauck. Discrete Simulation of NC Machining. *Algorithmica*, 4(1):33–60, 1989.

[35] A. Garcia-Alonso, N. Serrano, and J. Flaquer. Solving the Collision Detection Problem. *IEEE Comput. Graph. Appl.*, 14(3):36–43, May 1994.

[36] J. Goldsmith and J. Salmon. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Comput. Graph. Appl.*, 7:14–20, 1987.

[37] M. T. Goodrich and R. Tamassia. Dynamic ray shooting and shortest paths via balanced geodesic triangulations. In *Proc.* 9th *Annu. ACM Sympos. Comput. Geom.*, pp. 318–327, 1993.

[38] S. Gottschalk, M.C. Lin, and D. Manocha. OBB-Tree: A Hierarchical Structure for Rapid Interference Detection. In *Comput. Graphics (SIGGRAPH '96 Proc.)*, pp. 171–180, New Orleans, LA, USA, Aug. 1996.

[39] D. Greene. An Implementation and Performance Analysis of Spatial Data Access Methods In *Proc.* 5th *Int. Conf. Data Engineering*, pp. 606–615, Los Angeles, CA, USA, Feb. 1989.

[40] N. Greene. Detecting Intersection of a Rectangular Solid and a Convex Polyhedron. In P.S. Heckbert, editor, *Graphics Gems IV*, pp. 74–82. Academic Press, 1994. ISBN 0-12-336155-9.

[41] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching In *Proc. ACM SIGMOD Int. Conf. Management Data*, pp. 47–57, Boston, MA, USA, June 1984.

[42] T. He and A. Kaufman. Collision detection for volumetric models. In *Proc. IEEE Visualization '97*, pp. 27–34, Phoenix, AZ, USA, Oct. 1997.

[43] M. Held. ERIT – A Collection of Efficient and Reliable Intersection Tests. To appear *J. Graphics Tools*, 1998.

[44] M. Held, J.T. Klosowski, and J.S.B. Mitchell. Evaluation of Collision Detection Methods for Virtual Reality Fly-Throughs. In *Proc.* 7th *Canad.*

Conf. Comput. Geom., pp. 205–210, Québec City, Québec, Canada, Aug. 1995.

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

[46] M. Held, J.T. Klosowski, and J.S.B. Mitchell. QuickCD: An Efficient Collision Detection System Using BV-Trees. Manuscript, Dept. of Appl. Math. Stat., SUNY at Stony Brook, NY, USA, Mar. 1997.

[47] J. Hershberger and S. Suri. A pedestrian approach to ray shooting: Shoot a ray, take a walk. In *Proc. 4^{th} ACM-SIAM Sympos. Discrete Algorithms*, pp. 54–63, 1993.

[48] Y. Huang and J.H. Oliver. NC Milling Error Assessment and Tool Path Correction. In *Comput. Graphics (SIGGRAPH '94 Proc.)*, pp. 287–294, Orlando, FL, USA, July 1994.

[49] P.M. Hubbard. Collision Detection for Interactive Graphics Applications. *IEEE Trans. Visualizat. Comput. Graph.*, 1(3):218–230, Sep. 1995.

[50] P.M. Hubbard. Approximating Polyhedra with Spheres for Time-Critical Collision Detection. *ACM Trans. Graph.*, 15(3):179–210, July 1996.

[51] T.C. Hudson, M.C. Lin, J. Cohen, S. Gottschalk, and D. Manocha. V-COLLIDE: Accelerated Collision Detection for VRML. In *Proc. 2^{nd} Annu. Sympos. on the Virtual Reality Modeling Language*, Monterey, CA, USA, 1997.

[52] K.C. Hui. Solid Sweeping in Image Space – Application in NC Simulation. *Visual Comput.*, 10(6):306–316, 1994.

[53] R.B. Jerard, R.L. Drysdale, K. Hauck, B. Schaudt, and J. Magewick. Methods for Detecting Errors in Numerically Controlled Machining of Sculptured Surfaces. *IEEE Comput. Graph. Appl.*, 9(1):26–39, Jan. 1989.

[54] R.B. Jerard, S.Z. Hussaini, R.L. Drysdale, and B. Schaudt. Approximate Methods for Simulation and Verification of Numerically Controlled Machining Programs. *Visual Comput.*, 5(6):329–348, Dec. 1989.

[55] H. V. Jagadish. Spatial Search with Polyhedra In *Proc.* 6th *Int. Conf. Data Engineering*, Los Angeles, USA, pp. 311–319, Feb. 1990.

[56] J.T. Kajiya and T.L. Kay. Ray Tracing Complex Scenes. In *Comput. Graphics (SIGGRAPH '86 Proc.)*, pp. 269–278, Dallas, TX, USA, Aug. 1986.

[57] N. Katayama and S. Satoh. The SR-tree: An Index Structure for High Dimensional Nearest Neighbor Queries. In *Proc. ACM SIGMOD Int. Conf. Management Data*, Tucson, AZ, USA, 1997.

[58] Y. Kawashima et al. A Flexible Quantitative Method for NC Machining Verification Using a Space-Division Based Solid Model. *Visual Comput.*, 7(2–3):149–157, May 1991.

[59] J.T. Klosowski, M. Held, J.S.B. Mitchell, H. Sowizral, and K. Zikan. Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs. *IEEE Trans. Visualizat. Comput. Graph.*, 4(1):21–36, Mar. 1998.

[60] M.C. Lin. *Efficient Collision Detection for Animation and Robotics.* PhD thesis, EECS Dept., U. California, Berkeley, CA, USA, 1993.

[61] M.C. Lin and J.F. Canny. Efficient Algorithms for Incremental Distance Computation. In *Proc. IEEE Int. Conf. Robot. Automat.*, pp. 1008–1014, 1991.

[62] M.C. Lin and D. Manocha. Fast Interference Detection Between Geometric Models. *Visual Comput.*, 11(10):542–561, 1995.

[63] C. Liu, D.M. Esterling, J. Fontdecaba, and E. Mosel. Dimensional Verification of NC Machining Profiles Using Extended Quadtrees. *Comput. Aided Design*, 28(11):845–852, Nov. 1996.

[64] I.P. Logan, W.A. McNeely, J.J. Troy, and D.P. Wills. Voxel based real-time collision detection. Manuscript, Univ. Hull, 1997.

[65] W.A. McNeely. The Challenge of Industrial-Stength Haptic Simulation. Manuscript, The Boeing Company, 1996.

[66] J.P. Menon and D.M. Robinson. Advanced NC Verification via Massively Parallel Raycasting. *Manufact. Review*, 6(2):141–154, June 1993.

[67] J.P. Menon and H.B. Voelcker. Toward a Comprehensive Formulation of NC Verification as a Mathematical and Computational Problem. *J. Design & Manufact.*, 3:263–277, 1993.

[68] B. Mirtich. V-Clip: Fast and Robust Polyhedral Collision Detection. Submitted to *ACM Trans. Graph.*, MERL Technical Report TR-97-05, June 1997.

[69] J. S. B. Mitchell, D. M. Mount, and S. Suri. Query-sensitive ray shooting. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pp. 359–368, 1994.

[70] M. Moore and J. Wilhelms. Collision Detection and Response for Computer Animation. In *Comput. Graphics (SIGGRAPH '88 Proc.)*, pp. 289–298, Atlanta, GA, USA, Aug. 1988.

[71] D. M. Mount. Intersection detection and separators for simple polygons. In *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, pp. 303–311, 1992.

[72] B. Naylor, J. Amanatides, and W. Thibault. Merging BSP Trees Yields Polyhedral Set Operations. In *Comput. Graphics (SIGGRAPH '90 Proc.)*, pp. 115–124, Aug. 1990.

[73] H. Noborio, S. Fukuda, and S. Arimoto. Fast Interference Check Method Using Octree Representation. *Advanced Robotics*, 3(3):193–212, 1989.

[74] J.H. Oliver. *Graphical Verification of Numerically Controlled Milling Programs for Sculptured Surfaces.* PhD thesis, Department of CS, Michigan State U., MI, USA, 1986.

[75] J.H. Oliver and E.D. Goodman. Direct Dimensional NC Verification. *Comput. Aided Design*, 22(1):3–10, 1990.

[76] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, New York, 1994. ISBN 0-521-44592-2.

[77] I.J. Palmer and R.L. Grimsdale. Collision Detection for Animation Using Sphere-Trees. *Comput. Graph. Forum*, 14(2):105–116, June 1995.

[78] M. Pellegrini. Stabbing and ray shooting in 3-dimensional space. In *Proc.* 6*th* *Annu. ACM Sympos. Comput. Geom.*, pp. 177–186, 1990.

[79] M.K. Ponamgi, D. Manocha, and M.C. Lin. Incremental Algorithms for Collision Detection between General Solid Models. In *Proc. Symp. Solid Modeling Found. Applications*, pp. 293–304, 1995.

[80] F.P. Preparata and S.J. Hong. Convex hulls of finite sets of points in two and three dimensions. *Comm. ACM* 20:87–93, 1977.

[81] F.P. Preparata and M.I. Shamos. *Computational Geometry – An Introduction.* Springer-Verlag, New York, 1985. ISBN 0-387-96131-3.

[82] N. Roussopoulos and D. Leifker Direct Spatial Search on Pictorial Databases Using Packed R-trees In *Proc. ACM SIGMOD Int. Conf. Management Data*, pp. 17–31, 1985.

[83] T. Saito and T. Takahashi. NC Machining with G-Buffer Method. In *Comput. Graphics (SIGGRAPH '91 Proc.)*, pp. 207–216, July 1991.

[84] E. Schömer. *Interaktive Montagesimulation mit Kollisionserkennung.* PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 1994.

[85] E. Schömer and C. Thiel. Efficient collision detection for moving polyhedra. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pp. 51–60, Vancouver, BC, Canada, 1995.

[86] E. Schömer and C. Thiel. Subquadratic Algorithms for the General Collision Detection Problem. In *Abstracts* 12*th* *Europ. Workshop Computat. Geometry (CG'96)*, pp. 95–101, Münster, Germany, Mar. 1996.

[87] O. Schwarzkopf. Ray shooting in convex polytopes. In *Proc.* 8th *Annu. ACM Sympos. Comput. Geom.*, pp. 286–295, 1992.

[88] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R$^+$-Tree: A Dynamic Index for Multi-Dimensional Objects In *Proc.* 13th *Int. Conf. VLDB*, pp. 507–518, Brighton, England, Sep. 1987.

[89] SOLID url: `www.win.tue.nl/cs/tt/gino/solid/`.

[90] S. Suri, P. M. Hubbard, and J. F. Hughes. Collision Detection in Aspect and Scale Bounded Polyhedra. *Proc.* 9th *ACM-SIAM Sympos. Discrete Algorithms*, pp. 127–136, Jan. 1998.

[91] Y. Theodoridis and T. Sellis. Optimization Issues in R-Tree Construction. In *Proc. IGIS'94: Geographic Information Systems*, vol. 884 of *Lecture Notes Comput. Sci.*, pp. 270–273, 1994.

[92] T. van Hook. Real-Time Shaded NC Milling Display. In *Comput. Graphics (SIGGRAPH '86 Proc.)*, pp. 15–20, 1986.

[93] G. Vaněček, Jr. Brep-index: A Multidimensional Space Partitioning Tree. *Int. J. Comput. Geom. Appl.*, 1(3):243–261, 1991.

[94] G. Vaněček, Jr. Back-face Culling Applied to Collision Detection of Polyhedra. *J. Visualizat. and Comput. Animation*, 5(1), 1994.

[95] W.P. Wang and K.K. Wang. Geometric Modeling for Swept Volume of Moving Solids. *IEEE Comput. Graph. Appl.*, 6:8–17, Dec. 1986.

[96] W.P. Wang and K.K. Wang. Real-Time Verification of Multiaxis NC Programs with Raster Graphics. In *IEEE Int. Conf. on Robotics and Automation*, San Francisco, CA, USA, Apr. 1986.

[97] H. Weghorst, G. Hooper, and D.P. Greenberg. Improved Computational Methods for Ray Tracing. *ACM Trans. Graph.*, 3(1):52–69, 1984.

[98] D. A. White and R. Jain. Similarity Indexing with the SS-tree In *Proc. 12th Int. Conf. Data Engineering*, New Orleans, LA, USA, pp. 516–523, Feb. 1996.

[99] M. Yang and E. Lee. NC Verification for Wire-EDM Using an R-Map. *Comput. Aided Design*, 28(9):733–740, Sep. 1996.

[100] G. Zachmann. *Exact and Fast Collision Detection.* Diploma thesis, Institut für Graphische Datenverarbeitung, TU Darmstadt, Darmstadt, Germany, 1994.

[101] G. Zachmann and W. Felger. The BoxTree: Enabling Real-time and Exact Collision Detection of Arbitrary Polyhedra. In *Proc. 1st Workshop on Simulation and Interaction in Virtual Environments*, U. of Iowa, July 1995.

[102] K. Zikan and P. Konečný. Lower Bound of Distance in 3D. Technical report, Masaryk University, Czech Republic, Nov. 1996.