Evaluation of Collision Detection Methods for Virtual Reality Fly-Throughs

Martin Held* James T. Klosowski† Joseph S.B. Mitchell‡

Department of Applied Mathematics and Statistics
State University of New York, Stony Brook, NY 11794-3600

1 Introduction

Real-time collision detection is one of the most important problems that must be addressed in order to make *Virtual Reality* (VR) a "reality". Motivated by this need we address the following problem: Given a description of a 3D geometric environment, $\mathcal{E}$, preprocess it into a data structure of small size so that queries of the form, "*Does object F intersect any of the obstacles in $\mathcal{E}$?*" can be answered very rapidly. Further, we study the problem of *tracking* the motion of F within $\mathcal{E}$, in order to detect dynamically, in real time, when F collides with an obstacle. The environment $\mathcal{E}$ is given to us as a *boundary representation* of a set of polyhedral obstacles.

Our method for collision detection is motivated by recent computational geometry results on "ray shooting" using meshes (triangulations) of low "stabbing number"[1]. 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. (An excellent reference on the subject is [5].) From the *practical* point of view, however, the 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 [9, 13]. Then, the complexity of a query is the number of cells (tetrahedra) of the mesh that are met by the query ray before it hits an obstacle.

We have developed and implemented a collision detection method whose preprocessing step is to construct a decomposition of free space (the complement of the obstacles) into tetrahedra, marking the (triangular) facets that correspond to obstacle boundaries. For a given position of the object F, we then determine those tetrahedra intersecting the (triangular) facets of F, by using a simple search of the subdivision. In this paper, we give details of our algorithm, and we describe the experiments that we have conducted to test the efficiency of this method. A primary goal of our study was to determine if, indeed, the methods motivated by computational geometry have a practical impact on the collision detection problem. We report on our implementations of other hierarchical collision detection methods, and we compare results in a series of experiments.

Previous Work

Detecting intersections is of fundamental importance in computer graphics, solid modeling, and virtual reality. Thus, there have been many approaches to solving this problem, including the use of binary space partition (BSP) trees [16], "R-trees" and their variants [1], and octrees [14, 17].

One recent method for 3-dimensional intersection detection has been developed by Lin and Manocha [12, 4]. This method is based on decomposing solids into a union of convex polyhedra, constructing the Voronoi diagram of the space surrounding each such convex piece (which is particularly trivial), and then tracking the closest pairs of points between all pairs of convex pieces. This method is highly effective as long as there are not too many convex pieces involved; it has the advantage of not relying on any of the objects remaining stationary. Other recent work includes [7] and the thesis work of Hubbard [10, 11], who uses approximations of objects (covering by disks) to speed up collision detection.

2 A Mesh-Based Algorithm

2.1 Collision Detection within a Mesh

Suppose we are given a tetrahedral mesh $\mathcal{M}$ that conforms[2] to the obstacle environment, $\mathcal{E}$, and a flying object, F, which is a k-faceted[3] polyhedron. We want to determine for a query instance of F, at a given position and orientation, whether F intersects the environment.

Obviously, F 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 deal with this problem later

*held@ams.sunysb.edu; fax: (516) 632-8490. Supported by NSF Grant DMS-9312098. On sabbatical leave from Universität Salzburg, Salzburg, Austria.

†jklosow@ams.sunysb.edu; fax: (516) 632-8490. Supported by NSF grant CCR-9204585, and by a grant from Boeing Computer Services.

‡jsbm@ams.sunysb.edu; fax: (516) 632-8490. Partially supported by NSF grant CCR-9204585, and by a grant from Boeing Computer Services.

[1]The (line) *stabbing number* of the mesh is the maximum number of cells met by any query ray before it encounters an obstacle.

[2]I.e., the boundary of the obstacle environment corresponds to the union of some facets of the tetrahedra. Due to the fact that $\mathcal{M}$ conforms to $\mathcal{E}$, no tetrahedron of $\mathcal{M}$ can lie partially inside and partially outside of some obstacle.

[3]Without loss of generality, assume that all facets of the flying polyhedron are triangles.

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 bounding box of F, and enumerate all tetrahedra that lie partially inside the bounding box[4]. Provided that one tetrahedron containing 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 facets of these tetrahedra is part of an obstacle boundary, then we are done (no collision).

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. 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.

Note that a test[5] to see if F lies completely in the interior of some obstacle (or vice versa) can easily be carried out. Starting at the boundary of the environment, the mesh $\mathcal{M}$ is scanned and all tetrahedra that lie inside some obstacle are marked. Now, a test whether F lies completely in the 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 already during the collision check, and can thus be checked efficiently.

Implementation Issues The basic building blocks of this algorithm are primitives for checking whether two triangles (for the collision check), or a triangle and a line segment (for ray-shooting), intersect in 3D. These primitives were carefully implemented and tested, cf. [8]. They rely on floating point operations, where comparisons with respect to zero are carried out by means of conventional ϵ-based thresholds. Extensive tests gave us confidence that these primitives are reliable and efficient.

Small (numerical) errors may not matter when checking for collisions. However, they do matter for ray-shooting! We were surprised to learn how often a ray would hit an edge or a vertex of a tetrahedron, or how often a point would lie on the boundary of a tetrahedron rather than in its interior. In both cases there is some danger of making inconsistent topological decisions due to numerical inaccuracy, which, in the worst case, may result in a 'looping' of the code. These problems have been overcome by a careful implementation. For instance, we always pass a list of vertices to our primitives such that the vertices appear in sorted order (with respect to their indices) in the list, which guarantees that the numerical results are independent of the processing order.

[4]For the sake of simplicity, our implementation enumerates the (possibly larger) set of all tetrahedra whose bounding boxes overlap with $BB(F)$.

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

2.2 Mesh Generation

Our current implementation generates a conforming mesh based on using a Delaunay triangulation of the obstacle vertices, with Steiner points added in order to make the triangulation conform to $\mathcal{E}$. We start with the original vertices of $\mathcal{E}$ and compute their Delaunay triangulation. Afterwards, we check whether any[6] triangular facet of $\mathcal{E}$ is intersected by a facet of some Delaunay tetrahedron. If no intersection exists, then the mesh conforms.

Otherwise, for each 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.

Our approach is essentially similar to the concepts outlined by Sapidis and Perucchio [18]. The main difference is that they try to minimize the number of points added by cleverly selecting Steiner points which do not necessarily correspond to points of intersection, thus possibly getting rid of several intersections by adding only one Steiner point.

Clearly, the mesh we obtain is not necessarily of low stabbing number. (Bad examples exist.) Our goal in this preliminary study was to obtain *some* conforming mesh, hopefully "nice" in some respect. We are developing now new heuristics designed to give much lower stabbing numbers, e.g., based on the recent methods of [13].

Implementation Issues We note that the robustness of the mesh generation is a major issue. Since real-world models tend to have a lot of "degeneracies," such as four or more points being coplanar, an assumption of "general position" is inappropriate. Thus, for robustness in the generation of the Delaunay triangulation, we have adapted Mücke's implementation, which is based on "simulation of simplicity" [6].

Apart from (artificially introduced) coplanarity, we mainly struggled with robustness problems stemming from "bad data". Polyhedral models that are publically available on ftp-servers on the Internet may be less than ideal topological and geometric representations of "clean" polyhedra. Rather, they tend to have various degrees of "nearly coplanar" vertices, i.e., polygonal faces bounded by four or more vertices where the vertices only approximately lie on the same plane. Even worse, many models tend to have self-intersections, i.e., they contain (triangular) faces that intersect in places other than at their boundaries. Unfortunately, such deficiences are also common among models generated using popular CAD systems.

Inconsistent topologies of polyhedral models constitute a serious problem for our application — an intersection between an edge of one obstacle triangle and another obstacle triangle necessarily yields a Steiner point at the point of intersection. In particular, the scheme of [18] is not applicable in the case of self-intersections.

[6]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.

3 Some Simple "Box" Methods

We implemented and tested our mesh-based algorithm, 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 implemented for purposes of comparison.

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.) 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.)

Processing a collision query is done by means of a two-phase approach: In phase I, we compute the 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 of the flying object, we identify the voxels intersected by $BB(T)$ and check T 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, currently being implemented and tested, 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.

3.2 A k-d Tree Method

Another 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.

Each node of the tree is associated with a hyperrectangle (box), and (implicitly) with the set of obstacle triangles that intersect that box. (The root node is associated with the workspace.) 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, K, 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.

In our implementation, we make choice (2) 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? (Note that all of the obstacle vertices for the associated triangles may fall outside the box.)

We make choice (1) in four different ways and choose[7] the best one: (a) minimize $\max\{n_1, n_2\}$; (b) minimize $|n_1 - n_2|$; (c) minimize $n_1 n_2$; and (d) divide the longest side of the box. Here, n_1 and n_2 are the numbers of the associated obstacle triangles of the two new problems created at the children.

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 5 in the runs reported here.

Collision queries are again done in a 2-phase approach, similar to the algorithm outlined above for the regular grid.

3.3 R-trees of Boxes

An alternative tree-based representation of obstacles is based on the idea of "R-trees" and their variants [1]. We use a binary tree version; multi-ary trees are also possible.

Each node of the tree corresponds to a rectangular box and a subset of the obstacle triangles. The root is associated with the workspace and all of the obstacles. Consider a set $\mathcal{T}$ of obstacle triangles associated with a node. If $|\mathcal{T}| \leq K$, where K is some threshold (whose value was a parameter in our experiments), the node is a leaf, and we store the triangles $\mathcal{T}$ at 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 3 choices of splits based on either[8] 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.

Note that, at any one level of the tree, each obstacle triangle is associated with only a single node. Collision queries are again done in a 2-phase approach, similar to the algorithm outlined above for the regular grid.

4 Experimental Analysis

4.1 Set-Up

Environments and Flying Objects: We recorded flight statistics for three different groups of obstacle environments called "tetras" (for "tetrahedra"), "scenes", and "terrains". The first two groups were generated by means of the random BSP-tree partition (described below), where a random subset of the leaves were either filled with simple objects (for the seven environments in the "scenes" group) or with tetrahedra (for the eight environments in the "tetras" group). For the "tetras", the numbers of tetrahedra ranged

[7] Results reported here used choice (c), which was selected after running several comparisons.

[8] Results reported here minimized $\max\{V_1, V_2\}$, which was selected after running several comparisons.

from 25 to 4000. For the "scenes", we used various test objects[9], such as chess pieces, mechanical parts, aircraft, and models of animals. The number of triangles of the "scenes" ranged from 1376 to 27600, with an average of about 20000 triangles.

The third group of environments ("terrains") was obtained from real elevation data. We sampled so-called "1-degree DEM" data[10], 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, where each surface contains 28322 triangles.

These three groups of environments were tested with six flying objects of various complexities. We used the following polyhedral models (listed according to increasing order of complexity): a tetrahedron, a soccer ball, a chess piece (bishop), a balloon, and two aircraft (spitfire and 747-200M). The numbers of triangles ranged from four (for the tetrahedron) to 14643 (for the 747-200M). 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.

Random Generation of Environments Ideally, for purposes 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. Thus, for this set of experiments, our approach 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 the 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.

Flights: For our experiments, we implemented a form of "billiard paths": the flying object F is moved along a random path, and it is allowed 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.

[9]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.

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

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, with each step, the new bounding box is obtained efficiently by a simple hill-climbing algorithm applied to the (precomputed) convex hull of F.

4.2 Results

4.2.1 Mesh Properties

Number of Steiner Points: The final number of vertices (after inserting all Steiner points in order to achieve a conforming mesh) seldom was more than twice the original number of vertices. We also gathered data for the meshes of some smaller objects which were known to have lots of self-intersections and similar deficiences. As expected, some of those objects needed many more Steiner points in order to obtain a conforming mesh.

Number of Tetrahedra: It is well-known that the Delaunay triangulation of n points in 3D can consist of $\Omega(n^2)$ tetrahedra. Fortunately, in our tests, we did not observe quadratic growth; rather, we obtained an experimental lower bound of $5.5n$ tetrahedra and an experimental upper bound of $6.5n$ tetrahedra for a Delaunay triangulation of n vertices of (sets of) real-world objects.

Stabbing Number of a Mesh: We also conducted a series of experiments in order to determine the average (line) stabbing number of our meshes. For each of 10K line segments (defined by pairs of random points inside the unit cube) per environment, we computed the total number of tetrahedra intersected by the segment (ignoring obstacles that it crosses), normalized by dividing by the length of the segment; the mean of these normalized counts was recorded for each environment. There was no clear correlation between the experimental line stabbing numbers, on one hand, and the numbers of vertices (resp., numbers of tetrahedra) of the environments, on the other hand. Roughly, for meshes consisting of 50K to 250K tetrahedra, the average stabbing number was 55, with 30 being the minimum and 100 being the maximum among our tests.

4.2.2 Collision Detection Experiments

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, and no optimizing compiler was used. However, the tests were fair because all four methods implemented rely on the same basic primitives. Thus, a potential speed-up gained by fine-tuning and optimizing can be expected to benefit all methods uniformly. All tests were run on a Silicon Graphics Indigo 2, with 128MB of main memory.

Survey: For each method and each environment/object pair, we determined "frame-rates" as follows. All four methods recorded statistics for a flight of the object along the same billiard path within the same environment, for 10K 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.

Based on these frame-rates, we counted how many times a method was the fastest[11] among the four methods; see Table 1. Summarizing[12], 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.

Wins Per Method				
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 1: Number of wins for each method.

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 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 2. 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.

Scores				
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 2: Weighted ranking for each method.

In order 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[13] collision check. Roughly, the ranking of our four algorithms according to the number of full-resolution checks confirmed the rankings given in Tables 1 and 2.

Mesh-Based Method: In our test it became evident 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; all the wins in the "scenes" group were achieved by flights of the aircraft models (spitfire and 747-200M).

It also became apparent that, not surprisingly, a low stabbing number of a tetrahedral mesh for line segments does not necessarily translate to a low stabbing number for a solid (box). In order to improve the stabbing number for solids flying within the meshes of the terrains we added a regular grid of $10 \times 10 \times 10$ Steiner points to the original vertices of the terrains; the statistics presented are based on these improved meshes. Roughly, improving the meshes in this way yielded a speed-up of about 20%.

We expect that adding a few Steiner points in judiciously chosen positions – rather than simply adding a grid of points – may improve the stabbing number for solids flying within tetrahedral meshes significantly, and thus also speed-up the collision detection process. This expectation is supported by the fact that the mesh-based method took the first place within the "tetras" group. (An advantage of this group is that the environments did not suffer from self-intersections, or other artifacts of "bad" data.) Furthermore, these environments also yielded meshes that were visually pleasing and of fairly low stabbing numbers for the bounding boxes of the flying objects.

Box-Based Methods: The Tables 1 and 2 clearly show that the R-tree method was the fastest for all three of our test environments. A close examination of all our experimental data, however, reveals that as the flying object increases in size, all three of the box methods mostly tend to converge to nearly the same frame-rate.

In optimizing the various parameters for these methods, we chose $N = 40$ for the grid method, and a threshold $K = 10$ for the 3-d tree. The R-tree was given a threshold $K = 1$; therefore, the number of triangles stored equals the number of original triangles.

5 Extensions

(1) *Dynamic collision detection:* While our current implementation is pseudo-dynamic, it is easy to approximate[14] a fully dynamic algorithm. Instead of checking at a discrete set of placements of F for intersection with any obstacle,

[11] In case of close ties, several methods were counted as winners.

[12] Detailed charts have been omitted from this extended abstract due to lack of space; a full paper with detailed charts can be obtained from the authors.

[13] Of course, every collision check included a bounding-box pretest.

[14] Exact methods of treating dynamic collision detection have been addressed by [2, 3], by considering the four-dimensional time-space problem or by modeling the configuration space exactly.

we can do a linear interpolation for motion between discrete placements. In particular, we can take the convex hull of two consecutive (discrete) placements F_i and F_{i+1} of F along its (continuous) motion, and check this hull for intersection with obstacles. As long as the spacing between placements is small (i.e., we maintain a high frame-rate), this approximation to computing a swept volume for F will be fairly good.

(2) *Nested hierarchies:* In our current implementation, we use a two-phase approach to collision detection — first checking for obstacle intersection with the bounding box of F, and then, if necessary, checking for obstacle intersection with each facet of F. Obviously, our method extends to *nested hierarchies* of the flying object, where we compute a set of nested approximations to F, $Q_0 = F \subset Q_1 \subset \cdots \subset Q_K$, with, say, Q_K being the bounding box of F, and each Q_i being a constant factor more complex than Q_{i+1}. Computing such a nesting is itself a challenging research problem; but our collision detection methods easily apply to any such nesting, and could, potentially, be substantially faster with even a 3- or 4-level nesting.

(3) *Multiple flying objects:* Checking a flying object for intersection with the environment yields, with little additional computation, for each cell of our subdivision a list of triangles of the flying object that (partially) occupy this cell. (Subdivision cells are either tetrahedra, in the case of the mesh method, or boxes, in the case of the other methods.) If we maintain different lists for different flying objects then a single pass through all cells with non-empty lists reveals all potential collisions. In particular, two triangles of two different flying objects need only be checked for collision if they occupy the same cell. We expect this scheme to significantly cut down the number of collision checks between triangles of multiple flying objects.

6 Conclusions

We have proposed and implemented a simple new collision detection method based on the principles of low stabbing number meshes, a data structure devised for efficient ray-shooting. We have compared our mesh-based method to three other methods implemented by us. The results of the experiments are mixed: In some cases, the mesh-based method wins, while in other cases it loses to all three of the others. These results suggest a few directions for continued research: (1) Improved methods for taking additional advantage of the information stored in the mesh, without spending too much time on book-keeping operations needed for obtaining this information; (2) Practical methods for generating meshes of low stabbing number — clearly, our Delaunay-based triangulation can be far from optimal; and, (3) Hybrid methods that can be engineered to take advantage of the situations when one method is superior to another.

Acknowledgement

Jai Chakrapani implemented the first version of the mesh-based algorithm. We thank Claudio Silva and Nikolai Zsikov for valuable assistance in the project. We also thank Ernst Mücke for supplying us with the source code for the Delaunay triangulation. Our work has greatly benefited from the support of of the VR group at Boeing, including Jeff Heiserman, David Mizell, Henry Sowizral, and Karel Zikan.

References

[1] 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 International Conference on Management of Data*, pages 322–331, 1990.

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

[3] J. Canny. Collision detection for moving polyhedra. *IEEE Trans. Pattern Anal. Mach. Intell.*, PAMI-8(2):200–209, 1986.

[4] J. D. Cohen, M. C. Lin, D. Manocha, and M. K. Ponamgi. Exact collision detection for interactive environments. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 391–392, 1994.

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

[6] H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graph.*, 9:66–104, 1990.

[7] A. Garcia-Alonso, N. Serrano, and J. Flaquer. Solving the collision detection problem. *IEEE Computer Graphics and Applications*, 14:36–43, May 1994.

[8] M. Held. Reliable C code for computing triangle-triangle intersections and triangle-segment intersections in 2D and 3D. Technical report, Applied Math, SUNY Stony Brook, June 1994.

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

[10] P. M. Hubbard. Interactive collision detection. In *Proc. IEEE Symposium on Research Frontiers in Virtual Reality*, pages 24–31, 1993.

[11] P. M. Hubbard. Space-time bounds for collision detection. Technical Report CS-93-04, Dept. of Computer Science, Brown University, February 1993.

[12] M. Lin and D. Manocha. Efficient contact determination between geometric models. *Internat. J. Comput. Geom. Appl.*, To appear.

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

[14] M. Moore and J. Willhelms. Collision detection and response for computer animation. *Comput. Graph.*, 22(4):289–298, August 1988.

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

[16] B. Naylor, J. A. Amatodes, and W. Thibault. Merging BSP trees yields polyhedral set operations. *Comput. Graph.*, 24(4):115–124, August 1990.

[17] H. Noborio, S. Fukuda, and S. Arimoto. Fast interference check method using octree representation. *Advanced Robotics*, 3(3):193–212, 1989.

[18] N. S. Sapidis and R. Perucchio. Domain Delaunay tetrahedrization of solid models. *Internat. J. Comput. Geom. Appl.*, 1(3):299–325, 1991.