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

Abstract

We consider the problem of preprocessing a scene of polyhedral models in order to perform collision detection very efficiently for an object that moves amongst obstacles. This problem is of central importance in virtual reality applications, where it is necessary to check for collisions at real-time rates.

We give an algorithm for collision detection that is based on the use of a mesh (tetrahedralization) of the free space that has (hopefully) low stabbing number. The algorithm has been implemented and tested, and we give experimental results comparing its performance against three other algorithms that we implemented, based on standard data structures.

*held@ams.sunysb.edu; Supported by NSF Grant DMS-9312098. On sabbatical leave from Universität Salzburg, Salzburg, Austria.

†jklosow@ams.sunysb.edu; Supported by NSF grants ECSE-8857642 and CCR-9204585, and by a grant from Boeing Computer Services.

‡jsbm@ams.sunysb.edu; Partially supported by NSF grants ECSE-8857642 and CCR-9204585, and by a grant from Boeing Computer Services.

1 Introduction

Virtual Reality refers to the use of computers to simulate a physical environment in such a way that humans can readily visualize, explore, and interact with the "objects" in the environment. Since physical environments are inherently geometric, many of the computational problems involved in designing and building a VR system are geometric in nature. One particularly important problem that must be addressed in order to make VR a reality is the problem of real-time interactive *intersection detection*: Determine if two virtual objects intersect each other.

Motivated by this need to do collision detection in virtual reality, we address the following problem: *Given a description of a geometric environment, $\mathcal{E}$, in three dimensions, preprocess it into a data structure of small size so that queries of the form, "Does object A intersect any of the obstacles in $\mathcal{E}$?" can be answered very rapidly.* Further, we study the problem of *tracking* the motion of A within $\mathcal{E}$, in order to detect dynamically, in real time, when A collides with an obstacle. The environment $\mathcal{E}$ is given to us as a *boundary representation* of a set of polyhedral obstacles, each of which is a component part that has been designed and modeled within a CAD system.

Our method for intersection 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 [10]; see also [2, 3, 8, 11, 14, 23, 26, 30].) 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 [7, 16, 21]. 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).

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 A, we then determine those tetrahedra intersecting each (triangular) facet of the object A, by using a simple search of the subdivision, in time proportional to the number of tetrahedra intersecting the facet. If, during this search, an obstacle facet is detected that intersects A, we stop and report collision.

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 based on computational geometry have a practical impact on the collision detection problem. We report on our implementations of other (straightforward) hierarchical collision detection methods, and we compare results in a series of experiments.

A practical contribution of this paper is our work on systematic experimentation for the collision detection problem: Rather than presenting a method and running it on a few "pet" problems, we have been doing a careful experimental analysis and comparison, using a wide range of data, both real and simulated. We have an enormous amount of data, and continue to gather more each day. Further, through our joint efforts with Boeing, we are involved in integrating our code within their VR system.

Relation with Previous Work

The problem of intersection detection is of fundamental importance in computer graphics, solid modeling, and virtual reality. Thus, there have been many approaches to solving the problem, including the use of binary space partition (BSP) trees [24], "R-trees" and their variants (see [4]), and octrees [22, 25].

One recent method for 3-dimensional intersection detection has been developed by Lin and Manocha [20] (see also [9]). 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 work includes [13] and the thesis work of Hubbard [17, 18], who uses approximations of objects (covering by disks) to speed up collision detection. Very recent papers include [19], [27], [31], and [32].

An Application to VR in Aircraft Manufacturing

We were drawn into this line of research when the Boeing VR group came to us with a problem: How can one do *real-time*, *interactive* collision detection in a virtual world whose complexity is on the order of many millions of features (polygons)? The huge datasets that were involved in the CAD data for the Boeing 777 project were beyond the capacity of any known method. Even the visualization problem is a major challenge, which has recently been fairly successfully attacked by Boeing's "FlyThru" system. The next step in the VR system being developed at Boeing is to insert an object (e.g., a human hand, or figure, or a tool) into the virtual scene, and to track the motion of the object (e.g., using any of various 3-dimensional motion sensors), while continuously checking for intersections between the object and the obstacles comprising the scene. Ultimately, this will require that an object (of say 5,000–10,000 polygons) be checked for intersection with a set of obstacles (of say 5-10 million polygons), at a rate of 20 or more frames per second.

Static versus Dynamic Collision Detection

It is common to distinguish between three types of collision detection:

- *static* collision detection, where one is to check whether some "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.

Of course, the path taken by the flying object is not known a priori in a VR application. This study mainly focusses on pseudo-dynamic collision detection. We have not yet implemented any dynamic collision detection, but point out in this paper how our methods can be extended to dynamic detection.

2 A Mesh-Based Algorithm

In this section, we give a collision detection method based on tracking the motion of a flying object within a tetrahedral mesh. We begin by describing how, for a given mesh, one does an intersection query. We then describe a method we have developed and implemented for generating conforming tetrahedral meshes.

2.1 Collision Detection within a Mesh

Suppose we are given a tetrahedral mesh $\mathcal{M}$, which conforms[1] to the obstacle environment, $\mathcal{E}$, and a flying object, F, which consists of a k-faceted polyhedron. Without loss of generality assume that all facets of the flying polyhedron are 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 deal with 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 bounding box of F, and enumerate all tetrahedra which lie partially inside the bounding box[2]. 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[3] whether or not F lies completely in the interior of some obstacle (or vice versa) can easily be carried out, too. 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 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.

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. [15]. 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 reasonably efficient.

Note that small (numerical) errors usually do 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

[1] I.e., the boundary of the obstacle environment corresponds to the union of some faces 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.

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

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

tetrahedron, or how often a point would lie on the boundary of a tetrahedron rather than in its interior. In both cases there is the potential danger of making inconsistent topological decisions due to numerical inaccuracy, which, in the worst case, may result in a 'looping' of the code.

Fortunately, it does not really matter for ray shooting and the execution of the algorithm whether, e.g., any of two tetrahedra is returned as that one which contains a point if this point actually lies somewhere on their common face – as long as at least one of them is returned. For the point-in-tetrahedron test this goal would be achieved if we could guarantee that the query point is classified consistently to lie on the same side of the supporting plane of the face shared by the two tetrahedra, no matter whether this classification is invoked as part of the point-in-tetrahedron test for the first tetrahedron or for the second tetrahedron. To solve this problem, we assign indices to the vertices of the environment and of the flying object and make sure that we always pass a list of vertices to one of our primitives such that the vertices appear in sorted order (with respect to their indices) in the list.

2.2 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 conform to $\mathcal{E}$. 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.

Our approach is essentially similar to the concepts outlined by Sapidis and Perucchio [29, 28]. 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.

Implementation Issues

We note that the robustness of the mesh generation is a major issue. First of all, real-world models tend to have a lot of "degeneracies" such as four or more points being coplanar. Thus, any assumption of "general position" is inappropriate. Furthermore, in both our and Sapidis and Perucchio's algorithm the Steiner points added are coplanar with already existing points (namely the vertices of obstacle triangles on which they happen to lie), which increases the number of coplanar points even more.

In order to achieve a robust generation of the Delaunay triangulation, Edelsbrunner and Mücke's concept of "simulation of simplicity", cf. [12], is applied. This concept has been implemented by Mücke and used for a randomized-incremental computation of Delaunay triangulations. Our own code for meshing is largely based on Mücke's implementation.

Apart from artificially introduced coplanarity, we also 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. And even worse, a lot of the models tend to have self-intersections, i.e., they contain (triangular) faces which intersect in places other than at their boundaries. Unfortunately, the same deficiences seem to be common among models generated by some widely used (commercial) CAD systems, too.

Inconsistent topologies of polyhedral models may be acceptable for some rendering algorithms which still produce stunning images (because they do not rely on or need consistent topologies), but they 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 by Sapidis and Perucchio is not applicable in the case of self-intersections. Unfortunately, according to our experience, self-intersections tend to consume many more Steiner points than those necessitated by the actual points of intersection.

As a final remark, we would like to mention that care has to be taken when classifying whether or not a face of a tetrahedron intersects an obstacle triangle. For instance, no Steiner point has to be inserted if the two entities intersect only at a vertex. For this reason, our primitives for computing triangle-edge and triangle-triangle intersections not only compute intersections, but also classify them appropriately.

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 have also been implemented and tested, and which were used for comparison purposes in our experimental results (described in the next section).

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 box. A single obstacle triangle may be stored in the lists of many boxes; thus, the size of the resulting data structure could be large in comparison with the size of the input. (This issue is 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).

For a given query, we compute the bounding box, $BB(F)$, of the flying object, and we easily determine the set of grid boxes 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 grid boxes. 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 we check T against each (triangular) facet of F, stopping if we find an intersection. In the worst case, we could end up checking every obstacle triangle against every facet of F.

An alternative method, currently being implemented and tested, is, for a given query F, to compute the set of grid boxes intersected by F (e.g., by doing a 3-dimensional scan conversion of F, in time proportional to the number of grid boxes intersected by F), and to test, for each obstacle triangle in the associated lists, for intersection with F.

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, 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.

To process a query q (which is a region being tested for intersection with an obstacle), 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 2-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 $q = T$, for each triangular facet T of the flying object F, checking at the leaves for intersection between T 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 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 "sizes" (number of 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.

3.3 R-trees of Boxes

An alternative tree-based representation of obstacles is based on the simple idea of "R-trees" and their variants [4]. 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.

We describe 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 entire workspace and all of the obstacles. Consider a node having associated set $\mathcal{T}$ of obstacle triangles. If $|\mathcal{T}| \leq K$, where K 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 3 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.

4 Experimental Results

4.1 Set-up of Our Tests

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[4] such as chess pieces, mechanical parts, aircrafts, 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[5], 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.

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 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

[4] 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.

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

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, 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 Data

4.2.1 Mesh Properties

Number of Steiner Points: In Fig. 1 we plot the ratio of the final number of vertices (after inserting all Steiner points in order to achieve a conforming mesh) over the original number of vertices. As it can be seen, the final number of vertices seldom was more than twice the orginal number of vertices. Note that most of the environments in the "scenes" group suffered from obstacle objects with (a few) self-intersections. We also gathered data for some smaller objects which were known to have lots of self-intersections and similar deficiences, see Fig. 2. 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, this bound was not realized in any of our tests. Rather, we obtained an experimental bound of $6.5n$ tetrahedra for a Delaunay triangulation of n vertices of (sets of) real-world objects; see Table 1 (where the number of tetrahedra is denoted by $\#(M)$) and Fig. 3.

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 10,000 line segments (defined by pairs of random points inside the unit cube) per environment, we computed the numbers of tetrahedra that were 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. 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. See column "*#(stab)*" of Table 1 and Fig. 4.

4.2.2 Collision Detection Experiments

Survey: 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 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 128M of main memory. Note that the time consumed by rendering is not included.

Tables 5–7 present, in a condensed form, the flight statistics gathered for the four methods tested. For each method and each environment, we list the speed-rate and one additional item which covers some combinatorial complexity inherently connected with the method. The speed-rates listed have been determined as follows: for each scene/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. Apparently, the average frame-rate highly depends on the number of full-resolution collision checks, 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 with respect to the number of actual full-resolution collision checks. In the Tables 5–7, the column f/s shows the hypothetical average frame-rate for a flight where 5% of all steps give rise to a full-resolution collision[6] check.

Based on these normalized frame-rates, we counted how many times a method was the fastest[7] among the four methods; see Table 3. 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 pair, we a ssigned 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.

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 747-200M).

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.

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 the Tables 5–7, n_O denotes the average number of tetrahedra occupied by the bounding box of the flying object). 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. (Another 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: Tables 5–7 clearly show that the R-tree method was the fastest for all three of our test environments. In these tables, columns "c_V" and "n_V" refer to the average number of cells and the average number of nodes visited per step in the respective methods. Closer examination, 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. The numbers of triangles stored for the other

[6] This translates to 500 full-resolution checks for 10,000 steps; our flights averaged 532 full-resolution collision checks and 35 actual collisions.

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

two methods are given in the columns "$\#(T_{st})$" in our tables. The columns "*Hght*" refer to the heights of the various trees.

Full-Resolution Checks per Second Apart from the importance of intersection detection for VR applications, intersection detection also is of vital importance for computer graphics and CAD/CAM, mostly in the form of a static collision check. Thus, 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[8] 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.

5 Extensions

Several additional experiments are currently under way; in the full paper, we will have data to report on their results, including the following directions for extending our methods:
(1) *Dynamic collision detection:* While our current implementation is pseudo-dynamic, it is easy to approximate a fully dynamic algorithm. Instead of checking at a discrete set of placements of F for intersection with an obstacle, we can do a linear interpolation for motion between discrete placements. In particular, we can take the convex hull of F_i and F_{i+1}, for two consecutive (discrete) placements of F along its (continuous) motion, and check this hull for intersections 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 a very good one. (Exact methods of treating dynamic collision detection have been addressed by [5, 6], by considering the four-dimensional time-space problem or by modeling the configuration space exactly.)
(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. We are currently adapting surface simplification algorithms to be able to conduct experiments of this sort. The goal is then to develop a three-dimensional version of the algorithm of Mount [23], which is "query-sensitive" to, say, the number of facets required to separate the query object from the obstacles.
(3) *Multiple flying objects:* A direct method of allowing multiple flying objects is to do an all-pairs check for collisions among them, in addition to checking each flying object for collision with the environment. As long as there are not too many simultaneously flying objects or as long as all flying objects are of small complexity, the quadratic performance hit would not be too severe. However, one can do better. Checking a flying object for intersection with the environment yields, at basically no additional computational cost, for every cell of our subdivisions a list of triangles of the flying object which (partially) occupy this cell (which either is a tetrahedron in the case of the mesh or a box in the case of the box-based methods). If we

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

maintain different lists for the 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.
(4) *Hybrid methods:* Instead of subdividing the workspace into tetrahedra that are required to conform fully to the obstacles, we may find it more efficient to apply the mesh method to a coarse partitioning of the workspace, and then to use a different collision detection method (e.g., one of the simple box methods) within each subpiece of the workspace. Essentially, the grid-based method is doing something like this — partitioning the workspace into a few boxes, and then doing a brute-force search within each box. Hybrid methods of dealing with the data are also crucial to being able to handle really large datasets, where not all of the obstacle space can be held in memory at once. There remains a substantial amount of experimental research to determine an optimal hybrid mixture of methods.
(5) *Inserting/deleting obstacles:* It will be important in some applications for us to be able to insert and delete obstacles, without expending too much time recomputing the data structures.

6 Conclusions

We have proposed and implemented a simple new collision detection method based on the principles of low stabbing number meshes, which have been put forth in the computational geometry literature. This paper is really an experimental study of the practicality of this method.

We have compared our mesh-based method to three other methods we devised, implemented, and tuned. 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) Further experimentation on more and larger scenes, as well as integration into the VR system of Boeing; (2) Improved methods for generating meshes of low stabbing number — clearly, our Delaunay-based triangulation is not necessarily the best method. We are currently implementing a new method, based on the query-sensitive approach of [21], and hope to report the results of these experiments soon. (3) Hybrid methods that can be engineered to take advantage of the situations when one method is superior to another.

Acknowledgement

Jai Chakrapani is responsible for implementing the original version of the mesh-based algorithm described herein. We also thank Claudio Silva and Nikolai Zsikov for several useful discussions and for assistance with the software. We thank Ernst Mücke for supplying us with source code for the Delaunay triangulation (prior to making it publically available).

This project would not have been pursued without the collaboration and support of the VR group at Boeing. We thank Jeff Heiserman, David Mizell, Henry Sowizral, and Karel Zikan for their cooperation and collaboration on this project.

References

[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.*, pages 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*, pages 260–270, 1993.

[4] 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.

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

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

[7] B. Chazelle, H. Edelsbrunner, M. Grigni, L. Guibas, J. Hershberger, M. Sharir, and J. Snoeyink. Ray shooting in polygons using geodesic triangulations. In *Proc. 18th Internat. Colloq. Automata Lang. Program.*, volume 510 of *Lecture Notes in Computer Science*, pages 661–673. Springer-Verlag, 1991.

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

[9] 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.

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

[11] 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.*, pages 21–30, 1991.

[12] 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.

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

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

[15] 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.

[16] 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.

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

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

[19] P.M. Hubbard. Real-time Collision Detection and Time-critical Computing. In *Proc.* 1^{st} *Workshop on Simulation and Interaction in Virtual Environments*, U. of Iowa, July 1995.

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

[21] 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.

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

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

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

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

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

[27] M. Ponamgi, J. Cohen, M. Lin, and D. Manocha. Incremental Collision Detection for Polygonal Models. In *Proc.* 1^{st} *Workshop on Simulation and Interaction in Virtual Environments*, U. of Iowa, July 1995.

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

[29] N. S. Sapidis and R. Perucchio. Delaunay triangulation of arbitrarily shaped planar domains. *Comput. Aided Geom. Design*, 8(6):421–437, December 1991.

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

[31] G. Vanecek and C. Gonzalez-Ochoa. Representing Complex Objects in Collision Detection. In *Proc.* 1^{st} *Workshop on Simulation and Interaction in Virtual Environments*, U. of Iowa, July 1995.

[32] G. Zachmann and W. Felger. The BoxTree: Enabling Real-time and Exact Collision Detection of Arbitrary Polyhedra.

Environment Characteristics								
			Mesh		Grid	R-Tree	3d-Tree	
env. name	$\#(V)$	$\#(T)$	$\#(M)$	$\#(stab)$	$\#(T_{st})$	*Hght*	*Hght*	$\#(T_{st})$
buffalo-w	14400	28322	95714	57	50897	15	33	100383
denver-e	14400	28322	95908	55	51849	15	33	148528
denver-w	14400	28322	182507	59	62563	15	33	150360
eagle_pass-e	14400	28322	95659	54	51817	15	30	145789
grand_canyon-e	14400	28322	224779	61	61940	15	36	150425
jackson-w	14400	28322	109207	54	51711	15	35	127642
moab-w	14400	28322	144829	53	58045	15	33	151884
seattle-e	14400	28322	164679	53	68271	15	32	139390
25tetra	100	100	1164	18	12298	7	11	270
50tetra	200	200	1858	28	20584	8	12	519
100tetra	400	400	5482	38	37237	9	17	1761
250tetra	1000	1000	8016	33	9544	10	19	1804
500tetra	2000	2000	15776	41	10662	11	24	4220
1000tetra	4000	4000	33900	46	21676	12	27	9346
2000tetra	8000	8000	73345	70	30191	13	27	20163
4000tetra	15996	16000	147272	89	49881	14	30	40374
s9.20rp	5680	11200	58625	56	26178	14	35	84253
s20.40rp	11360	22400	134090	80	76884	15	35	181021
s22.150room	10492	13800	73430	38	17740	14	42	45107
s23.300room	20995	27600	146240	47	34395	15	40	79816
s16.6easy	11604	22092	107048	99	71039	15	34	135833
s15.4cows	11612	23216	167408	31	33422	15	44	146502
s10.2each	748	1376	6920	22	5507	11	31	8933

Table 1: Characteristics of the Test Environments.

Object Complexities			
object name	$\#(V)$	$\#(T)$	$\#(V_{CH})$
tetra	4	4	4
socbal	60	116	60
bishop	250	496	41
balloon	1141	2094	395
spitfire	2479	5152	51
747-200M	7594	14643	499

Table 2: Complexities of the Flying Objects.

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 3: Number of Wins for Each Method.

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 4: Weighted Ranking for Each Method.

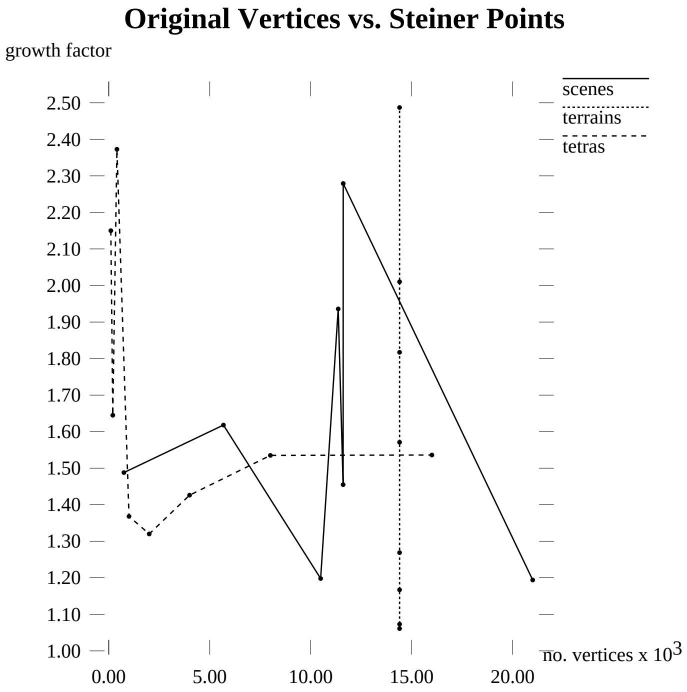

Figure 1: Growth Rate of Number of Vertices.

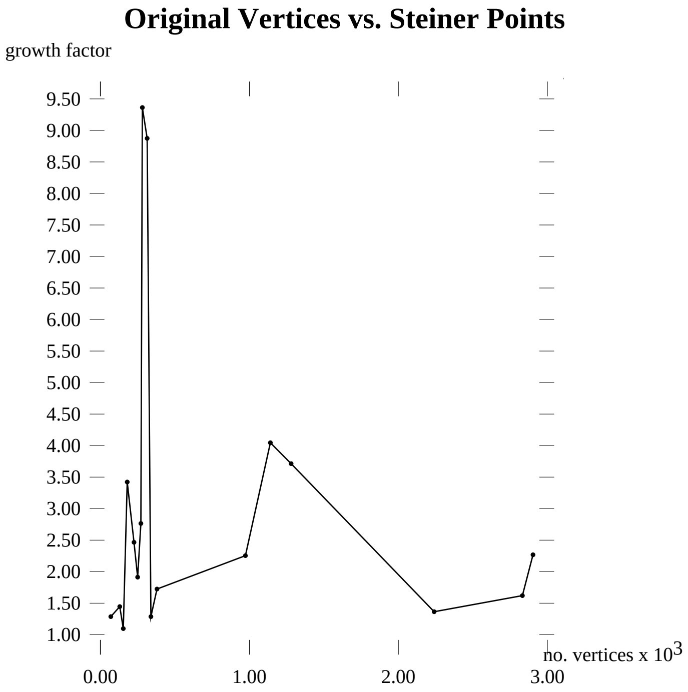

Figure 2: Growth Rate of Number of Vertices (Small Objects).

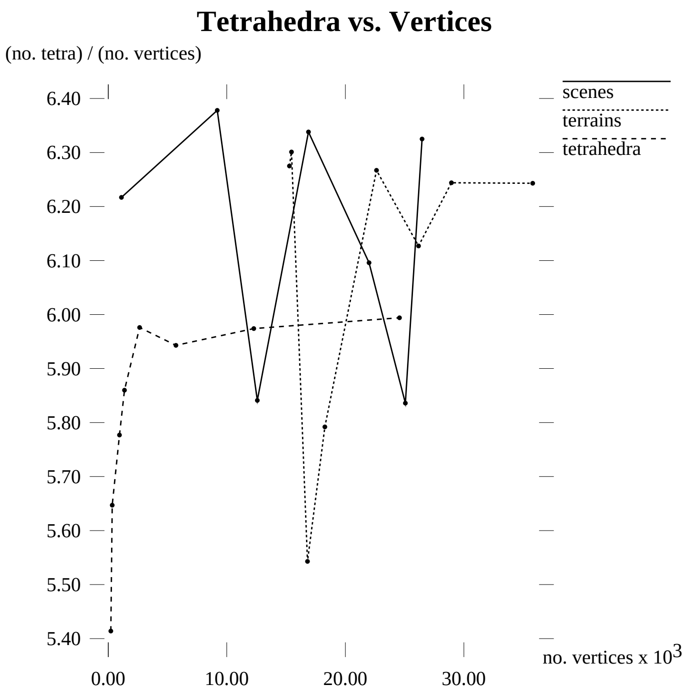

Figure 3: Tetrahedra versus Vertices.

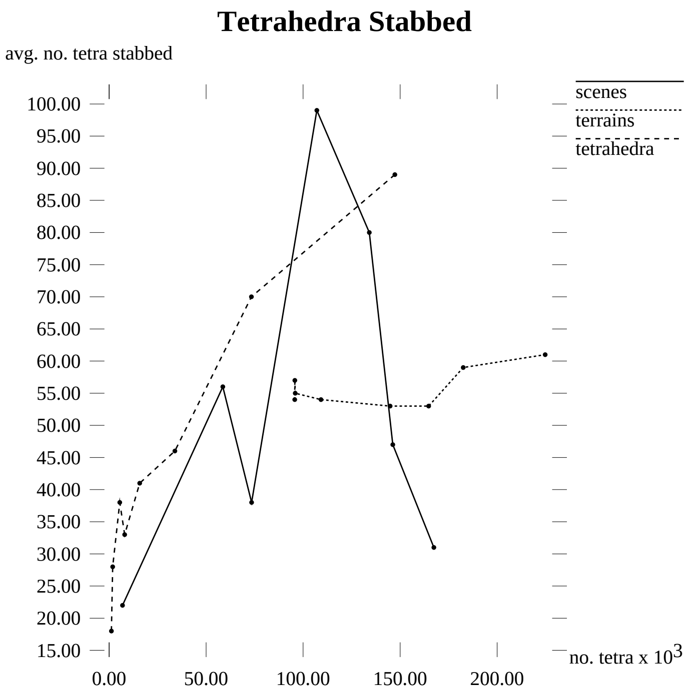

Figure 4: Average Line Stabbing Number.

Flight Statistics												
	Mesh			Grid			R-Tree			3d-Tree		
env. name / obj. name	f/s	f_N/s	n_O	f/s	f_N/s	c_V	f/s	f_N/s	n_V	f/s	f_N/s	n_V
s9.20rp / tetra	1122.3	814.8	61	4444.4	3226.7	38	5882.4	4270.6	12	5181.3	3761.7	24
s9.20rp / socbal	909.1	567.3	53	1964.6	1225.9	33	2590.7	1616.6	28	1934.2	1207.0	67
s9.20rp / bishop	742.4	218.3	44	1224.0	359.9	21	1186.2	348.8	55	753.0	221.4	99
s9.20rp / balloon	209.1	97.9	51	242.6	114.0	71	271.1	127.4	236	190.9	89.7	596
s9.20rp / spitfire	95.5	45.3	51	96.1	45.7	126	96.4	45.9	809	88.7	42.2	1491
s9.20rp / 747-200M	48.1	15.4	47	37.5	12.0	230	38.9	12.4	1901	33.9	10.8	2967
s20.40rp / tetra	874.1	2007.0	74	2702.7	6232.4	41	2873.6	6626.4	27	2881.8	6645.5	48
s20.40rp / socbal	708.2	1301.7	59	954.2	1757.6	48	994.0	1831.0	91	846.0	1558.4	175
s20.40rp / bishop	565.3	482.8	45	648.9	555.5	38	473.3	405.1	202	485.4	415.5	314
s20.40rp / balloon	87.5	149.9	56	97.5	168.2	209	66.9	115.5	1541	77.6	133.9	2206
s20.40rp / spitfire	57.5	90.4	54	49.5	77.9	393	32.2	50.7	3270	30.4	47.8	4790
s20.40rp / 747-200M	22.7	29.7	55	19.1	25.2	859	12.9	17.0	8109	13.7	18.0	11290
s22.150room / tetra	925.9	625.9	67	3076.9	2080.0	37	3367.0	2276.1	19	4201.7	2840.3	23
s22.150room / socbal	1340.5	136.7	38	3861.0	393.8	26	5291.0	539.7	7	4878.0	497.6	13
s22.150room / bishop	728.3	125.3	47	1536.1	264.2	17	1531.4	263.4	35	1006.0	173.0	56
s22.150room / balloon	124.7	19.5	61	347.0	54.1	35	452.1	70.5	81	414.6	64.7	198
s22.150room / spitfire	89.0	9.3	42	140.8	14.6	38	231.1	24.0	128	203.6	21.2	297
s22.150room / 747-200M	39.2	4.4	43	54.2	6.1	82	64.5	7.2	688	64.2	7.2	861
s23.300room / tetra	786.2	445.0	83	2732.2	1546.4	36	3311.3	1874.2	22	3846.2	2176.9	20
s23.300room / socbal	581.4	174.4	55	2681.0	804.3	29	3048.8	914.6	23	2331.0	699.3	37
s23.300room / bishop	483.1	78.3	55	1177.9	190.8	17	1512.9	245.1	39	1264.2	204.8	63
s23.300room / balloon	79.5	32.8	115	245.2	101.0	65	258.7	106.6	275	237.6	97.9	520
s23.300room / spitfire	43.5	11.0	76	99.3	25.0	76	147.7	37.2	401	117.4	29.6	762
s23.300room / 747-200M	33.4	5.2	55	56.5	8.8	115	61.8	9.6	753	49.4	7.7	1315
s16.6easy / tetra	606.1	1343.0	121	2941.2	6764.7	42	2451.0	5637.3	35	1941.7	4466.0	38
s16.6easy / socbal	598.8	952.1	86	1084.6	1828.6	47	762.8	1286.0	122	949.7	1601.1	132
s16.6easy / bishop	377.5	469.6	80	385.8	557.1	59	199.4	288.0	487	265.5	383.3	442
s16.6easy / balloon	113.4	202.8	84	84.8	167.7	242	40.5	80.1	2469	55.5	109.8	2165
s16.6easy / spitfire	74.6	83.2	89	58.1	68.4	300	28.8	33.9	3431	38.6	45.4	3270
s16.6easy / 747-200M	25.8	25.3	79	22.0	23.0	704	12.3	12.8	7758	16.5	17.2	8419
s15.4cows / tetra	880.3	197.2	82	4587.2	1027.5	35	10526.3	2357.9	4	8695.7	1947.8	11
s15.4cows / socbal	1221.0	7.3	51	4132.2	33.1	25	7874.0	63.0	1	7751.9	62.0	3
s15.4cows / bishop	739.1	47.3	58	1512.9	96.8	14	2309.5	147.8	10	1193.3	76.4	28
s15.4cows / balloon	298.0	20.3	62	504.5	34.3	26	561.5	38.2	27	279.2	19.0	88
s15.4cows / spitfire	193.7	13.6	67	247.0	17.8	30	249.8	18.0	84	245.5	17.7	208
s15.4cows / 747-200M	66.6	7.3	61	64.2	7.6	86	71.4	8.4	540	60.7	7.2	1026
s10.2each / tetra	1960.8	278.4	35	6849.3	986.3	35	16666.7	2400.0	2	14705.9	2117.6	8
s10.2each / socbal	1838.2	55.1	29	3846.2	115.4	25	7633.6	229.0	1	6944.4	208.3	7
s10.2each / bishop	1215.1	14.6	30	2178.6	26.1	12	2666.7	32.0	2	2666.7	32.0	7
s10.2each / balloon	466.4	33.6	31	538.5	38.8	26	518.4	37.3	49	504.0	36.3	70
s10.2each / spitfire	209.7	24.3	36	227.7	26.4	42	211.1	24.5	144	198.3	23.0	286
s10.2each / 747-200M	79.0	9.9	33	69.2	8.7	100	66.5	8.4	493	60.8	7.7	937

Table 5: Flight Statistics (Scenes).

Flight Statistics												
	Mesh			Grid			R-Tree			3d-Tree		
env. name / obj. name	f/s	f_N/s	n_O	f/s	f_N/s	c_V	f/s	f_N/s	n_V	f/s	f_N/s	n_V
buffalo-w / tetra	500.8	440.7	153	2544.5	2249.4	37	3891.1	3439.7	15	275.4	243.5	28
buffalo-w / socbal	327.3	609.5	124	650.6	1221.9	44	1050.4	1972.7	78	54.6	102.6	225
buffalo-w / bishop	256.8	211.6	107	271.3	225.2	34	770.4	639.4	84	93.7	77.8	251
buffalo-w / balloon	94.1	85.8	120	181.6	166.7	96	216.0	198.3	353	93.7	86.0	841
buffalo-w / spitfire	55.0	46.7	59	66.4	57.1	202	72.8	62.6	1212	55.2	47.5	2636
buffalo-w / 747-200M	43.7	16.1	46	39.4	14.8	206	53.8	20.2	919	32.6	12.2	2465
denver-e / tetra	894.5	982.1	82	2907.0	3267.4	38	4902.0	5509.8	12	4184.1	4702.9	26
denver-e / socbal	1111.1	280.0	43	3164.6	797.5	27	6024.1	1518.1	4	4132.2	1041.3	22
denver-e / bishop	423.9	250.1	49	578.7	341.4	28	1094.1	645.5	60	736.4	434.5	172
denver-e / balloon	181.1	103.6	46	234.0	133.9	68	319.4	182.7	173	193.5	110.7	527
denver-e / spitfire	57.0	48.8	49	59.3	50.8	200	79.8	68.3	1068	67.2	57.5	2464
denver-e / 747-200M	25.0	20.1	45	25.3	20.7	473	30.9	25.2	2485	23.3	19.0	6584
denver-w / tetra	727.3	1552.0	88	1923.1	4130.8	42	2227.2	4784.0	29	2197.8	4720.9	47
denver-w / socbal	569.2	960.7	56	905.8	1554.3	47	1216.5	2087.6	75	797.4	1368.4	173
denver-w / bishop	522.2	334.2	29	632.9	415.2	32	777.0	509.7	105	646.0	423.8	234
denver-w / balloon	82.0	103.3	40	108.6	138.2	157	114.3	145.4	814	94.9	120.7	1679
denver-w / spitfire	45.6	45.0	32	55.4	55.6	244	55.0	55.2	1727	48.0	48.1	3273
denver-w / 747-200M	10.9	17.2	45	12.7	20.2	967	13.6	21.7	7354	12.7	20.3	14323
eagle_pass-e / tetra	772.2	665.6	95	2949.9	2542.8	38	3787.9	3265.2	11	4739.3	4085.3	26
eagle_pass-e / socbal	691.1	716.0	66	1158.7	1207.4	38	2375.3	2475.1	30	1264.2	1317.3	103
eagle_pass-e / bishop	384.5	259.9	42	485.2	329.0	33	860.6	583.5	89	646.0	438.0	247
eagle_pass-e / balloon	144.3	94.6	48	186.8	122.5	81	297.9	195.4	187	186.4	122.3	654
eagle_pass-e / spitfire	64.5	62.1	47	53.3	51.2	218	47.6	45.8	956	65.6	63.1	2646
eagle_pass-e / 747-200M	31.0	24.4	45	29.0	22.8	432	37.4	29.4	1839	29.6	23.3	5354
grand_canyon-e / tetra	607.5	942.9	100	1612.9	2506.5	41	2164.5	3363.6	29	2207.5	3430.5	44
grand_canyon-e / socbal	661.4	877.0	63	1183.4	1588.2	40	1369.9	1838.4	51	1243.8	1669.2	125
grand_canyon-e / bishop	286.6	244.8	41	536.8	472.4	39	666.2	586.3	127	556.8	490.0	307
grand_canyon-e / balloon	57.5	85.8	62	50.3	78.8	192	101.3	158.8	950	88.3	138.5	2055
grand_canyon-e / spitfire	39.9	48.6	44	51.1	62.4	274	61.8	75.5	1416	48.0	58.6	3471
grand_canyon-e / 747-200M	17.7	17.6	45	21.3	21.8	593	22.9	23.5	3862	19.9	20.4	8131
jackson-w / tetra	825.1	493.4	84	3300.3	2000.0	36	5405.4	3275.7	10	5025.1	3045.2	19
jackson-w / socbal	738.0	646.5	68	1589.8	1405.4	35	2907.0	2569.8	22	2028.4	1793.1	72
jackson-w / bishop	592.1	311.4	29	722.5	382.9	26	1153.4	611.3	54	839.6	445.0	148
jackson-w / balloon	113.8	74.9	50	207.4	137.7	81	270.4	179.6	232	189.7	125.9	657
jackson-w / spitfire	65.5	43.3	37	92.2	61.0	151	109.7	72.6	644	91.1	60.3	1635
jackson-w / 747-200M	34.6	19.2	35	32.8	18.2	259	47.5	26.4	1165	39.9	22.2	3195
moab-w / tetra	829.2	1313.4	84	2369.7	3772.5	39	3246.8	5168.8	20	3115.3	4959.5	37
moab-w / socbal	927.6	712.4	42	1600.0	1248.0	35	2475.2	1930.7	29	1901.1	1482.9	75
moab-w / bishop	394.8	300.0	34	453.5	347.4	36	689.7	528.3	122	572.7	438.7	280
moab-w / balloon	92.5	82.0	39	148.6	134.7	121	171.5	155.4	485	130.9	118.6	1167
moab-w / spitfire	30.5	38.0	44	42.9	54.1	313	54.0	68.1	1748	39.9	50.3	4116
moab-w / 747-200M	22.4	19.8	33	24.1	21.4	547	28.8	25.6	2649	23.7	21.1	6759
seattle-e / tetra	967.1	1431.3	63	2421.3	3646.5	38	3802.3	5726.2	17	3257.3	4905.5	32
seattle-e / socbal	673.4	940.1	55	1100.1	1584.2	42	1730.1	2491.3	46	1013.2	1459.0	130
seattle-e / bishop	370.0	421.8	40	436.7	503.1	40	730.5	841.5	104	496.5	572.0	327
seattle-e / balloon	96.4	61.3	38	181.0	115.8	84	266.9	170.8	230	189.5	121.3	787
seattle-e / spitfire	17.3	33.3	55	32.2	63.9	459	30.7	61.0	3395	28.1	55.7	6229
seattle-e / 747-200M	19.7	13.4	42	21.5	14.7	422	35.8	24.6	1928	28.0	19.2	5272

Table 6: Flight Statistics (Terrains).

Flight Statistics												
	Mesh			Grid			R-Tree			3d-Tree		
env. name / obj. name	f/s	f_N/s	n_O	f/s	f_N/s	c_V	f/s	f_N/s	n_V	f/s	f_N/s	n_V
25tetra / tetra	3021.1	3939.6	18	5000.0	7310.0	39	9259.3	13537.0	6	9434.0	13792.5	8
25tetra / socbal	1736.1	1788.2	17	1626.0	1775.6	40	1468.4	1603.5	41	1510.6	1649.5	47
25tetra / bishop	651.0	673.2	17	390.5	432.6	48	329.5	365.1	203	358.8	397.6	189
25tetra / balloon	175.1	207.3	17	109.6	136.1	174	83.3	103.4	825	81.2	100.8	866
25tetra / spitfire	62.0	99.9	16	33.7	59.2	473	23.2	40.7	3079	25.7	45.1	2853
25tetra / 747-200M	33.4	26.4	17	19.8	16.7	651	13.0	11.0	5315	14.5	12.3	4210
50tetra / tetra	2155.2	7534.5	22	3663.0	12981.7	46	4651.2	16483.7	14	5208.3	18458.3	13
50tetra / socbal	998.0	2934.1	18	694.9	2093.1	68	593.5	1787.5	131	657.0	1979.0	129
50tetra / bishop	420.9	809.8	17	292.9	575.9	77	173.7	341.6	432	215.0	422.6	360
50tetra / balloon	111.0	242.3	19	69.3	159.0	306	47.1	108.1	1748	48.7	111.8	1689
50tetra / spitfire	41.2	102.0	17	24.8	63.9	698	14.5	37.2	5881	17.7	45.6	4981
50tetra / 747-200M	27.4	30.3	20	16.5	19.3	845	10.9	12.7	6405	13.4	15.7	5587
100tetra / tetra	1901.1	8619.8	27	2832.9	14141.6	48	3389.8	16922.0	21	4201.7	20974.8	18
100tetra / socbal	739.6	3146.4	25	433.8	2102.4	94	263.8	1278.3	351	344.9	1671.6	265
100tetra / bishop	302.8	959.7	22	149.2	541.8	129	82.1	298.1	980	97.0	352.3	801
100tetra / balloon	138.4	258.8	23	72.1	153.8	285	39.9	85.2	2132	50.6	108.0	1867
100tetra / spitfire	43.3	116.9	20	20.5	62.2	795	11.7	35.5	7619	15.1	45.7	6217
100tetra / 747-200M	14.6	38.4	23	7.1	21.3	2191	3.7	11.1	23725	5.0	15.1	18359
250tetra / tetra	2145.9	4240.3	28	4524.9	9086.0	41	6711.4	13476.5	10	6993.0	14042.0	12
250tetra / socbal	1305.5	1885.1	22	1053.7	1591.1	47	932.8	1408.6	77	1191.9	1799.8	65
250tetra / bishop	1005.0	496.5	21	909.9	476.8	29	691.6	362.4	97	758.2	397.3	118
250tetra / balloon	222.3	195.2	22	149.1	132.4	129	115.2	102.3	613	113.9	101.1	685
250tetra / spitfire	90.3	102.7	25	55.3	71.0	357	35.8	45.9	2352	37.5	48.2	2961
250tetra / 747-200M	38.9	35.4	27	22.6	22.6	734	14.8	14.8	5152	16.1	16.1	6606
500tetra / tetra	1647.4	2197.7	37	5291.0	7079.4	39	7092.2	9489.4	10	7462.7	9985.1	15
500tetra / socbal	1123.6	2476.4	35	1206.3	2680.3	52	907.4	2016.3	92	988.1	2195.7	134
500tetra / bishop	742.4	555.3	29	687.8	553.0	38	451.5	363.0	180	517.9	416.4	210
500tetra / balloon	253.4	175.8	30	194.3	138.3	103	141.8	100.9	520	103.4	73.6	683
500tetra / spitfire	90.6	102.6	34	62.2	76.5	323	39.9	49.1	2258	42.1	51.8	3184
500tetra / 747-200M	39.4	32.6	33	27.6	23.1	589	17.0	14.2	4724	20.0	16.7	5599
1000tetra / tetra	1222.5	4369.2	51	3436.4	12343.6	47	3448.3	12386.2	24	3831.4	13762.5	27
1000tetra / socbal	855.4	2249.8	50	1050.4	2821.4	60	675.2	1813.6	152	723.6	1943.6	186
1000tetra / bishop	338.1	726.2	46	330.1	709.8	78	172.6	371.1	601	205.7	442.3	632
1000tetra / balloon	137.8	256.5	35	60.2	112.3	247	47.3	88.3	2047	65.5	122.3	1669
1000tetra / spitfire	86.6	103.2	39	61.3	73.4	312	40.1	48.1	2310	39.8	47.7	3211
1000tetra / 747-200M	43.9	26.8	44	31.2	19.2	445	20.9	12.9	3857	23.0	14.2	4240
2000tetra / tetra	712.8	3388.5	97	2164.5	10350.6	47	1901.1	9091.3	43	2100.8	10046.2	49
2000tetra / socbal	1040.6	1444.3	43	1457.7	2134.1	44	1182.0	1730.5	73	1302.1	1906.2	106
2000tetra / bishop	341.3	824.6	48	305.8	767.0	92	113.8	285.4	818	187.8	471.1	761
2000tetra / balloon	104.2	254.2	56	84.1	211.9	311	38.7	97.5	2662	51.2	129.1	2793
2000tetra / spitfire	55.5	114.4	47	39.3	83.9	546	20.2	43.0	5050	25.4	54.2	5600
2000tetra / 747-200M	21.7	42.1	55	14.7	29.5	1395	7.1	14.2	14671	9.2	18.5	16350
4000tetra / tetra	362.2	901.1	178	2631.6	6557.9	36	3906.2	9734.4	18	3663.0	9128.2	35
4000tetra / socbal	460.4	2060.8	78	599.9	2827.8	84	295.1	1391.0	367	384.8	1813.8	379
4000tetra / bishop	217.2	734.7	79	234.5	800.0	115	117.2	400.0	963	144.7	493.7	1074
4000tetra / balloon	94.5	172.1	101	100.8	183.5	216	101.0	183.7	896	81.8	148.9	2092
4000tetra / spitfire	32.6	111.7	70	26.8	92.9	875	13.5	46.9	8227	16.0	55.5	10410
4000tetra / 747-200M	22.3	37.4	150	16.2	28.0	1215	6.6	11.5	12533	10.7	18.5	14444

Table 7: Flight Statistics (Tetras).

Full Resolution Checks per Second					
env. name	fly. obj.	R-tree	Grid	3d-Tree	Mesh
eagle_pass-e	tetra	477.3	709.2	743.5	92.3
eagle_pass-e	socbal	293.7	183.3	209.9	66.3
eagle_pass-e	bishop	44.5	28.7	45.5	16.8
eagle_pass-e	balloon	114.0	63.2	67.2	37.1
eagle_pass-e	spitfire	20.6	16.4	19.1	6.7
eagle_pass-e	747-200M	16.3	8.6	15.9	4.8
grand_canyon-e	tetra	492.6	809.7	790.5	99.1
grand_canyon-e	socbal	379.5	240.1	278.9	88.3
grand_canyon-e	bishop	48.3	25.5	47.1	25.4
grand_canyon-e	balloon	24.9	33.5	36.3	9.9
grand_canyon-e	spitfire	17.1	19.9	21.4	7.4
grand_canyon-e	747-200M	6.4	7.6	8.3	3.4
1000tetra	tetra	1123.6	1342.3	1739.1	481.9
1000tetra	socbal	309.6	306.7	265.6	282.5
1000tetra	bishop	69.8	72.1	47.4	109.2
1000tetra	balloon	9.7	17.9	12.1	27.4
1000tetra	spitfire	7.1	14.2	6.4	24.2
1000tetra	747-200M	7.5	20.4	15.0	32.1
4000tetra	tetra	1342.3	2150.5	2197.8	187.6
4000tetra	socbal	277.4	189.9	188.7	92.9
4000tetra	bishop	48.8	63.8	57.1	82.5
4000tetra	balloon	37.2	32.5	30.9	10.9
4000tetra	spitfire	10.0	9.3	7.4	3.8
4000tetra	747-200M	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	747-200M	10.7	15.3	15.5	18.1
s23.300room	tetra	196.7	315.0	363.6	47.4
s23.300room	socbal	189.8	229.4	208.3	68.5
s23.300room	bishop	48.2	31.8	34.9	11.9
s23.300room	balloon	14.3	14.5	18.7	5.5
s23.300room	spitfire	7.2	4.4	8.7	1.2
s23.300room	747-200M	18.8	11.4	10.3	4.7

Table 8: Number of Full Resolution Checks per Second for Each Method.