Spatial Data Structures in Simulation and Planning*

Ali Boroujerdi and **Bernard M.E. Moret**
Department of Computer Science
University of New Mexico
Albuquerque, NM 87131-1386

Abstract

Many mission plans for aircraft (and other mobile objects) require detailed simulation for a rigorous analysis; automated generation of the plans themselves is also likely to require simulation. One of the problems encountered in simulating or planning an aircraft mission is the intersection of detection (or other) volumes—a standard problem in identification, where the volumes represent error bounds. We present work towards the efficient determination of whether an aircraft is visible to a fixed radar emplacement and the identification of pairs of intersecting detection volumes among a large collection. Our solutions make use of recent results in computational geometry and demonstrate the usefulness of sophisticated combinatorial algorithms and data structures in solving practical questions.

1 Introduction

When reviewing the mission plan for, say, a strike mission, a detailed simulation is invaluable to the planner, as it can reveal unsuspected interactions and also allow probabilistic estimation of various cost objectives. In carrying out a detailed simulation, one faces a number of data processing questions related to geometric concepts: at each time step, which aircraft are visible to which radar installations? which aircraft are within detection range of each other? of a collection of detected items, with some estimation errors, how many are distinct? If the number of aircraft in the simulation is at all large or the terrain description complex, such questions require sophisticated approaches to be solved within a time frame that allows the simulation to proceed at a useful pace. (For instance, a standard approach to the line-of-sight problem simply divides the space into orthogonal cells and precompute the visibility for each cell; this approach cannot attain sufficient resolution without generating a prohibitive number of cells.)

In this paper, we present results of work on two facets of these intersection problems:

- The line-of-sight problem: given a fixed radar emplacement and known terrain around it, when is an aircraft visible to the radar?
- The pairwise detection problem: given a large collection of 3-dimensional volumes (taken to be axes-parallel rectangular boxes), report all pairs of intersecting volumes.

Our solutions incorporate recent theoretical results in computational geometry and demonstrate the usefulness of such results in a practical setting. The algorithms are useful not only in military applications, but also in database query processing, motion-planning, and others.

2 The Line-of-Sight Problem

Given a stationary radar and a 3-dimensional map of the terrain, the problem is to determine efficiently whether any given point in space is visible to the radar—once this determination is made, specific radar and aircraft characteristics can be used to determine the probability of detection. Since the radar emplacement and the surrounding terrain are fixed, it is sensible to preprocess the data so that queries can be answered as efficiently as possible. Since many radars have a revolving beam, the query should be able to take the current beam position into account. While real radars can "peek" around corners, we address here a line-of-sight problem; however, adaptation of our technique to a real radar is straightforward.

The main characteristic of this problem is that it is, in fact, only 2.5-dimensional: it can be reduced to a 2-dimensional problem, with the third coordinate handled entirely separately, by a procedure developed for computer graphics. We model the terrain surface with polygons and preprocess the terrain information from the point of view of the radar to eliminate hidden lines and surfaces. (See Cole and Sharir (1989) for a discussion of visibility problems in polyhedral terrains.) What remains is essentially 2-dimensional information—one can think of it as projected onto the surface of a half-sphere centered at the radar emplacement—with distance information. The remaining segments partition the surface of the half sphere into regions; each region is described by a simple polygon and corresponds to the visible part of a polygonal facet of the terrain. We further process the result to partition any non-convex polygonal pieces into

*This work is supported by the Naval Research Laboratory under contract N00014-92-C-2144.

convex pieces—a standard and easy procedure in computational geometry. The result is a partition of the half-sphere into a collection of convex polygons, in turn defined by a collection of segments (the boundaries of the polygons).

We now use results from computational geometry on the problem of planar point location: "given a planar subdivision (a partition of the plane into convex polygons, for instance), preprocess it so as to determine efficiently in which polygon an arbitrary query point falls." A simple and efficient solution due to Sarnak and Tarjan (1986) makes use of persistency, a property of data structures that allows them to be searched or otherwise manipulated not only in their most recent version (as in standard data structures), but also in earlier versions (before the last few insertions or deletions, for instance). Sarnak and Tarjan (1986) showed that a persistent binary search tree could be used to solve the planar point location problem with linear storage and logarithmic time—which is optimal. The data structure records the history of a plane sweep along one axis; as a vertex is encountered in the sweep, marking the beginning or end (as seen from the sweep) of one or more segments, these segments are inserted into the tree or deleted from it; insertion is made with a key determined by the other coordinate, thereby maintaining a secondary ordering. Point location is achieved by two successive binary searches, each requiring at most logarithmic time: the first determines, by binary search along the first coordinate, the two sweep points between which the query point falls, while the second searches the binary search tree as it existed at that point in the sweep to find the two segments between which the query point falls. Knowing the two segments, each of which belongs to the boundary of exactly two polygons, one can determine in constant time which is the polygon common to both segments and thus where the query point falls.

To adapt the result to the half-sphere, we let the two angles of spherical coordinates be the two coordinates of the sweep, with elevation the secondary coordinate and horizontal angle determining the sweep. We have to take special precautions because our sweep makes a complete circle—we need to impose an artificial starting point and handle some special cases when the sweep ends at that point. To answer our visibility query, we begin by locating, in logarithmic time, the convex polygon in the subdivision of the half-sphere in which the query aircraft lies; since this polygon represents a piece of terrain surface, it has associated parameters (radial distance and slope, for instance), so that, by using the third spherical coordinate of the query aircraft—its radial distance from the radar—, we can determine in constant time whether the aircraft is currently behind or in front of the surface represented by the convex polygon. If the aircraft is behind, it is masked by the terrain; if it is in front, it is on the line-of-sight of the radar, at which point specific characteristics of the radar and of the aircraft can be used to determine the probability of detection. If the radar is very directional, its cone of visibility in horizontal angle can be used to avoid the search when the radar is not "looking" in the right direction.

The space required by our algorithm is linear in the number of segments that define the planar partition of the half-sphere—generally a tiny fraction of the total terrain information. This very small space requirement is a major advantage of our method, as is its easy scalability—the level of detail can easily be modified by adapting the polygonal surface model (for instance as a function of distance from the radar) and the storage requirements remain modest. Preprocessing requires sorting the endpoints of the n line segments, which takes $O(n \log n)$, and building the persistent structure, which takes $O(n \log n)$, and so is inexpensive. Queries (i.e., determining whether a point is seen by the radar) take $O(\log n)$, with absolutely minimal overhead, and thus are very fast. (These numbers assume that the persistent data structure is built using the node-copying method rather than the "fat node" method—see the paper of Driscoll *et al.* (1989) for a discussion of both methods; the fat node method would add a logarithmic factor to the running times. The fat node method may remain of interest, since it only loses efficiency if, at many points during the sweep, a large fraction of the segments are present within the data structure.)

We have implemented our version of planar point location for the half-sphere (but not the production of the planar partition) using a persistent red-black tree (one needs a balanced search tree for worst-case performance guarantees and one particularly needs a balanced search tree in which deletions and insertions only cause, in the amortized sense, a constant number of changes in the tree, so as to preserve the space guarantees of persistency). While implementing a persistent red-black tree is a significant undertaking, such a package should become part of standard libraries, since persistency finds a multitude of applications in data structures and algorithms, particularly in computational geometry.

We conclude this section by noting that our method lends itself well to parallelism (the same persistent binary search tree can be used concurrently by many processors sharing the storage), thereby allowing us to handle large number of queries at once, and to distributed solutions, since each radar emplacement has its own data structure and can be treated (for the purpose of visibility determination) entirely separately from others.

3 The Pairwise Detection Problem

Reporting all intersections among a collection of 3-dimensional objects is an important problem in computational geometry, due to applications in many domains—simulations of battle conditions where objects may represent detection envelopes, database queries, and many others. In some applications, the number of objects may grow very large—although the objects then tend to be elementary in their basic description. How to handle a problem with a million objects is a difficult computational task. Indeed, in a general setting, where the objects may be any convex volumes, the problem is not solvable efficiently—the best intersection algorithms in 3 dimensions run in $n \log n$ time for just one pair of objects with n vertices, edges, and faces. In the special case where the objects are simple rectilinear boxes aligned with the axes, however, the problem is greatly simplified and a number of reasonably efficient algorithms were developed (although, for the most part, not implemented or tested) in the early 1980s (see Section 8.8 of the text of Preparata and Shamos (1985). The simplification seems extreme—from arbitrary convex objects to a very rigid geometry and objects defined by just two points (the opposite corners of the box); however, it is a successful first approximation in many cases and a valid model for applications such as database queries.

3.1 Approaches

The problem can be divided into two versions: a static version, where the single goal is to find and report all pairs of intersecting boxes in the given collection; and a dynamic version, where the collection is subject to changes (adding new boxes or removing existing ones) and the set of intersecting pairs must be maintained as changes occur. The second problem is much harder than the first (not surprisingly); at this time, all approaches to the second problem use brute-force dynamization techniques (due to Bentley and Saxe (1980), which essentially consist of breaking up the data structure into small pieces and maintaining the pieces) on data structures designed to handle static problems. We focus on the static version of the problem.

Early algorithms for the problem [Bentley and Wood, 1980, Six and Wood, 1980] worked on the 2-dimensional version. The first good solution was given by Edelsbrunner [Edelsbrunner, 1980] and independently by McCreight [McCreight, 1980], with the development of the interval (or priority search) tree. With this data structure, the 2-dimensional problem can be solved in linear space and $O(n \log n + k)$ time, where k is the number of intersecting pairs—which is theoretically optimal. (Chazelle and Guibas (1989) offer a comprehensive survey of 2-dimensional intersection problems and their solutions.) These algorithms all generalize to d-dimensional problems, where they run in $O(n \log^{d-1} n)$ space and $O(n \log^{d-1} n + k)$ time; in particular, in three dimensions, the space requirements are proportional to $n \log^2 n$. These requirements are not acceptable for really large instances: with a million boxes, for instance, the $\log^2 n$ factor, which may normally seem negligible, could easily be on the order of 100—not a factor that can be easily overcome by an upgrade in hardware. On the other hand, these algorithms incur very little overhead and can be expected to run very efficiently.

3.2 Achieving Linear Storage

Edelsbrunner and Overmars proposed in 1985 a collection of general techniques to reduce the space requirements of large families of geometric algorithms. Among these families is that of box intersection in d-dimensional space. Their theoretical results indicate that d-dimensional box intersection can be solved in optimal linear storage and acceptable $O(n \log^{d-1} n + k)$ time. However, no practical algorithm for box intersection that would embody their technique had been developed.

In order to achieve linear storage and overcome the most serious limitations of intersection algorithms, we set out to turn their idea into a practical algorithm. The main obstacle was their apparent use of very complex dynamized data structures that had to be traversed in just the proper order. What we found was that such did not need to be the case; on the contrary, only the high-level principle behind these data structures needed to be emulated, while everything could in fact be achieved with very simple data structures (linked lists).

Our algorithm implements a traversal of a segment tree (a standard data structure in computational geometry) on one coordinate and exploits the fact that, whenever a box or segment fills the elementary interval corresponding to the current segment tree node in the traversal, we can reduce its dimensionality for the purposes of intersection. (Think of a box spanning the height of a room, from floor to ceiling: then intersections of this box with any other boxes can be detected by considering only the projection of these boxes on the floor of the room—ignoring the height dimension.) The segment tree is not explicit; the traversal creates its nodes only as needed, in depth-first search fashion (which is why we can avoid the logarithmic storage penalty of real segment trees). The 2-dimensional problem created in this way is solved by the same method—traversing a segment tree along the next coordinate. The 1-dimensional problem is where the intersections are really detected, in $O(k)$ time, where k is the number of intersections reported.

At a high-level, our algorithm (for 3 dimensions) proceeds as follows:

1. Create a node for the beginning and end of each box along each of the three dimensions; sort these nodes in order of increasing coordinates into three lists, one for each dimension. (In 3D, we shall use only one of the three sorted orders, but the other two will be used in 2D and 1D, respectively; they are computed initially and maintained in linear time through the levels of the algorithm.) Set the left and right endpoints of the current interval to the z coordinates of the beginning of the lowest box and of the end of the highest box, respectively.
2. Determine whether any box spans the current z interval from left endpoint to right endpoint—call any such box a spanning box. Pass to a 2-dimensional routine a list of the spanning boxes (the query boxes) and a list of the other boxes (the object boxes). The 2D routine will look for intersections between any two query boxes and between a query box and an object box—but not between two object boxes. On return from the 2D routine, discard all spanning boxes.
3. Cut the current interval down the middle (i.e., in the middle of the list of sorted endpoints of boxes); recursively solve the problem on the top (all of those boxes at least partially contained in the top half space) and on the bottom, with two recursive calls. To prepare a recursive call, set up the three lists used by the procedure—sorted on each of the axis. Note that a box can appear in both top and bottom lists.

The two-dimensional version differs slightly, because it receives two lists of boxes, not just one—in three dimensions, every box is both a query and an object, but in two dimensions, we have a distinction between the two. Only the list of queries is used in determining cuts for recursive calls, although we continue to detect spanning boxes (this time boxes that span an elementary interval along the y axis). The 2D recursive calls use all boxes that are not spanning boxes (and that impinge on the left or right half-plane, respectively), otherwise preserving the query vs. object distinction. The spanning boxes that are query boxes in 2D are passed on to the 1D routine to be checked for intersection against all other boxes within this interval (query or object); the spanning boxes that are object boxes in 2D are passed on to the 1D routine to be checked for intersection against all query boxes. Finally, the one-dimensional version gets a list of query segments and a list of object segments, both sorted, and thus can report all intersections in linear time with a single sweep (with a scanning method similar to that used in the merging step of mergesort).

3.3 Algorithmic Design and Analysis

One problem with the approach described so far is that the same two boxes could be seen to intersect in many intervals—up to a logarithmic number of them, in fact, since each box can ultimately be cut into a logarithmic number of slices by the recursive process. Not only is multiple reporting of intersections undesirable, it is also very inefficient—in this case, it would add at least a logarithmic factor to the running time. To avoid this problem, we only consider a box for potential intersection with some other box if its top corner falls within the current interval; we call these boxes "strict;" a box whose top corner falls within a preceding (in this case, higher) interval is deemed to be "loose." We will consider only intersections among pairs in which at least one member is a strict box—note that two loose boxes both exist in the preceding interval and thus could be seen to intersect there. In other words, we only report—in fact, we only look for—an intersection in the first elementary interval in which it exists.

The algorithm thus runs in time $O(k)$ at the one-dimensional level, using, at each call, at most linear storage. At the two-dimensional level, the algorithm calls itself recursively on the two half-intervals and also spends $O(k)$ time reporting all intersections found through calls to the one-dimensional routine. Because the depth of recursion is logarithmic and because it takes linear time to do the division of lists, the algorithm at the two-dimensional level runs in $O(n \log n + k)$ time. To see that it runs in linear space, we need one critical observation: a box (in 2D as well as in 3D) is at all times divided in at most three pieces: a left endpiece, which will be passed to the left recursive call; a right endpiece, which will be passed to a right recursive call (possibly at a higher level in the recursion tree); and a middle piece, which must be a spanning box within its interval and thus will be processed by the lower-dimensional routine and discarded before the recursive calls are placed. Hence, the storage required for each box is constant, so that the total storage requirement for the two-dimensional routine is linear. The same line of reasoning shows that the three-dimensional routine also uses only linear storage; however, it has its own logarithmic depth of recursion and may call the two-dimensional routine from any level of the recursion, so that its running time is a logarithmic factor higher than that of the two-dimensional routine, or $O(n \log^2 n + k)$ time.

The resulting code is quite simple. Our C implementation uses only about 2 pages of code for the 1D routines, 4 pages for the 2D routines, and 3 pages for the 3D routines; the driver need only prepare the three sorted lists. The resulting program is simpler than any of the methods discussed earlier, runs in only linear storage, and promises low overhead and thus has the potential to be applied to large problems.

3.4 Preliminary Results

To measure the "real" performance of this algorithm requires a lot of precautions. The output-dependent term k should at first be eliminated by considering problems that give rise to very few intersections; once the effect of the $O(n \log^2 n)$ behavior is understood, linearity in the output size can be verified. Another problem posed by the testing is that k can grow very large for very dense problems—to a quadratic value. When dealing with tens or hundreds of thousands of boxes, a quadratic growth is impossible to handle in main memory—the output should be written directly out to disk rather than stored internally, but for testing purposes, one can simply report the number of intersecting pairs rather than list all such pairs. Finally, since the main reason for this program is its linear space requirements, the implementation should be tuned very carefully to minimize any storage overhead.

Our preliminary results indicate that the algorithm runs fast, although not as fast (in 3 dimensions) as methods based on priority trees; the storage grows linearly as predicted, but remains somewhat larger than desirable, because of the proliferation of lists. A hybrid version, using our algorithm as a first pass to break the data down into smaller sets then a priority-based method to solve the smaller subproblems, should benefit from the characteristics of both methods and allow us to solve truly large instances, with several hundred thousand to a million boxes.

Finally, we note that our algorithm, unlike most others, lends itself very naturally to parallel or distributed implementations, since it is essentially a nested form of divide-and-conquer. Each dimension can be treated separately, while recursive calls give rise to a natural forking of active processes. Parallel implementations should allow us to handle several millions of boxes in very low running times.

4 Conclusion

We have presented practical algorithms, derived from recent theoretical results in computational geometry, that solve problems of importance in simulation and planning. Along with the two algorithms discussed in this paper, we are developing others for various spatial tasks; recent results in computational geometry have made such significant advances that much of the traditional approach to spatial data structures (grids, $k-d$ trees, etc.) is no longer competitive.

References

[Bentley and Saxe, 1980] Bentley, J.L, and J.B. Saxe, "Decomposable searching problems I: static to dynamic transformation," *J. Algorithms* **1** (1980), 301–358.

[Bentley and Wood, 1980] Bentley, J.L., and D. Wood, "An optimal worst-case algorithm for reporting intersection of rectangles," *IEEE Trans. Computers* **29** (1980), pp. 571–577.

[Chazelle and Guibas, 1989] Chazelle, B., and L.J. Guibas, "Visibility and intersection problems in plane geometry," *Discr. Comput. Geom.* **4** (1989), 551–581.

[Cole and Sharir, 1989] Cole, R., and M. Sharir, "Visibility problems for polyhedral terrains," *J. Symbolic Comput.* **7** (1989), 11–30.

[Driscoll *et al.*, 1989] Driscoll, N. Sarnak, D.D. Sleator, and R.E. Tarjan, "Making data structures persistent," *J. Comput. Syst. Sci.* **38** (1989), 86–124.

[Edelsbrunner, 1980] Edelsbrunner, H., "Dynamic data structures for orthogonal intersection queries," Technical Report #59 (1980), Institute für Informationsverarbeitung, Techn. Univ. Graz.

[Edelsbrunner and Overmars, 1985] Edelsbrunner, H., and M.H. Overmars, "Batched dynamic solutions to decomposable problems," *J. Algorithms* **6** (1985), 515–542.

[McCreight, 1985] McCreight, E.M., "Priority search trees," *SIAM J. Computing* **14** (1985), 257–276.

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

[Sarnak and Tarjan, 1986] Sarnak, N., and R.E. Tarjan, "Planar point location using persistent search trees," *Commun. ACM* **29** (1986), 669–679.

[Six and Wood, 1980] Six, H.W., and D. Wood, "The rectangle intersection problem revisited," *BIT* **20** (1980), 426–433.