PAPER *Special Section on Discrete Mathematics and Its Applications*

Designing Efficient Geometric Search Algorithms Using Persistent Binary-Binary Search Trees*

Xuehou TAN†, Tomio HIRATA†† *and* Yasuyoshi INAGAKI††, *Members*

SUMMARY Persistent data structures, introduced by Sarnak and Tarjan, have been found especially useful in designing geometric algorithms. In this paper, we present a persistent form of binary-binary search tree, and then apply this data structure to solve various geometric searching problems, such as, three dimensional ray-shooting, hidden surface removal, polygonal point enclosure searching and so on. In all applications, we are able to either improve existing bounds or establish new bounds.

key words: *computational geometry, persistent data structures, persistent binary-binary search trees, ray-shooting*

1. Introduction

Ordinary data structures are *ephemeral* in the sense that an update (insertion or deletion) to the structure destroys the old version, leaving only the new version available for use. In contrast, a *persistent structure* allows accesses to past versions. The structure is *fully persistent* or *partially persistent* depending upon whether any previously existing version can also be updated or not. In other words, the persistent structure embeds all versions of the ephemeral structure so that access to any version can be effectively simulated. Driscoll et al. [8] have given general techniques for making an ephemeral data structure persistent, provided that each node of the ephemeral structure has constant out-degree. Persistent data structures have been found especially useful in designing geometric algorithms [8], [21].

For higher dimensional problems, multi-layer data structures, such as binary-binary search trees and segment-segment trees [17], are often used. Specifically, operations on multi-layer structures include not only the node updates but also the substructure updates. Thus, Driscoll et al.'s technique cannot be directly applied to making multi-layer data structures persistent. However, for specific problems, it is possible to develop some special forms of persistent multi-layer structures. In this paper, we present a persistent form of binary-binary search tree and apply it to solving various geometric searching problems. In the following we briefly review the essentials of persistent data structures.

1. 1 Persistent Data Structures

Driscoll et al. [8] have given general techniques for making an ephemeral data structure persistent, provided that each node of the ephemeral structure has constant out-degree. Specifically, they add the partial persistence to an ephemeral data structure through the method of *node copying*. Each node of an ephemeral structure is expanded to hold k extra pointer slots in addition to the original ones. When a pointer change made to a node v, if there is an empty slot in node v, the new pointer is stored, along with a version stamp indicating when the change occurred; otherwise, a copy $c(v)$ of v is created, which is filled with the newest pointer values of v and thus has k new empty slots. Since the copy $c(v)$ is required to store a pointer in the latest parent of v, node copying can ripple backwards through the structure. However, amortized over a sequence of pointer changes, there are only $O(1)$ nodes copied per pointer change. If an update operation (insertion or deletion) requires only $O(1)$ pointer changes, the partially persistent structure can then be built with an amortized space cost of $O(1)$ per update. Simulation of an access to the ith version of the structure is simply accomplished by following at each node the appropriate pointer with the maximum version stamp no greater than i.

A main difference between partially persistent structures and fully persistent structures is that the various versions of a partially persistent structure have a natural linear ordering, whereas the versions of a fully persistent structure are only partially ordered. To build a fully persistent data structure, we first impose a total ordering on its versions. The total ordering is represented by a list, called the *version list*. When a new version, say i, is created, it is inserted in the version list immediately after its parent $p(i)$; $p(i)$ is the version that is updated to obtain version i. Thus for any version i, the descendants of i occur consecutively in the version list, starting with i. The version list is represented in Dietz and Sleator's structure [6].

Manuscript received September 13, 1993.

† The author is with the School of High-Technology for Human Welfare, Tokai University, Numazu-shi, 410-03 Japan.

†† The authors are with the Faculty of Engineering, Nagoya University, Nagoya-shi, 464 Japan.

* A preliminary version of this paper can be found in the Proceedings of the SIGAL International Symposium on Algorithms (LNCS 450), August, 1990, Tokyo.

The structure is able to determine, given two versions i and j, whether i precedes or follows j in the version list in $O(1)$ worst-case time. It takes $O(1)$ worst-case time for an insertion. Navigation through the persistent structure is then the same as in the partially persistent case, except that versions are compared with respect to their positions in the version list rather than their numeric values. With a variant of node copying, a fully persistent structure can be obtained with the same bounds as in the partially persistent case. The variant of node copying is called *node splitting*; when a node runs out of slots for new pointers, it is split into two, putting the first half of the pointers in one copy and the remainder in the other. The node splitting process can thus cascade through the structure.

When the persistence-addition techniques are applied to a particular kind of search tree, the *red-black* tree [11], considerable simplifications can be made. In summary, Driscoll et al. [8] obtain a way to build both partially and fully persistent search trees with a worst-case logarithmic time per update or access and a worst-case space of $O(1)$ per update.

In the companion paper [21], partially persistent search trees have been applied to give a simple implementation of efficient point location for planar subdivision. Suppose that S is a planar subdivision of size n. If we were to draw a vertical line through every vertex of S, then S would be broken into a number of vertical slabs. Within each slab, S looks like a series of vertically ordered trapezoids. See Fig. 1. Consider sweeping a vertical line through the plane from left to right and store the edges of S intersected with the sweep line in an *active list*. As the boundary from one slab to the next is crossed, certain edges are deleted from the sorted list and certain new ones are added. Over the entire sweep, there will be $O(n)$ insertions and deletions, one insertion and one deletion per edge of S. In other words, a new version of the active list is created when a vertex of the subdivision is reached. The history of the active list gives all the data for planar point location. The problem is then reduced to storing and accessing all the versions of the active list in a partially persistent search tree. Clearly, the space requirement of the structure is $O(n)$. Given a query point p, we can locate the slab containing p in $O(\log n)$ time, and then perform another binary search to find p within the slab in $O(\log n)$ further time.

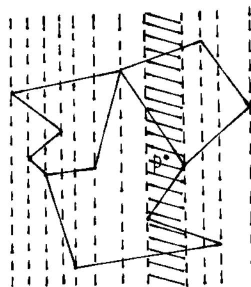


Fig. 1 A planar subdivision.

1.2 Overview of This Paper

We will first present our persistent binary-binary search trees by examining a basic problem, called the *three-dimensional ray-shooting problem*. Let q denote an arbitrary point in the 3-dimensional space and S a set of planar polygonal faces that do not intersect. (The faces may be concave or even have holes.) The ray-shooting problem requires reporting the face of S which is first hit by the open z-ray emanating from q in the positive z direction. Note that the problem we considered is restricted to the rays parallel to the z axis. Let N be the total number of edges of the faces in S, and let K be the number of edge intersections of S in the image (x, y)-plane. Obviously K is bounded above by $O(N^2)$. We present a persistent form of binary-binary search tree to solve this problem in $O(\log N)$ query time and $O(N+K)$ space. Our persistent structure makes use of a partially persistent search tree and a fully persistent search tree. An advantage of our structure is that it is conceptually simple and suitable for actual implementation.

Our structure also supports the *stabbing search* in which we are requested to report all the faces intersecting a query line segment in the z direction and the *topmost face search* in which the face last hit by the z-ray emanating from a query point is asked for. We then apply these searching algorithms to solve other geometric problems such as hidden surface removal, polygonal point enclosure searching and so on. In all applications, we are able to either improve existing bounds or establish new bounds.

Before closing this section, we make a comparison of our data structure with other similar data structures. First, we note that the solution for the ray-shooting problem can be applied to the *spatial subdivision searching problem* in which the polygonal faces of S form a subdivision of three dimensional-space and we are requested to find the polyhedron containing a query point p. (After the face first hit by the ray emanating from p in the z direction is found, the polyhedron containing p can be easily reported in constant time.) But, the known spatial subdivision searching algorithms [5], [20] are not suitable for our purpose. Combining the partial persistence-addition technique [8] and the dynamic planar point-location technique [18], [19], Preparata and Tamassia [20] give an $O(\log^2 N)$ query time and $O(N \log^2 N)$ space solution for the spatial subdivision searching problem. Althougth their algorithm is space-efficient, the subdivision must be *convex*. Thus, it cannot be simply applied to the ray-shooting problem. Cole's similar lists method [5] has been applied to the spatial subdivisions formed by n planes in the space. Although

Cole's similar lists method can be used to solve the ray-shooting problem, a total z order of all faces is required. (Note that making a set of faces in the space acyclic is not an easy work [16].) Moreover, the similar lists method cannot be applied to the stabbing search problem and the topmost face search problem without increasing the query time. Finally, we note that although our algorithm is not space-efficient for the spatial subdivision searching problem, it attains the optimal query time and is space-efficient for the other problems (see Sect. 3).

2. Three-Dimensional Ray-Shooting Using a Pair of Persistent Binary Search Trees

For a set S of planar polygonal faces and a query point q, the three-dimensional ray-shooting problem requires to find the element of S which is first intersected with the open z-ray emanating from q in the positive z direction. Let us now give a simple solution to this problem [7]. The edges are first projected onto the (x, y) plane. We assume, without loss of generality, that no edge is vertical in the (x, y) plane. We define an *interval* in the (x, y) plane as the line segment along a projected edge e that have the endpoint(s) of e or the intersection(s) of projected edges as its two endpoints and does not contain any intersection in its interior. We define a *region* as the area bounded by a set of intervals. Thus, the projected edges are intersected into *intervals*, which define a planar graph G with $O(N+K)$ *regions* (Fig. 2), where K is the number of edge intersections in the (x, y) plane. For each region, we maintain a list of faces whose projections contain that region. In a region's face list, the faces are then sorted into the z order. Given a query point q, we can perform a planar point location on G to locate the region containing the image of q in the (x, y) plane, then search the face list of that region for the face immediately above q, and thus obtain the face first hit by the z-ray emanating from q. Obviously the query time is $O(\log N)$. If an efficient planar point-location structure, such as Sarnak and Tarjan's [21], is used, the space requirement becomes $O((N+K)N)$.

By noticing that the face lists of adjacent regions are similar, we can further reduce the space bound to $O(N+K)$. Let us now approach the problem with the plane-sweep paradigm. A vertical line is swept through the (x, y) plane from left to right. The regions crossed by the sweep line are called the *current regions*. For each current region R, we make a note of the boundary intervals that are intersected with the sweep line. These intervals are R's *upper* and *lower current intervals*, depending on whether the region is below or above the intervals. (A concave region may have several intersections with the sweep line.) A face list is *active* if it is the face list of a current region. During the sweep, the current regions with their face lists are maintained in a data structure for ray-shooting. A binary-binary search tree can be used for this purpose; The current regions are vertically stored in a balanced binary search tree, for instance, a red-black tree; each node of this tree denotes a region, whose face list again gets attached to a balanced binary search tree over the z order of the faces. An example of this binary-binary search tree is shown in Fig. 3. To obtain the $O(N+K)$ space bound, we give a way to make this binary-binary search tree persistent.

When the sweep line is moved left-right through the graph G, the current regions and their face lists change. As a vertex v is passed, the regions to the left of v are deleted from the set of current regions and their face lists become inactive, the regions to the right of v are inserted into the set of current regions and the face lists of these newly started regions are created, and the region above or below v changes its lower or upper current interval. (It may happen that neither deletion nor insertion occurs when a vertex is passed.) Note that the change of the current regions is just the same as that in the planar point location problem. Thus, the first level search tree can be implemented as the partially persistent search tree for planar point location developed by Sarnak and Tarjan [21]. The operations on the second level search trees are a sequence of face list creations and cancels. Since region R's face list becomes inactive when region R is deleted from the set

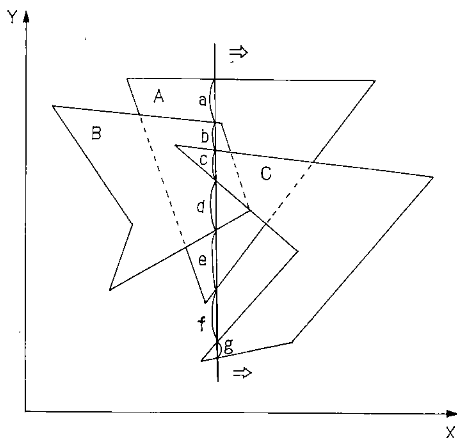


Fig. 2 The projection of a set of polygonal faces.

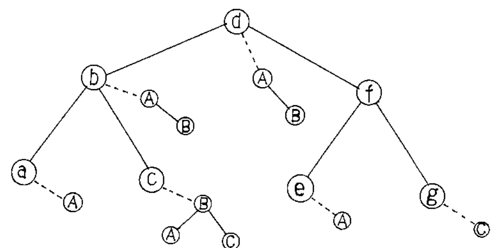


Fig. 3 The binary-binary search tree for ray-shooting.

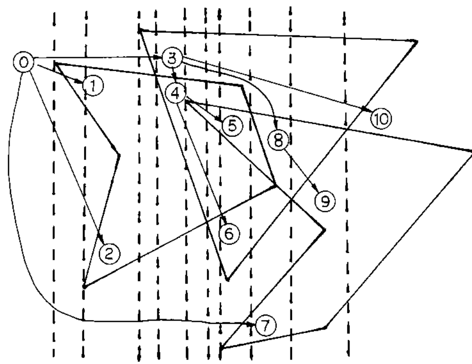


Fig. 4 The order of regions' face lists created.

of current regions, the remaining task is how to carry out face list creations. A face list creation needs at first glance up to $O(N \log N)$ time and $O(N)$ space. However, we can reduce the cost to $O(\log N)$ time and $O(1)$ space.

We will first give a general construction of our data structure and then make it precise. Observe that between the face lists of two adjacent regions R and R', there exists only one difference. That is, the face list of R' can be obtained by modifying the face list of R with an insertion or a deletion of the face bounded by the interval between R and R', depending on whether the face's projection contains region R' or not. Suppose that we encounter a vertex v during the sweep. For the newly started regions at v, i.e., the regions to the right of v, we can iteratively use the face list of a region to build the face list of adjacent region. Let R_0 denote the unbounded region of the graph G. Clearly, R_0's face list is empty. All other regions' face lists can be then obtained from a sequence of updates on R_0's face list. Figure 4 shows such an example. Numbers attached to regions show the order of regions' face lists created. (The places where a face list is created are marked with a vertical dashed line.) Arrows between regions define a binary relation α: $R_i \alpha R_j$ means that region R_j's face list is obtained by updating region R_i's face list. Observe that relation α gives a partial order on the set of created face lists. This observation suggests that the sequence of face list creations can be implemented in a fully persistent structure.

Let the ith face list in the sequence of creations have *list number* i. Since a face list creation contains a face list insertion or deletion, the ith face list is just the ith version of the fully persistent search tree. To set up a one-to-one relation between the regions and their face lists' numbers, we save list number i, or exactly, the number of the corresponding version of the fully persistent search tree, in the nodes of the partially persistent search tree (first level tree) that denote the corresponding region. Thus, a face list can be created in $O(\log N)$ time and $O(1)$ space.

Let us now make our data structure into details. Recall that a region in the partially persistent search tree is represented by its upper intervals, a concave region may be represented by more than one interval in the same version of the partially persistent search tree (see Fig. 1). In this case, a current region may have several intersections with the sweep line. But it does not affect the planar point location algorithm, because we only need to find the interval immediately above the query point (and the region containing the query point can then be easily determined). See also [21].

For the construction of the fully persistent search tree, we first note that the unbounded region R_0 is concave and even has holes. Due to the representation of concave regions in the partially persistent search tree, a concave region may have several face lists although all of them have the same content (see Fig. 4 for examples). When encountering a vertex v during the sweep, we perform a binary search in the current version of the partially persistent search tree to find the interval and thus the region above v. Then the face list of the found region can be employed to create the face lists of the regions to the right of v.

In summary, the first and second level search trees are implemented as a partially and fully persistent search trees, respectively. By the aid of face lists' numbers, we can easily set up the connection between these two different persistent structures. That is, a number of face list corresponds a version of the fully persistent search tree, and is stored in the nodes of the partially persistent tree that represent the corresponding region. From the size of the graph G, we conclude:

Theorem 1: Given a set S of polygonal face in the space, it is possible to preprocess it in $O((N+K)\log N)$ time and $O(N+K)$ space, so that the ray-shooting query can be answered in $O(\log N)$ time, where N is the total number of edges and K the number of edge intersections in the image plane.

Proof: First, all the edges of S are projected onto the (x, y) plane. These projected edges are intersected into intervals, which define a planar graph G. The planar graph G can be simply constructed in $O((N+K)\log N)$ time or even $O(N \log N+K)$ time and $O(N+K)$ space [4], [14]. Then we sweep a vertical line through the plane, and build a partially persistent search tree for the planar graph G and a fully persistent search tree for all of the regions' face lists. The upper intervals of a region are stored in some nodes of the partially persistent search tree, which hold a pointer to the face list(s) of the corresponding regions, i.e., the corresponding version number of the fully persistent tree. At each sweep step, we need $O(\log n)$ time and $O(1)$ space to update the persistent data structure. Thus, it requires $O((N+K)\log N)$ time and $O(N+K)$ space to build the persistent structure.

Given a query point p, we first perform a planar point location on G to find the region contain the image of p in the (x, y) plane. Then using the version

number found in the partially persistent search tree, we perform a binary search on the fully persistent search tree to find the face immediately above p. Clearly, the query time is $O(\log N)$. This completes the proof. □

We end this section with several remarks about the generality of our structure. Our data structure also supports the *topmost face search* in which the face last hit by the z-ray emanating from a query point is asked for and the *stabbing search* in which the queries are of the following form: given a line segment (a, b) parallel to the z axis, report all faces segment (a, b) intersects. By noticing that these two kinds of searches can be performed in any version of the fully persistent search tree, we can respectively solve them in $O(\log N)$ and $O(\log N+t)$ query time where t is the number of reported polyhedra.

In applying our data structure to the spatial subdivision searching problem, one drawback of our algorithm is that is requires $O(N+K)$ space. (Note that $K=\Omega(N^2)$ in the worst case.) To reduce the complexity of K, we can find a good projection plane, e.g., by a number of random choices. (In many cases, it is even possible to make K be linear to or slightly greater than N.) Furthermore, there is an interesting and important class of spatial subdivisions such that the number of intersections between the projections of the edges on the (x, y) plane is subquadratic. Let S be a subdivision of 3-space into n convex regions (e.g., the Voronoi diagram of n points in 3-space). It is well known that the complexity of S is $\theta(n^2)$ [9]. The intersections of these $\theta(n^2)$ edges in the projection (x, y) plane are trivially bounded by $O(n^4)$. A careful study shows that the complexity of the projected image of S is $\theta(n^3)$ [12]. Thus, for a spatial subdivision with n convex regions, our data structure immediately leads to an $O(\log n)$ query time and $O(n^3)$ space solution. For comparison, if we use Preparata and Tamassia's algorithm [17], then we can obtain the $O(\log^2 n)$ query time and $O(n^2 \log^2 n)$ space solution for the same problem.

3. Applications

This section presents applications of the data structure developed in Sect. 2 to problems that can be formulated in terms of ray-shooting, or that relate to such problems; in all cases we are able to improve existing bounds or establish new bounds.

3.1 Translating a Set of Faces or Polyhedra in Three Dimensions

Given a set of faces or polyhedra in three dimensions. The *translation problem* requires moving them in a given direction, one at a time, without collisions occurring between them. To solve this problem, we are required to perform a sequence of ray-shooting queries (see [15]). As an immediate consequence of Theorem 1, we obtain:

Theorem 2: The translation problem for a set of faces or polyhedra in three dimensions can be solved in $O((N+K)\log N)$ time and $O(N+K)$ space, where N is the total number of edges and K the number of edge intersections in the image plane.

This result improves Nurmi's result of $O((N+K)\log N)$ time and space [15].

3.2 Hidden Surface Removal

Given an environment of nonintersecting opaque polyhedra in three dimensions, we wish to compute the image visible from a given viewpoint. To produce a realistic image of the environment, we first project all the environment's edges onto the image plane. Suppose that the image plane is the (x, y) plane. This establishes a line segment subdivision of the (x, y) plane. Each region of this subdivision is covered by some of the environment's faces. The *hidden surface removal problem* is the task of reporting the topmost face for each region of the subdivision. Remember that the persistent structure developed in Sect. 2 supports queries not only about the z neighbors of a point but also about the topmost face in the z direction. After finding the topmost faces for every region, we have to remove all the subdivision segments whose two incident regions have common topmost faces and all the vertices that are in the interior of an edge or a region. See [13] for details. These yield:

Theorem 3: The hidden surface removal problem can be solved in $O((N+K)\log N)$ time and $O(N+K)$ space, where N is the total number of the environment's edges and K the number of edge intersections in the image plane.

Although the same resource bounds have been obtained by Schmitt et al. [22], their algorithm requires various complicated data structures. In [13] McKenna showed that the hidden surface removal problem can be solved in $O(N^2)$ time, which is an improvement to the $O((N+K)\log K)$ bound in cases where $K=\Omega(N^2)$. The drawback of his algorithm is that it requires $\Omega(N^2)$ space. For recent results on ray shooting and hidden surface removal, see [1].

3.3 Polygonal Point Enclosure Searching

The *polygonal point enclosure search problem* is: given a set of N simple polygons, each with a bounded number of edges, and a query point in the plane, determine all the polygons enclosing the point.

The N polygons consist of $O(N)$ edges. The collection of edges defines a planar graph G. Each region of the graph G is overlapped by some of the original polygons. Our intention is to precompute for

each region the list of overlapping polygons. Assume that each polygon has a pre-defined z-coordinate. We can thus build a persistent binary-binary search tree for the graph G. Having located the region containing a given point in the partially persistent search tree, we report all the covering faces of that region in the corresponding version of the fully persistent search tree. This yields:

Theorem 4: Given a set of N simple polygons in the plane, each with a bounded number of edges, there exists a data structure such that the t polygons that contain a given point can be reported in $O(\log N+t)$ time, using $O(N+K)$ space where K is the number of edge intersections.

This result improves Edelsbrunner et al.'s result of $O(\log N+t)$ query time and $O(NK)$ space [10].

3.4 Polygon Intersection Reporting

Let P denote a set of simple polygons in the plane, each with a bounded number of edges. The *polygon intersection problem* requires to report all pairs of polygons which have at least one point in common.

It is obvious that the problem can be reduced to two sub-problems [2]: The line segment intersection problem and the batched polygonal point enclosure search problem. The first subproblem is already solved by Chazelle and Edelsbrunner [4]. Their algorithm runs in $O(N \log N+K)$ time and $O(N+K)$ space, where K is the number of edge intersections. The second can be solved by applying Theorem 4. We conclude:

Theorem 5: All T intersections among N simple polygons can be reported in $O((N+K)\log N+T)$ time and $O(N+K)$ space, where K is the number of edge intersections.

A challenging open problem is to determine whether this problem can be solved in optimal time $O(N \log N+T)$, analogous to the optimal time for computing line segment intersections [4].

4. Conclusions

We have presented a persistent form of binary-binary search tree by considering the ray-shooting problem. For a set of polygonal faces with a total of N edges in the space and K edge intersections in the image plane, our structure supports a ray-shooting query with $O(\log N)$ time, and requires $O(N+K)$ space and $O((N+K)\log N)$ preprocessing time to build. The data structure also gives new better solutions for various geometric searching problems. Besides these applications, we believe that the data structure is general enough to serve as a stepping-stone for other geometric problems as well.

Acknowledgement

We would like to thank Dr. Takeshi Tokuyama of IBM Tokyo Research Laboratory for his valuable comments and discussions.

References

[1] Agarwal, P. K. and Matousek, J., "Ray shooting and parametric search," in *Proceedings, 24th Annu. ACM Symp. Theory of Computing*, pp. 517–526, 1992.

[2] Bentley, J. L. and Wood, D., "An optimal worst-case algorithm for reporting intersections of rectangles," *IEEE Trans. Comput.* vol. C-29, pp. 571–577, 1980.

[3] Chazelle, B., "How to search in history," *Inform. Control* vol. 64, pp. 77–99, 1985.

[4] Chazelle, B. and Edelsbrunner, H., "An optimal algorithm for intersecting line segments in the plane," in *Proceedings, 29th Annu. IEEE Symp. Found. of Comput. Sci.*, pp. 590–600, 1988.

[5] Cole, R., "Searching and storing similar lists," *J. Algorithms*, vol. 7, pp. 202–220, 1986.

[6] Dietz, P. and Sleator, D. D., "Two algorithms for maintaining order in a list," in *Proceedings, 19th Annu. ACM Symp. Theory of Computing*, pp. 365–372, 1987.

[7] Dobkin, D. and Lipton, R. J., "Multidimensional search problems," *SIAM J. Comput.*, vol. 5, pp. 181–186, 1976.

[8] Driscoll, J. R., Sarnak, N., Sleator, D. D. and Tarjan, R. E., "Making data structures persistent," *J. Comput. Sys. Sci.*, vol. 38, pp. 86–124, 1989.

[9] Edelsbrunner, H., *Algorithms in Combinatorial Geometry*, Springer-Verlag, 1987.

[10] Edelsbrunner, H., Maurer, H. A. and Kirkpatrick, D. G., "Polygonal intersection searching," *Inform. Process. Lett.*, vol. 14, pp. 74–77, 1982.

[11] Guibas, L. J. and Sedgewick, R., "A dichromatic framework for balanced trees," in *Proceedings, 19th Annu. IEEE Symp. Found. of Comput. Sci.*, pp. 8–21, 1978.

[12] Hirata, T., Matousek, J., Tan, X. and Tokuyama, T., "Complexity of projected images of convex subdivisions," *Proc. 4th Canadian Conference on Computational geometry*, pp. 121–126, 1992.

[13] McKenna, M., "Worst case optimal hidden surface removal," *ACM Trans. Graphics*, vol. 6, pp. 19–28, 1987.

[14] Nivergelt, J. and Preparata, F. P., "Plane sweep algorithms for intersecting geometric figures," *Comm. ACM*, vol. 25, pp. 739–747, 1982.

[15] Nurmi, O., "On translating a set of objects in 2- and 3-dimensional space," *Comput. Vision Graphics Image Process.*, vol. 36, pp. 42–52, 1986.

[16] Paterson, M. S. and Yao, F. F., "Binary partitions with applications to hidden-surface removal and solid modelling," *Proc. 5th ACM Symposium on Computational Geometry*, pp. 23–32, 1989

[17] Preparata, F. P. and Shamos, M. I., *Computational Geometry*, Springer-Verlag, 1985.

[18] Preparata, F. P. and Tamassia, R., "Fully dynamic techniques for point location and transitive closure in planar structures," in *Proceedings, 29th Annu. IEEE Symp. Found. of Comput. Sci.*, pp. 558–567, 1988.

[19] Preparata, F. P. and Tamassia, R., "Fully dynamic point location in a monotone subdivision," *SIAM J. Comput.*, vol. 18, pp. 811–830, 1989.

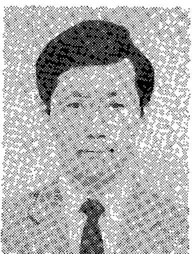
[20] Preparata, F. P. and Tamassia, R., "Efficient spatial point

location," in Algorithms and Data Structures (WADS'89), *Lect. Notes in Comput. Sci*. vol. 382, pp. 3–11, Springer-Verlag, 1989.

[21] Sarnak, N. and Tarjan, R. E., "Planar point location using persistent search trees," *Comm. ACM*, vol. 29, pp. 669–679, 1986.

[22] Schmitt, A., Müller, H. and Leister, W., "Ray tracing algorithms—theory and practice, *Proc. NATO Advanced Study Inst. Theoret. Found. Comput. Graphics and CAD*, pp. 997–1029, Springer-Verlag, 1987.

Xuehou Tan was born in Jiangsu, China in 1962. He received the B.S. degree in 1982 and the M.S. degree in 1985 from Nanjing University, and the Ph.D. degree in 1991 from Nagoya University. He is currently an Assistant Professor in the School of High-Technology for Human Welfare, Tokai University. From 1985 to 1987 he was a Research Associate of the Computer Science Department, Nanjing University. From 1992 to 1993, he was a visiting scientist with the University of Montreal and McGill University. His current research interests are in computational geometry and VLSI theory.



Tomio Hirata received the B.S. degree in electrical engineering, and the M.S. and Ph.D. degrees in Computer Science, all from Tohoku University in 1976, 1978, and 1981, respectively. He is currently a Professor in the Electronics Department, Nagoya University. From 1981 to 1986 he was an Assistant Professor of Computer Science Department, Toyohashi University of Technology. His research interests include graph algorithms, data structures, and geometric algorithms.

Yasuyoshi Inagaki was born in Nagoya, Japan in 1939. He received the B.Sc. degree in 1962, the M.Sc. degree in 1964, and the Dr. of Engineering degree in 1967 from Nagoya University. He was an Assistant Professor from 1970 to 1976 at the Department of Electrical Engineering and Information Engineering, Nagoya University, and a Full Professor from 1977 to 1980 at the Department of Electronics, Mie University. Since 1980, he has been a Full Professor at the Department of Electrical Engineering and Information Engineering, Nagoya University. His current research interests are in the area of algebraic theory of software specification, verification, and implementation, automata and languages theory, design and analysis of algorithms, and fundamental theory of artificial intelligence. He received the 1966 Inada Award from the Institute of Electrical Communication Engineers of Japan. Dr. Inagaki is a member of IEEE, ACM, EATCS, the Institute of Electrical Engineers of Japan, the Information Processing Society of Japan, Japan Society for Software Science and Technology, Japanese Society for Artifitial Intelligence, the Operations Research Society of Japan.