Programming Techniques and Data Structures

M. Douglas McIlroy* Editor

Plane-Sweep Algorithms for Intersecting Geometric Figures

J. Nievergelt
E.T.H., Zurich, Switzerland

F. P. Preparata
University of Illinois

Algorithms in computational geometry are of increasing importace in computer-aided design, for example, in the layout of integrated circuits. The efficient computation of the intersection of several superimposed figures is a basic problem. We consider plane figures defined by points connected by straight line segments, for example, polygons (not necessarily simple) and maps (embedded planar graphs). The regions into which the plane is partitioned by these intersecting figures are to be processed in various ways such as listing the boundary of each region in cyclic order or sweeping the interior of each region. Let n be the total number of points of all the figures involved and s be the total number of intersections of all line segments. We present two plane-sweep algorithms that solve the problems above; in the general case (no convexity) in time $O((n + s)\log n)$ and space $O(n + s)$; when the regions of each given figure are convex, the same can be achieved in time $O(n \log n + s)$ and space $O(n)$.

CR Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*geometrical problems and computations*; E.1 [**Data Structures**]—*arrays, lists, trees*

General Terms: Algorithms, Design, Theory

Additional Key Words and Phrases: plane-sweep algorithms, intersection problems, maps, polygons

* Former editor of Programming Techniques and Data Structures, of which Ellis Horowitz is the current editor.

This work was supported in part by the National Science Foundation under Grant MCS 78-13642 and Joint Services Electronics Program under Contract N00014-79-C-0424. This is a revised version of Report R-863, Coordinated Science Laboratory, University of Illinois, Urbana, October 1979.

Authors' Present Addresses: J. Nievergelt, Informatik, ETH, CH-8092 Zurich, Switzerland; F. P. Preparata, Coordinated Science Laboratory, University of Illinois, Urbana, Illinois 61801.

1. Introduction

One type of algorithm that emerged from recent research in computational plane geometry promises to be efficient for all problems to which it is applied. It sweeps the plane "from left to right," advancing a "front" or "cross section" from point to next point. All processing is done at this moving front, without any backtracking, with a horizon or look-ahead of only one point. Algorithms of this type are often called "greedy," in contrast to exhaustive search algorithms that create tentative partial results perhaps to be discarded later. An important issue in computational complexity is to understand which problems can be solved by greedy algorithms.

We present two efficient algorithms for problems that have such diverse applications as computer graphics, geographic data processing, and integrated circuit design and layout. In computer graphics, plane-sweep techniques play a central role in raster scan conversion and permit a precise definition of the previously vague notion of "coherence" (what pixels a raster line shares with the preceding one.) Traditional scan conversion algorithms are vector-oriented and keep track of the identity of areas bounded by these vectors only with difficulty. The area-oriented plane-sweep algorithm presented in this paper shows that scan conversion can include the processing of areas at little additional cost compared to processing vectors only. In geographic applications, such as the compilation of zoning maps, as well as in the design of masks for integrated circuit fabrication, a plane-sweep technique naturally solves the problem of superposing two or more partitions of the plane.

Shamos and Hoey [7, 8] presented an algorithm that, by sweeping the plane unidirectionally, determines in time $O(n \log n)$ whether or not n line segments are free of intersections. Bentley and Ottmann [2] have extended this algorithm to report all s intersections of n line segments within time $O((n + s)\log n)$. We elaborate on this type of algorithm in two directions. First, we show that within the same asymptotic effort, it can compute several kinds of topological and geometric results about the connected regions into which the plane is divided by the line segments. For example, a plane-sweep algorithm can output a cyclic list of all their boundary vertices and/or segments. Second, we show that assumptions of convexity allow one to improve these asymptotic bounds. Specifically, we present an algorithm that computes all s intersections of two convex maps (embedded planar graphs with convex regions) with a total of n points in space $O(n)$ and time $O(n \log n + s)$.

2. Regions Formed by a Polygon

Later sections of this paper refer to several versions of the problem of computing regions of the plane formed by embedded graphs. Here we present in detail the

Fig. 1. A polygon of five line segments and regions. R_0: $V_4W_1V_1V_5$; R_1: $V_1V_5V_1W_1V_3V_2W_2$; R_2: $V_2V_3W_2$.

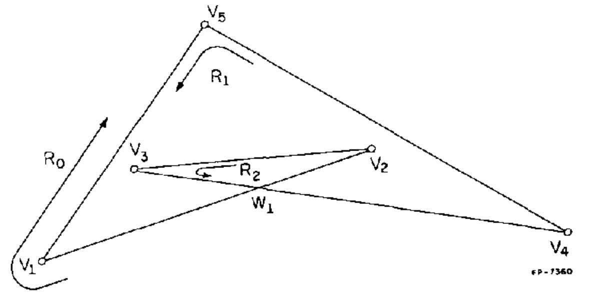
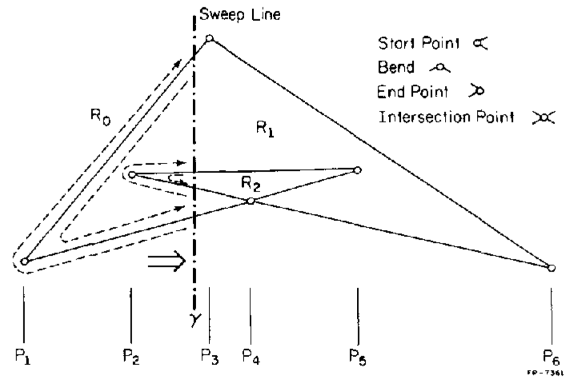


Fig. 2. Figure 1 revisited. Points have been ordered according to increasing x coordinates. Cross section γ contains one interval for each of the regions R_0 and R_2, and two intervals for two branches of R_1 that have split. Oriented "sticky tapes" trailing the cross section γ are shown as broken lines.



version which is simplest for expository purposes: let the given graph be a possibly self-intersecting polygon, i.e., a closed chain of n line segments or equivalently, a cyclic list of n points in the plane (for an application of self-intersecting polygons, see [6].) The reader who has understood how the algorithm works in this case will have no difficulty in following the brief description of how it applies to the intersection of more general plane-embedded graphs.

2.1 Statement of the Problem and Terminology

Given a sequence of n points $V_i = (x_i, y_i)$, $i = 1, 2, \ldots, n$, in the plane, a polygon with vertices V_i is the sequence of line segments $V_1V_2, V_2V_3, \ldots, V_nV_1$. These n line segments in general define s intersection points $W_j = (x_j, y_j)$, $j = 1, 2, \ldots, s$. When $s = 0$, the polygon is called simple and divides the plane into two regions, an internal bounded region R_1 and an external unbounded region R_0. In general, $s = O(n^2)$ and the polygon divides the plane into $r + 1 \geq 2$ disjoint regions, namely, the external unbounded region R_0 and r simply connected internal regions $R_1, \ldots, R_r$ (when the polygon is nondegenerate, $r = s + 1$.) Each region is itself a simple polygon that has as its vertices some subset of $\{V_1, \ldots, V_n, W_1, \ldots, W_s\}$. The desired result is a list of all regions, where each region is given by a cyclic list of its vertices, starting with the rightmost vertex; the external region in clockwise order, the internal regions in counterclockwise order. Figure 1 illustrates these concepts.

As Fig. 1 suggests, we make certain assumptions of nondegeneracy, namely, all points (the ones originally given as well as the intersection points) are distinct, a point lies on only those line segments on which it must lie, and no others (e.g., V_2 does not lie on V_4V_5). Section 4 discusses the modifications required to handle degenerate pictures.

The above notions are sufficient to describe the problem and its solution. In order to describe the algorithm that computes the solution, we introduce auxiliary concepts that reflect the dynamic aspects—a unidirectional sweep of the plane. Call the originally given points V_i and the intersection points W_j simply points, $P_1, P_2, \ldots, P_{n+s}$, sorted in order of increasing x coordinate. (We assume for ease of exposition that no two points have equal x coordinates; if P_i and P_{i+2} have equal x coordinates, a lexicographic ordering on the pair (x, y) suffices for the following discussion to apply.) The x axis is being distinguished as the direction of the plane-sweep, i.e., the sweep line is orthogonal to the x axis. Therefore, under our assumptions of nondegeneracy and distinct x coordinates, it is natural that each point be classified uniquely into one of four categories: start point, bend, end point, intersection point. (See Fig. 2.)

A cross section is a vertical line in the plane along with all the information about which line segments and regions it cuts and in what order. The line segments cut by a cross section partition it into intervals. We are mainly concerned with cross sections that do not pass through any points. The set of all (topologically equivalent) cross sections that lie between two adjacent points is called a slice. A cross section that passes through (exactly) one point is called a transition.

As the current cross section sweeps the figure, it drags along "sticky tapes" that hug the periphery of regions, as shown in Fig. 2. Much of the specification of the region-finding algorithm is concerned with properly maintaining these tapes across transitions.

2.2 Data Structures Maintained by the Algorithm

The region-finding algorithm to be presented in Sec. 2.3 operates upon three data structures. Two of these, the x structure and the y structure, are common to all plane-sweep algorithms. As the line that sweeps the plane advances in the direction of the x axis, the x structure represents a queue of tasks to be accomplished. The y structure represents the state of the current cross section. The third data structure is specific to the goal to be achieved by the plane-sweep. In order to process regions we introduce an r structure that represents the state of regions cut by the current cross section.

These three data structures are crucial for understanding the algorithm as well as for its efficiency. We present them at a fairly abstract level, by postulating what operations must be performed and how much time is available for them (asymptotically in terms of the problem parameters n and s.) We refer the reader to standard textbooks [1, 4] for concrete implementations that realize the postulated time bounds.

2.2.1 *The x structure X*

At any moment, X contains those points that have been discovered so far and are yet to be processed, sorted according to increasing x coordinate. Points are assigned a type according to the classification of Sec. 2.1. The x structure is a priority queue that must support the following operations within time bound $O(\log k)$ when it contains k entries.

- MIN: find and remove the entry with minimal x coordinate
—INSERT: insert a new entry with given x coordinate.
Heaps or balanced trees are suitable for implementing priority queues.

Initial content: the n originally given points, sorted, with their classification. Final content: empty.

At each transition, the point which defines this transition is removed and, at most, two intersection points are inserted into X. During execution a total of $n + s$ points move through the x structure, hence the maximal number of entries at any given time is $< n + s$. Since $s = O(n^2)$, any operation on the x structure can be done in time $O(\log(n + s)) = O(\log n)$.

2.2.2 *The y structure Y*

Y contains all the information about a cross section which is representative of its entire slice. It has an entry for each interval of the cross section, including the intervals that extend to $y = +\infty$ and $y = -\infty$, and thus it never has more than $n + 1$ entries. An equivalent description is that Y has an entry for each line segment intersected by the slice, including two sentinels. The line segment entry is a linear formula that defines this segment, so that for any x, the corresponding value $y = ax + b$ can be obtained in time $O(1)$.

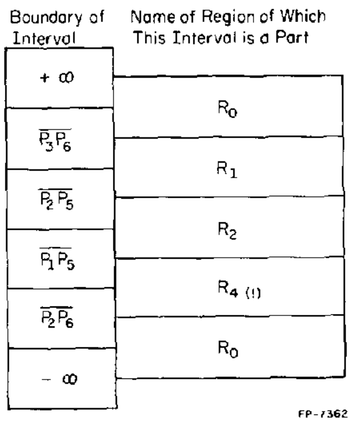
Initial and final content of Y: the single interval R_0 bounded by line segments $y = +\infty$ and $y = -\infty$.

Figure 3 shows the y structure of the slice between P_4 and P_5 in Fig. 2.

Y is a dictionary (see [1]) that must support operations FIND, INSERT, DELETE, PREDECESSOR, and SUCCESSOR within time bound $O(\log k)$ when it contains k entries. We tailor the exact definition of these operations to the specific use we make of them, thus postulating dictionary operations that are easily synthesized from the standard ones.

—FIND(P): given a point $P = (x, y, T)$, obtain one of the following results depending on the type T of P.
bend: the unique line segment s whose right end point is P;
end: the two line segments s (above) and t (below) whose common end point is P;
start: the two adjacent line segments s (above) and t (below) in whose interval $[s, t]$ the point P lies;

Fig. 3. The y structure of the slice between points P_4 and P_5 in Fig. 2. The field "P_3P_6" contains the definition of the line segment connecting its endpoints P_3 and P_6. Given an x value, this entry allows evaluation of the corresponding y value in time $O(1)$. Notice the entry $R_4(\infty)$ under "Name of region." It refers to a tentative region which was created at transition P_4. In a left-to-right scan of the plane it will only become known at transition P_5 that region R_4 is to be identified with region R_1.



intersection: the two line segments s and t whose intersection is P.

—INSERT(s, I): given a line segment s and an interval I adjacent to s from above or below, insert the pair (s, I) at the proper place determined by the y value of s at the current x value.

—DELETE(s, I): given a line segment s and an interval I adjacent to s, delete the pair (s, I).

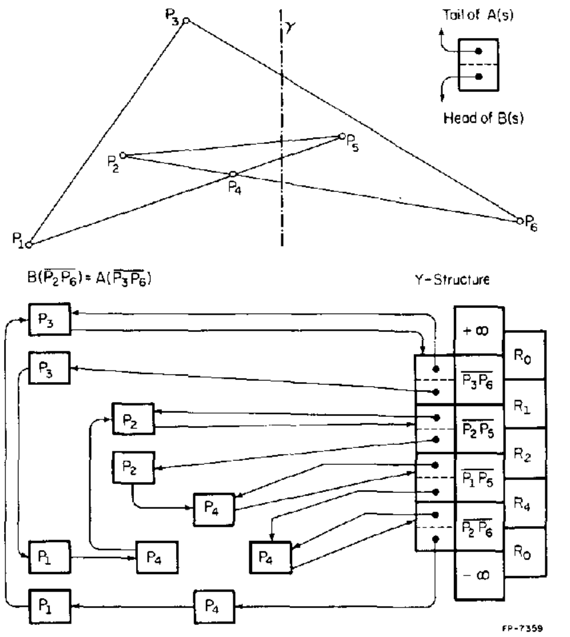
—SUCCESSOR(s), PREDECESSOR(s): given segment s and the current x value, return the neighboring segment above or below s.

A dictionary with k entries can be implemented so as to support the above operations within time $O(\log k)$ by any of several types of balanced trees. Binary search for a given y value is performed by evaluating the linear formulas $y = ax + b$ stored as segment entries along a root-to-leaf path. By means of additional pointers a dictionary can easily be implemented so that SUCCESSOR and PREDECESSOR work in time $O(1)$. For the region-finding algorithm such a refinement yields no overall asymptotic speed-up. For the convex map-intersection algorithm of Sec. 3, on the other hand, fast SUCCESSOR and PREDECESSOR operations are essential.

2.2.3 *The r structure R*

The r structure is the key to the region-finding algorithm. It integrates information about the regions and their peripheries as it is accumulated during the unidirectional sweep. It is initialized to be empty and terminates empty. For any given cross section it contains information about exactly those regions that are cut by this cross section. Specifically, R associates with each line segment s in the cross section two cyclic lists $A(s)$ and $B(s)$. These consist of the vertices on the boundaries of the regions which lie above and below s, respectively. Equivalently, with each interval determined by two adjacent segments, R associates the cyclic list of the bound-

Fig. 4. The r structure of a cross section γ in slice P_4P_5, shown attached to the y structure. Lists can be thought of as sticky tapes that hug the boundaries of regions. Access to the lists is provided by two pointers called "Tail of A(s)" and "Head of B(s)."



ary vertices of that part of the region which lies to the left of the cross section.

R is accessed from the y structure in the manner shown in Fig. 4. Each line segment entry s in Y points to the head of the list $B(s)$ and to the tail of $A(s)$. The latter points back to the segment entry for s in Y. Since R is attached to Y, the two could be considered to be a single data structure. We prefer to describe them separately, since Y is common to most plane-sweep algorithms, while R is specific to region identification. The above description is a static picture of R. Section 2.3 presents the dynamic picture of how R is updated at each transition and how region boundaries are formed.

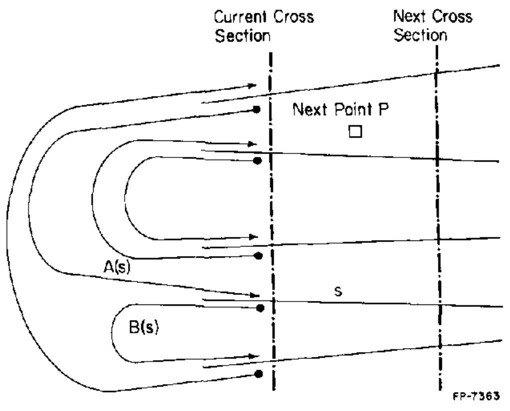
2.3 The Region-Finding Algorithm

The algorithm that sweeps the plan and forms the region boundaries has the following simple overall structure. Here X, Y, and R are the three data structures previously described.

```
Procedure SWEEP:
$X \leftarrow n$ given points, sorted by $x$ coordinate
$Y \leftarrow (-\infty, +\infty)$,   name of region $\leftarrow R_0$
$R \leftarrow \emptyset$
while $X \neq \emptyset$ do   begin
                    $P \leftarrow$ MIN($X$)
                    TRANSITION($P$)
                  end
end of SWEEP;
```

Procedure TRANSITION is the advancing mechanism of SWEEP. It encompasses all the work involved in processing one point P and moving the "front" from the slice to the left of P to the slice immediately to the right of P; in the process, it updates the corresponding data structures and builds up the result in an output structure. TRANSITION is invoked exactly $(n + s)$ times. We show that one invocation uses $O(\log n)$ time and thus establishes an $O((n + s)\log n)$ time bound on the performance of SWEEP. Figure 5 illustrates the situation when TRANSITION is invoked; given a current cross section, update all data structures so as to represent the new cross section correctly.

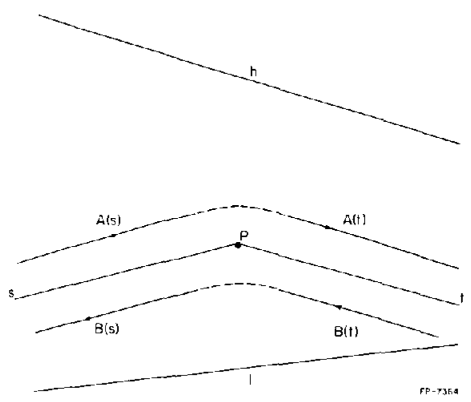
Fig. 5. The situation faced by TRANSITION. The "next point" P is of one of four types: bend, start, end, intersection.



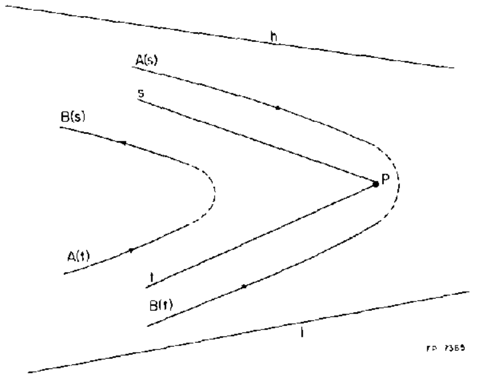
The following notation is needed to understand the detailed description of TRANSITION presented below. $A(s)$ and $B(s)$ are the cyclic lists of vertices of the regions bordering s above and below, respectively, with the orientation shown in Fig. 5. Lists are thought of as being ordered from left to right; thus, for a point P and a list L, "$P*L$" denotes that P has been added to L as its new head, whereas "$L*P$" denotes that P is the new tail. Similarly for two lists L_1 and L_2, L_1*L_2 denotes their concatenation. The function INTERSECT(s, t) checks in time $O(1)$ whether two segments s and t intersect, and if so, inserts the intersection point into X in time $O(\log|X|)$. We assume that the sentinels $+\infty$ and $-\infty$ do not intersect any line segments. If s and t are adjacent line segments, $[s, t]$ denotes the interval enclosed by them.

Procedure TRANSITION(P) breaks into four cases depending on the type of P. We present each case separately.

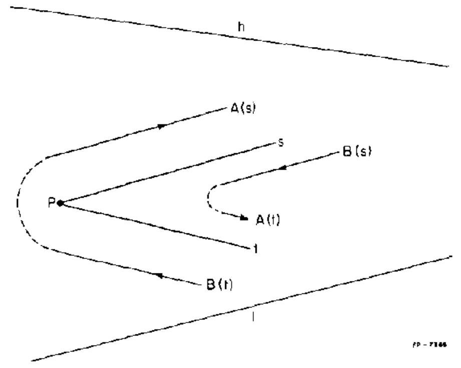
case "bend":
  FIND(P) yields the unique line segment s whose right endpoint is P;



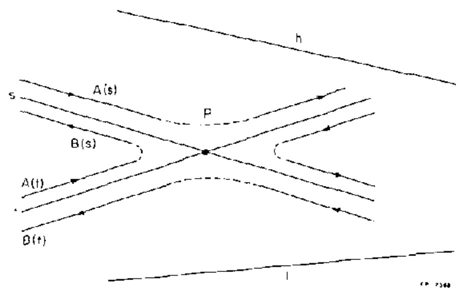
```
  $t$ ← segment starting at $P$
  $h$ ← SUCCESSOR($s$)
  $l$ ← PREDECESSOR($s$)
  INTERSECT($t$, $h$)
  INTERSECT($l$, $t$)
  $A(t)$ ← $A(s)*P$
  $B(t)$ ← $P*B(s)$
  replace $s$ with $t$ in $Y$
end of case "bend";

case "end":
  FIND($P$) yields the two line segments $s$ and $t$ whose common
          endpoint is $P$;
  $h$ ← SUCCESSOR($s$)
```



```
  $l$ ← PREDECESSOR($t$)
  INTERSECT($l$, $h$)
  catenate $A(t)*P*B(s)$
  catenate $A(s)*P*B(t)$
  replace [$t$, $s$] by [$l$, $h$], whereby the region name of [$l$, $h$] is the
          smaller of the names of [$s$, $h$] and [$l$, $t$]
  DELETE($s$, [$s$, $h$])
  DELETE($t$, [$l$, $t$])
end of case "end";

case "start":
  FIND($P$) yields the two adjacent line segments $h$ and $l$ in whose
          interval [$l$, $h$] point $P$ lies;
```



```
  $s$ ← high segment starting at $P$
  $t$ ← low segment starting at $P$
  INTERSECT($h$, $s$)
  INTERSECT($t$, $l$)
  INSERT($s$, [$s$, $h$]), whereby the region name of [$s$, $h$] is inherited
              from [$l$, $h$]
  INSERT($t$, [$l$, $t$]), whereby the region name of [$l$, $t$] is inherited from
          [$l$, $h$]
  modify [$l$, $h$] to represent [$t$, $s$], which receives as its region name the
          name of $P$
  initialize lists
          $B(t) = A(s) \leftarrow P$
          $B(s) = A(t) \leftarrow P$
end of case "start";

case "intersection":
  FIND($P$) yields the two line segments $s$ and $t$ whose intersection is
          $P$;
```



```
  $h$ ← SUCCESSOR($s$)
  $l$ ← PREDECESSOR($t$)
  INTERSECT($t$, $h$)
  INTERSECT($l$, $s$)
  catenate $A(t)*P*B(s)$
  $A(t)$ ← $A(s)*P$
  $B(s)$ ← $P*B(t)$
  permute $s$ and $t$, and modify the old entry [$t$, $s$] to
      represent the new interval [$s$, $t$], which receives as its region
      name the name of $P$
  initialize list $B(t) = A(s) \leftarrow P$
end of case "intersection".
```

All four cases of procedure TRANSITION are built from the same building blocks in slightly different combinations. Operations FIND, INSERT, DELETE, SUCCESSOR, and PREDECESSOR on the y structure are performed in time $O(\log|Y|)$. Other operations on the y structure that we have called "replace," "modify," or "permute" do not alter the structure of Y and can be done in time $O(1)$. INTERSECT is the only operation that involves the x structure (by means of INSERT into X) and can be done in time $O(\log|X|)$. All operations on the r structure take place at the head or tail of a list and are done in time $O(1)$.

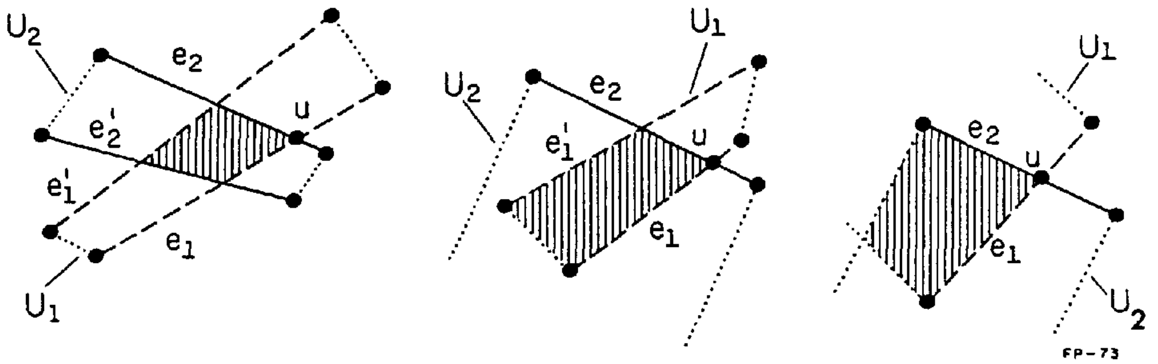
We have seen in Sec. 2.2 that $|X| = O(n + s) = O(n^2)$, and thus all the operations in TRANSITION can be done in time $O(\log n)$. Since the algorithm that sweeps the plane makes $n + s$ transitions, it runs in time $O((n + s)\log n)$ as stated earlier.

3. Intersection of Convex Maps

A map is a planar graph G embedded in the plane. Each vertex of G is represented as a point and each edge is represented as a straight line segment in such a way that edges intersect only at common vertices. A map subdivides the plane into r simply connected internal regions $R_1, \ldots, R_r$ and one external unbounded region R_0. A map is convex if each internal region is convex and the complement of the external region is convex.

Given two convex maps G_1 and G_2, with n_1 and n_2 vertices, respectively, and s intersections of edges of G_1 with edges of G_2, we show that the set of these s intersections can be computed in time $O(n \log n + s)$ and space $O(n)$, where $n = n_1 + n_2$. A straightforward application of the plane-sweep algorithm described in [2], which does not take advantage of convexity, yields this

Fig. 6. The domain U is shown cross-hatched.



result in time $O((n + s)\log n)$ and space $O(n + s)$. The most widely known geometric intersection problem that can be solved in time $O(n \log n + s)$ is that of computing intersections of rectangles with parallel sides; it has been studied by Bentley and Wood [3], McCreight [5], and others. Their results are not directly comparable to those of this section, since the restricted nature of the geometric objects suggests the use of specialized techniques tailored to the problem of handling vertical and horizontal line segments.

For $i = 1, 2$, let G_i be a convex map, and e_i an edge of G_i. Assume that no other edge passes through the intersection point u of e_1 and e_2 (degenerate cases can be handled with no difficulty.) Define r_i as the region of G_i such that $r_1 \cap r_2$ lies entirely to the left of a vertical line through u. (See Fig. 6.) We now define a plane domain U as follows. Let $e_1' \neq e_1$ be an edge—if it exists—on the boundary of r_1 which intersects e_2, and let e_2' be analogously defined. If e_i' exists, define U_i as the convex hull of e_i and e_i', otherwise U_i is the unbounded half-plane-strip orthogonal to e_i on the side of r_i; then let $U = U_1 \cap U_2$ (Fig. 6).

We claim that U contains in its interior no edge, nor any portion of edge, either of G_1 or of G_2. It suffices to show that U_i contains in its interior no edge, nor any portion of edge, of G_i. This is obvious when U_i is unbounded, because in this case U_i is contained in the unbounded exterior region of G_i; when U_i is bounded, then $U_i \subseteq r_i$ by convexity, and obviously the claim holds.

In order to aid the reader's intuition we first present an informal description of the technique. The intersection of the two maps G_1 and G_2 will be a map whose vertices are of two types: (*i*) original vertices either of G_1 or of G_2, and (*ii*) intersections of an edge of G_1 with an edge of G_2. For ease of reference, we call them V vertices and I vertices, respectively. The plane-sweep is conveniently broken down into the alternation of two activities; a phase of a primary sweep and one of a secondary sweep. The primary sweep is characterized by a scanning pointer that advances from V vertex to V vertex, proceeding from left to right. The processing associated with this primary sweep is very simple and reduces to an updating of the y structure, by deleting the edges incident to the current vertex v from the left and inserting those which are incident to v from the right. The visit of the I vertices, on the other hand, is assigned to the secondary sweep, which processes the edges issuing from v towards the right. In contrast to the general case examined in Sec. 2, where intersections found had to be stored in the x structure (a priority queue) for future processing, here the convexity of the regions of G_1 and G_2 allows a substantial simplification, i.e., the immediate processing of I vertices found in a march along the edges issuing from v toward the right. It is convenient to view the plane-sweep as characterized by a frontier line which cuts the plane and has to its left all the vertices—V vertices and I vertices—visited by the algorithm. The frontier is in general a polygonal line, which advances from left to right and plays the role of the cross section of Sec. 2. We now justify the simplification mentioned above.

Consider a vertical cross section γ (corresponding to the current position of the scanning pointer) passing through a vertex v of, say, G_1 [Fig. 7(a)] and suppose that edges e_1 and e_2, of G_1 and G_2, respectively, are adjacent in γ and intersect at I vertex u. By a preceding argument, the wedge comprised between the line γ, e_1, and e_2 does not contain edges, nor any portion of edges, of G_1 and G_2; therefore we immediately visit u and complete the region of which u is the "end" point. In addition, we advance the frontier to include u at its left [see Fig. 7(*b*)] and update the y structure. This update corresponds to exchanging the order of e_1 and e_2. After that we may proceed with the verification of whether e_1 and e_2 have further intersections with edges of G_2 and G_1, respectively. To be more specific, if the edges issuing to the right of the current V vertex are ordered counter clockwise, we start by testing whether the first and last elements of this sequence are involved in intersection-producing edge adjacencies. To organize the correct visits

Fig. 7. The current sweep section advances by one point.

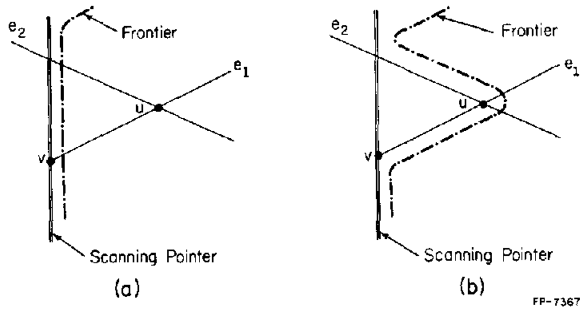
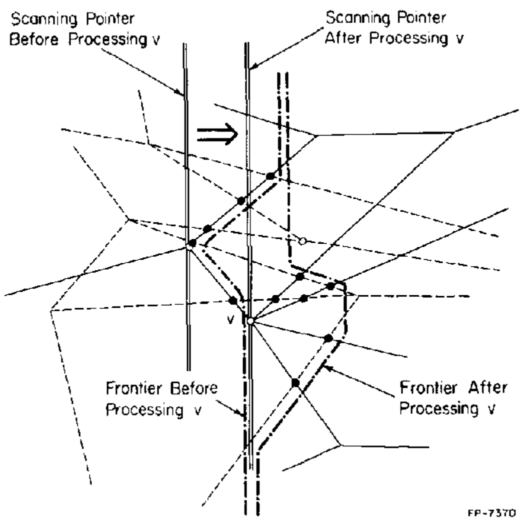


Fig. 8. Illustration of the respective displacements of sweep-line and frontier during one alternation of primary and secondary sweep. Edges of G_1 are shown in solid lines, those of G_2 in broken lines.


Scanning Pointer Before Processing v
Scanning Pointer After Processing v
v
Frontier Before Processing v
Frontier After Processing v
FP-7370

of the intersections, the adjacencies found are naturally placed into a first-in-first-out queue Q (in contrast to the priority queue used in Sec. 2). Processing any adjacency extracted from this queue may generate at most two new intersections—producing adjacencies, which, in turn, have to be placed into Q. It is clear, therefore, that a phase of secondary sweep is completed when Q becomes empty. In Fig. 8 we illustrate the positions of the scanning pointer and of the frontier before and after one alternation of primary and secondary sweep phases.

A description of the procedure is given below. For each vertex v (either of G_1 or G_2), we denote as $L(v)$ and $R(v)$ the sets of edges incident to v and lying, respectively, to its left and to its right. For all vertices—except the left and right extreme vertices of either map—the hypothesis of convexity guarantees that both $L(v)$ and $R(v)$ are nonempty. The y structure Y is as usual a dictionary and supports the operations of INSERT and DELETE in logarithmic time, and PREDECESSOR and SUCCESSOR in constant time. The x structure is now an array X. For simplicity of exposition we omit the r structure, which can be handled as in Sec. 2. Instead, we introduce the set I of intersections heretofore found, to be implemented as a linear list. The main algorithm SWEEP has the following structure.

```
procedure SWEEP:
  1. for each vertex $v$ of $G_1 \cup G_2$ do
sort counterclockwise the edges of $L(v)$ and $R(v)$
  2. sort the vertices of $G_1 \cup G_2$ by increasing abscissa and place them
in $X[1:n]$
  3. $Y \leftarrow \emptyset, I \leftarrow \emptyset$
  4. for $i := 1$ until $n-1$ do TRANSITION($X[i]$)
end of SWEEP;
```

Again, TRANSITION is the advancing mechanism of the sweep. It performs a complete cycle (a primary sweep followed by a secondary sweep) by advancing both the scanning pointer and the frontier line. It uses all data structures of SWEEP except X. In addition it employs a first-in-first-out queue Q of edge pairs, implemented as a linear list, for which "$\Leftarrow Q$" and "$Q \Leftarrow$", respectively, denote the operations of "remove" and "insert." The same notation applies to linear lists $L(\,)$ and $R(\,)$. P denotes a generic vertex of $G_1 \cup G_2$.

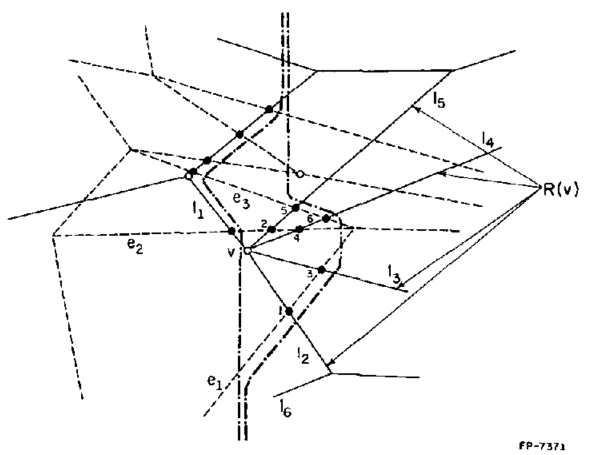
```
procedure TRANSITION($P$):
  1. $Q \leftarrow \emptyset$
  2. PRIMARY SWEEP
  3. SECONDARY SWEEP
end of TRANSITION;

procedure PRIMARY SWEEP:
  2. while $L(P) \neq \emptyset$ do
  3.    begin $e \Leftarrow L(P)$
  4.          $e_1 \Leftarrow$ PRED($e$)
  5.          DELETE($e$)
        end (* all edges incident to $P$ from the left have been deleted
        from $Y$ and $e_1$ is not incident to $P$ *)
  6. while $R(P) \neq \emptyset$ do
  7.    begin $e \Leftarrow R(P)$
  8.          if $e_1$ and $e$ belong to different maps
               then $Q \Leftarrow (e_1, e)$ (* this can only happen at the start of
               the while-loop* )
  9.          $e_1 \leftarrow e$
 10.          INSERT($e$)
 11.          $e_2 \leftarrow$ SUCC($e$)
        end (* all edges incident to $P$ from the right have been inserted
               into $Y$ and $e_2$ is not incident to $P$ *)
 12.    if $e_2$ and $e$ belong to different maps then $Q \Leftarrow (e, e_2)$
        (* at most two pairs of edges have joined $Q$ *)
end of PRIMARY SWEEP;

procedure SECONDARY SWEEP:
 13. while $Q \neq \emptyset$ do
       begin
 14.     $(e_1, e_2) \Leftarrow Q$
 15.     if $e_1$ and $e_2$ intersect then
          begin
 16.        $I \leftarrow I \cup (e_1, e_2)$
 17.        $e' \leftarrow$ PRED($e_1$), $e'' \leftarrow$ SUCC($e_2$)
 18.        if $e'$ and $e_2$ belong to different maps
              then $Q \Leftarrow (e', e_2)$
 19.        if $e_1$ and $e''$ belong to different maps
              then $Q \Leftarrow (e_1, e'')$
 20.        exchange $(e_1, e_2)$ in $Y$
          end (* at most two new edge pairs have joined $Q$ *)
       end
end of SECONDARY SWEEP;
```

It is convenient to illustrate the working of one cycle of TRANSITION by referring to Fig. 8, whose relevant portion is repeated in Fig. 9. Here, I vertices are numbered to reflect the order in which they are visited, that is, added to set I in step 16. Loop 2–5, executed just once, deletes edge l_1 and identifies e_1; next, loop 6–11 inserts l_2, l_3, l_4, l_5, and identifies e_2. The pairs (e_1, l_2) and (l_5, e_2) are placed in Q in steps 8 and 12, respectively. At this point the primary phase has initialized Q and the secondary phase starts. I vertex 1 is the first to be found and it generates the pair (e_1, l_3), whereas (l_6, l_2) is not a legal pair since both edges belong to the same map. By continuing this process, the I vertices are discovered in the displayed order and the edge pairs are input to Q in the following order:

$$(e_1, l_2), (l_5, e_2), (e_1, l_3), (l_4, e_2), (l_5, e_3), (e_1, l_4), (l_4, e_3)$$

Note that some pairs input to Q do not correspond to

Fig. 9. Ilustration of the working of algorithm TRANSITION on a portion of Fig. 8.



actual adjacencies such as (e_1, l_4) above. However, since adjacency is a necessary condition for intersection, no incorrect result is produced. As to the possible inefficiency, observe that each virtually visited I vertex generates at most two pairs; hence the total number of pairs generated is proportional to the number of I vertices. Intersections are found while advancing on edges from left to right and when an edge is deleted (step 5) all I vertices on this edge have already been found. Indeed, by referring to Fig. 7 and the pertaining discussion, one can visualize the advancement of the frontier one I vertex at a time, which stops when no more intersections are found. Moreover, when edges are deleted no adjacencies are produced. Indeed, suppose the contrary; assuming that the edges incident from left to $v \in G_1$ are being deleted, there is an edge e_1 of, say, G_1 right above v and an edge of e_2 of G_2 right below v. The existence of e_1 implies that v is not the rightmost vertex of G_1, whence $R(v) \neq \emptyset$, i.e., there is an edge e_3, issuing from v to the right, which separates e_1 from e_2.

The running time of the procedure SWEEP is easily shown to be $O(n \log n)$. By Euler's theorem on planar graphs, the number of edges is proportional to the number of vertices. Thus, steps 1 and 2 of SWEEP each use time $O(n \log n)$. As to TRANSITION, loops 2–5 and 6–11 each use time $O(\log n)$ per edge and thus time $O(n \log n)$ globally. Loop 13–20 (secondary sweep)—as well as steps 8 and 12—are executed $O(s)$ times, but each uses time $O(1)$. While this claim is obvious for steps 8 and 12 (additions to a FIFO queue), in loop 13–20 this performance can be achieved by specifying that access to e_1 and e_2 in Y be implemented by pointers rather than by standard dictionary manipulation. Thus we conclude that the map intersection algorithm runs in time $O(n \log n + s)$. The $O(n)$ space bound is obvious (disregarding list I).

4. Comparison of the Two Plane-Sweep Algorithms

In order to assess the generality of plane-sweep algorithms, we cast the two instances used in Secs. 2 and 3 into a common frame. Both algorithms have the following structure:

```
Algorithm SWEEP:
  1. Initialize x structure
                y structure
                task-specific data structures, such as R or Q
  2. while x structure not empty do
     2.1 P ⇐ next point from x structure
     2.2 TRANSITION(P)
where TRANSITION(P) is of the form
  1. with y locate an interval in the y structure;
     locally update the y structure,
  2. compute some new intersections and process these.
```

The y structure is identical for both algorithms. It stores the current cross section consisting of $O(n)$ entries in a data structure that supports the operations FIND, INSERT, DELETE in logarithmic time and the operations PREDECESSOR and SUCCESSOR in constant time.

The x structure is rather different for the two problems we have discussed. The simple case is illustrated by the convex map problem; all relevant transitions are known a priori, that is, the $n = n_1 + n_2$ vertices of the two given graphs. After they have been sorted they can be stored in any static data structure suitable for sequential processing (i.e., the operation NEXT takes constant time), for example, an array. The reason is that each intersection being computed can be processed entirely (an $O(1)$ operation) when it is encountered. Since it need not be considered a transition, it does not need to be stored for deferred processing and retrieved from the x structure (an $O(\log n)$ operation). By contrast, in the regions-of-a-polygon problem, a computed intersection must be treated as a transition, to be stored into and retrieved from the x structure. This requires a dynamic data structure, which supports the operations MIN and INSERT and cannot be as efficient as a static data structure. The mere fact that operations on the x structure now require logarithmic as opposed to constant time, however, would not affect the asymptotic time requirement of the algorithm, since this access time gets absorbed in the $n \log n$ term. The difference between $O(n \log n + s)$ and $O((n + s)\log n)$ is merely due to the fact that $n + s$ transitions move through the x structure as opposed to n.

The two algorithms presented can be combined to compute the regions of the intersection of two arbitrary maps (nonconvex) in time $O((n + s)\log n)$. In order to do this, however, the classification of points into the four categories: "bend," "end," "start," and "intersection" of Sec. 2.3, must be changed to deal with one general type of point where an arbitrary number of edges meet. This modification resolves the problem of degeneracy mentioned in Sec. 2.1. An intersection between more than two edges in the same point is simply treated as a vertex of high degree. By means of the same generalization, the regions of the intersection of two convex maps can be computed in time $O(n \log n + s)$.

Acknowledgments. We are grateful to the following people for helpful comments during the development of this paper: G. Beretta, H. R. Gnägi, P. Läuchli, T. M. Liebling, E. M. McCreight, K. Lieberherr, J. Waldvogel, and an anonymous referee.

Received 11/79; revised 9/81; accepted 12/81

References
1. A. Aho, J.E. Hopcroft, and J.D. Ullman. *Analysis and Design of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.
2. J.L. Bentley and T.A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Computers, C-28,* 9, (Sept. 1979) 643–647.
3. J.L. Bentley and D. Wood. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Trans. Compters. C-29,* 7, (July 1980) 571–576.
4. D.E. Knuth. *The Art of Computer Programming.* Vol. 3, Sorting and searching. Addison-Wesley, Reading, Mass., 1973.
5. E.M. McCreight. Efficient algorithms for enumerating intersecting intervals and rectangles. Report CSL 80-9 XEROX PARC, June 1980.
6. M.E. Newell and C.M. Sequin. The inside story on self-intersecting polygons. *LAMBDA, 1,* Second Quarter 1980, 20–24.
7. M.I. Shamos and D. Hoey. Closest-point problems. *16th IEEE Annual Symposium on Foundations of Computer Science.* Berkeley, CA. 1975, 151–162.
8. M.I. Shamos and D. Hoey. Geometric intersection problems. *17th IEEE Annual Symposium on Foundations of Computer Science.* Houston, TX. 1976, 208–215.

Technical Note
Management Science and Operations Research
Harvey Greenberg
Editor

Comment on Gamma Deviate Generation

Philip A. Houle
Drake University

Recent papers have presented methods for the generation of random variables from the gamma distribution function using a rejection method. A hazard due to unclear notation is examined which may have led practitioners to use an incorrect method.

CR Categories and Subject Descriptors: G.3.[**Probability and Statistics**]—*random number generation, probabilistic algorithms (including Monte Carlo)*
General Term: Algorithm
Additional Key Words and Phrases: gamma variates, rejection method, simulation

Tadikamalla [3] has presented a method for generating random variables from the gamma distribution with a nonintegral shape parameter α. Pritsker and Pegden [2] have used the method for $1 \le \alpha \le 5$ as the sampling procedure for gamma variables in SLAM. As reproduced by Pritsker and Pegden, the procedure is incorrect. The difficulty appears to stem from the choice of symbols in Tadikamalla's presentation.

Tadikamalla has presented the method for generating gamma variables in the following steps.

Step 1. Set $m = [\alpha]$, the integer portion of α; set $p = \alpha - m$, the fractional portion of α.

Step 2. Generate m independent uniform deviates $U_1, U_2, U_3, \ldots, U_m$ and compute $X = (-\log(\prod_{i=1}^{m} U_i))(\alpha/m)$.

Step 3. Generate another uniform deviate r.

Step 4. If $r \le T(X) = (X/\alpha)^p \exp(-p[(X/\alpha) - 1])$, return X as the required gamma variate. Otherwise go to Step 2.

Author's Present Address: Philip A. Houle, Computer Information Systems, College of Business Administration, Drake University, Des Moines, IA 50311.