CVGIP: GRAPHICAL MODELS AND IMAGE PROCESSING
Vol. 54, No. 1, January, pp. 1–12, 1992

A Polygonal Approach to Hidden-Line and Hidden-Surface Elimination*

MICHAEL T. GOODRICH†

Department of Computer Science, Johns Hopkins University, Baltimore, Maryland 21218

Received November 4, 1988; accepted June 4, 1991

We present algorithms for the well-known hidden-line and hidden-surface elimination problems. Our algorithms are optimal in the worst case, and are also able to take advantage of problem instances that are "simpler" than in the worst case. Specifically, our algorithms run in $O(n \log n + k + t)$ time, where n is the number of edges, and k (resp. t) is the number of intersecting pairs of line segments (resp. polygons) in the projection plane π. Our algorithms are based on a polygon-based strategy, rather than an edge-based strategy, and are quite simple. © 1992 Academic Press, Inc.

1. INTRODUCTION

The hidden-line and hidden-surface elimination problems are well known in computer graphics [9, 15, 16, 19, 22, 23, 28–31]. In the hidden-line elimination problem one is given a set of simple, nonintersecting planar polygons in 3-dimensional space, and a projection plane π, and wishes to determine which portions of the polygonal boundaries are visible when viewed in a direction normal to π, assuming all the polygons are opaque. (See Fig. 1.) In the related hidden-surface elimination problem one is also interested in determining which portions of the interiors of the polygons are visible. That is, if one colored each polygon with a unique color, then the problem would be to determine the color of each face of the drawing produced by a solution to the hidden-line elimination problem. Using the terminology of [31], we are interested in the *object space* versions of these problems; i.e., we want solutions that are independent of any specific rendering device.

We briefly review some of the efficient algorithms for these problems. In [9] Dévai gives an algorithm for hidden-line elimination running in $O(n^2)$ time and $O(n^2)$ space. In [19] McKenna shows how to solve the hidden-surface elimination in these same bounds. Both of these algorithms are optimal in the worst case, because there are problem instances that have $\Omega(n^2)$ output size [9, 19]. However, these algorithms always take $O(n^2)$ time, even if the size of the output is small (e.g., $O(1)$). In [22] Nurmi gives an algorithm for hidden-line elimination that runs in $O((n + k) \log n)$ time and $O((n + k) \log n)$ space, where k is the number of intersecting pairs of line segments in π (k is at most $O(n^2)$). Schmitt [28] is able to achieve this same bound using only $O(n + k)$ space. When the number of intersecting edges is not too large (i.e., $k \ll n^2/\log n$), these algorithms clearly run faster than $O(n^2)$. They are not worst-case optimal, however.

Recently, Chazelle and Edelsbrunner [6] have shown how to construct the graph of intersections of n line segments in the plane in optimal $O(n \log n + k)$ time, where k is as above. Since segment intersection is important in hidden-line elimination, one might think that this immediately improves the previous hidden-line elimination algorithms, but this is not the case. One exception is a hidden-line elimination algorithm by Schmitt [29], which runs in $O(n + k + r)$ time given the intersection graph, where r is the number of (*edge*, *polygon*) intersections in π. His algorithm makes a global visibility test for each edge, however, making his algorithm less desirable from a practical point of view.

In this paper we give an algorithm for the hidden-line elimination problem that is optimal in the worst case, and also takes advantage of problem instances that are "simpler" than in the worst case. Intuitively, our approach is to exploit the polygonal properties of the input, whereas previous algorithms concentrate more on edges. Our algorithm for the hidden-line elimination problem runs in $O(n \log n + k + t)$ time, where t is the number of (*polygon*, *polygon*) intersections (which is at most $O(n^2)$). Note that t is always less than r, the number of (*edge*, *polygon*) pairs. In fact, r can be significantly larger than t in general. For example, one can easily construct problem instances where t is $O(n)$ while r is $\Omega(n^{3/2})$ (e.g., $\sqrt{n}$ polygons having $\sqrt{n}$ edges each and projecting so as to be nested inside one another). Also note that t is in some sense independent of k, the number of (*segment*, *segment*) intersections, since there are problem instances where t is $O(n^2)$ and k is $O(1)$ (e.g., n triangles projecting so as to be nested) and other problem instances where t is

* Portions of this work appeared in preliminary form as M. T. Goodrich, "A Polygonal Approach to Hidden-Line Elimination," *Proc. of 25th Annual Allerton Conference on Communication, Control, and Computing*, Allerton IL, 1987, 849–858.

† Research supported by the National Science Foundation under Grants CCR-8810568 and CCR-9003299.

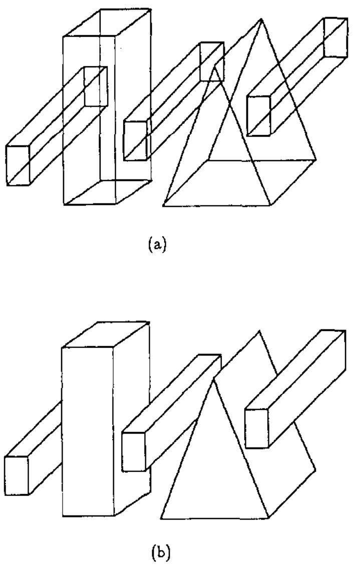

FIG. 1. An example of hidden-line elimination. (a) Shows the set of polygons before hidden-line elimination, and (b) shows it after.

$O(1)$ and k is $O(n^2)$ (e.g., two polygons shaped like forks with many tines and these tines form a cross-hatched pattern in π).

Our hidden-line method has been implemented [17] and benchmarked against the methods of Dévai [9] and Schmitt [28, 29]. The results of these benchmark tests suggest that our method is superior or competitive in running time to the algorithms of Dévai and Schmitt in practice. In addition, our method appears easier to implement than Schmitt's method [17].

We also show how to apply our polygonal approach to the hidden-surface elimination problem. Our algorithm for this problem also runs in $O(n \log n + k + t)$ time, assuming that the "overlap" relation, defined by the polygons and viewing direction, does not contain any cycles. Our algorithm does not need the back-to-front ordering of the polygons as input, however, since it embeds this computation as one of the steps of the algorithm. Besides being "better" than worst-case optimal, our algorithm is also of interest in that it provides an object-space version of the famous "painter's algorithm" (which is also known as the "list-priority" or "depth sorting" algorithm) [16, 31], where one renders the polygons in order by their distance from the viewing "eye," back to front, so that the low-level polygon-filling routines for the rendering device automatically eliminate nonvisible polygonal regions.

Both of our algorithms are quite simple, and, given the graph of line–segment intersections, can be implemented so that the only underlying data structures needed are linked lists and arrays. If one is willing to live with an algorithm that runs in $O(n \log n + k + t)$ expected time (rather than in the worst case), then one can use the methods of Clarkson [7] or Mulmuley [21] for constructing the graph of line–segment intersections. This would allow one to completely implement our algorithms in a simple fashion using just linked lists and arrays. One could also use the plane-sweeping method of Bentley and Ottmann [3], which runs in $O((n + k) \log n)$ time.

The remainder of this paper is divided into four sections. In the next section we describe the main data structure used in our algorithms, the polygon arrangement. In Section 3 we give our algorithm for hidden-line elimination, and in Section 4 we show how to solve the hidden-surface elimination problem. We conclude in Section 5.

2. THE POLYGON ARRANGEMENT

We begin our discussion by defining the polygon arrangement of a set of polygons, a structure that represents how a collection of polygons intersect in the plane. Let a set $\Gamma = \{P_1, P_2, \ldots, P_m\}$ of simple polygons in the xy-plane be given. For any polygon P_i we let ∂P_i denote the boundary of the polygon P_i, and assume, without loss of generality, that the vertices of ∂P_i are listed so that the interior of P_i would be on the lefthand side if we were to traverse the vertices of ∂P_i in the given order. We define the *representative vertex* of P_i, denoted $rep(P_i)$, to be the vertex with smallest y-coordinate from all the vertices of P_i with smallest x-coordinate, i.e., $rep(P_i)$ is the vertex that would be first if the vertices of P_i were sorted in increasing order lexicographically by (x, y)-coordinates. For any point p, we define the *downward* (*resp, upward*) *vertical shadow* of p in Γ to be the first point belonging to the boundary of a polygon in Γ that is intersected by the vertical ray emanating downward (resp., upward) from p, parallel to the y-axis. If no such point exists, then we take the downward (resp., upward) vertical shadow to be $-\infty$ (resp., $+\infty$). The *polygon arrangement* of Γ is defined on the following embedded planar graph $G = (V, E)$:

1. V consists of all points v that satisfy one of the following:
   (a) v is a vertex of a polygon in Γ,
   (b) v is an intersection point of the boundaries of two polygons in Γ, or
   (c) v is the vertical shadow of the representative vertex of a polygon in Γ;
2. E consists of all the (undirected) pairs (v, w), $v, w \in V$, that satisfy one of the following:
   (a) v and w are connected by a polygonal edge s and there is no point $z \in V$ between v and w on s, or
   (b) w is the vertical shadow of v and v is the representative vertex of some polygon in Γ.

To reflect the polygonal nature of the input one also must add some data structures that relate the edges and vertices of this graph to the polygons in Γ.

There are a number of ways one can represent the polygon arrangement, e.g., by generalizing the "winged edge" structure of Baumgart [2], the "quad edge" structure of Guibas and Stolfi [14], or the "doubly-connected edge list" structure of Muller and Preparata [20, 25]. In any case, the polygon arrangement would be stored as a collection of cross-referenced adjacency lists and arrays. In order to be specific in how one can implement the various aspects of our algorithms we give an implementation of the polygon arrangement here. The implementation we choose borrows ideas from each of the above data structures, but probably is most similar to the "winged edge" structure.

We store the vertices of V in an array VERT, the edges of E in an array EDGE, and the polygons of Γ in an array POLY. Each record of VERT corresponds to a vertex v, and contains the following fields: the x-coordinate of v, the y-coordinate of v, and a pointer to an adjacency list ADJACENCIES, which lists the indices of the edges in EDGE which are incident to v. Each record of POLY corresponds to a polygon in Γ, and contains a list, BOUNDARY, of the indices of vertices in VERT that are on the boundary of P_i, listed as they would occur if one were to traverse P_i from $rep(P_i)$ so as to keep the interior of P_i on the lefthand side. Each record of EDGE corresponds to an edge (v, w), and contains the following fields: (1) the indices of v and w in VERT, (2) a pointer, SIDE, which stores the index in POLY of the polygon (if any) that contains (v, w) on its boundary, (3) pointers to the positions of v and w in the BOUNDARY list for the SIDE polygon (assuming SIDE is defined), and (4) two lists, ENTER1 and ENTER2 (which we will define shortly). Note that using the SIDE pointer and the pointers into the BOUNDARY list for the SIDE polygon one can determine whether the SIDE polygon for (v, w) is on its right or on its left. Note that this also means that we consider an edge to belong to the boundary of at most one polygon. To allow for more general cases one would simply maintain several instances of the "same" edge, each one indicating that it belongs to the boundary of a different polygon. The list ENTER1 (resp., ENTER2) lists the indices of each polygon P_i in POLY such that v (resp., w) is on ∂P_i and the edge (v, w) intersects the interior of P_i. Intuitively, ENTER1 is the list of all the polygons that one enters in traversing (v, w) from v to w, and ENTER2 lists the polygons one enters in traversing (v, w) from w to v.

In addition to the above lists and arrays, we store all the representative vertices that do not have vertical shadows in Γ in a list COMP (since each entry corresponds to a connected component in G). See Fig. 2 for an example polygon arrangement.

Let us now turn to the construction of the polygon arrangement. As one might suspect, the bottle-neck computation is the construction of the graph of segment intersections and vertical shadows. Chazelle and Edelsbrunner prove the following:

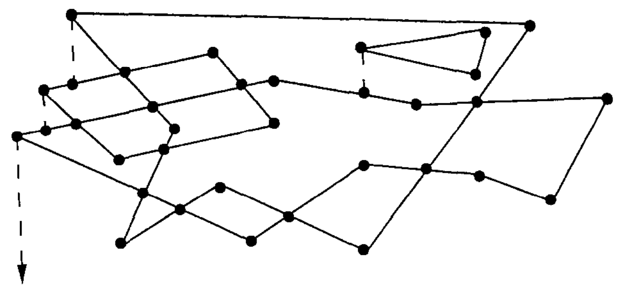
FIG. 2. An example of polygon arrangement. In this figure there are four polygons, and their arrangement represents 4 (*polygon, polygon*) intersections, 10 (*segment, segment*) intersections, and 3 vertical shadows. The circles denote the vertices of V and the straight lines denote the edges of E, where a solid edge denotes part of a polygonal segment and a dashed line denotes a vertical shadow edge. The vertical shadow edge drawn as an arrow denotes the situation where a representative vertex has $-\infty$ as its vertical shadow (hence is in the COMP list).

LEMMA 2.1 [6]. *Given a set S of n line segments in the plane, one can construct the graph of segment intersections and vertical shadows determined by S in $O(n \log n + k)$ time and $O(n + k)$ space.*

The algorithm of Chazelle and Edelsbrunner relies on a number of beautiful algorithmic techniques (including dynamic binary search trees (e.g., red–black trees [12, 32]), topological sweeping [10, 13], and segment trees [4]), but in somewhat involved. The authors have an implementation that consists of approximately 1,500 lines of C code, discounting driver and I/O routines, and claim that it is competitive with existing methods [6]. If one wants to use a simpler method to construct the polygon arrangement, and does not mind having an algorithm that is not worst-case optimal, then we recommend substituting the algorithm of Chazelle and Edelsbrunner by the randomized algorithm of Clarkson [7] or Mulmuley [21], which relies exclusively on the use of simple data structures such as linked lists and arrays. The worst-case complexity of the algorithms by Clarkson and Mulmuley is not as good as that of the algorithm by Chazelle and Edelsbrunner, but its expected running time matches the worst-case bound of their algorithm, as it runs in $O(n \log n + k)$ expected time (independent of the distribution of segments and intersection points). For completeness, and to illustrate its simplicity, we include a description of the algorithm by Clarkson.

LINE-SEGMENT INTERSECTION ALGORITHM [7]. Given a set S of n line segments in the plane, the line segments of S are added in random order, one by one, to a set U. An undirected graph, $H(U)$, of the intersection points, the segment endpoints, and the (upward and downward) vertical shadows of segment endpoints and intersection points is maintained as U grows. Note that $H(U)$ decomposes the plane into a collection of cells (i.e., faces) that are, more or less, trapezoidal. These trapezoidal cells are maintained in a list Q. For each edge e of $H(U)$ one stores pointers to the two cells in Q that are adjacent to e, and for each cell c in Q one stores a list of the (at most four) edges of $H(U)$ that bound c. A bipartite "conflict graph" $C(U)$ is also maintained as U grows. Its two vertex sets are the set of segments in $S - U$ and the set of cells in Q, respectively. There is an edge in $C(U)$ between a cell c in Q and a segment s in $S - U$ if $c \cap s \neq \emptyset$. When a segment s is added to U the cells that are adjacent to s in $C(U)$ must be deleted from Q, since s "cuts" each of them into smaller cells. For each cell c that is cut by s (as determined by the adjacency list for s in $C(U)$) one deletes c from Q and $C(U)$ and inserts into $H(U)$ and Q the (at most four) new cells that s cuts c into. For each such new cell c' one examines the list of segments in $C(U)$ that were adjacent to c in $C(U)$ to see which of these segments intersect with c', adding the appropriate adjacencies to $C(U)$ and $H(U)$ as necessary. One iterates this procedure until $U = S$.

Clarkson [7] shows that this simple procedure runs in $O(n \log n + k)$ expected time and space (he also shows how the space can be reduced to $O(n + k)$).

While the construction of the graph of segment intersections and vertical shadows is being performed it is important that for each line segment s one keep track of the polygon, P_i, that contains s on its boundary, as well as maintaining the edge on the boundary of P_i that immediately follows s. This allows one to easily construct the POLY array and all its accompanying BOUNDARY lists once the graph of segment intersections is constructed.

Let us suppose we have constructed this graph, with the extra information as just described. One can easily construct the VERT array, and its accompanying lists, by taking the vertices of this graph and deleting all vertical shadow vertices that are not the downward vertical shadows of representative vertices. Note that we can then have the COMP list, as well, by taking all vertices that have $-\infty$ as their downward vertical shadow. Similarly, it is fairly straightforward to construct the EDGE array by "stitching" back together any edges divided by vertices we deleted in constructing the VERT array. Using the extra information maintained during the construction of the graph of segment intersections and vertical shadows, as well as the input specifications of the polygons, we can then construct the POLY array and its accompanying BOUNDARY lists. Constructing the fields of each entry in the EDGE array is also straightforward, given this information, except for the construction of the ENTER1 and ENTER2 fields. To construct, say, the ENTER1 list for an edge $e = (v, w)$ one examines each edge $f = (v, u)$ that is incident to v (using the ADJACENCIES list for v), and determines whether the SIDE polygon for f contains e in its interior (using the pointers into the BOUNDARY list for the SIDE polygon of f). One inserts each such polygon into the list ENTER1. Constructing the ENTER2 list is similar.

Let us examine the time and space complexity of constructing the polygon arrangement. As already mentioned, constructing the graph of line segment intersection points and vertical shadows can be done in $O(n \log n + k)$ time using $O(n + k)$ space, where k denotes the number of (*segment*, *segment*) intersections. Given this graph, constructing all the lists and arrays of the polygon arrangement, except for the ENTER1 and ENTER2 lists, requires an additional $O(n + k)$ time, since the methods used in these constructions examine each edge and vertex in this graph $O(1)$ times. The construction of the ENTER1 and ENTER2 lists for each edge $e = (v, w)$ takes time proportional to the number of edges incident on v plus the number of edges incident on w. Since v (resp., w) is a vertical shadow, a segment endpoint, or an intersection point, the time of this construction is bounded by $O(1) * k_e$, where k_e is the number of segments that intersect the segment containing (v, w) at v, plus 1 (to account for the case when v is not an intersection point). Note that $\sum_{e \in E} k_e = O(k)$, since each intersection will be counted only twice by this accounting scheme. Thus, the time to construct all the ENTER1 and ENTER2 lists is $O(n + k)$. Therefore, the total time needed to construct the polygon arrangement is $O(n \log n + k)$ using $O(n + k)$ space. We summarize this section with the following theorem:

THEOREM 2.2. *Given a set Γ of simple polygons in the xy-plane, the polygon arrangement for Γ can be constructed in $O(n \log n + k)$ time and $O(n + k)$ space, where n denotes the number of polygonal edge segments in Γ and k is the number of pairs of intersecting line segments.*

We next describe how we use the polygon arrangement to do hidden-line elimination.

3. HIDDEN-LINE ELIMINATION

Suppose we are given a set $\Gamma = \{P_1, P_2, \ldots, P_m\}$ of simple, planar polygons in 3-dimensional space, as well as a projection plane π. The hidden-line elimination problem is to determine which portions of the polygonal boundaries are visible when viewed in a direction normal to π. Without loss of generality, we assume that π is the xy-plane, that the view direction is toward $(0, 0, -\infty)$, and that the vertices of each polygon P_i are listed so that the interior of P_i would be one the left if we were "walking" around the boundary of P_i in the order given with our feet pointing down toward $(0, 0, -\infty)$. In this section we present an algorithm for this problem which is both simple and efficient.

Since we will be dealing with 3-dimensional objects as well as their 2-dimensional projections, we make the following definitions. For any point p we let $x(p)$, $y(p)$, and $z(p)$ denote the x-, y-, and z-coordinates of p, respectively. We denote the set of polygons in Γ projected to π by Γ_π, and use P_i' to denote the projection of the polygon P_i to π. Given a point p in π, we use $\pi_i(p)$ to denote the point on the plane containing P_i that projects to p, and define the *coverage of p with respect to P_i* to be the number of polygons in Γ that obscure $\pi_i(p)$, i.e., the number of polygons intersected by the ray emanating out from $\pi_i(p)$ normal to π in the direction of the viewing "eye" $(0, 0, \infty)$. We present a high-level description of our algorithm below.

HIDDEN-LINE ELIMINATION ALGORITHM (HIGH-LEVEL DESCRIPTION).

Step 1. *Constructing the Polygon Arrangement.* We construct the polygon arrangement of Γ_π in this step, as described in the previous section. Since we are dealing with 3-dimensional polygons projected to the plane, we augment the polygon arrangement to keep track of some of the 3-dimensional information. Namely, for each polygon P_i' in POLY we store some additional fields to represent the plane which contains P_i. For example, if the plane containing P_i is determined by an equation, $ax + by + cz + d = 0$, then one could store the coordinates a, b, c, and d as fields in the POLY entry for P_i' to represent this plane.

Step 2. *Computing the Coverage of Representative Vertices.* In this step we use the polygon arrangement to compute the *coverage*, c_i, of each representative vertex $rep(P_i')$ with respect to P_i. We perform this step by traversing the polygon arrangement in a depth-first search fashion, storing the names of all the polygons that contain our current position as we go. Each coverage computation is done by examining the polygons in this set. The total time for this traversal is $O(n + k + t)$.

Step 3. *Computing Visible Edges.* In this step we use the polygon arrangement to "walk" around the boundary of each polygon P_i', starting at its representative vertex, computing the coverage of each edge portion as we go. All the portions of $\partial P_i'$ that have zero coverage are marked "visible" (or displayed). This takes $O(n + k)$ time and completes the algorithm.

END OF HIGH-LEVEL DESCRIPTION. We now describe in more detail how to perform each of the above steps. Since we have already shown how to perform Step 1, we begin with Step 2.

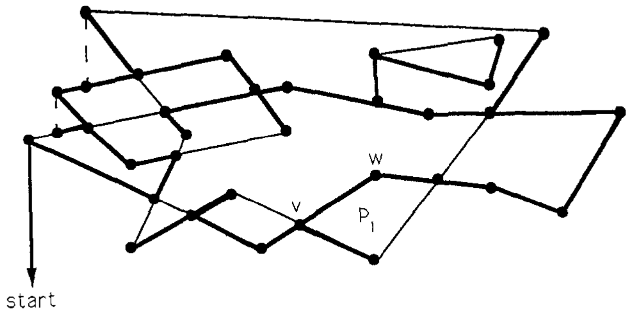

FIG. 3. The depth-first traversal. In going from v to w we are entering P_1 and in going from w to v (on the way back) we are leaving P_1.

3.1. *Step 2: Computing the Coverage of Representative Vertices*

We begin with some definitions. During any traversal of the polygon arrangement, suppose we are currently at a node v and moving to a node w. We say that we are *entering* (resp., *leaving*) the polygon P_i' along (v, w) if v (resp., w) belongs to the boundary of P_i' and (v, w) intersects the interior of P_i'. (See Fig. 3.) Note that for any edge (v, w) in the polygon arrangement the set of polygon(s) one enters (resp., leaves) in traversing the edge (v, w) from v to w corresponds exactly to the ENTER1 (resp., ENTER2) list for (v, w).

Step 2 can be performed by traversing the polygon arrangement of Γ_π in the following way. Let G denote the polygon arrangement of Γ_π, and recall that COMP is the set of all representative vertices that do not have vertical shadows in Γ_π. Starting with w, the first vertex in COMP, we remove w from COMP and begin traversing G in a depth-first-search fashion [1] starting with w. As we perform the traversal we maintain a list, D, of the polygons in Γ_π that contain our current position in the plane. Since w has $-\infty$ as its downward vertical shadow, there can be no polygons in Γ_π that properly contain w in their interior. Thus, we begin with D being empty. We represent each polygon in D by its index in the POLY list, and maintain a back-pointer from each polygon in POLY to its position in D (or store a **nil** pointer, if the polygon is not in D). As we are traversing G, each time we enter the interior of a polygon we insert its index in D, and each time we leave the interior of a polygon we delete its index from D. Also, when we are "returning" along already traversed paths, i.e., popping off the stack of visited nodes in the depth-first search, we reverse any operations we made along the way. Note that using the ENTER1 and ENTER2 lists we can immediately determine which polygons we are entering and leaving in traversing an edge. Thus, using these lists and the POLY array, we can implement the insertion or deletion of the index of each polygon we are entering or leaving in $O(1)$

time per polygon. At the time we encounter a representative vertex $rep(P_i')$ each polygon P_j' in D is such that $rep(P_i')$ lies in P_j'. This is because we include vertical shadows as well as intersection points in the polygon arrangement—so that if two polygons in Γ_π intersect, then their vertices are contained in the same connected component. Thus, we can compute the coverage, c_i, of this $rep(P_i')$ by examining all the polygons that are currently in D and count how many of them correspond to polygons in $\Re^3$ that obscure $\pi_i(rep(P_i'))$. In order to check if a polygon P_j obscures $\pi_i(rep(P_i'))$ one need only check whether the plane containing P_j lies before or behind $\pi_i(rep(P_i'))$, which can be determined in $O(1)$ time using the extra information we store with each polygon in POLY. When we return to w, completing a traversal of the connected component of G containing w, we continue the depth-first search starting with the next vertex in COMP. We repeat this traversal until we have visited all the vertices in COMP.

Let us analyze the time complexity of this traversal. Since we are traversing the polygon arrangement in a depth-first fashion, the set D will only change when we enter or leave a polygon, and then D only gains or loses one item, which can be performed in $O(1)$ time. The only other operation we perform in the traversal is computing the coverage of each representative vertex, which, for each $rep(P_i')$, can be performed in time proportional to the number of polygons stored in D when we encounter $rep(P_i')$. Since there is only one representative vertex for each polygon, and for each $rep(P_i')$ we only examine those polygons that properly contain $rep(P_i')$ in their interior, the number of comparisons we make for each P_i' will be at most the number of polygons in Γ_π that intersect P_i'. Thus, we make at most $O(t)$ comparisons overall, where t is the number of pairs of intersecting polygons in Γ_π. Since we are traversing the polygon arrangement in a depth-first fashion, we will visit each edge at most twice. Thus, this step takes $O(n + k + t)$ time overall.

3.2. *Step* 3: *Computing Visible Edges*

In this step we use the polygon arrangement to "walk" around the boundary of each polygon P_i' in Γ_π, in turn, starting with $rep(P_i')$. Specifically, we start with $rep(P_i')$ and walk around P_i' using its BOUNDARY list, always maintaining the coverage, with respect to P_i, of the point on P_i' that corresponds to our current position. We initialize a counter c to the value c_i, the coverage of $rep(P_i')$ with respect to P_i (which was computed in Step 2), and maintain the property that c is the coverage of our current position in the walk through the BOUNDARY list as follows: Let p be our current position in the BOUNDARY list and let q be the point we are moving to next in the traversal. If in going from p to q we enter a polygon P_j' such that P_j obscures $\pi_i(q)$, we increment the counter c, and if we leave a polygon P_j' such that P_j obscures $\pi_i(p)$, we decrement c. Note that we can determine the polygons we are entering using the ENTER1 and ENTER2 lists for the edges we are traversing, and that determining whether a polygon P_j obscures $\pi_i(p)$ or not can be done in $O(1)$ time by checking if the plane containing P_j is in front or behind $\pi_i(p)$. Any edges of BOUNDARY we traverse with $c = 0$ we mark as being "visible." This gives us all the visible portions of $\partial P_i'$, since there can be no polygons that obscure the corresponding portions of ∂P_i. The total time needed to traverse all the polygons in Γ_π in this manner is $O(n + k)$, since we traverse each vertex once and each intersection point twice. This completes the algorithm.

We summarize this section in the following theorem.

THEOREM 3.1. *Given a set* Γ *of simple, planar polygons in* $\Re^3$, *consisting of n total edges, and a projection plane* π, *the hidden-line elimination problem for* Γ *can be solved in* $O(n \log n + k + t)$ *time and* $O(n + k)$ *space, where k* (*resp. t*) *is the number of intersecting pairs of line segments* (*polygons*) *in* Γ_π.

4. HIDDEN-SURFACE ELIMINATION

In this section we show how to use the polygon arrangement to solve the problem of eliminating hidden surfaces. Let the input be as for the hidden-line elimination problem. We present a high-level description of our algorithm below.

HIDDEN-SURFACE ELIMINATION ALGORITHM (HIGH-LEVEL DESCRIPTION).

Step 1. *Constructing the Polygon Arrangement.* We construct the polygon arrangement of Γ_π in this step, as described in Section 2. As in the hidden-line elimination algorithm, for each polygon P_i' in POLY we store some additional fields to represent the plane which contains P_i.

Step 2. *Constructing the "Overlap" Relation.* In this step we construct a directed graph R that represents the "overlap" relation determined by the polygons in Γ. In particular, each vertex of R corresponds to a polygon in Γ, and there is an edge (i, j) in R if P_i obscures some part of P_j. (See Fig. 4.) We classify each edge (i, j) as to whether P_i' completely contains P_j' in its interior, or not. Also, we "prune" out each polygon P_i such that P_i is completely invisible because there is some other polygon P_j that completely obscures P_i, i.e., P_i' is inside P_j' and P_j is in front of P_i. We construct this relation by traversing the polygon arrangement similar to the way we did in the hidden-line elimination algorithm Steps 2 and 3. The total time for this traversal is $O(n + k + t)$.

Step 3. *Sorting the Polygons.* In this step we use the overlap relation computed in Step 2 to construct the list

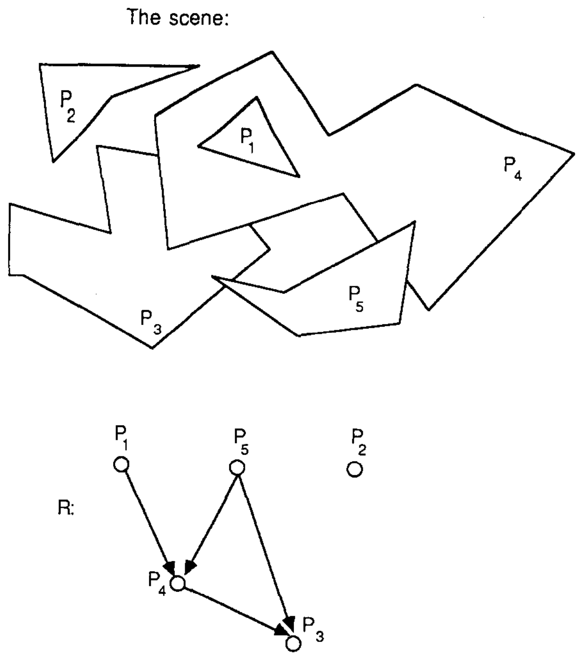

FIG. 4. The overlap relation R.

priority of the polygons. That is, we find a labeling of the polygons such that if P_i is in front of P_j then P_i has a higher label than P_j. This takes $O(t)$ time.

Step 4. *"Drawing" the Polygons Back-to-Front.* In this step we use the polygon arrangement and the information computed in the previous steps to simulate the painter's algorithm in object space. Initially, all the edges and faces in the polygon arrangement are marked "invisible." Starting with the polygon that is farthest from the viewing position, we activate each polygon one by one in order based on the list priority labels. When we activate a polygon P'_i we traverse the edges on the boundary of P'_i, marking them as being "visible," and mark which polygons are visible on their left and right sides. In addition, we traverse all the previously "visible" edges that are contained in the interior to P'_i and mark them as "invisible," since the activation of P'_i, in essence, covers them up. This takes $O(n + k)$ time.

End of High-Level Description. We begin the discussion of the details of the hidden-surface algorithm with Step 2, since the implementation of Step 1 was given in Section 2.

4.1. *Step 2. Constructing the "Overlap" Relation*

In this subsection we give the details for constructing a directed graph R that represents the "overlap" relation determined by the polygons in Γ. In particular, each vertex of R corresponds to a polygon in Γ, and there is an edge (i, j) in R if P_i obscures some part of P_j. Recall that we also classify each edge (i, j) as to whether P'_i completely contains P'_j in its interior, or not. In addition, we remove each polygon that is completely obscured by another. The method is as follows.

Use the COMP list as in the hidden-line algorithm to begin a depth-first traversal of the polygon arrangement. As before, we maintain a list D of all the polygons that contain our current position as we perform this traversal. Again, we insert into D the index of each polygon we enter during the traversal, and delete the index of each polygon we leave. It is when we come to a representative vertex that we perform a computation different to that used in the hidden-line elimination algorithm. Namely, upon reaching a representative vertex $rep(P'_i)$ (for the first time) we suspend the process of performing the depth-first search temporarily and call a procedure OVERLAP on P'_i, which will compute all the polygons that P'_i overlaps, i.e., all the polygons whose boundaries intersect the boundary of P'_i.

Procedure OVERLAP(P'_i).

Step 1. We begin by copying the entire list D into a workspace list C (we will discard C at the end of the OVERLAP procedure). We associate three bit fields with each polygon P'_j in C: INT, which will be **true** if and only if the boundary of P'_j intersects the boundary of P'_i; *OBSCURES*, which will be **true** if P_j obscures some part of P_i; and OBSCURED, which will be **true** if some part of P_j is obscured by P_i. Initially, for each polygon P'_j in C, its INT bit is **false,** and its OBSCURES bit (resp., its OBSCURED bit) is **true** if and only if P_j is in front of (resp., behind) $\pi_i(rep(P'_i))$. We then use the BOUNDARY list for POLY[i] to walk around the boundary of P'_i and update these bits as we go. Each time we encounter an intersection point p, say with a polygon P'_j, we check if P'_j is in C (which can be done in $O(1)$ using backpointers from the POLY array). If P'_j is not in C, then we add it to C. In either case, we set the INT bit associated with P'_j to **true.** Then, we set the OBSCURES bit for P'_j to **true** if P_j is in front of $\pi_i(p)$. Similarly, we set the OBSCURED bit for P'_j to **true** if P_j is behind $\pi_i(p)$.

Step 2. After we complete the walk around P'_i and return to $rep(P'_i)$ we search through the entire list, C, to verify that the obscuring relations we have just discovered are consistent. Specifically, for each P'_j in C, we check if the OBSCURES and OBSCURED bits for P'_j are both set. If both these bits are set, then we stop the hidden-surface procedure and either query the user or apply a heuristic as explained by Hearn and Baker [16] to resolve this ambiguity (by splitting P_i or P_j). For the remainder of the discussion let us assume that the OBSCURES and OBSCURED bits are consistent.

Step 3. In this final step of the OVERLAP procedure we determine whether or not P_i is completely obscured by some other polygon, and if it is not, we add the edges involving P_i to R (i.e., edges of the form (i, j) or (j, i)) and set the INT bits of each such edge. In particular, we search through C to see if the INT bit associated with a polygon P_j' in C is **false** while its OBSCURES bit is **true**. If there is such a polygon, then we mark P_i' as being "invisible," we delete all the space used for the list C, we delete all references to P_i from R, and we return back to the depth-first search procedure. If P_i is not completely obscured by some P_j, then we search through C one more time. For each P_j' in C, if P_j' is not marked "invisible," then we add the edge (j, i) to R if OBSCURES (P_j') is **true** and we add the edge (i, j) to R if OBSCURED (P_j') is **true.** We then set the INT bit for this new edge to the INT bit associated with P_j' in the list C (which was computed in the previous step). As soon as we are done with a P_j' in C, we delete all the space used for P_j' in C. When we have performed this computation for each P_j' in C we return to the depth-first search procedure, picking up where we left off. This completes the OVERLAP procedure.

Since at the time we reach $rep(P_i')$ in the depth-first search the list D is the list of all polygons that contain $rep(P_i')$, and each polygon has exactly one representative, the amount of work we perform in the depth-first traversal is $O(n + k + t)$. The total amount of work spent in the OVERLAP procedure for each polygon is proportional to the number of vertices in BOUNDARY plus the number of polygons that completely contain P_i'. Thus, the total amount of work spent in Step 2 is proportional to $O(n + k + t)$. In the following lemmas we characterize what the directed graph R represents.

LEMMA 4.1. *Let P_i be a polygon in Γ. P_i is completely obscured by some polygon P_j if and only if P_i' has been marked "invisible."*

Proof. ($\Leftarrow$:) Suppose P_i' has been marked "invisible." Then there is a polygon P_j' in C such that P_j obscures $\pi_i(rep(P_i'))$ and the INT bit for P_j' is set to **false.** The only way that the INT bit for P_j' can be set to **false** is if the boundary of P_j' does not intersect the boundary of P_i'. That is, P_i' is completely contained in the interior of P_j'. Thus, either P_i is completely obscured by P_j or P_i is in front of P_j. Since P_j obscures a point, $\pi_i(rep(P_i'))$ on P_i, P_j must obscure all of P_i.

($\Rightarrow$:) Suppose P_i is completely obscured by some polygon P_j. Then, when we reach $rep(P_i')$ in the depth-first traversal of the polygon arrangement, P_j' must be in the list D. Also, P_j' must completely contain P_i' in its interior. Thus, the INT bit for P_j' will not be set to **true** in the OVERLAP procedure for P_i'. Therefore, since P_j must obscure $\pi_i(rep(P_i'))$, P_i' must be marked "invisible" during the OVERLAP procedure. ∎

LEMMA 4.2. *Let P_i and P_j be two polygons in Γ such that neither P_i nor P_j completely obscures the other. Then P_i obscures some part of P_j if and only if there is an edge (i, j) in R.*

Proof. The "if" direction follows immediately from the discussion of the OVERLAP procedure, so suppose P_i obscures some part of P_j. We want to show that the edge (i, j) is in R. There are two cases.

Case 1. P_i' is completely contained in P_j'. Then when we reached $rep(P_j')$ in the depth-first traversal the polygon P_j' was in D. Thus, when we initialized the OBSCURED bit for P_j' it was set to **true.** Since it is always the case that once such a bit is set to **true** it is never set to **false,** we must have added an edge (i, j) to R at the end of the OVERLAP procedure for P_j'.

Case 2. The boundary of P_i' intersects the boundary of P_j'. Since P_i obscures some part of the polygon P_j, there must be some point p that is an intersection point of P_i' and P_j' such that P_i obscures $\pi_j(p)$. This must have been discovered in the OVERLAP procedure for P_j'; hence, we must have added an edge (i, j) to R during the OVERLAP procedure for P_j'. ∎

Note that these two lemmas imply that if the overlap relation, defined by the polygons in Γ and the projection plane π, does not contain cycles, then the graph R is a directed acyclic graph that represents it, except that all pairs (i, j) such that P_i completely obscures P_j are absent from R.

The alert reader may also have noted that an edge (i, j) may appear twice in R—inserted once when we discovered that P_i obscures P_j and once when we discovered that P_j is obscured by P_i. This does not cause any trouble for us, though, since the size of R is $O(t)$ regardless of whether some edges appear twice or not, and, as we will see, the fact that an edge can appear twice in R will not corrupt the sorting step (Step 3), which comes next.

4.2. *Step* 3. *Sorting the Polygons*

In this step we construct the list priority of the polygons. From the previous step we have a graph R that represents the polygon-overlap relation. We begin by constructing a depth-first search tree of R, starting from those nodes that do not have any edges coming into them. The graph R is acyclic if and only if there are no back edges in the depth-first search tree (i.e., nontree edges (v, w) such that the DFS number for v is greater than that for w) [1]. Note that the existence of two copies of some edges does not corrupt this test. Thus, we can check in $O(t)$ time whether the graph R is acyclic or not. If it is not acyclic, then we stop the hidden-surface elimination procedure and print out enough information about the cycles that were discovered to enable the user or some heuristic procedure (e.g., [16]) to resolve the ambiguities (by splitting the appropriate polygons). Let us proceed with the discussion assuming that R is acyclic.

The graph R represents a partial order. To construct a valid list priority of the polygons we must embed this partial order in a total order. That is, we must assign integer labels to the polygons so that if (i, j) is in R then the label for P_i is greater than the label for P_j. Note that this is exactly the topological sorting problem; hence, can easily be solved in $O(t)$ time given the depth-first search tree for R (see Aho, Hopcroft, and Ullman [1] or Knuth [18]). Let $prio(P_i)$ denote the list priority label of polygon P_i. The only computation left is that of Step 4.

4.3. *Step* 4. *"Drawing" the Polygons Back-to-Front.*

In this step we use the polygon arrangement and the information computed in the previous steps to simulate the painter's algorithm in object space.

We give each edge e in the polygon arrangement three more fields: VIS, a bit that is **true** if and only if e is visible; LEFT, the name of the polygon that is visible on the lefthand side of e; and RIGHT, the name of the polygon that is visible on the righthand side of e. Initially, for each edge in the polygon arrangement, its VIS is **false,** and its LEFT and RIGHT fields are undefined. Let PRIO be the list of all the polygons in R sorted in increasing order by their *prio* labels computed in the previous step. Note that the first polygon in PRIO is farthest from the viewing direction and the last polygon in PRIO is the closest. The main computation for Step 4 is to "activate" the polygons in PRIO, one by one, starting with the first polygon in PRIO. In activating a polygon P_i we update the VIS, LEFT, and RIGHT fields of edges in the polygon arrangement to indicate that the polygon P_i is currently the frontmost polygon. When the algorithm completes we will have a representation of a solution to the hidden-surface elimination problem. The details of this polygon activation procedure follow.

We begin Step 4 by deleting from R all those edges whose INT bit is set to **true,** and let $\hat{R}$ denote the graph that is left. Thus an edge (i, j) is in $\hat{R}$ if and only P_i' is completely contained in P_j' and P_i obscures some part of P_j. (The relation $\hat{R}$ will be used to compute "background" polygons.)

The remainder of Step 4 involves the iterative examination of each of the polygons in PRIO. Suppose that we have already activated i polygons, and let P_j be the $(i + 1)$st polygon in PRIO. We begin the activation of P_j by determining the background polygon for P_j, i.e., the polygon P_k with highest priority among those polygons that completely contain P_j' in their interior. We do this by examining all the edges emanating out from j in $\hat{R}$ and find the polygon P_k in this group with highest $prio(P_k)$ value. If there is no such polygon P_k, then we say that the background for P_j' is $-\infty$.

The activation of P_j' consists of two steps: (1) indicating that P_j' is visible, and (2) "painting" out all the edges that P_j' makes invisible. In the first step we start with $rep(P_j')$ and traverse the edges of the BOUNDARY list for P_j' setting their VIS, LEFT, and RIGHT fields as we go. Before we begin this traversal we initialize a variable, CURRIGHT, to the background face for P_j'. For each edge e in BOUNDARY we set e's VIS bit to **true,** we set e's LEFT field to P_j, and we set e's RIGHT field to CURRIGHT. If we come to an edge that crosses P_j' and has its VIS bit set to **true,** then we update CURRIGHT to be the polygon that will be on our right after crossing that edge. We can determine that polygon by examining the LEFT and RIGHT fields for that edge. There is one problem, however, and that is that upon encountering the first edge e whose VIS bit is set to **true** we may discover that the CURRIGHT value we started with was wrong (based on the LEFT and RIGHT fields for that edge). Such a mistake can occur if, as may often be the case, the boundary polygon for P_j is not the polygon directly below $rep(P_j')$. If this should happen, then we mark our current position in BOUNDARY and march back to $rep(P_i')$, updating each edge's RIGHT field to the correct value. In particular, if we set any RIGHT fields to the background polygon, then we must update these fields to this newly discovered polygon. When we complete this "backpatching" computation we return to where we left off in BOUNDARY and continue our traversal. When we complete the traversal of the BOUNDARY list for P_j' we will have set all the edges of BOUNDARY to indicate that P_j is not the closest polygon to the viewing direction. We have not, however, "painted" out the edges that P_j' has made invisible. To do that we must perform the second step in our activation procedure.

In the second step of our activation procedure we again traverse the BOUNDARY list for P_j'. When we come to cross an edge e (intersecting P_j') that has its VIS bit set to **true,** then we call a "little" depth-first search procedure that traverses the polygon arrangement starting with e and restricts its movements to those edges in the interior of P_j' that have their VIS bits set to **true.** For each edge we traverse in this depth-first search procedure we set its VIS bit to **false.** Note that for this procedure to be efficient we must be able to traverse the edges whose VIS bits are set to **true** without having to consider edges whose VIS bits are set to **false.** This is not a problem, however, if we keep two versions of the ADJACENCIES list—one for edges with their VIS bit set to **true** and one for edges with their VIS bit set to **false.** When we complete this DFS procedure we continue our traversal of the BOUNDARY list, looking for other edges that cross P_j' and have their VIS bit set to **true.** When we complete this second traversal of BOUNDARY we consider P_j' to be activated, and we repeat the above activation procedure with the next polygon in PRIO.

When we complete the activation of the last polygon in PRIO we will have a representation of a solution to the hidden-surface elimination problem. Namely, we have a graph of visible edges, and for each visible edge e we have the name of the polygon that is visible on the lefthand side of e and the name of the polygon that is visible on the righthand side of e.

Let us examine the running time of Step 4. Determining the background polygon for all the activation steps takes at most $O(t)$ steps. Since, for any BOUNDARY list, we traverse an edge in BOUNDARY at most two times (not just once, because of possible back-patching), traversing the edges in all the BOUNDARY lists can be done in $O(n + k)$ time. In performing the little depth-first searches to "paint" out edges that become invisible we restrict our computation to edges that have their VIS bits set to **true.** After this computation, these edges have their VIS bits set to **false,** and they are never set back to **true;** hence, never traversed again. Thus, the total amount of time spent in all these little depth-first search procedures is $O(n + k)$. Therefore, Step 4 can be implemented in $O(n + k + t)$ time. The next lemma establishes the correctness of Step 4.

LEMMA 4.3. *After Step* 4 *completes, and edge in the polygon arrangement is visible if and only if its VIS bit is set* ***true.*** *Moreover, if an edge is visible, then its LEFT and RIGHT fields correctly identify the polygonal faces that are on its left and right.*

Proof. The proof is by induction on the number of polygons activated from the PRIO list. The claim is vacuously true when there are zero polygons activated, since all the edges in the polygon arrangement have their VIS bits initialized to **false.**

Suppose the lemma is true after the first i polygons have been activated from PRIO. Consider the activation of the $(i + 1)$-st polygon, P_j, in PRIO. Because of the way we constructed the PRIO list, the polygon P_j is not obstructed by any of the polygons that have already been activated (this is the reason why the painter's algorithm works correctly). In activating P_j' we set the VIS bit of each edge in the BOUNDARY list for P_j' to be **true.** This is clearly correct. It is also correct that we set each of their LEFT fields to be the polygon P_j, since P_j is defined to be on the left of all the edges on its boundary. By the induction hypothesis, all the edges that are not contained in the interior of P_j' are correctly labeled. Thus, in the first traversal of BOUNDARY, if we ever cross an edge that has its VIS bit set to **true,** then we correctly set the RIGHT fields of the edges in BOUNDARY (since their values are determined by the visible edges we cross in traversing BOUNDARY). If, on the other hand, we never cross any visible edges in traversing BOUNDARY, then there can be only one polygon on the righthand side of the edges in BOUNDARY, and that polygon must be the background polygon, which we computed in initializing CURRIGHT. Therefore, we correctly assign the RIGHT fields of all the edges in BOUNDARY. The only other edges that are affected by the activation of P_j' are those previously-visible edges contained in the interior of P_j'. Recall that we deleted edges in the interior of P_j' by performing "little" depth-first searches on edges whose VIS bits are set to **true** starting from edges that cross the boundary of P_j'. Hence, the only way we could have missed an edge e contained in the interior of P_j' whose VIS bit was set to **true** is if all paths that lead to e from the boundary of P_j' contain an edge whose VIS bit is set to **false** (and each such path visits the "false" edge before e). By the induction hypothesis, the only way that an edge can have its VIS bit set to **false** is if it is not visible. Let H be the component of G consisting of each currently visible edge e' (i.e., each edge with its VIS bit set **true**) such that there is a path of currently visible edges from e to e'. Then the edges of H must be completely contained in the interior of P_j', and all the edges of H must belong to polygons that are completely contained in the interior of P_j'. This of course implies that P_j completely obscures these polygons. But there cannot be any polygons in PRIO that are completely obscured by P_j, since we removed all such polygons from the graph R in Step 3 (to construct the graph $\hat{R}$). Therefore, the edge e cannot exist. That is, we correctly "paint out" all the edges that become invisible as a result of activating P_j'. This completes the proof. ■

We summarize the discussion of this section in the following theorem.

THEOREM 4.4. *Suppose one is given a set* Γ *of simple, planar polygons in* $\Re^3$, *consisting of n total edges, and a projection plane* π, *such that the "overlap" relation determined by* Γ *and* π *does not contain any cycles. Then the hidden-surface elimination problem for* Γ *and* π *can be solved in* $O(n \log n + k + t)$ *time and* $O(n + k + t)$ *space, where k (resp. t) is the number of intersecting pairs of line segments (polygons) in* π.

5. CONCLUSION

In this paper we gave algorithms for hidden-line elimination and hidden-surface elimination that are optimal in the worst case and are also able to take advantage of problem instances that are "simpler" than in the worst case. Our approach was based on the idea of using a structure that we called the polygon arrangement and is an example of the paradigm from computational geometry of reducing geometric problems to graph problems. Our algorithms both run in $O(n \log n + k + t)$ time, where

n is the number of polygon edges, k is the number of (*segment*, *segment*) intersections in the projection plane, and t is the number of (*polygon*, *polygon*) intersections in the projection plane.

Our hidden-line elimination algorithm was implemented by Hostetler [17] and compared with the methods of Dévai [9] and Schmitt [28, 29] on a number of benchmark examples. For various "randomly" generated problem instances, our method was always competitive with these other methods, and on some instances its running time was 88% that of Schmitt's algorithm and only 12% that of Dévai's algorithm [17]. Moreover, these improvements were for scenes made up of relatively few polygons (e.g., between 40 and 80 polygons). Also, Hostetler reports that our method was easier to implement than Schmitt's method. Thus, we expect our method to be quite efficient in practice.

Our hidden-surface elimination algorithm provides an object space analog to the famous "painter's algorithm" [31, 16], and achieves the above performance bounds assuming that the "overlap" relation, defined by the input polygons and the projection plane, does not contain any cycles (we made no restrictive assumptions about the input for our hidden-line elimination algorithm). It is difficult to characterize the running time of our algorithm if the "overlap" relation does contain cycles, because the method for resolving such situations is a heuristic based on the idea of "cutting" polygons into smaller pieces. Thus, the best known worst-case efficient algorithm for the most general version of the hidden-surface elimination problem is still the algorithm by McKenna, which runs in $O(n^2)$ time and space [19]. It is an open problem to improve these worst-case bounds without making any restrictive assumptions about the input.

Algorithms such as ours, where the running time is proportional to the size of the input and the number of intersections the input determines, are known as "intersection-sensitive" algorithms. Another important class of algorithms are those that are "output-sensitive," where the running time depends on the size of the input and the size of the output. The only known output-sensitive algorithms for hidden-line elimination are either suboptimal [24] or are for the restricted case when the each edge on the input polygons is parallel to one of the coordinate axes [5, 8, 15, 11, 26], or the input is a terrain [27]. Thus, the existence of an optimal output-sensitive algorithm (i.e., one that runs in $O(n \log n + a)$ time, where a is the size of the input) remains an open problem.

ACKNOWLEDGMENTS

We thank Mikhail J. Atallah, Larry Blythe Hostetler, Michael McKenna, Joseph O'Rourke, and E. S. Panduranga for conversations concerning the topics of this paper.

REFERENCES

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison–Wesley, Reading, MA, 1974.
2. B. G. Baumgart, A polyhedron representation for computer vision, in *Proceedings, 1975 AFIPS National Computer Conference 44*, pp. 589–596, AFIPS Press, 1975.
3. J. L. Bentley and T. Ottmann, Algorithms for reporting and counting geometric intersections, *IEEE Trans. Comput.* **C-28,** 1979, 643–647.
4. J. L. Bentley and D. Wood, An optimal worst case algorithm for reporting intersections of rectangles, *IEEE Trans. Comput.* **C-29,** 1980, 571–576.
5. M. Bern, Hidden surface removal for rectangles, *J. Comput. Syst. Sci.* **40,** 1990, 49–69.
6. B. Chazelle and H. Edelsbrunner, An optimal algorithm for intersecting line segments in the plane, in *Proceedings, 29th IEEE Symposium on Foundations of Computer Science, 1988*, pp. 590–600.
7. K. L. Clarkson, Applications of random sampling in computational geometry, II, in *Proceedings, 4th ACM Symposium on Computational Geometry, 1988*, pp. 1–11.
8. M. de Berg and M. H. Overmars, *Hidden Surface Removal for Axis-Parallel Polyhedra*, Technical Report RUU-CS-90-21, Dept. of Computer Science, Utrecht Univeristy, 1990.
9. F. Dévai, Quadratic bounds for hidden-line elimination, in *Proceedings, 2nd ACM Symposium on Computational Geometry. 1986*, pp. 269–275.
10. H. Edelsbrunner and L. J. Guibas, Topologically sweeping an arrangement, in *Proceedings, Proc. 18th ACM Symposium on Theory of Computing, 1986*, pp. 389–403.
11. M. T. Goodrich, M. J. Atallah, and M. Overmars, An input-size/output-size trade-off in the time-complexity of rectilinear hidden surface removal, in *Proceedings, 17th International Conference on Automata, Languages, and Programming, 1990.*
12. L. J. Guibas and R. Sedgewick, A dichromatic framework of balanced trees, in *Proceedings, 19th IEEE Symposium on Foundations of Computer Science, 1978*, pp. 8–21.
13. L. J. Guibas and R. Seidel, Computing convolutions using reciprocal search, in *Proceedings, 2nd ACM Symposium on Computational Geometry, 1986*, pp. 90–99.
14. L. J. Guibas and J. Stolfi, Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams, *ACM Trans. Graphics* **4,** 1985, 75–123.
15. R. H. Güting and T. Ottmann, New algorithms for special cases of the hidden line elimination problem, *Comput. Vision Graphics Image Process.* **40,** 1987, 188–204.
16. D. Hearn and M. P. Baker, *Computer Graphics*, Prentice–Hall, Englewood Cliffs, NJ, 1986.
17. L. B. Hostetler, Jr., *A Comparison of Three Hidden Line Removal Algorithms*, Technical Report 89-02, Dept. of Computer Science, Johns Hopkins Univ., 1989.
18. D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison–Wesley, Reading, MA, 1968.
19. M. McKenna, Worst-case optimal hidden-surface removal, *ACM Trans. Graphics* **6** No. 1, 1987, 19–28.
20. D. E. Muller and F. P. Preparata, Finding the intersection of two convex polyhedra, *Theoret. Computer. Sci.* **7,** No. 2, 1978, 217–236.
21. K. Mulmuley, A fast planar partition algorithm, I, in *Proceedings, 29th IEEE Symposium on Foundations of Computer Science, 1988*, pp. 580–589.

22. O. Nurmi, A fast line-sweep algorithm for hidden line elimination, *BIT* **25,** 1985, 466–472.

23. T. Ottmann and P. Widmayer, Solving visibility problems by using skeleton structures, in *Proceedings, 11th Symposium on Mathematical Foundations of Computer Science, 1984,* pp. 459–470.

24. M. H. Overmars and M. Sharir, Output-sensitive hidden surface removal, in *Proceedings, 30th IEEE Symposium on Foundations of Computer Science, 1989,* pp. 598–603.

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

26. F. P. Preparata, J. S. Vitter, and M. Yvinec, *Computation of the Axial View of a Set of Isothetic Parallelepipeds*, Laboratoire d'Informatique de L'Ecole Normal Supérieure, Départment de Mathématiques et d'Informatique, Report LIENS-88-1, 1988.

27. J. H. Reif and S. Sen, An efficient output-sensitive hidden-surface removal algorithm and its parallelization, in *4th Symposium on Computational Geometry, 1988,* pp. 193–200.

28. A. Schmitt, *On the Time and Space Complexity of Certain Exact Hidden Line Algorithms*, Universität Karlsruhe, Fakultät für Informatik, Report 24/81, 1981. Cited in [15].

29. A. Schmitt, Time and space bounds for hidden line and hidden surface algorithms, *EUROGRAPHICS '81,* pp. 43–56.

30. S. Sechrest and D. P. Greenberg, A visibility polygon reconstruction algorithm, *ACM Trans. Graphics* **1,** No. 1, 1982, 25–42.

31. I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, A characterization of ten hidden-surface algorithms, *Comput. Surveys* **6,** No. 1. 1974, 1–25.

32. R. E. Tarjan, *Data Structures and Network Algorithms*, SIAM, Philadelphia, PA, 1983.