RECONSTRUCTING VISIBLE REGIONS FROM VISIBLE SEGMENTS

W.R. Franklin and V. Akman

RUU-CS-86-5

March 1986

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

RECONSTRUCTING VISIBLE REGIONS FROM VISIBLE SEGMENTS

W.R. Franklin and V. Akman

Technical Report RUU-CS-86-5
March 1986

Department of Computer Science
University of Utrecht
P.O. Box 80.012, 3508 TA Utrecht
the Netherlands

RECONSTRUCTING VISIBLE REGIONS FROM VISIBLE SEGMENTS

Wm. Randolph Franklin (*)
Electrical, Computer, and Systems Eng. Dept.
Rensselaer Polytechnic Institute
Troy, N.Y. 12180, USA

Varol Akman
Dept. of Computer Science, University of Utrecht
Budapestlaan 6, P.O. Box 80.012
3508 TA Utrecht, the Netherlands

March, 1986

ABSTRACT

An algorithm is presented for reconstructing visible regions from visible edge segments in object space. This has applications in hidden surface algorithms operating on polyhedral scenes (e.g. W.R. Franklin, "A linear time exact hidden surface algorithm," *ACM Computer Graphics* 14(3), 117-123, 1980). A special case of reconstruction can be formulated as a graph problem: "Determine the faces of a straight-edge planar graph given in terms of its edges." This is accomplished in $O(n \log n)$ time using linear space for a graph with n edges, and is worst-case optimal. (The graph may have separate components but the components must not contain each other.) The general problem of reconstruction is then solved by applying our algorithm to each component in the containment relation.

Keywords: hidden surface removal, polyhedra, holes, object space, straight-edge planar graph, point-location.

INTRODUCTION

The hidden surface problem in computer graphics has been an important area of study for the last two decades. For a somewhat dated but otherwise excellent survey see [16]; a bibliography is provided by Griffiths [10]. The majority of the currently used algorithms for hidden surface removal operate in *image space*, the realm of (raster) display devices. When the accuracy of the output (as opposed to a particular rendering) is crucial, algorithms which perform the visibility calculations at object resolution are needed. These *object space* algorithms include (but are not limited to) those by Loutrel [11], Fuchs et al [8], Weiler and Atherton [18], Sechrest and Greenberg [15], and Franklin [1, 2]. Various analyses of the hidden surface problem from the viewpoint of computational complexity can be found in F.F. Yao [19], Schmitt [14], Ottmann and Widmayer [13], and Nurmi [12].

In this paper, we present an algorithm, at object space, for reconstructing the visible regions in a polyhedral scene, i.e. joining the visible edge segments output from a hidden line program to find the visible regions. Such an algorithm is used by e.g. Franklin's hidden surface algorithm [2] as well as applications in cartography [3]. The input to the algorithm is a set of visible edge segments computed as in [2]. (This will be detailed in the sequel.) The regions output by the algorithm may be stored (along with their intensities which must be computed) for later display. Although the algorithm is based on a simple graph problem, namely, "Determine the faces of a straight-edge planar graph given in terms of its edges," this to our knowledge is the first published implementation in computer graphics. Our algorithm to solve the above problem is worst-case optimal and spends $O(n \log n)$ time and linear space for a graph with n edges. (It is required that the graph has no separate components containing each other.) The reconstruction is done by solving the above problem for each component in the containment relation of the graph under consideration. The algorithm has been implemented in each of Ratfor, Franz Lisp, and Prolog. For brevity, we only give the Prolog implementation (1) in this paper (cf. Appendix); others are available upon request.

(*) Address until July 1986: Computer Science Division, Electrical Eng. and Computer Science Dept., 543 Evans Hall, University of California, Berkeley, CA 94720, USA. Research of this author is supported by the National Science Foundation under grant no. ECS-8351942, and by the Schlumberger-Doll Research Labs, Ridgefield, CT.

(1) The advantages of Prolog for computational geometry and graphics have recently been studied; for some views and implementations see Franklin [6], Gonzales et al [9], and Swinson [17].

PROBLEM DEFINITION AND TRANSFORMATION

Given a family of polyhedra in the three-dimensional Euclidean space, the following algorithm (2) computes a hidden surface picture assuming that the viewpoint is at ∞ in the positive z-direction and orthographic projections of the polyhedra are taken in the xy-plane:

(i) Compute in the xy-plane the intersections of the projected edges of the given polyhedra to subdivide each edge into edge segments (hereinafter, segments). An edge with no intersections is just one segment. Each segment is completely visible or else completely hidden.

(ii) Compute the visibility status of each segment (i.e. take the midpoint of the segment and determine its status which necessarily is the segment's status).

(iii) Compute the regions in the xy-plane described the visible segments.

(iv) For each computed region, determine the three-dimensional face which gave rise to it and shade the region accordingly.

In the above algorithm, step (iii) where the polygons in the xy-plane corresponding to visible parts of faces must be found from a set of visible segments is called *polygon reconstruction* and will be the subject matter of this paper. For the upcoming discussion, it is better to cast this problem in a more abstract setting, i.e. in terms of straight-edge planar graphs. (From now on, when talking about the problem in the visibility context we shall use the terms "segment" and "region" whereas in the graph context we shall employ the terms "edge" and "face," respectively.)

Let E be a set of edges in the xy-plane. It is assumed that the members of E are such that they constitute a *legal* planar subdivision, i.e. one in which every face is bounded save the outer one which is infinite. If E is obtained as a result of the above hidden surface algorithm then the subdivision is necessarily legal (under the reasonable assumption that the polyhedral input to the hidden surface algorithm is meaningful). Notice that the notion of legality is similar to constructive solid geometry in the sense that we do not allow dangling edges (Figure 1). The *polygon reconstruction problem* asks for a listing of the faces of the planar subdivision given in terms of its edges. By listing, we mean specifying the vertices of the faces in order (clockwise (cw) or counterclockwise (ccw)) starting with any of them. Thus, in Figure 2(a), we are given a picture of three polyhedra in space. The visible segments have been shown in Figure 2(b). Notice that they are in no particular order, i.e. $E=\{e_1,e_2,\cdots,e_{25}\}$. In Figure 2(c) the reconstructed faces have been shown. Denoting them by F, it is seen that $F=\{f_1,f_2,\cdots,f_8\}$. (It will be shortly clear why the vertices are labeled in Figure 2(c) in that particular way.)

A crucial point in Figure 2 is that the graph has two components. In this case, the separate parts are caused by the fact that the pyramid projects individually while the cubes overlap in projection. Our algorithm can handle such separate components as long as they do not enclose each other. However, it is also possible to have separate components due to holes in the given polyhedra or due to components one of which projects inside the other. This is demonstrated in Figure 3(a) and Figure 3(b), respectively. Our algorithm will not be able to handle this. Before we proceed any further, we must therefore resolve this issue.

The problem can be solved by finding the connected components of the graph using standard algorithms. As long as two connected components do not enclose each other they can be included in the same input set to our algorithm. The question of containment can be decided by repeated applications of standard point-location algorithms. In particular, consider two separate components C_1 and C_2. If one takes any point p of C_1 and finds out that C_2 encloses p then it is seen that C_2 contains C_1. However note that we are still left with the problem of specifying an order of the computed faces which must eventually be painted in a hidden surface display. Consider the faces in Figure 4. Here we have a "tower" of three connected components enclosing each other. The right approach is to start the painting with the "outer" component and progress toward the "innermost" component. In other words here we must paint the computed faces in the following order: $f_1,f_2,\cdots,f_{10}$. (Within each component, the order of painting its faces can be made arbitrary.) With these explanations, we assume that the *general* problem of reconstruction is solved as soon as we give our algorithm for reconstructing a planar graph with no containment. Accordingly, in the sequel we shall assume that we are dealing with a legal straight-edge planar subdivision with no "holes." Separate components are on the other hand allowed as long as they do not violate this requirement.

Another important issue to be resolved is that of real coordinates. The fact that one is in fact dealing with numbers output by a line intersection algorithm makes it necessary that we take care of the small discrepancies

(2) Although Franklin's algorithm is an extension of this naive algorithm and uses an adaptive grid to reduce its complexity from quadratic worst-case to linear expected time, it is conceptually the same [2, 4, 7].

introduced in the way the intersections were computed and the set of visible segments were arrived. Specifically, refer to Figure 5 where there must be a tiny region to be output by the polygon reconstruction algorithm assuming that everything is correct. However, this triangle is most probably due to numerical errors in computing the vertex it in fact "corresponds" to and should in many cases be discarded. (We refer the reader to Franklin [5] for an account of such numerical errors caused by the underlying computer algebra.) Thus, a preconditioning step prior to reconstruction is needed. This process will take as input a value ε and will output a new set of segments so that if there are vertices in the segment set within ε-neighborhood of each other then they are "reduced" to a single common vertex. The proper value of ε is dependent on the underlying numerical errors. (Nevertheless, since situations such as Figure 5 can actually occur, there is no way to guarantee a good ε.) This preconditioning process is assumed in the following description. In fact, we shall, without loss of generality, describe our polygon reconstruction algorithm assuming that the graph vertices all have integer coordinates.

THE ALGORITHM

(The reader may follow the example in the next section while reading this section.) The input to the algorithm consists of n edges in the xy-plane (grid) which are specified by their endpoint coordinates

$$E=\{((x_{i1},y_{i1}),(x_{i2},y_{i2})):\ i=1,\cdots,n\}$$

It is emphasized that no particular order is assumed in E. As a matter of fact, the operations of our algorithm can be carried out on the abstract data type "set" in an environment supporting set operations efficiently.

Given E, we first obtain the graph specified by E. To do this, we create $E''=E\cup E'$ where E' is the "reverse" of E, that is

$$E'=\{((x_{i2},y_{i2}),(x_{i1},y_{i1})):\ ((x_{i1},y_{i1}),(x_{i2},y_{i2}))\varepsilon E\}$$

As a matter of fact, E'' corresponds to the edges of the *directed* graph specified by E now. Next we sort E'' by the first key in lexicographic order in x and y values. Specifically, consider an element $((x1,y1),(x2,y2))$ of E''. Then the sort takes place on key $(x1,y1)$ and the lexicographic order $\leq$ is such that

$$((x1,y1),(x2,y2))\leq((X1,Y1),(X2,Y2))\ \textit{iff}\ (x1<X1)\ \textit{or}\ (x1=X1\ \textit{and}\ y1\leq Y1)$$

for another element $((X1,Y1,),(X2,Y2))$ of E''. Now, consider the elements of E'' with the same key (3). These are the edges of the planar graph which have an endpoint (vertex) equal to their common key. We shall call the totality of the other endpoints (vertices) of these edges a "row" and the common vertex a "pivot" for descriptive purposes in the following algorithm. (The pivot is not included in the row.) Thus the planar graph can be visualized as a table made of a family of rows each having a different pivot. Finally, we sort the elements of each row about their pivots in angular order (4). (The angle θ of a vertex satisfies $0\leq\theta<2\pi$.) We shall refer to this final table as the "navigation" table since the algorithm to be given below will navigate through this table to obtain the faces one after another. Associated with each row of the table we hold a counter which is initialized to the cardinality of the row in the beginning of the navigation.

Algorithm Navigate

```
; There are n rows (and thus n pivots) in the navigation table
; Initially all elements of the rows are marked as "unused"
; We use integers instead of the vertex names (e.g. 1 means v1)
; Count field of each row has also been initialized
CurrentPivot <-- 1
; Come here after outputting a face
; Find new pivot to start with
L1:
IF Count(CurrentPivot) = 0 THEN
  CurrentPivot <-- CurrentPivot + 1
  IF CurrentPivot > n THEN STOP ELSE GO TO L1 FI
FI
CurrentRow <-- CurrentPivot
CurrentVertex <-- First "unused" element of CurrentRow
```

(3) For convenience, we assume that we renamed the vertices so that the graph is now represented by vertex names and not vertex coordinates.

(4) If the vertices are sorted in cw order then the final faces output by the algorithm will be in ccw order (save the infinite face which will be cw). If the vertices are sorted in ccw order then the final faces will be cw (with a ccw infinite face). In the sequel we assume that the latter approach is taken.

```
Face <-- {CurrentPivot} ; Initialize the face to be output
DO FOREVER              ; This loop corresponds to the navigation
  Append CurrentVertex to Face
  Mark CurrentVertex as "used"
  Count(CurrentRow) <-- Count(CurrentRow) - 1
  IF CurrentVertex = CurrentPivot THEN
    OUTPUT Face         ; This face finished
    GO TO L1
  FI
  PreviousRow <-- CurrentRow
  CurrentRow <-- CurrentVertex
  CurrentVertex <-- The element of CurrentRow following PreviousRow (with wrap-around)
OD
End
```

Correctness: The steps prior to the execution of the algorithm (i.e. building the graph) are obviously correct. The correctness of the algorithm is easy to establish under our assumption that the graph is a legal subdivision. In particular, we start with the first pivot and the correct handling of Count guarantees that the right pivots are always chosen after that. As soon as face is output the algorithm starts anew with a partially marked navigation table. Each element in each row is marked "used" once and only once. The navigation stops as soon as there are no pivots with a nonzero Count field.

Complexity: The initial sorting of E'' to obtain the pivot vertices takes $O(n \log n)$ time. (Renaming the vertices and obtaining the graph also takes this much time since we simply take each coordinate pair and via binary search find its vertex name (binary search on first coordinate followed by a binary search on the second coordinate).) Sorting in angular order around a pivot takes $O(d_i \log d_i)$ for pivot i with degree d_i. (The degree of a pivot is the cardinality of its associated row.) An upper bound on the total time spent in this process is again $O(n \log n)$ since the sum of the degrees is linear in n. Now, consider the navigation process. Assume that we are constructing one of the faces and suppose that it will eventually have v vertices. The cost of constructing this face is then only $O(v \log n)$ since the only non-constant time operation in the algorithm is locating the first "unused" element of CurrentRow and this clearly takes $O(\log n)$ time by binary search (since a row is in sorted angular order). Each vertex i of the graph will appear in exactly d_i faces, giving again a total time of $O(n \log n)$ for all the faces to be output. Thus, after the navigation table is built the algorithm uses $O(n \log n)$ time. The total space used is clearly $O(n)$.

Optimality: We shall show that the algorithm is worst-case optimal by demonstrating that it can be used to sort real numbers $r_1, r_2, \cdots, r_n$. Assuming that the numbers are lying on the x-axis in the xy-plane the following construction is made (Figure 6(a)): For each r_i include in the set of edges to be submitted to the algorithm the edges of the triangle $A_{r_i}B_{r_i}C$ where $C=(0,1)$ and, $A_{r_i}=(r_i-\varepsilon,0)$ and $B_{r_i}=(r_i+\varepsilon,0)$. (Here ε is a small positive constant.) When terminated the algorithm would have output the boundary

$$C, B_{n1}, A_{n1}, C, B_{n-1,1}, A_{n-1,1}, \cdots, C, B_{1}, A_{1}$$

although not necessarily in this order. However from this boundary polygon one can infer in linear time the ascending sorted order of the given numbers. Note that in this proof, we have assumed that the polygon reconstruction algorithm always outputs the boundary of the infinite face of the subdivision. Clearly, this boundary is not necessitated and one can argue that there may exist another algorithm which only the outputs the proper faces. However, even an algorithm which does not output the infinite face would require $O(n \log n)$ time since one can make a "thicker" single polygon out of the polygons in Figure 6(a). This is shown in Figure 6(b).

Before closing this section we shall briefly treat the problem of shading the regions obtained by the algorithm above. Consider any region f found by the algorithm. Let p be an interior point of f (5). Compare p against the projections of all faces. (Franklin's algorithm does this much more efficiently via adaptive grid [2] but this is not the issue here.) Let F be the closest face whose projection contains p. Then f corresponds to a visible part of F and should be given an appropriate shading value (e.g. proportional to the surface normal of F). If p is not on any face then it corresponds to the background and is given the default shading value.

(5) *Caveat:* Notice that since there may be separate components enclosing each other the point must be taken very close to the boundary of the face to guarantee a correct picture.

AN EXAMPLE

We now give an example to clarify how the algorithm works. Consider the graph shown in Figure 7. The input is

E = {((2,4),(3,4)),((3,2),(3,4)),((4,2),(3,4)),((3,2),(4,2)),
((3,2),(2,2)),((2,2),(2,4)),((1,3),(2,4)),((2,1),(3,1)),
((2,1),(2,2)),((3,2),(3,1)),((1,3),(2,2))}

After "reversing" E to get E' and uniting these two we get

E'' = {((2,4),(3,4)),((3,2),(3,4)),((4,2),(3,4)),((3,2),(4,2)),
((3,2),(2,2)),((2,2),(2,4)),((1,3),(2,4)),((2,1),(3,1)),
((2,1),(2,2)),((3,2),(3,1)),((1,3),(2,2)),
((3,4),(2,4)),((3,4),(3,2)),((3,4),(4,2)),((4,2),(3,2)),
((2,2),(3,2)),((2,4),(2,2)),((2,4),(1,3)),((3,1),(2,1)),
((2,2),(2,1)),((3,1),(3,2)),((2,2),(1,3))}

After lexicographic sorting we get new E'' as

((1,3),(2,4)),((1,3),(2,2)),
((2,1),(3,1)),((2,1),(2,2)),
((2,2),(2,4)),((2,2),(3,2)),((2,2),(2,1)),((2,2),(1,3)),
((2,4),(3,4)),((2,4),(2,2)),((2,4),(1,3)),
((3,1),(2,1)),((3,1),(3,2)),
((3,2),(3,4)),((3,2),(4,2)),((3,2),(2,2)),((3,2),(3,1)),
((3,4),(2,4)),((3,4),(3,2)),((3,4),(4,2)),
((4,2),(3,4)),((4,2),(3,2))

Note that we have omitted the set signs and listed E'' such that the elements in each row above have the same pivot. Now we can rename the graph vertices (i.e. deal with names instead of coordinates). The renaming is as follows

v_1 = (1,3)
v_2 = (2,1)
v_3 = (2,2)
v_4 = (2,4)
v_5 = (3,1)
v_6 = (3,2)
v_7 = (3,4)
v_8 = (4,2)

Thus we have the following graph (notice that this corresponds to the adjacency list representation)

v_1: v_4,v_3
v_2: v_5,v_3
v_3: v_4,v_6,v_2,v_1
v_4: v_7,v_3,v_1
v_5: v_2,v_6
v_6: v_7,v_8,v_3,v_5
v_7: v_4,v_6,v_8
v_8: v_7,v_6

After angular sorting of the rows about the pivots we get the navigation table

v_1: v_4,v_3

v_2: v_5,v_3

v_3: v_6,v_4,v_1,v_2

v_4: v_7,v_1,v_3

v_5: v_6,v_2

v_6: v_8,v_7,v_3,v_5

v_7: v_4,v_6,v_8

v_8: v_7,v_6

In the above table the pivots are the first elements in each row before the column sign. It is emphasized that there is a wrap-around for each row, e.g. in row 7 it is implicit that v_4 follows v_8. The regions created by the navigation algorithm are then

f_1=(v_1,v_4,v_3)

f_2=(v_1,v_3,v_2,v_5,v_6,v_8,v_7,v_4)

f_3=(v_2,v_3,v_6,v_5)

f_4=(v_3,v_4,v_7,v_6)

f_5=(v_6,v_8,v_7)

Note that f_2 is the boundary of the infinite face. Also note that all proper faces are in cw order whereas f_2 is ccw.

It may also be instructive for the reader to try the algorithm on a graph with separate components, say two squares. We omit such an example here.

SUMMARY

We presented an optimal algorithm and its Prolog implementation for reconstructing visible regions from a set of visible segments output by a hidden line program. This is an important operation in object space hidden surface algorithms operating on polyhedral scenes.

Note: We have recently learned that R.I. Hartley of University of Waterloo (Dept. of Computer Science) has a 1985 technical report titled "Reassembling polygons from edges." Although not yet seen by us, this title sounds suggestive of the topic of the present paper.

References

1. W.R. Franklin, *Combinatorics of hidden surface algorithms*, Ph.D. dissertation and report TR-12-78, Harvard University, Center for Research in Computing Technology, Cambridge, Mass. (1978).
2. W.R. Franklin, "A linear time exact hidden surface algorithm," *ACM Computer Graphics* 14(3) pp. 117-123 (1980).
3. W.R. Franklin, "A simplified map overlay algorithm," *Harvard Computer Graphics Conf.*, (1983).
4. W.R. Franklin, "Adaptive grids for geometric operations," *Proc. Sixth International Symposium on Automated Cartography* 2 pp. 230-239 (1983).
5. W.R. Franklin, "Cartographic errors symptomatic of underlying algebra problems," *Proc. International Symposium on Spatial Data Handling* 1 pp. 190-208 (1984).
6. W.R. Franklin, "Computational geometry and Prolog," pp. 737-749 in *Fundamental Algorithms for Computer Graphics*, ed. R.A. Earnshaw,Springer-Verlag, Heidelberg (1985).
7. W.R. Franklin and V. Akman, *Adaptive grid for polyhedral visibility in object space: An implementation*, Manuscript (1986).

8. H. Fuchs, Z.M. Kedem, and B.F. Naylor, "On visible surface generation by a priori tree structures," *ACM Computer Graphics* **14** pp. 124-133 (1980).

9. J.C. Gonzales, M.H. Williams, and I.E. Aitchison, "Evaluation of the effectiveness of Prolog for a CAD application," *IEEE Computer Graphics and Applications* **4**(3) pp. 67-75 (1984).

10. J.G. Griffiths, "Bibliography of hidden line and hidden surface algorithms," *Computer Aided Design* **10**(3) pp. 203-206 (1979).

11. P.P. Loutrel, "A solution to the hidden line problem for computer drawn polyhedra," *IEEE Transactions on Computers* **19**(3) pp. 205-213 (1970).

12. O. Nurmi, "A fast line-sweep algorithm for hidden line elimination," *BIT* **25** pp. 466-472 (1985).

13. T. Ottmann and P. Widmayer, "Solving visibility problems by using skeleton structures," *Proc. Mathematical Foundations of Computer Science*, pp. 459-470 Springer-Verlag, (1984).

14. A. Schmitt, "Time and space bounds for hidden line and hidden surface algorithms," *Proc. Eurographics-81*, North-Holland , (1981).

15. S. Sechrest and D.P. Greenberg, "A visible polygon reconstruction algorithm," *ACM Transactions on Graphics* **1**(1) pp. 25-42 (1982).

16. I.E. Sutherland, R.F. Sproull, and R. Schumacker, "A characterization of ten hidden surface algorithms," *ACM Computing Surveys* **6**(1) pp. 1-55 (1974).

17. P.S.G. Swinson, "Logic programming: A computing tool for the architect of the future," *Computer Aided Design* **14**(2) pp. 97-104 (1982).

18. K. Weiler and P. Atherton, "Hidden surface removal using polygon area sorting," *ACM Computer Graphics* **11**(2) pp. 214-222 (1977).

19. F.F. Yao, "On the priority approach to hidden surface algorithms," *Proc. 21st Annual IEEE Symposium on the Foundations of Computer Science*, pp. 301-307 (1980).

APPENDIX

Here we give the complete code of an implementation of the polygon reconstruction problem in C-Prolog. We divided the code into two parts: UTIL and PLANAR. The former consists of several useful programs used by the polygon reconstruction program which is given in the latter.

After defining a planar graph (via *edge* and *vertex* data structures as shown at the end of PLANAR) the polygons are found by just running the main program *main*. The polygons are stored in *polygon*.

```
%=============================================================================
% UTIL -- General Utility Procedures.
%=============================================================================
% Double indentation => Not user callable usually.
%
%           append                  Append one list to another
%             bag_to_set            Remove dupls from unsorted list
%             find                  Find elt in list, or append it
%           findset                 Find set of all elts w property
%           findall                 Find bag of all elts w property
%                 my_collect_bag    (internal)
%                 getnext
%           maplist                 Apply fn to each elt of list
%             quick_sort            Quick sort
%                 qsort
%                 partition
%             lessp                 (quick and dirty)
%           remove_duplicates       Remove dupls from sorted list
%           reverse                 Reverse order of list
%                 reverse2
%           second                  Return 2nd elt of list
%-----------------------------------------------------------------------------
% APPEND -- Append a list to the end of another.
% Notice that any two of the arguments of "append" can be instantiated,
% and "append" will instantiate the third argument to the appropriate
% result. This property is known as "reversible programming."

append([],L,L).
append([H|T],L,[H|S]) :- append(T,L,S).

%-----------------------------------------------------------------------------
% BAG_TO_SET -- Remove duplicates from unsorted list.

bag_to_set(L0,L) :- quick_sort(L0,L1),remove_duplicates(L1,L).

%-----------------------------------------------------------------------------
% FIND(Plist,Elt) -- Find elt in list, or append it.

find([E|_],E).
find([_|T],E) :- find(T,E).

%-----------------------------------------------------------------------------
% FINDSET(E,R,S) -- Find the set of elements E with property R.

findset(E,R,S) :- findall(E,R,S0),bag_to_set(S0,S).

%-----------------------------------------------------------------------------
% FINDALL -- Find bag (multiset) of all elts with certain property.

findall(X,P,_) :- asserta(bag(mark)),P,asserta(bag(X)),fail.
findall(_,_,L) :- my_collect_bag([],M),!,L=M.
```

```
% "collect_bag" changed to "my_collect_bag" since there is a predefined
% "collect_bag" that is not documented and that doesn't appear to do
% the same thing.

my_collect_bag(S,L) :- getnext(X),!,my_collect_bag([X|S],L).
my_collect_bag(L,L).

getnext(X) :- retract(bag(X)),not(X=mark).

%-----------------------------------------------------------------------
% MAPLIST(A,F,B) -- Apply proc F to each elt of list A to create B.

maplist([A1|A],F,[B1|B]) :- !,Z=..[F,A1,B1],call(Z),maplist(A,F,B).
maplist([],_,[]).

%-----------------------------------------------------------------------
% QUICKSORT -- Note that '<' works for atoms as well as numbers.

quick_sort(L0,L) :- qsort(L0,L,[]).
qsort([X,..L],R,R0) :- !,partition(L,X,L0,L1),
                       qsort(L1,R1,R0),qsort(L0,R,[X,..R1]).
qsort([],R,R).

partition([X,..L],Y,[X,..L0],L1) :- lessp(X,Y),!,partition(L,Y,L0,L1).
partition([X,..L],Y,L0,[X,..L1]) :- !,partition(L,Y,L0,L1).
partition([],_,[],[]).

% "lessp" is a quick and dirty general comparison for anything.
% Atoms are less than lists. Lists are compared on first elt only.

lessp([A|B],[C|D]) :- !,lessp(A,C).
lessp(_,[C|D]) :- !.    % Cut in case of retry.
lessp(A,B) :- A < B.

%-----------------------------------------------------------------------
% REMOVE_DUPLICATES -- Remove duplicate entries from sorted list.

remove_duplicates([X,X|L0],L) :- remove_duplicates([X|L0],L),!.
remove_duplicates([X|L0],[X|L]) :- remove_duplicates(L0,L),!.
remove_duplicates(X,X).

%-----------------------------------------------------------------------
% REVERSE -- Reverse order of list.

reverse(A,B) :- reverse2(A,B,[]).
reverse2([A0|A],B,L) :- reverse2(A,B,[A0|L]).
reverse2([],B,B).

%-----------------------------------------------------------------------
% SECOND -- Return the second element of list.

second([_,A|_],A).
```

```
%=============================================================================
% PLANAR -- Find polygons of planar graph.
%=============================================================================
% Purpose:
% This takes the vertices and edges defining a planar graph and
% calculates the polygons.
%
% Sample Application:
% Joining the visible edge segments output from a hidden line program
% to find the visible regions.
%
% Algorithm Limitations:
% Cannot handle containment among separate graph components.
%
% Implementation Limitations:
% Because of the lack of real arithmetic, this program does not do
% actual geometry, only topology.  Hence it wants as input the angle
% (in degrees in the positive (ccw) direction) that each edge leaves its
% first vertex.
%
% Input data structures:
%     vert(vname,x,y)
%     edge(ename,v1,v2,angle)
%
% Order of procedures:
%           split_vert1
%           split_vert2
%           split_all_vert
%           makecorner
%           otherend
%           join1
%           join2
%           joinall
%           orderedges
%           order_a_vertex
%           reverseangle
%           main
%
%-----------------------------------------------------------------------
% Separate out the individual corners at the vertices.
% SPLIT_VERT1(vertex,list of edges from it).
split_vert1(V,[E|T]) :- split_vert2(V,E,[E|T]).   % Need 1st elt later.

% SPLIT_VERT2(vertex,1st edge,list of unprocessed edges).
split_vert2(V,E1,[E2,E3|T]) :- makecorner(V,E2,E3),
                               split_vert2(V,E1,[E3|T]).
split_vert2(V,E1,[E2]) :- makecorner(V,E2,E1).

split_all_vert :- retractall(corner(_,_,_)),fail.
split_all_vert :- edgeorder(V,L),split_vert1(V,L),fail.
split_all_vert.

%-----------------------------------------------------------------------
% MAKECORNER -- Given a vertex and its 2 adjacent edges, find the
% adjacent vertices and assert that fact.

makecorner(V,E1,E2) :- otherend(E1,V,V0),otherend(E2,V,V2),
                       assert(corner([V0,V,V2],V,V2)).
```

```
otherend(E,U,V) :- edge(E,U,V,_).
otherend(E,U,V) :- edge(E,V,U,_).

%-----------------------------------------------------------------
% JOINALL -- Join edge segments into a long edge.
% Data structure: corner([v1,v2,..vn],v(n-1),vn)
% The list contains all the vertices. The last two are repeated
% outside the list for efficiency.

join1 :-        % Join 2 polygon chains that match on two vertices.
        corner(L,VN1,VN),corner([VN1,VN|M],WN1,WN),
        join2(L,VN1,VN,M,WN1,WN).

join2(L,VN1,VN,M,VN1,VN) :-     % We've got a complete polygon.
        L=[VN1,VN|M],
        !,
        retract(corner(L,_,_)),
        assert(polygon(M)),
        !.
join2(L,VN1,VN,M,WN1,WN) :-     % This is a separate proc of the cut.
        append(L,M,N),
        retract(corner(L,_,_)),
        retract(corner([VN1,VN|M],WN1,WN)),
        assert(corner(N,WN1,WN)),
        !.

joinall :- join1,fail.
joinall.

%-----------------------------------------------------------------
% ORDEREDGES -- Given the angles of the edges, find the set of edges
% in clockwise order around each vertex and assert facts of the
% form: edgeorder(v,[e,e,...])

orderedges :- retractall(edgeorder(_,_)),fail.
orderedges :- vert(V,_,_),order_a_vertex(V),fail.
orderedges.

order_a_vertex(V) :-            % Order the edges around one vertex.
                        % Edges leaving this vertex.
        findall([A,E],(edge(E,V,_,A0),reverseangle(A0,A)),S1),
                        % Edges entering this vertex.
        findall([A,E],edge(E,_,V,A),S2),
        append(S1,S2,AElist),
        quick_sort(AElist,Sorted_AElist),
        maplist(Sorted_AElist,second,Edges0),
        reverse(Edges0,Edges),
        assert(edgeorder(V,Edges)),
        % Don't take chances. No redoing within
        % this code is wanted after the vertex is found.
        !.

%-----------------------------------------------------------------
% REVERSEANGLE -- Find the angle of an edge going in the opposite
% direction to an edge with this angle. 0 <= output < 360.

reverseangle(A,B) :- A<180,B is A+180,!.
reverseangle(A,B) :- B is A-180.
```

```
%-----------------------------------------------------------------------

main :-
        orderedges,       % and assert edgeorder.
        split_all_vert,   % and assert polygons.
        joinall.

%-----------------------------------------------------------------------

% Example:

vert(v1,0,0).
vert(v2,1,0).
vert(v3,1,1).
vert(v4,0,1).
vert(v5,2,0).
edge(e1,v1,v2,0).
edge(e2,v2,v3,90).
edge(e3,v3,v4,180).
edge(e4,v4,v1,270).
edge(e5,v2,v5,0).
edge(e6,v5,v3,120).
```

FIGURES

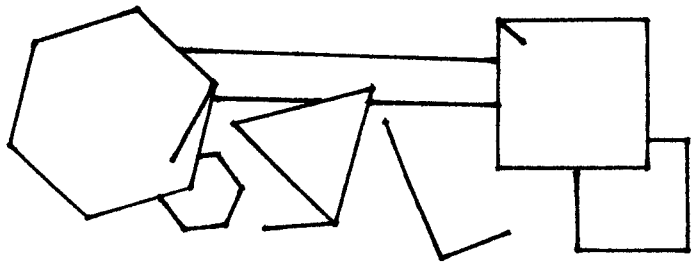

Figure 1. Dangling edges are not allowed.

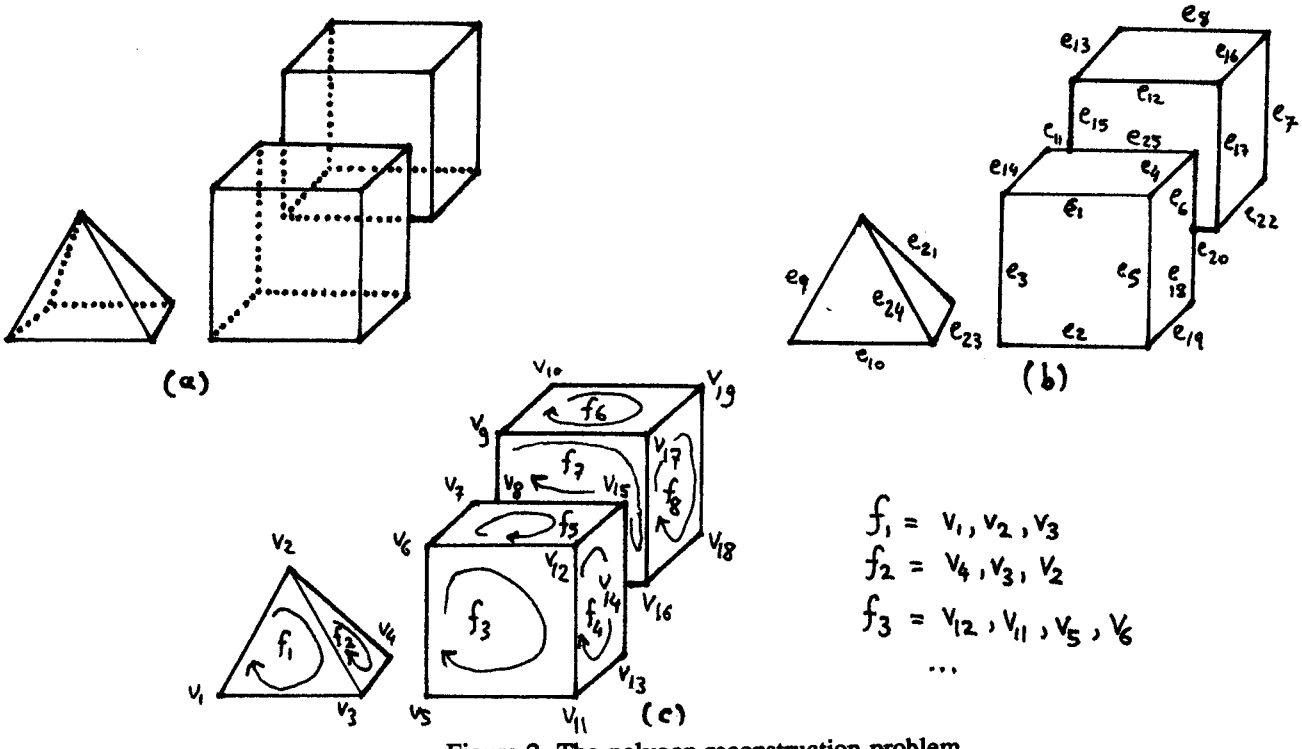

Figure 2. The polygon reconstruction problem.

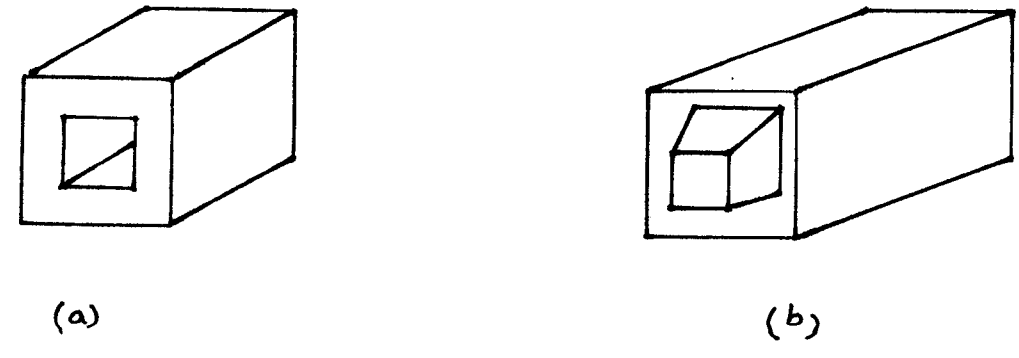

Figure 3. Two separate components with containment.