A Simple and Efficient Haloed Line Algorithm for Hidden Line Elimination

W.R. Franklin[*] and V. Akman[†]

Abstract

An efficient algorithm, HALO, is given to compute haloed line drawings of wire frame objects. (Haloed line drawings are described by Appel et al. [1]

HALO has two parts: CUT and DRAW. CUT uses an adaptive grid to find all edge intersections. It overlays a square grid, whose fineness is a function of the number and length of the edges, on the scene. It determines the cells that each edge passes through, sorts these by cell to obtain the edges in each cell, and then, in each cell, tests each pair of edges in that cell for intersection. For broad classes of input this takes time linear in the number of edges plus the number of intersections. CUT writes a file containing all the locations where each edge is crossed in front by another. Given a halo width, DRAW reads this file edge by edge. For each edge, it subtracts and adds the halo width to each intersection to get the locations where the edge becomes invisible and visible. It sorts these along the edge, and then traverses the edge, plotting those portions where the number of "visible" transitions is equal to the number of "invisible" transitions. DRAW takes time linear in the number of edge segments. Dividing HALO into two parts means that redrawing a plot with a different halo width is fast, since only DRAW need to be rerun.

CR Categories and Subject Descriptions: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling - *geometric algorithms, languages, and systems*; F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems - *geometrical problems and computations*

General Terms: Algorithms, design.

Additional Key Words and Phrases: Hidden line elimination, haloed line effect, wire frame.

[*]Electrical, Computer and Systems Engineering Department
Rensselaer Polytechnic Institute
Troy, New York 12180
USA

[†]Centre for Mathematics and Computer Science
Kruislaan 413
1098 SJ Amsterdam
The Netherlands

1. Introduction

As computer aided design (CAD) deals with more complicated databases, it becomes crucial to display the data effectively so that people can comprehend it. A suitable method must be efficient since users will be interactively manipulating and displaying their data. With special purpose hardware becoming less expensive, and even custom VLSI design becoming as easy as writing software (for those with the appropriate facilities), a suitable algorithm should lend itself to parallelism and implementation in silicon. Since a CAD database may contain wire frame models without any surface information, the algorithm should be able to handle them.

This paper offers an efficient algorithm called HALO to solve this problem via the technique of haloed line elimination. Haloed lines are introduced in Appel et al.[1] which cites many reasons for using them and gives good examples. Briefly, we assume that each line has a narrow region, or halo, that runs along it on both sides. If another more distant line intersects this first line, then that part of the farther line that passes through the first line's halo is blotted out. For example, see figure 1 which shows four drawings of a pair of cubes. Figure 1(a) gives all the edges. Figure 1(b) draws the visible edges solidly, but removes the hidden edges. Figure 1(c) draws the visible edges solidly, but dashes the hidden edges. Finally, figure 1(d) gives all the edges, but adds the haloed line effect. It should be clear that the haloed drawing shows more 3-dimensional relationships than the other three. Figure 1(b) and 1(c) do not give the 3-dimensional relationship between two edges that are both hidden, since either both will be omitted or both will be drawn dashed. Further, if we dash the hidden edges, we must be able to tell which edges are hidden, so we must know what the faces of the objects are. In contrast, with haloed lines we produce a gap on an edge where it passes behind another edge, so we need only the edge data and not the faces.

To be fair, Markowsky and Wesley[2] show how to calculate the faces from just the edges, but the process is slow and subject to ambiguities. To observe that distinct objects can have the same wire frame, refer to figure 2 which shows an object with 9 vertices (the corners and the center of a rectangular block) and 20 edges (the edges of the block and those connecting each block corner to the center). This represents a closed

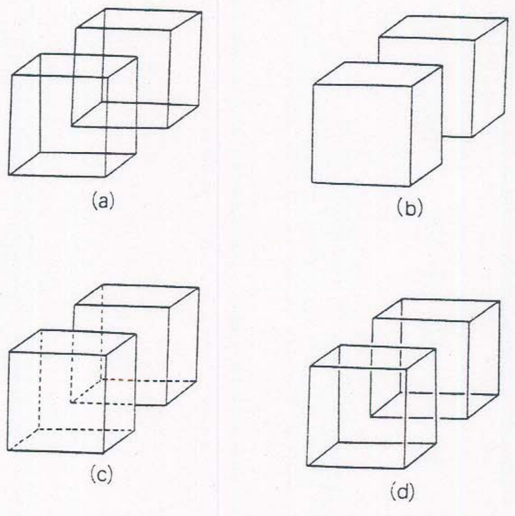

Figure 1. Four ways of looking at a pair of cubes: (a) show all edges, (b) remove hidden edges, (c) dash hidden edges, (d) use haloed lines

object which is a rectangular block with two opposite pyramids (extending from to opposite faces to the center) subtracted. However, all three possible choices of pairs of pyramids create an object with the same wire frame. Thus it is difficult to use general hidden surface methods without face information.

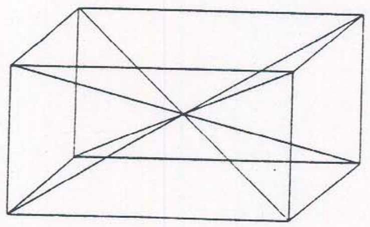

Figure 2. A wire frame model that corresponds to three different objects

The algorithm presented here has several advantages:

1. It is fast, especially on complex scenes with thousands of short edges. The execution time is proportional to the number of projected edge intersections plus the number of edges, for broad classes of input.
2. The computation is split into two parts. The first, which is the slower, uses just the edges, and not the halo width. Thus if we wish to draw the same object repeatedly with various sizes of halos (perhaps to pick the best looking plot) we avoid most of the computation for the second and later plots.
3. The algorithm lends itself to being executed in parallel. Most of the steps of HALO either i) Perform the same operation independently on each element of a set, writing out elements of a new set, or ii) Sort the elements of a set. The former can obviously be made parallel N ways for an N element set.[3] As for the latter, there is currently much effort on parallel sorting algorithms.[4]

After this introduction, the structure of the paper is as follows. In Section 2 we present HALO in its entirety. Section 3 deals with some implementation issues. Section 4 discusses its efficiency and mentions the results of our implementation. Finally, Section 5 suggests ways of improving and generalizing HALO.

2. The Algorithm

HALO is divided into two parts: CUT and DRAW. CUT uses an adaptive (variable) grid to determine which projected edges intersect. (Adaptive grids are described elsewhere[5, 6, 7, 8]). Then it writes a file of them. DRAW uses this intersection file and the desired halo width to compute and draw the visible parts of the edges.

The input to CUT is a set of N edges in 3-dimensional space, $\{(X_1,Y_1,Z_1,X_2,Y_2,Z_2)\}$. Assume without loss of generally that for all edges $0<X_i,Y_i<1$ and $Z_i>0$ for $i=1,2$. (Thus X and Y coordinates have been scaled to a 1 by 1 screen.) The viewpoint is placed at $(0,0,\infty)$, so the projection of point (X,Y,Z) is (X,Y). We are not concerned with the preliminary rotation, scaling, and perspective transformation since efficient techniques are well known.

```
Algorithm CUT:
C1.  Read the input set to determine the average projected edge length, $L$.
C2.  Overlay a regular grid on the screen of $B$ by $B$ squares (of sides $G=1/B$). Here $B$ is an integer
     and $G$ is supposed to be $\lambda L$ where $\lambda$ is a constant about 1. (This will be made clearer in
     Section 4.) We assume that each cell is identified by a unique integer in $[1,B^2]$ which will be called
     the "cell number".
C3.  Initialize a set CEPAIR as $\phi$. For each edge $E$ in the input set, repeat C3.1-C3.2:
     C3.1. Determine the cells that the projection of $E$ passes through.
     C3.2. For each cell C found in C3.1 add an ordered pair $(C,E)$ to CEPAIR.
```

```
C4.  Sort CEPAIR (i.e. the set $\{(C,E)\}$) by the cell number (i.e. C) to make a linear list.
C5.  Collect all the elements in this linear list with the same cell number to make a new set NEWCE consisting of $\{(C,\{E\})\}$.
C6.  Initialize a new set TRIPLE as $\phi$. For each cell C in NEWCE repeat C6.1-C6.5:
     C6.1. Pair up the edges in C in every combination.
     C6.2. For each pair of projected edges $E_1$ and $E_2$ determine whether they intersect.
     C6.3. If $E_1$ and $E_2$ intersect then calculate the intersection point and determine if it is in cell C. If it is not in this cell, this intersection is ignored. This ensures that if $E_1$ and $E_2$ have several cells in common, then the intersection will be noted only once.
     C6.4. If $E_1$ and $E_2$ are still considered to intersect then determine which one is in front at the point of intersection.
     C6.5. Assume without loss of generality that $E_1$ is behind $E_2$ at the intersection point $P$. Calculate the angle between projected $E_1$ and $E_2$ and call it $\theta$. Add an ordered triple $(E_1,P,\theta)$ to TRIPLE.
C7.  Sort TRIPLE (i.e. the set $\{(E,P,\theta)\}$) by the edge number (i.e. $E$).
C8.  Collect together all the points for each edge in TRIPLE to make a new set CUTSET. Thus CUTSET is given as $\{(E,\{(P,\theta)\})\}$.
C9.  Determine which edges $E$ have no elements in CUTSET. These are edges that never pass behind another edge. For each such edge $E$ add a dummy element $(E,\phi)$ to the set.
C10. Write out CUTSET.
End CUT.
```

DRAW reads CUTSET and the desired halo width δ. Then it draws the visible parts of the edges on an edge by edge basis.

Algorithm DRAW:

```
D1.  Initialize and scale the plotter for a screen of coordinates in the range (0,1) by (0,1).
D2.  For each edge $E$ in CUTSET repeat D2.1-D2.5:
     D2.1. Initialize a set SIGN as $\phi$. SIGN will contain elements of the form $(P, \text{sign})$ where $P$ is a point on $E$ and sign is + or -.
     D2.2. For each intersection point $P$ of $E$, subtract and add an effective halo width $\overline{\delta}$ (determined by $\delta$ and $\theta$) to obtain points $P_1$ and $P_2$ where the halo on $E$ respectively starts and stops. Add two elements $(P_1,-)$ and $(P_2,+)$ to SIGN. (This assumes a direction on $E$ from the first to the second endpoint of $E$.)
     D2.3. Let $PE_1$ and $PE_2$ be the first and second endpoints of $E$, respectively. Add two final elements to SIGN: $(PE_1,+)$ and $(PE_2,-)$.
     D2.4. Sort SIGN so that the points in its elements will be in order along $E$.
     D2.5. Process the elements of the sorted SIGN in order counting the accumulated number of +'s and -'s up to each point. At a point where the excess of +'s over -'s becomes one, start drawing a visible part of $E$. Stop drawing it at the first point where the number of +'s and -'s becomes equal. Any part where there are more -'s than +'s is a region of overlapping halos, or else a halo going off the end of $E$. There is never more than one excess + and at the least point the number of +'s will be equal to the number of -'s.
End DRAW.
```

Informally, the validity of these two algorithms must be clear from above descriptions. Basically, the correctness of CUT follows from the provision in C6.3 to note each intersection only once. The correctness of DRAW readily follows from the way + and - signs are assigned and D2.5. An aesthetically important case that DRAW can handle (but does not as presented above) will be discussed in Section 5.

3. Implementation Notes

We now look at the several implementation issues. The comments below refer to the steps of CUT or DRAW.

C3.1-C3.2: If we include a few extra cells, the final result will still be correct. Thus a convenient method is to draw a rectangular box around E and include all the cells that the box passes through. This is only noticeably suboptimal for edges much longer than L, which are statistically infrequent.

It is noted that the cells must partition the space exactly, i.e. each point must fall in exactly one cell. This can be satisfied by considering each vertical grid line between two cells to be inside its right neighbor, and considering each horizontal grid line between two grid cells to be inside its upper neighbor.

It is noted that, unlike Warnock's algorithm,[9] we are not cutting or recursively dividing on the edges. We are merely noting which cells each edge passes through. There is a discussion of recursive subdivision for curved edges in Section 5.

C4: In this step one can use a linear bucket sort since the keys are in $[1,B^2]$.

C5: Each element of NEWCE contains both a cell and all the edges passing through that cell (and possibly a few more edges also). In an efficient implementation this compaction step can be combined with the sorting step (C4), by combining adjacent elements whenever they have the same cell. An alternative way is to establish a linked list for each cell and then read CEPAIR, adding each edge to its appropriate list. However, this way even empty cells will have a list header that requires space, must be initialized, and then must be read. Besides, with linked lists intertwining through storage, there is no longer any locality of reference.

C6.1: Notice that from the statement "pair up edges in every combination" it is clear that the process has quadratic time efficiency in N in the worst case. Thus in principle a sweep approach (cf. Section 5) should be faster for very large N. Our experience is that the adaptive grid method never becomes that slow since the practical scenes are mostly homogeneous.

C6.2-C6.3: A convenient way involves precalculating the edges' equations. E_1 and E_2 intersect if and only if the endpoints of E_1 are on the opposite sides of E_2 (possibly extended) and also the endpoints of E_2 are on the opposite sides of E_1 (possibly extended). A point (X,Y) is on one side or the other of a projected edge with equation $\alpha X+\beta Y+\gamma=0$ according to the sign of this expression. Thus a good way may be to precalculate the edges' equations at the start and store them with edge.

An important point to observe is that if two edges intersect at an endpoint then this intersection must be ignored. This is conceptually easy but in practice numerical problems may warrant special care.

C6.4: The easiest way is to precalculate an equation for each edge, in three dimensions, of the form $Z=\alpha X+\beta Y+\gamma$. This equation is singular for edges perpendicular to the screen, but as those edge project to points, we are not concerned.

C9: A good way is to allocate a bit array with N elements, one per edge, and scan the set, setting bits. In practice, we could set this array earlier as we found the intersections.

C2.4: Unless E is truly vertical, one can sort the points along E by their X coordinates. When E is vertical, the points P must be sorted by their Y coordinates. In general, only a small number of halos are expected for many edges; thus one can legitimately resort to a naive sorting algorithm such as bubble sort.

4. Efficiency

We shall assume that the input edges are independently and identically distributed. In practice this assumption can be relaxed considerably so that scenes of correlated and structured edges can be calculated fast.

Choosing the suitable λ for a given scene can be done within a broad range without really affecting the efficiency of the algorithm. (Our experience shows that a B value between 10 to 30 is in general a good choice, regardless of the scene characteristics.) The optimal λ would depend on the relative speed of various parts of CUT so it is hard to predict a priori. The reader is referred to Franklin[7] for a thorough discussion of this issue.

In CUT, two things dominate the execution time. The first involves deciding which cells each edge falls in. The second is the processing of each cell. We have N edges of average length L in a B by B grid of sides $G=1/B$, $G=\lambda L$. To find the cells an edge falls in takes time proportional to a constant plus the actual number of those cells. The expected number of cells covered by the bounding box of an edge is $O(L^2B^2)$. Thus the total time to place the edges in cells is $O(NL^2B^2)$, or after using $B=\frac{1}{\lambda L}$, $O(N)$.

For the second part of CUT, there are $O(N)$ (C,E) pairs distributed among B^2 cells, for an average of $O(N/B^2)$ edges per cell. This gives an average of $O(N^2G^4)$ pairs of edges to test for intersection per cell. (This must hold since the edges are independently distributed. Thus their number in any cell is Poisson distributed, whence the mean of the square is the square of the mean.) The time to process all cells becomes $O(B^2N^2G^4)$, or after using $G=\lambda L$, $O(N^2L^2)$. Since the latter is equal to the expected number of intersections of the edges CUT behaves linearly in the sum of input and output elements.

The execution time of DRAW is linear in the number of edges and points, assuming that one can use a linear sorting algorithm for the points along each edge. (Note that DRAW uses CUTSET which has one element per intersection plus one per edge that is not crossed in front.)

We implemented HALO in Ratfor, a Fortran processor.[10] Our implementation runs on a Prime 750. A single precision floating multiplication takes 2-3 microseconds on a Prime 750. Figure 3 shows a simple scene made of cubes. Figure 4 shows another cube plot which has 9408 edges. (The whole plot, which is a regular array of 28 by 28, is rather large to be duplicated here.) This took about 4 CPU minutes to compute. What makes HALO efficient is its use of the fast edge intersection algorithm, CUT. On a separate test, the first author implemented EDGE (a version of CUT written in Flecs, another Fortran preprocessor)[11] which creates random edges of varying lengths and angles, finds all the intersections, and then plots the edges with intersections marked. In the following table, sample statistics are given on the timing of EDGE.

N	L	G	V	T1	T2	T
100	0.100	0.100	15	0.17	0.26	0.43
300	0.100	0.100	153	0.54	0.93	1.47
1000	0.010	0.010	11	1.73	3.62	5.35
1000	0.030	0.030	163	1.72	2.54	4.25
1000	0.100	0.100	1720	1.71	4.46	6.18
3000	0.010	0.010	149	5.24	8.05	13.29
3000	0.030	0.030	1487	5.41	8.82	14.22
3000	0.100	0.100	15656	5.19	27.93	33.12
10000	0.003	0.010	156	16.36	16.45	32.82
10000	0.010	0.010	1813	17.38	26.02	43.40
10000	0.030	0.030	16633	17.68	44.78	67.45
30000	0.001	0.010	149	48.33	43.95	92.28
30000	0.003	0.010	1797	48.46	54.21	102.66
30000	0.010	0.010	16859	52.85	98.93	151.78
50000	0.001	0.010	315	77.71	75.75	153.46
50000	0.003	0.010	4953	79.49	92.37	171.87
50000	0.010	0.010	47222	86.23	278.49	364.72

LEGEND

N:	Number of edges
L:	Average length of edges, assuming a 1 by 1 screen
G:	Length of side of each grid cell
V:	Number of intersections found
T1:	CPU time in seconds to preprocess edges
T2:	CPU time in seconds to find the edge intersections
T:	T1 plus T2

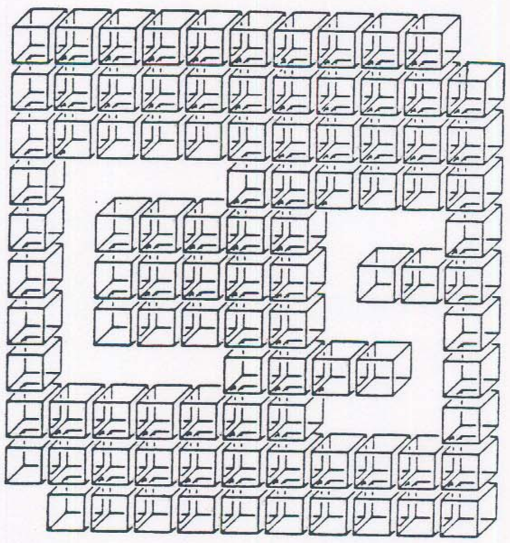
Figure 3. A small example output of HALO

Note, for instance, that even for a large case with 50000 edges and 47222 intersections, EDGE is taking very reasonable time. Some other statistics for this biggest case are: 98753 (C,E) pairs and 11534 rejected intersections.

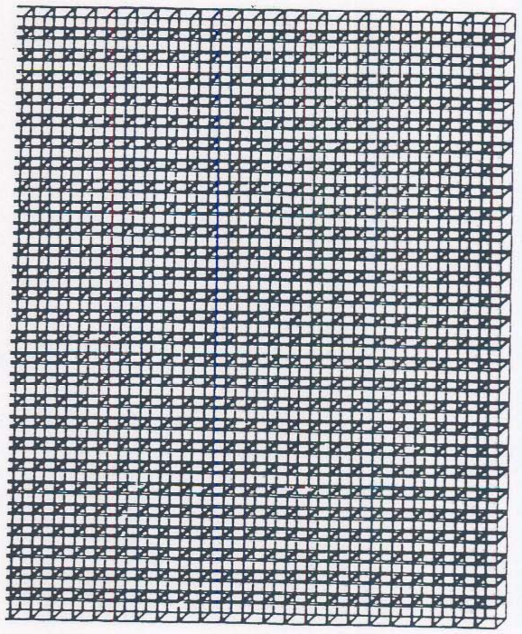
Figure 4. A large example output of HALO

5. Extensions

The first thing that comes to mind is to use a hierarchical grid to accommodate regions of the plot where the edges are clustered more. This would save time in scenes where there are orders of magnitude variation in the edge density. On the other hand, as soon as the cells become hierarchical, parts of CUT such as determining the cells an edge falls in become more complicated and slower. It is also noted that in practice scenes are resolution limited, that is, people do not create scenes with enormous variations. If there is a large blank expanse, some detail will be added there. If there is a crowded region, simplifying notation and approximations will be used.

In general, if pathological input is a problem, one can use more complicated yet proven computational geometry techniques such as those given in.[12, 13] For example, Bentley and Wood's algorithm[13] finds intersections of rectangles in worst case time proportional to the number of actual intersections. As applied to HALO, we would put a box around each edge and use their algorithm instead of the adaptive grid to produce pairs of edges likely to intersect. (Nevertheless, an adversary can defeat this by creating a scene with parallel slanted lines whose boxes intersect even though the edges themselves do not intersect.) The sweep line technique of Bentley and Ottmann[12] reports all K intersections among N line segments in $O((N+K)\log N)$ time. A recent result of Chazelle[14] shows that it is in fact possible to compute all K intersections among N line segments in time $O(K+f(N))$ where f is a subquadratic function of N. Nevertheless, all these algorithms use sophisticated data structures whose practical implementation may pose some difficulties. The reader may consult to Shamos and Hoey,[15] and Nievergelt and Preparata[16] for an in-depth review of the sweep idea. The latter paper and Berata[17] also include implementations of plane sweep.

Another extension is to handle curved edges. This can be done without modifying the general structure of HALO. The following rather low level changes become fundamental:

1. The edges are no longer defined by endpoints but by the coordinates of the splines, say.
2. It is more difficult to tell which cells a given curve falls in. If the curve is smooth, it can be enclosed in a box. (It does not matter if a few extra cells are also included.) If the curve is complicated, one can subdivide it until it is smooth, and then use the bounding box.
3. One must determine whether two curves in the same cell intersect, and if so, the parameter value. Efficient curve intersection is an area of current research. One can split the curves into line segments, intersect them to get approximate crossing points, and then refine them with a few iterations of Newton's rule. It is also possible to approximate the curves by quadratic parametrics for which closed form solutions are known.
4. One must sort the intersection points along each curve. If the curve is parametric, this means that we need the point as a parameter value, not just as (X,Y). If the curve is in some other form but is single valued in X (or Y) one can sort the points in X (or Y).

HALO takes no account of the possible effects between nonintersecting line segments. There may be cases where this is aesthetically important. Consider two line segments, E_1 and E_2, situated such that the halo of the first affects the second, although E_1 does not intersect E_2 (Figure 5). This may easily be resolved with further processing but the additional work would probably lessen the advantage of haloed line drawings. After all, one resorts to haloed lines for quick but approximate hidden line elimination and for easy perception of relative depth of lines, axes, curves, and lettering. Appel et al.[1] summarize this nicely when they state "While some haloing is inconsistent, the overall effect is vivid ..."

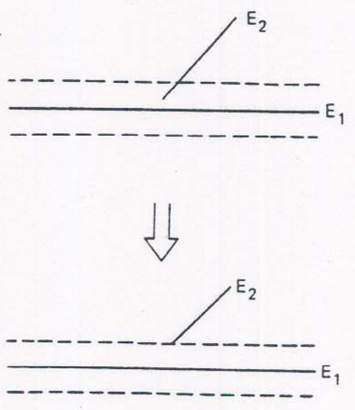

Figure 5. An aesthetically important case which can be handled with additional processing

Acknowledgments

This material is based upon work supported by the National Science Foundation under grant number ECS 83-51942. The second author was also partly supported by a Fulbright grant while he was a student at Rensselaer Polytechnic Institute, Troy.

References

1. A. Appel, F.J. Rohlf, and A.J. Stein, "The Haloed line effect for hidden line elimination," *ACM Computer Graphics (SIGGRAPH'79 Proceedings)* **13**(2), pp. 151-156 (August 1979).
2. G. Markowsky and M.A. Wesley, "Fleshing out wire frames," *IBM Journal of Research and Development* **24**(5), pp. 582-597 (September 1980).
3. L.G. Valiant, "Parallel computation," *(Distributed Systems: Part 1, Algorithms and Complexity)*, Amsterdam, pp. 35-48, Mathematical Center Tracts (1983).
4. J. van Leeuwen, "Distributed computing," *(Distributed Systems: Part 1, Algorithms and Complexity)*, Amsterdam, pp. 1-34, Mathematical Center Tracts (1983).
5. W.R. Franklin, *Combinatorics of hidden surface algorithms*, TR-12-78, Center for Research in Computing Technology, Harvard University, Cambridge, MA (May 1978).

6. W.R. Franklin, "A linear time exact hidden surface algorithm," *ACM Computer Graphics (SIGGRAPH'80 Proceedings)* **14**(3), pp. 117-123 (July 1980).

7. W.R. Franklin, "An exact hidden sphere algorithm that operates in linear time," *Computer Graphics and Image Processing* **15**(4), pp. 364-379 (April 1981).

8. W.R. Franklin, "Adaptive grids for geometric operations," *Proceedings of the Sixth International Symposium on Automated Cartography* **2**, pp. 230-239 (October 1983).

9. I.E. Sutherland, R.F. Sproull, and R. Schumacker, "A characterization of ten hidden line algorithms," *ACM Computing Surveys* **6**(1), pp. 1-55 (March 1973).

10. B.W. Kernighan and P.J. Plauger, *Software Tools,* Addison-Wesley, Reading, MA (1976).

11. T. Beyer, *Flecs user's manual,* Computing Center, University of Oregon (October 1975).

12. J.L. Bentley and T. Ottmann, "Algorithms for reporting and counting geometric intersections," *IEEE Transactions on Computers* **28**(9), pp. 643-647 (September 1979).

13. J.L. Bentley and D. Wood, "An optimal worst-case algorithm for reporting intersections of rectangles," *IEEE Transactions on Computers* **29**(7), pp. 571-577 (July 1980).

14. B. Chazelle, "Intersecting is easier than sorting," *Proceedings of the Sixteenth ACM Symposium on Theory of Computing*, pp. 125-134 (April/May 1984).

15. M.I. Shamos and D. Hoey, "Geometric intersection problems," *Proceedings of the Seventeenth IEEE Conference on Foundations of Computer Science*, pp. 208-215 (1976).

16. J. Nievergelt and F.P. Preparata, "Plane-sweep algorithms for intersecting geometric figures," *Communications of the ACM* **25**(10), pp. 739-747 (October 1982).

17. G.B. Berata, "An implementation of a plane-sweep algorithm on a personal computer," *Dissertation ETH No. 7538*, Zurich, Swiss Federal Institute of Technology (ETH) (1984).