CHAPTER 4

THE PRISM-MAP SPECIAL CASE

4.1 INTRODUCTION

This chapter describes an efficient algorithm for displaying 3-D scenes showing discrete spatially varying data. Given a 2-D map or planar graph composed of polygons where each polygon has a positive real number attribute, a prism is erected on each polygon with height proportional to that attribute. The resulting 3-D scene is plotted with shading and hidden lines removed. Thus the spatial variation of the attribute may be quickly and intuitively grasped by the nontechnical observer. This has applications to areas such as geography if the map is a cartographic map, or to physics if the map diagrams the periodic table.

The algorithm takes time $\Theta(N*\log(N))$ where N is the number of edges in the map. Most of the calculations can be done without knowing the prism heights so extra plots with different attributes for the prisms can be produced quickly. This algorithm has been implemented and tested on maps of up to 12000 edges.

Consider a scene such as the base map of the USA shown in Figure 4-1. This algorithm shows how to put a prism on each state as shown in Figure 4-2 where the height of each state is the per capita alcoholism in that state. Thus it is easy to intuitively see the spatial variation of alcoholism.

This example illustrates both what a PRISM-MAP is and why it is so useful. With the information explosion, it is no longer enough to produce data; the data must be in a form that a casual observer can easily and intuitively appreciate or else it is worthless. As computing power becomes cheaper, powerful display techniques like this become more important both because there is more data to display and because the display techniques are less expensive to use.

Figure 4-1: Base map of the USA

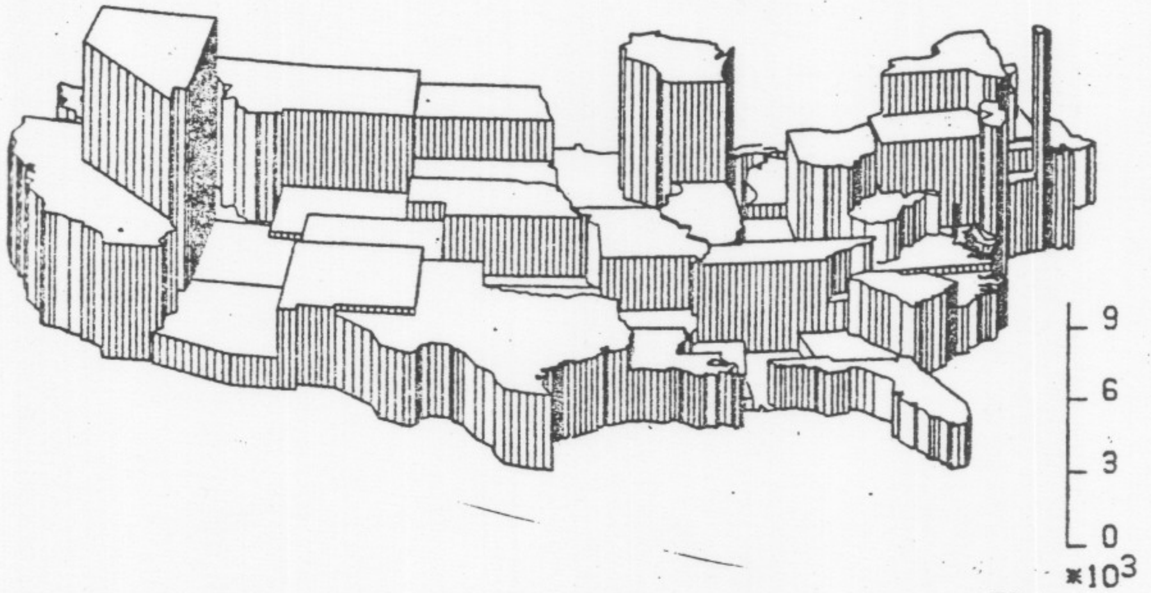

Figure 4-2: Prism-map of the USA showing estimated alcoholism per 100,000 people by state, with vertical shading

This chapter describes a new, faster algorithm to produce prism maps. Indeed, most of the calculation can be performed on the two dimensional base map, producing an intermediate file that can be combined with different sets of heights to produce new plots.

This algorithm takes time $\Theta(N*\log(N))$ where N is the number of edges in the map. If cost were no object this algorithm would not be necessary, since a three dimensional scene could be generated from the base map and heights and then fed into a general hidden surface routine. The only previous published solution did just this. Tobler [19??] took Omega(N^2) time, and could only process a few hundred edges. This algorithm solves part of the problem of displaying a varying 3-D surface of a function of two variables. The surface can either vary discretely or continuously. This algorithm handles the former case while hidden surface contouring algorithms such as ASPEX by Rens and Tobler before 1967, described in Lab for Computer Graphics, [1977] and [1978] handle the continuous case. (ASPEX is an updated version of SYMVU). This algorithm adapts the concept of a horizon line, used in the continuous case, to the discrete case. Although problems like this are not that well known in the computer science community, there have been attempted solutions for several years by cartographers and geographers. Even though three dimensional plots are more appealing, because of their difficulty, various two dimensional methods have heretofore necessarily been used.

4.2 THE ALGORITHM

4.2.1 Definition

Prism: A polyhedron that is the extension of a polygon in the XY plane, into the Z direction. The top face is congruent to, parallel to, and straight above the bottom face. The side faces are vertical rectangles. If the 2-D polygon has N sides then the prism has 2N vertices, 3N edges, and N+2 faces. A simple polygon and the prism derived from it are shown in Figure 4-3.

4.2.2 Basic Algorithm

The algorithm is basically this:

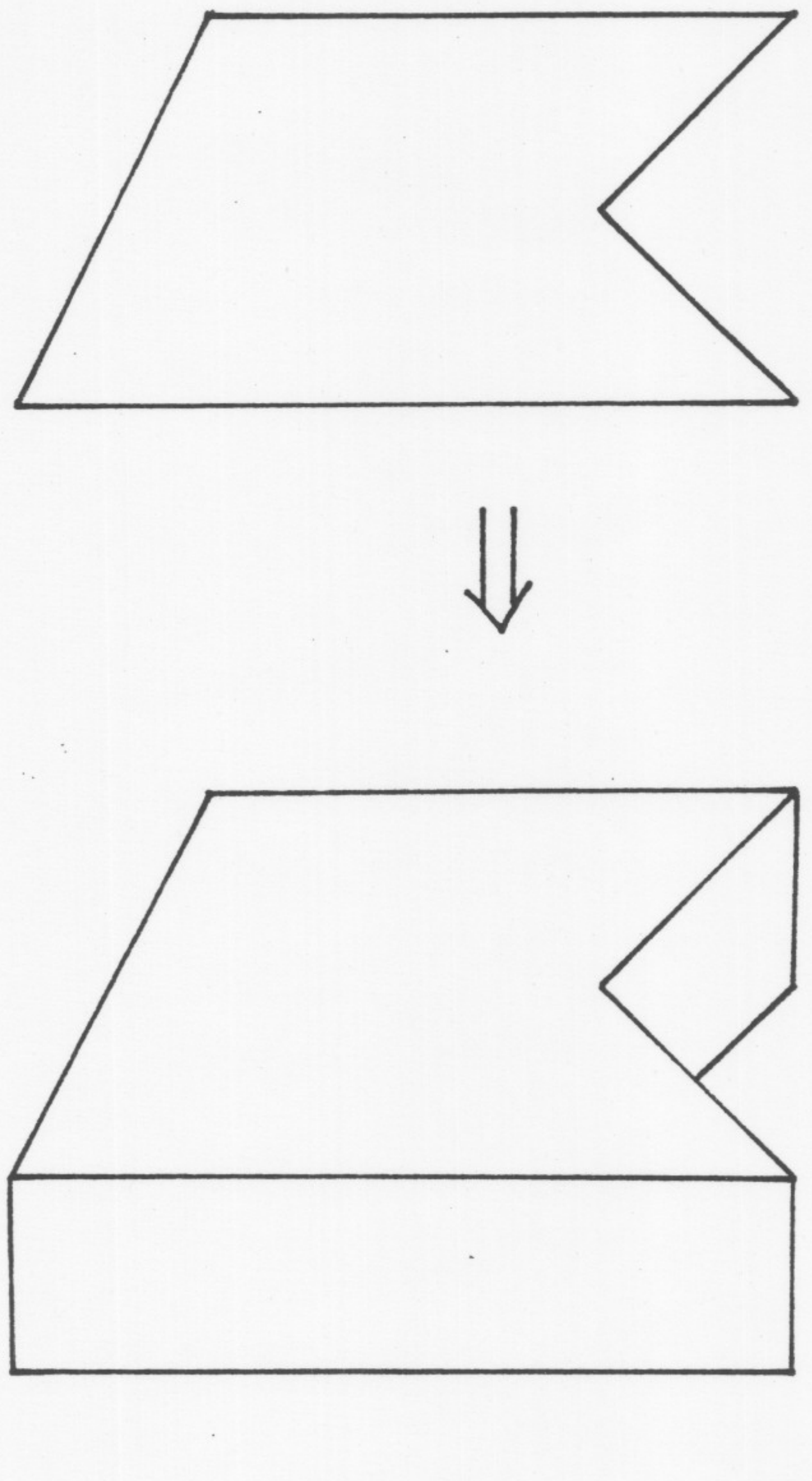

Figure 4-3: Simple polygon and the prism derived from it

1. Read the input map and normalize it.

2. Write its edges to a file, one edge per record.

3. Sort the file by minimum Y value of each edge.

4. Process the resulting file with a local processor that:

    1. Reads edges in order, into memory,

    2. Reorders them while in memory so that if edge E_1 <u>hides</u> (defined later) E_2, then E_1 occurs before E_2.

    3. Writes them out.

5. Repeat the following as often as desired, once per plot:

    1. Read the set of prism values or heights into memory.

    2. Read the final sorted file and as each edge is read, draw part of the plot.

The steps will now be explained in more detail.

4.2.3 Input

The map consists of points, lines and polygons. However the only explicit datatype is a set of straight edges or line segments that form a planar graph. The polygons, marked by unique identification numbers, can be obtained from the edges. Each edge, E_i, represents a quadruple (A_i, B_i, L_i, R_i). A_i and B_i are the coordinates of the two endpoints. L_i and R_i are the two polygons, on the left and right of E_i (looking from A_i towards B_i). The nonexistent polygon on the exterior is numbered zero.

4.2.4 Normalization

The map can be observed in perspective in two dimensions from some general point (X,Y) in the plane. It is rotated, scaled and perspectively transformed to make the viewpoint be at (Ø, -infinity). So now the projection is orthogonal. For the actual 3-D scene, the viewpoint is given in 3-space with a line from it to the origin forming a given altitude angle with the horizontal. Newman & Sproull [1973] gives a thorough description of 3-D transformations and perspective projections.

Each edge also has two bits of supplementary information calculated that tell whether the adjacent edges at each end continue on in the same X direction or double back. In Figure 4-4, edge B doubles back with respect to A but C doesn´t. If there is more than one other edge adjacent to A at a given endpoint, this is treated as a double back on A. These bits are used later for shading.

4.2.5 First Sort

The first sort is by the minimum Y coordinate of each edge. It is very simple and can be done quickly enough by any reasonable external sorting algorithm, such as Knuth [1973].

4.2.6 The Partial Order

4.2.6.1 The Partial Order In 2-D -

Definition: Edge A on the input map directly hides edge B iff there exists a vertical line which intersects both A and B with the B intercept being higher, and with no other edges intersecting that line between the A and B intercepts. The vertical line can intersect either edge at an endpoint.

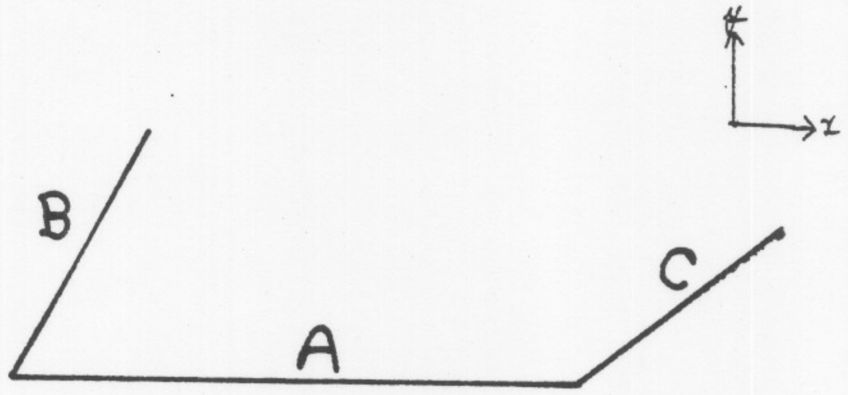

Figure 4-4: Adjacent map edges doubling back on an edge

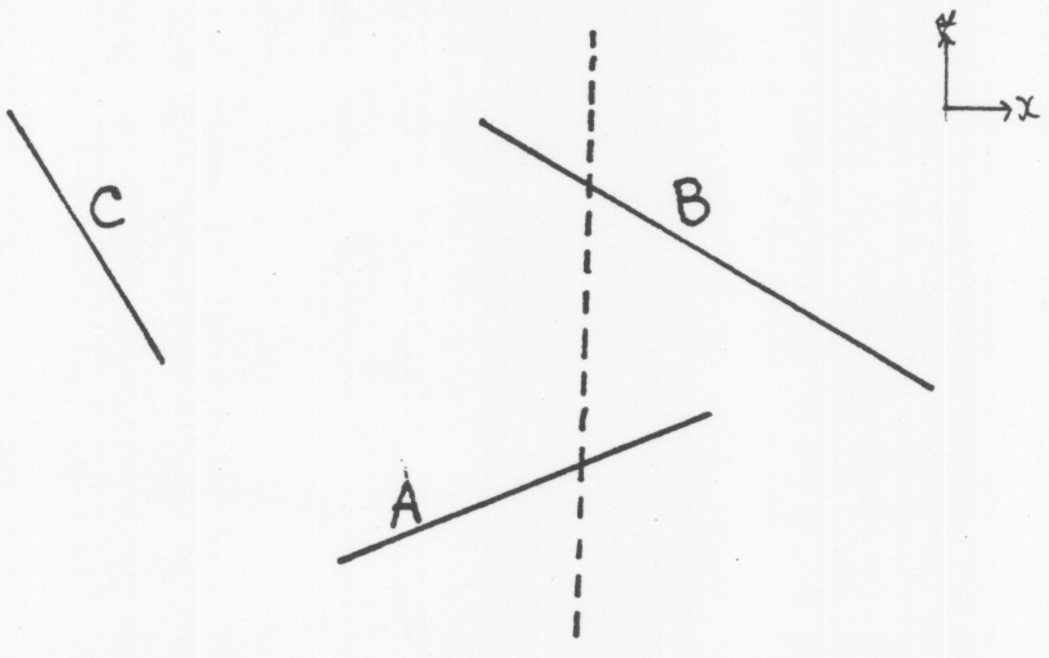

Figure 4-5: The hidden relation between map edges

This means that at that line, A obscures B as seen from the viewpoint and that no other edge is between them at that point.

Definition: A <u>indirectly hides</u> B iff there is a sequence of $N>2$ edges C_i with $A=C_i$, $C_N=B$, C_i directly hiding C_{i+1}, and A does not hide B directly.

Definition: A <u>hides</u> B iff A directly hides B or A indirectly hides B.

Notice that "hides" is closed under transitive completion. Thus "hides" is a partial order on the edges of the input map. In Figure 4-5, A directly hides B but doesn´t hide C, directly or indirectly. Note that A can indirectly hide B even though there is no vertical line intersecting both A and B.

Hiding induces a partial order in 2-D because there cannot exist a finite sequence of edges, each hiding the next and the last hiding the first. This is not true in 3-D since three rectangles, A, B, and C, can be arranged so that A directly hides B, B directly hides C, and C directly hides A.

4.2.6.2 The Partial Order Extended Into 3-D -

However, under some restrictions, polygons in 3-D must satisfy this partial order. In particular,

Theorem: if a vertical rectangle is erected on each edge of the 2-D map, and the whole scene viewed in 3-D, then the rectangles cannot violate the partial order.

Proof: This is because if a 3-D rectangle extended from 2-D edge A hides one extended from B in 3-D, then A must hide B in 2-D. This applies regardless of the relative heights of the rectangles. But these rectangles are just the prism sides.

Now consider the prism tops, that is the 2-D polygons.

Theorem: The prism tops can be split into polygons that can be interspersed with the prism sides and satisfy the partial order.

Proof: The polygon's edges can be divided into top and bottom edges. In Figure 4-6, the top edges of the polygon are dashed and the bottom solid. Each edge that is not vertical has a top and a bottom side and the top edges are those with the polygon adjacent to their bottom. Vertical edges of the polygon are considered to be bottom edges. Now split each polygon into several slices as in shown Figure

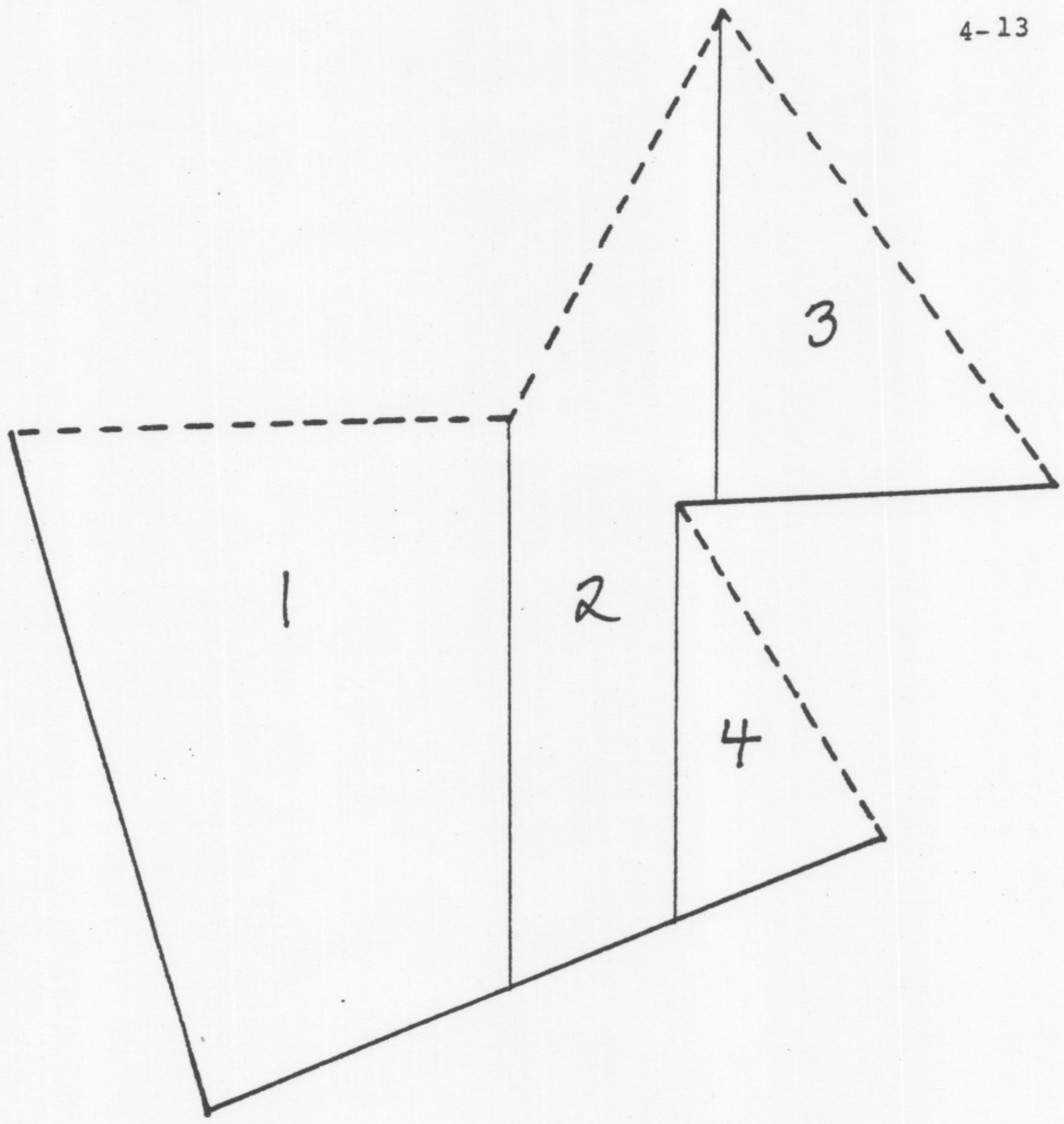

Figure 4-6: Slicing a prism top by dropping lines from the vertices between the top edges

4-6 by dropping thin lines from each vertex that is between two top edges down until the line passes out of the polygon. Return to 3-D and consider the prism tops, split into slices. The prism sides obeyed the partial order but do these top slices also? For a given prism, consider the slice, S, derived from top edge T.

Theorem: The only polygon that S can hide directly is the vertical face, V, on the back of the prism, also derived from T.

Proof: Consider a ray from the viewpoint going through S to intersect another polygon that S hides. But this ray is entering the prism through S and so the next polygon it passes through must be another face (or the bottom) of the prism. But this other prism face must be a back face (it clearly isn't a front face which is the only other choice). If the ray leaves the prism through the bottom then it is below level of the whole scene now so it will never intersect anything else.

Theorem: The only polygons that directly hide S are vertical faces of prisms below it, and those would hide T directly if S were not present.

Proof: As before, a ray from the viewpoint that intersects S after passing through another face must be leaving some other prism through that face. Because of the relative orientations of the viewpoint and polyhedra, a ray can only leave a prism through a back face or through the bottom. So any face directly hiding S is a back vertical face of another prism. Since the viewpoint is at (0, -infinity), this other prism is below it.

Thus S fits neatly into the partial ordering and does not cause any circularity. Note that it was necessary to split the prism tops or else under a circularity could happen.

Since the prism top slices always immediately precede the prism vertical faces in the ordering, when sorting all the polygons to fit the partial order, it is sufficient to sort the 2-D edges to fit the 2-D partial order and then from each edge that is a polygon top edge to create two 3-D polygons, a top slice and a vertical wall, and from each other edge one polygon, a vertical wall.

4.2.6.3 The Simplified Algorithm -

Now given that the prism map´s polygons can be ordered as described above, a possible hidden surface algorithm would be to paint them in order onto an initially blank screen, taking care to paint only blank parts of the screen and never to overpaint anything. But first the partial ordering has to be calculated.

4.2.7 The Final Sort

4.2.7.1 Outline -

The algorithm for the final sort is, briefly:

1. Run a scan line up the screen, from y=Ø (the bottom) to y=1 (the top).

2. As the scan line rises above the bottom of an edge, E, read it into memory.

3. When E is read into memory, compare it against the edges adjacent to it on the scan line to determine if it directly hides them or they directly hide it. If one, say A, hides another, say B, add a link between A and B so each knows about the other.

4. If the scan line rises above the top point of E, then:

    1. If E is has been determined to be hidden by any edges still in memory, then do nothing.

    2. Otherwise write E to the final sorted edge file.

5. If E was written, possibly there are some edges that were remaining in memory only because they were hidden by E and no other edges. If so, write them out, and repeat the process until there are no edges completely below in the scan line remaining in memory unless they are hidden by some edge still in memory.

The steps will now be expanded:

4.2.7.2 The Scan Line -

Since an edge A that hides edge B usually has a smaller minimum Y coordinate, the initial sorting is mostly correct. However, there can be violations as shown in Figure 4-7. Here A hides B and has a smaller minimum Y which is normal, but C hides D while it has a larger minimum Y than D. This second sorting step finds and corrects these cases.

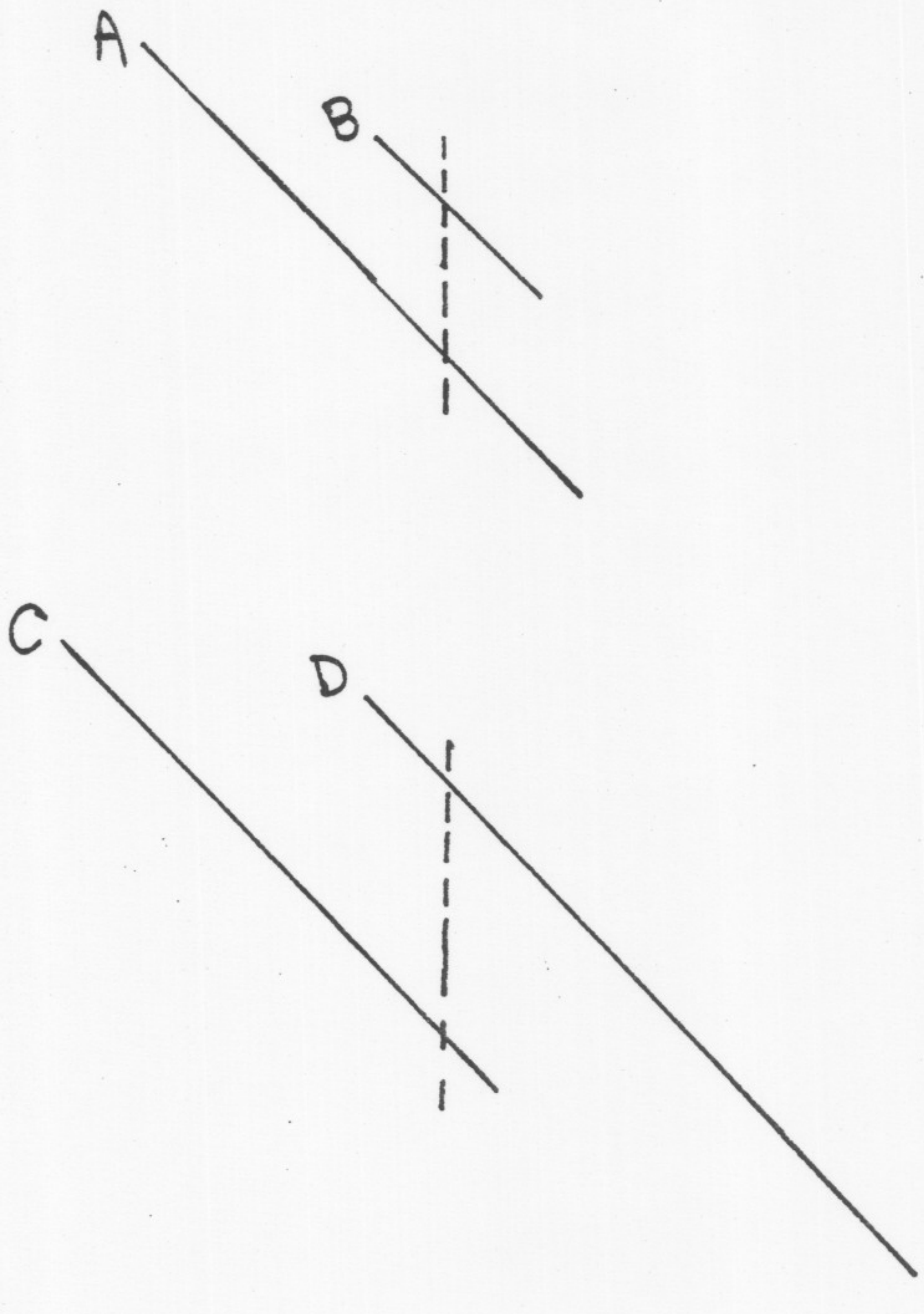

Figure 4-7: Edges misordered after the first sort

Edges are read from the presorted edge file one by one. They are kept in memory for a while, during which time they are called <u>active edges</u>. While in memory the active edges are reordered if necessary and then are written out, one by one.

Imagine a scan line $Y=Y_s$ running across the map. Assuming the map's Y values run from from Ø to 1, the scan line initially has Y_s=Ø and it moves up the page until $Y_s=1$. Since the edges have been sorted by minimum Y, as the scan line moves up the page, it will encounter them in order. At any given time, those edges above it have not been read yet, those crossing it are the active edges, and those below it have generally been reordered and written out. To determine when an edge is to be written, it is necessary to know if its maximum Y value is below Y_s. For this, all the active edges are arranged into a <u>priority list</u> data structure, implemented as a <u>heap</u> as described in AHU [1974]. The key of each edge is its maximum Y. Every time Y_s is increased, the mimimum element of the heap is compared with Y_s and written out if it is less. This is repeated until the minimum element of the heap is greater than Y_s.

Y_s is not raised continuously. Instead, a new edge, E, is read from the file and Y_s is raised to the minimum Y value of E. E is held until the necessary edges from the heap are written and then it is processed and among other actions added to the heap. Then the next new edge is read, and so on until the end of file.

4.2.7.3 Detection Of Ordering Violations -

Theorem: Two edges, A and B, with A directly hiding B but B before A in the sorting order, must both be active edges for some value of Y_s.

Proof: If not, since B was before A, B's maximum Y value would be below A's minimum Y value. But then it would be impossible for A to hide B. So both edges will be in memory together at some time, that is must both be active edges for some Y_s.

Further, consider when $A<B$ directly and there are no other edges between them. Then either they are adjacent for some Y_s, or there is a chain of edges from A to B, with each pair adjacent for some scan line, and with the first of each pair hiding the second. Thus it is only necessary to check adjacent edges along the scan line for violations. However,

adjacency relationships along the scan line only change when an edge is added or deleted. Therefore when a new edge, E, is added to the active set, it is only necessary to compare E with its one or two neighbours. When Y_s is high enough so that E is written out, then its two neighbours must be compared against each other.

A tree data structure is used to hold the adjacency information along the scan line. Each edge is entered with a key that is the X value of its intersection with the scan line. This value changes whenever Y_s increases, but the keys need not be recalculated since they are not stored explicitly. Instead each edge´s equation is stored in the form x=ay+b so that the keys can be calculated from the current value of Y_s whenever the tree is accessed. This method of handling the keys would only cause problems if the edges changed order as Y_s increased. But this cannot happen since the edges in the original map are forbidden from intersecting. Instead of the input data containing two intersecting edges, it should contain four edges and an extra vertex (the intersection point).

If edges A and B are compared and it is discovered that A hides B, nothing is done immediately except to note the fact. There is a counter attached to every edge in memory telling how many other edges have been found to hide it. When an edge is read into memory, this counter is zero. In this case, B´s counter is incremented by one. B is also added to a list attached to A of all the edges that A has been found to hide. Then when the time comes that Y_s is above B´s maximum Y value, B´s counter is checked to determine the number of active edges hiding it. There might not be any left now even if there were some before, since if their maximum Y values were smaller than B´s, they would already have been written out. If B is no longer hidden by any active edges, then it is written. Otherwise it is nevertheless deleted from the active edge set, but remains in memory (called a _semiactive edge_), accessible through its membership on A´s list of edges that A hides.

The semiactive edges are written after the active edges that hide them. Whenever any edge is written to the final sorted edge file, its list of edges that it hides is traversed in order. Any edge whose hidden counter is greater than zero cannot yet be written out since some other edge still hides it and must be written first. In this case, the counter is decremented by one. But if the counter is zero, the edge can be written out and finally deleted from memory. This edge itself has a list (possibly empty) of edges that it hides and after it is written out they are also tested and possibly written. Thus the active edge being written is the root of a tree of semiactive edges. This tree is traversed in depth-first order, writing out any edges with zero hidden counts.

A given edge may occur more than once in the tree if it is hidden indirectly more than one way. Then every time is it accessed but for possibly the last time, it is not written since its hidden count is positive. The last time it is accessed in the tree from this active edge it will be written out unless it is hidden by still another remaining active edge.

After the last edge has been read into memory and processed, Y_s is raised to the top of the screen to force the processing and writing out of any remaining semiactive edges. There will be no edges remaining at the end that cannot be written because they have hidden counts greater than zero. Indeed, an edge's hidden count reflects the number of edges left in memory that hide it. If no edge that was left could be written because they all were hidden by some edge then there would have to be a circularity in the hiddenness relation which is impossible since it is a partial order.

4.2.8 Making The Plot

The sorted edge file that was produced in the previous section can now be used with any set of prism heights to produce a plot. The basic algorithm uses a concept of a _horizon line_ that has been used previously to draw hidden surface plots of net representations of bivariate functions.

A horizon line is a function Y=F(X,T) where X and Y address the plotter screen and T is the elapsed time. The line stretches across the plot from left to right and since it is a function never doubles back on itself. At T=Ø, it lies along the bottom edge of the plot and it increases with T. At any time, it cuts the plots into two regions: The area below has been calculated and plotted while the area above has not been touched yet. As a new part of the plot is calculated, the horizon line is raised above it to include it. Thus this is simply an implementation of the simple algorithm mentioned above in section 4.2.6.3.

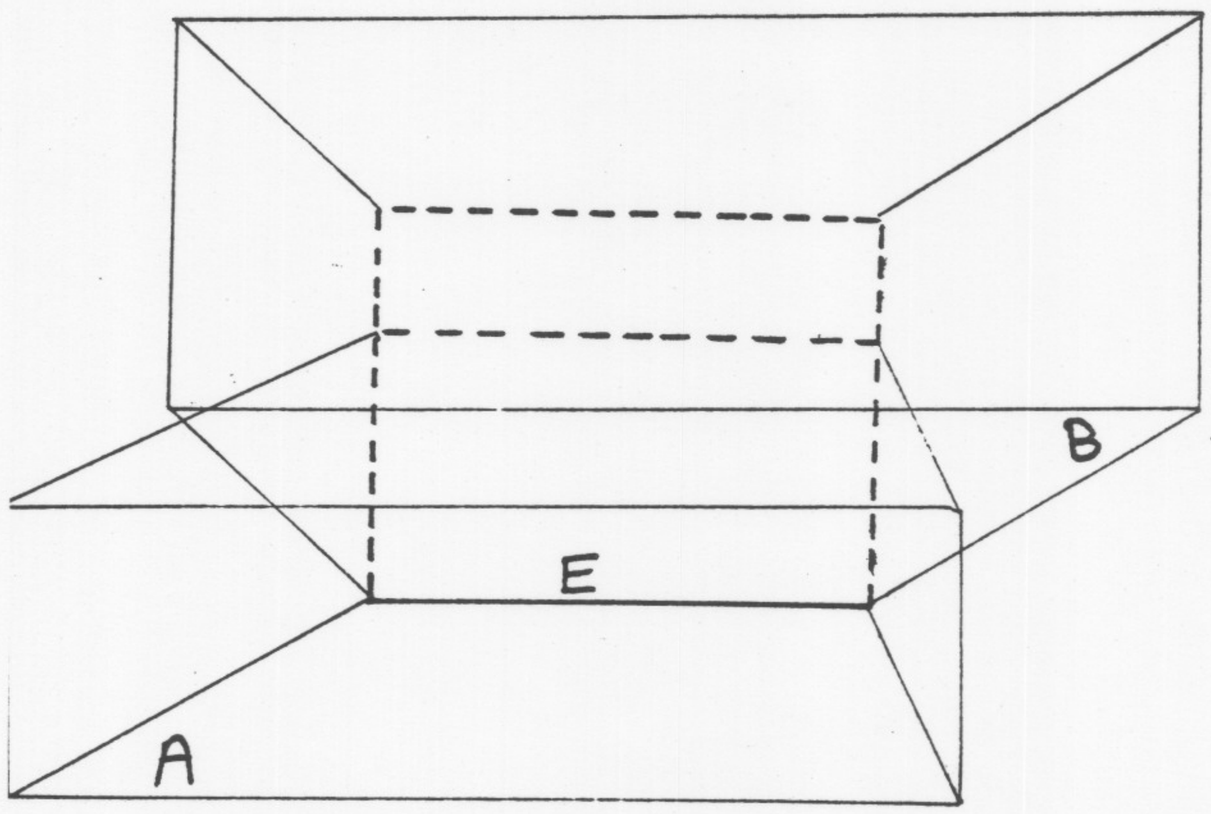

Figure 4-8: Prism edges induced by one map edge

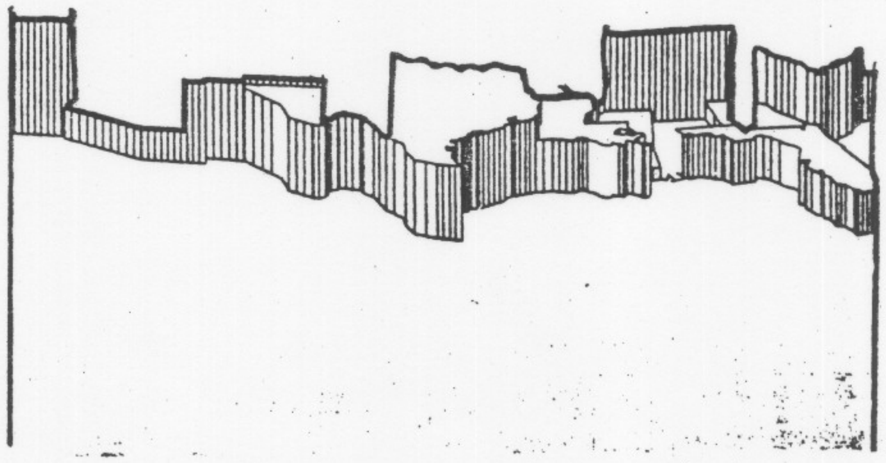

Figure 4-9: The horizon line in a partly completed plot

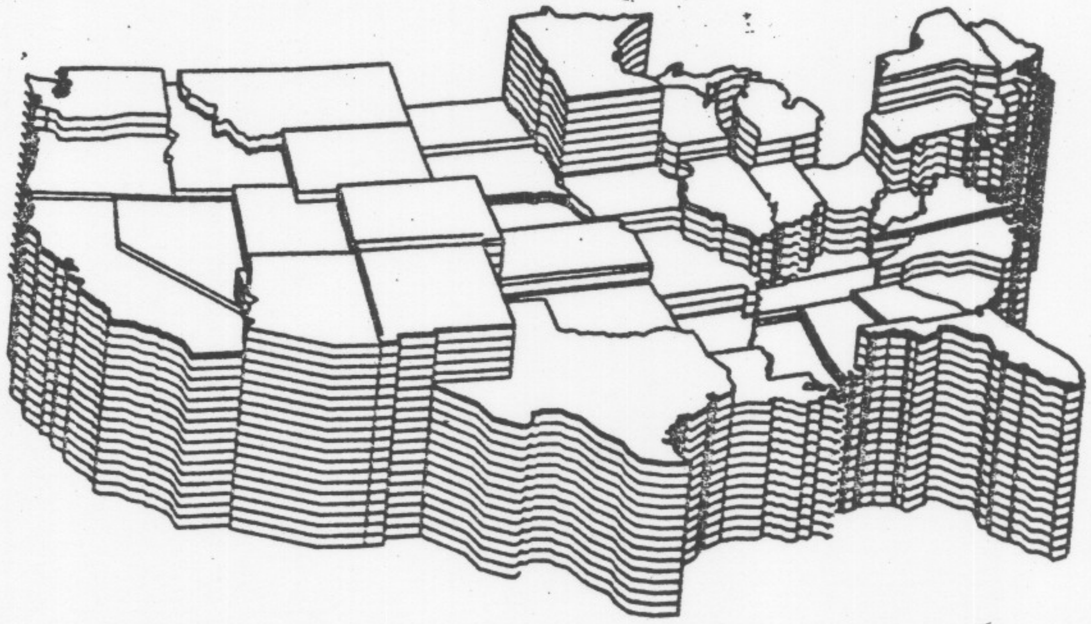

Figure 4-10: Prism-map of public school expenditures in the USA, by state, per capita, showing contour line shading

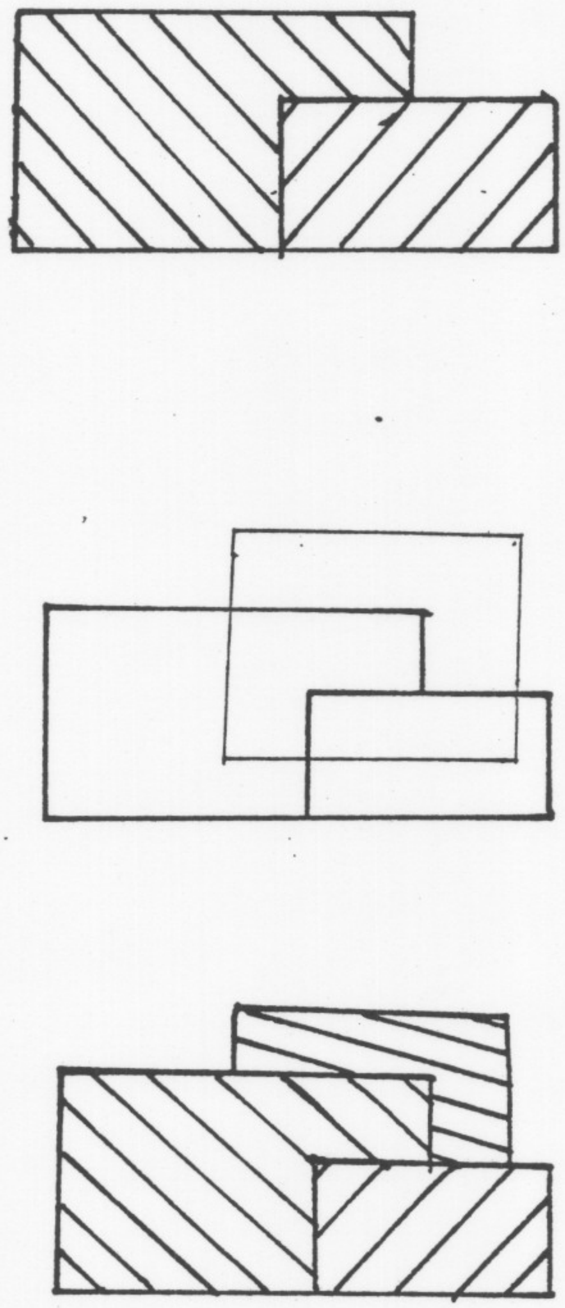

Figure 4-11: plotting a polygon and raising the horizon line

To produce the plot, the prism heights are read and stored in memory in a hash table indexed by polygon number. Then the sorted edge file is read, and each edge induces part of the plot. Consider for example edge E in Figure 4-8. It has polygon A on its left and B on its right. E causes four lines to be drawn - the dashed lines in the figure. They are two vertical edges common to the two prisms and a top edge of each prism. The heights at which to draw the lines are known since each edge knows the polygons on each side. If the horizon line should cut across the lines, only the part above the horizon line is drawn. This is the way hidden edges are prevented from being drawn. After the lines are drawn, the horizon line is raised above them. Figure 4-9 shows Figure 4-2 halfway through its plot with the current horizon line sketched in.

In Figure 4-8, the left prism is higher than the right one. If it were lower, then only one top edge would be visible to be drawn since the higher right prism would hide the left one. Also note that every vertical edge of a prism can be induced by two or more edges of the map. Nevertheless it is drawn only once since after the first time it is drawn, the horizon line is raised high enough that it is not drawn again. Figure 4-11 shows how the horizon line is raised as only the visible portion of a new polygon is drawn.

4.2.9 Shading

The last section described how the plot was drawn and how hidden lines were calculated; this section describes how it it shaded. Two different types of shading are possible: contour lines or vertical shading that assumes an imaginary light source. In either case, extra lines that were not part of the original plot are added to highlight it.

4.2.9.1 Contour Lines -

These lines run along the sides of the prisms and in the original 3-D scene would be horizontal. If the prisms were cut from thick layers of plywood, the contour lines would be the joins between the plies. They are equidistant and enable the user to count up the side of the prism to determine its height.

Contour lines may produced by not drawing the top edges of the prisms immediately. Instead the top edge is raised gradually from the bottom edge in increments of the contour spacing until its proper value. The complete calculation involving the horizon line is performed for each contour line. The edges are still processed in order: all the contour lines for each edge are drawn before the next edge is read. This seems slow but is necessary to determine which parts of the contour lines are visible. Figure 4-1Ø is an example of contour shading. It shows relative per capita public school expenditures by state. Figure *** shows per capita public school expenditures by state with the contour lines at multiples of $5Ø.

4.2.9.2 Vertical Shading -

The sides of each prism can be shaded with vertical hatch lines that create a grey scale approximating illumination from a light source. However, the intent here is not to approximate physical reality, (for which see Newell [1977]), but to suggest contrasts so as to make the plot easier to understand. As an analogy, it is easier to learn to recognize a person from a skilled caricature than from a photograph since the cartoon emphasizes the features, be they a large nose or whatever, that are not average and plays down the normal ones. Here it is desired to highlight the indentations in the boundaries. To do this, a cosine law raised to a power is used. With a cosine law, the shading on a face is directly proportional to the cosine of the angle between the normal to the face and the direction of an imaginary light source. Raising the cosine to a power increases the amount of very light and very dark areas at the expense of the middle intensity grey areas that would normally cover most of the plot.

In Figure 4-8, edge E induced two areas to be shaded. They are a vertical side face of prism A and a slice of the top of prism B. The areas in each case are precisely the area between the corresponding top edge and the current horizon line. The top edge of B is drawn first and then the top of B below the edge down to the horizon line. Since the other edges of B that are below E have already been processed, the horizon line cannot be below the bottom of B´s top. Thus shading down from the top edge doesn´t cause a streak down to the bottom of the plot. After the top edge of B has been drawn, E causes the top corresponding top edge of A to be processed. By now the horizon line is at the top edge of B so that shading down from the top edge of A shades precisely that part of the side face of A which is above B.

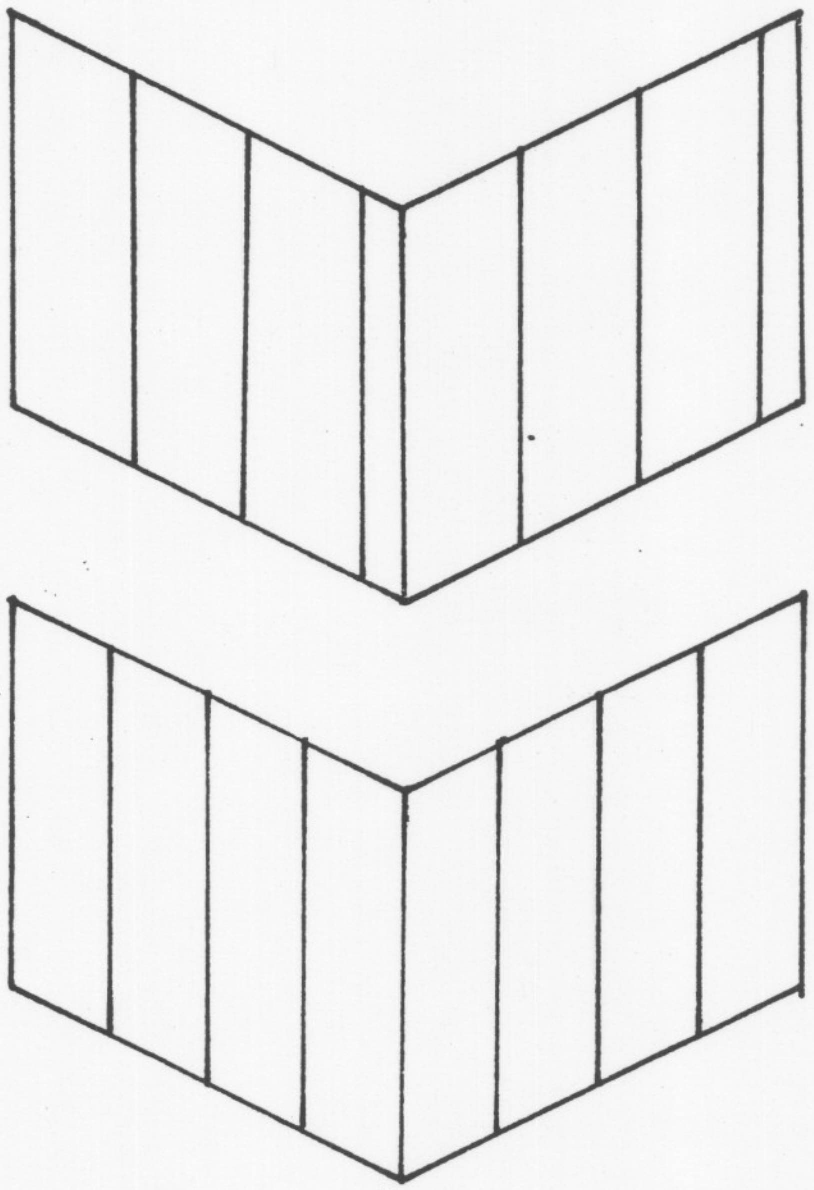
Figure 4-32: Perturbing shading spacing to smooth it

Various details must be handled to make the shading attractive. For instance, the length of E may be not many times the shading spacing. To make the shading blend more smoothly from one edge to the adjacent one, the shading spacing is adjusted slightly so that there are an integral number of shading bands along E. This prevents the dark band that would occur if two shading lines were very close together because one was induced by E and one by E´s neighbouring edge. Similarly a light band could arise. Figure 4-12 shows two adjacent faces before and after the shading spacing was perturbed slightly to as to create an integral number of bands along the face. The resulting shading appears much smoother.

A worse problem arises when E is actually shorter than the desired shading spacing. In this case, a random number generator is used to decide whether to draw one shade line or none. This keeps the average density of shading correct but introduces another problem; that of random clustering. When a long border between two polygons is broken into short straight edges, these edges are separated so that they are not plotted in order. Thus there is no correlation between how the vertical wall due to one face is shaded and how its neighbouring wall is shaded. Because of the random number generator, several consecutive faces may have shade lines followed by several in a row that have none. This is the same type of clustering that is observed when a polygon is shaded by random independent points.

If in fact the whole chain (the sequence of edges forming the border between two polygons) were shaded as a unit, then there would be no problem. Starting at the chain's beginning, the accumulated light or darkness on the chain could be measured as the edges twisted and meandered. When the accumulated light on the chain passed one unit, the accumulated light could be decremented by one, and one line of light drawn. If shading were by lines of darkness, as with ink, an analogous operation would be performed.

This process can be done even though the chains are split up by calculating the shading at the time the map is normalized and before the chains are split into edges. Then each edge has stored with it the starting location and increment, of any shade lines of vertical walls induced by it. However this method requires that the shading algorithm be fixed at sorting time. Previously the angle of illumination and the exact relation between the illumination and the shading spacing could be decided just before the plot was produced without resorting.

Some plot time freedom in the shading algorithm can be obtained by extra information with the edges when they are split off from the chain. This extra information contains quantities such as the edge′s length along the chain, length along the projected chain etc. It allows faces resulting from adjacent edges to be shaded continuously. Assume the shading function can be decomposed into a weighted sum of functions of the light angle and certain fixed basis functions of the edge′s position on the chain. Then if the values of the basis functions are stored with each edge, any shading law obtained by varying the weights can be chosen at plot time. The only problem with this approach is the extra storage required to store the sorted edge file.

4.2.9.3 Silhouette Edges -

The vertical edges of the prisms may or may not be drawn, but either way causes problems. If they are drawn then in places where the map edges are very short, there are more lines due to the vertical edges than due to the shading. This makes the effective shading dependent on the length of the edges which is unreasonable. On the other

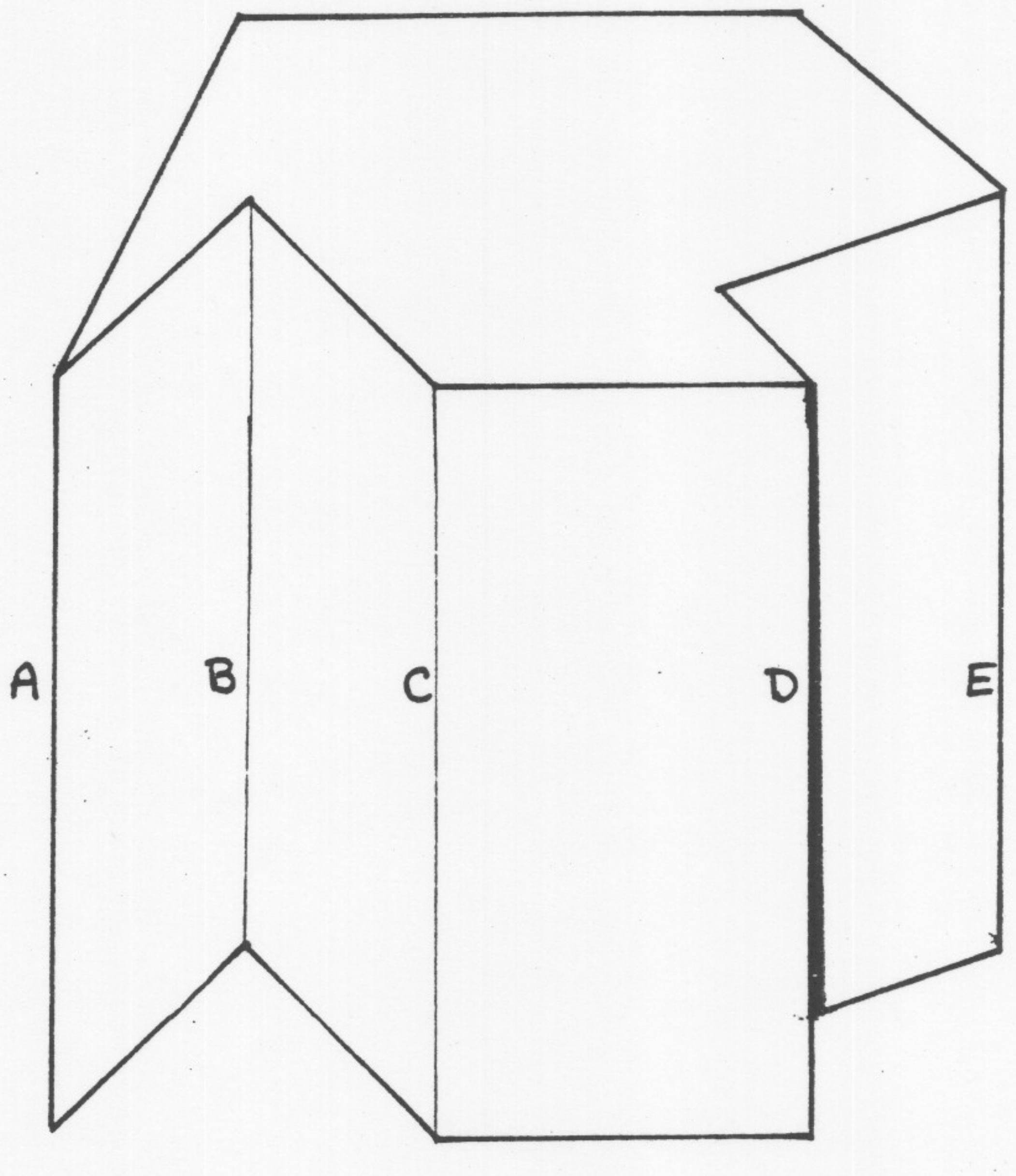

Figure 4-13: Silhouette edges of a prism

hand, if no vertical prism edges are drawn, then there will be no edge to mark where the prism doubles back, unless perhaps there is a shading line right at the edge. To solve this problem, only the <u>silhouette</u> <u>edges</u> are drawn. They are vertical edges rising from points where the 2-D polygon doubles back. In 3-D, these edges generally delimit areas of the plot where there are two different prisms, since if prism P doubles back, then P will be on one side of the vertical edge, in the plot, and whatever is behind it will be on the other side. These silhouette edges are identified by the marking bits that were stored with the edges when the map was normalized and split into separate edges.

In Figure 4-13, the nonsilhouette edges are B and C. The silhouette edges are A, D, and E. The silhouette edges are not the final answer to whether or not to drawn the vertical edges. This is shown by edge D where we might want to omit the part drawn in double thickness.

4.3 STATISTICAL ANALYSIS OF INPUT DATA

4.3.1 Theoretical Analysis

To analyze the time required by this algorithm, it is necessary to know the statistical distribution of the input maps. This is very difficult to determine from first principles. All we can do is to determine a sufficiently robust set of statistics for the algorithm and hope that any novel applications are not too ill conditioned.

Geographic boundaries fall into two categories: natural and man-made. The natural boundaries, according to Mandlebrot [1977], are probably scale invariant and in fact form fractional dimensional curves. Scale invariance means that statistically the boundaries look the same regardless of their scale: an inch to a mile is no different from an inch to an inch. Fractional dimension means that under a suitably generalized notion of dimension, these curves have a dimension between one and two. Scale invariance makes even statistics such as how often a curve crosses a given scan line are meaningless since the curve's length becomes infinite. However this is irrelevent since even though the original boundary may be scale invariant, after it has been digitized and generalized to a given level of accuracy it is no longer scale invariant. This can be proven but is intuitively reasonable since a boundary digitized to a given accuracy, e, will have its form determined by e, and so its properties will depend on its scale.

In contrast, there is no theory at all for madmade borders. They may be straight or smoothly curving lines along parallels and meridans. They may also be gerrymandered without rhyme or reason. Probably the only thing to do here is to take a statistical sample and attempt to derive a heuristic law.

What can be done theoretically is to determine how a map might get more complicated; that is what might happen to the number of polygons, P, and the total length of all the edges, L, as the number of vertices, N, increases. Two things may happen:

1. The map may have the same polygons as before but the boundaries may be represented more accurately so that P stays constant but L increases with N. The exact relation depends on the form of the borders.

2. Not only may be boundaries get more accurate, but there may be more divisions. For instance if the first map is of states, the second, more accurate map may include counties and parishes.

Statistics for the first case were gathered and will be summarised in the next section. For the second case, let a map with a given N, P and L be replaced by 4 copies of itself reduced to half the scale so that the total size remains constant.

Then $N' = 4N$

$P' = 4P$

$L' = 4L/2 = 2L$

Thus $P = \Theta(N)$

$L = \Theta(N^{1/2})$

These statistics are independent but there are various important dependent statistics such as the average number of edges crossing any scan line, or the average number of active edges, M. Another is E, the number of edges. Most of the vertices are incident on 2 edges and rarely a vertex is incident on as many as 4. As N increases the fraction of vertices incident on only 2 edges tends to one. This is because most of the vertices are separating short straight line segments on a long continuous boundary. So E = N. Now an edge of length L that is assumed to be randomly oriented has a projected vertical length of 2L/pi $\doteq$ Ø.637 L. Since the screen is of height one, this is its probability of intersecting a given random scan line. Thus the average number of edges intersecting a scan line is

$$\begin{aligned} M &= .637\ L \\ &= \Theta(L) \\ &= \Theta(N^{1/2}) \end{aligned}$$

4.3.2 Heuristics

To better determine the distribution of input maps, some measurements were made of parts of the World Data Bank II, described in Anderson [1977], which contains national boundaries and of the aforementioned map of the USA with state boundaries. To test the first way a map might get bigger, WDB-II with 14378 edges was generalized to 7 different levels of accuracy, the last reducing the number of edges to 24Ø9. The generalization algorithm of Douglas [1973] was used. The number of active edges, M, was related to the number of edges, N, with a 9% error by $M=cN^{Ø.15}$.

This is a much slower rate of increase than the theoretical analysis showed for the second case so the second case is indeed a more stringent test of the algorithm.

Next the robustness of the number of active edges was tested by rotating the USA map to 6 different angles and measuring the average and maximum number of active edges. To the nearest integer, the average was always 15 and the maximum varied from 29 to 35.

It is somewhat surprising that the average should be so constant since the USA contains so many borders that run either north-south or east-west. There seem to be three reasons:

1. The north-south edges tend to balance the east-west edges to the first order leaving only a second order variation in the total projected edge length (which is proportional to the average number of active edges) as the map is rotated.

2. Even though the predominant direction of an edge may be north-south, there are many little diversions that do not appear too important on the map but which even out the statistics.

3. Although the straight man-made borders stand out, still most of the map consists of natural borders whose statistics are independent of orientation.

However, this invariance of M with angle or orientation fails to hold with maps where in one part the size of the polygons is greatly different than in another part. One such example is Chicago Standard Metropolitan Statistical Areas which are much smaller in the city centre than in the suburbs. Then a scan line that runs along a main street can cut many more edges than a scan line that has been rotated even slightly.

4.4 RESOURCES REQUIRED BY THE ALGORITHM

4.4.1 Time

This section analyses the theoretical performance of the algorithm. Luckily it turns out that the results are robust and independent of the detailed distribution of the input data. Let

T_N = Time to calculate and plot one scene.

A_N = Initial formatting time.

B_N = First sorting time.

C_N = Second sorting time.

D_N = Horizon array processing and plotting time.

Then $T_N = A_N + B_N + C_N + D_N$

Now the initial formatting time is one sequential pass so

$$A_N = \Theta(N)$$

and like all reasonable sorts,

$$B_N = \Theta(N*\log(N))$$

The final sorting time requires accessing a heap and a tree, both of average size M, for $\Theta(N)$ times. Each access takes time $\Theta(\log(M))$. So

$$C_N = \Theta(N*\log(M))$$

This is no larger than B_N so long as $\log(M) \leq \log(N)$, that is so long as $M \leq N^c$ for some c. In the last section on statistics of the input data, this was determined to be true with $c \leq 1/2$. Note that although the actual value of c doesn´t affect the rate of growth of the time, it does have a dramatic effect on the multiplicative constant since in practice the final sort is by far the slowest part of the algorithm. Note also that the restriction of M to be a polynomial function of N is no restriction at all since M is bounded above by N. This would give c=1. If this actually happened, the algorithm´s asymptotic growth would be the same but it would use much more storage. In any case

$$C_N = O(N*\log(N))$$

The final plotting time requires one pass through the data so

$$D_N = \Theta(N)$$

Thus $T_N = \Theta(N*\log(N))$

which, for a hidden surface algorithm, is a satisfactory time.

4.4.2 Storage

The whole input file is never in memory at any one time. During the preprocessing stage, three edges need be in memory together. One is being processed and the other two are its neighbours that are needed to set the silhouette bits stored with the edge for shading. The external sorting runs better the more storage it gets, but only needs a small constant amount. The only variable part of the algorithm is the final sort during which the active edges all must be in memory. This is $M = \Theta(N^{1/2})$ edges or for the 4641 edge USA map, an average of 15 and a maximum of 32. The final plotting requires a constant amount of storage.

4.5 IMPLEMENTATION

The algorithm described in this chapter has been implemented and is described in Appendix B. This appendix also contains many more examples of prism plots.

CHAPTER 5

THE HIDDEN SPHERE SPECIAL CASE

5.1 INTRODUCTION

This chapter gives an algorithm for the hidden surface problem in which the scene is a set of non-overlapping spheres in perspective projection. If the assumptions mentioned later are satisfied, the algorithm executes in time $\Theta(N^{5/3}*\log(N))$. Under these same assumptions, X, the number of intersections among the intersections of the projected spheres, is $\Theta(N^{4/3})$ so the time is $\Theta(X^{5/4}*\log(X))$ which is not too much faster than $\Theta(X)$.

A typical scene might be a ball model of a molecule, each atom of which is represented by a sphere. This problem is important because molecular models for chemists are more than just toys. Chemists need to construct them to obtain an understanding of the spatial relationships between the different parts of complex molecules. The repetitive construction of mechanical models is slow and tedious, so it is useful to automate the problem. Such an algorithm can also provide a graphic output mechanism for other computer programs in chemistry, such as those that assist the chemist in developing new synthesis paths. These work by combining a large computer database of chemical knowledge with the chemist´s intuition and creativity. Thus any means of making the interface freer are useful.

To date, much of the work on the design of hidden surface algorithms has been done on the case in which the objects have straight edges and flat faces. A good summary of the various algorithms with many references is Sutherland, Sproull & Schumacker [1974b]. Curved surfaces (modeled by splines), reflections and semitransparant objects have been handled by Blinn [1976], Crow [1976], Gouraud [1971], Levin [1976], Mahl [1972] and Phong [1975] but the algorithms are very slow. Wright [1974] plots irregularly shaped objects by dividing a 3-space box containing the object into cells with a 3-D grid. Then a bit array with one bit per cell tells which cells the object is in. Finally a generalized horizon line problem draws the visible portions of the cells, front to back. This algorithm was designed to draw electron orbital clouds.

Here I consider another special case where the 3-D object is a collection of nonoverlapping identical spheres such as the atoms in a ball model of a molecule. Because of the difficulty of drawing such models, chemists' programs have generally been restricted to plotting line outlines of the molecules that show only the bonds, such as in Nir, Garduno & Rein [1977]. Since there are no hidden lines, there are few depth cues in the plot so cues must be added by mechanisms such as viewing stereo pairs and varying the thickness of the lines.

A special case hidden sphere algorithm should run more efficiently than a general hidden surface algorithm because:

1. It considers the hidden spheres as spheres and not as straight line approximations or as special case splines or patches.

2. There is a partial order spheres: spheres A and B are related when A is in front of B as seen from the viewpoint. This partial order doesn't exist for general objects in three dimensions. For example, in Figure 5-1, three bars, A, B and C, are

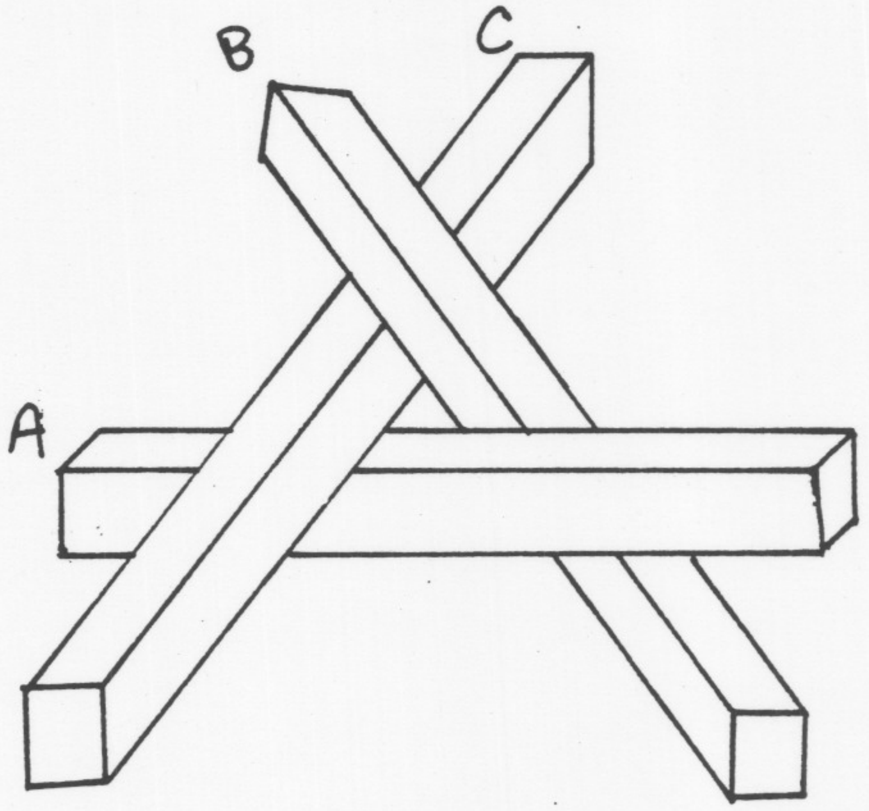

Figure 5-1: Three objects that cannot be ordered by distance

arranged in space so that A is in front of B at some point, B is in front of C, and C is in front of A. General hidden surface algorithms that wish to use this partial order must detect such violations and split the offending objects into several pieces until a partial order exists.

3. Since only spheres and not straight lines and planes are allowed in the input, the algorithm can be much simpler than a general purpose one that handles more general scenes. This is not to say that the algorithm would necessarily be slower if it handled more special cases, only that it would be more complicated.

5.2 NOTATION

Let N = number of spheres

s_i = the i-th sphere.

c_i = the i-th circle, the projection of s_i onto the perspective plane. c_i will refer to either the perimeter or the surface area, depending on the context.

r_i = radius of circle c_i in the plot.

r = r_i when all the r_i are the same.

d_i = distance of sphere s_i from the viewpoint.

p_i = location of the centre of c_i on the screen.

5.3 DEFINITIONS

1. constant density packing of the spheres: As N tends to infinity, the total volume of the spheres is Θ(the volume of the smallest enclosing cube).

2. average depth of circles on the screen: This is the average number of circles a uniformly distributed random point on the screen falls within. Given that the screen has a fixed size, this is proportional to the total area of the circles.

5.4 ASSUMPTIONS

1. The size of the screen is 1 by 1.

2. As N increases, the r_i are scaled so that the projected scene fills the screen.

3. No two spheres intersect in 3 dimensions (although of course their projected circles frequently overlap in 2 dimensions).

4. The spheres are equal sized and the projection is isometric so all the spheres project onto circles of the same size.

5. The spheres are ordered so that $i<j = d_i \leq d_j$.

6. The spheres are packed with a constant density as N tends to infinity.

7. The complexity of the plot is taken to be the number of intersections between the c_i.

5.4.1 Accuracy Of Assumptions

The first two assumptions are that the screen is a fixed size and that the projected scene fills it. They just assert that the projection and scale are normalized.

The third assumption, that no two spheres intersect in three dimensions, restricts the utility of the algorithm somewhat since it forbids molecular models where the electron clouds around the atoms overlap each other. It is made so that the intersections of the spheres do not have to be calculated. Also, if the spheres intersect, the projections of their perimiters are no longer circles, but now are combinations of sections of general conics. Nevertheless, it would be worthwhile to relax this assumption sometime.

The fourth assumption, that the projected circles are of equal sizes, also simplifies the algorithm and its analysis. The analysis is simplified because if the circles are of different sizes, either because the spheres are different sized or because the projection is not isometric, then there is another parameter that must be modeled and analyzed. Since, as has been stated before, there is no clean logical definition of, and probability measure on, the sample space of input scenes, this is not just a mathematical problem. It involves experiments on actual scenes likely to be used.

The problem here is that there is no clear axiomatic definition of the hidden surface problem since it is an applied problem and must solve the problems users wish it to solve. Determining the statistics of the scenes the algorithm will be used on, before the algorithm exists, is impossible since even the users don´t know. After the algorithm becomes available, a demand will be created that did not exist before. The space of problems that the algorithm will be used on will also become favourably biassed as time goes on since it will be used more often on those cases it handles the most efficiently. Because of this, the best that can be done is to pick some reasonable scenes that are neither too easy nor too hard and design the algorithm to solve them. This is why assumption 4 is made. Nevertheless it will be relaxed later.

Assumption 5, that the spheres are sorted, states that a simple preprocessing step taking time $\Theta(N*\log(N))$ has been performed already.

Assumption 6, that the spheres are packed with constant density, is major and is not obviously true. It is satisfied by molecules that grow like "blobs" equally in all directions without sending out long streamers. Regular crystals usually satisfy this. The implication of this assumption is that only a few of the atoms in the molecule are visible, even partly. For instance in a K by K cubic crystal, of the K^3 atoms, only the $\Theta(K^2)$ atoms (those on the front surface and a small distance in) are visible. However, many organic molecules are long and stringy. As they get bigger, they get longer but no wider. Thus when observed from the side, most of the atoms are visible. Of course, for any given molecule, the radius of the atoms can be changed so that any proportion of the atoms are visible. In the limit as $r \to 0$, the atoms become points, and thus they all can be seen.

The last assumption, 7, that the measure of complexity is the number of intersections among the projected circles, sets a standard of complexity against which the performance of different algorithms can be measured. It is insufficient to use the number of spheres, N, since then the r_i must also be considered. This is an attempt to combine them into one measure of complexity. It is a reasonable one since the number of arcs is proportional to the number of intersections. Another reasonable measure of complexity would be the number of visible arcs of spheres but this is much more difficult to handle statistically.

Under the equal density assumption, the number of intersections is related to the number of spheres by $X=\Theta(N^{4/3})$ so this assumption is useless since N can be used just as easily. This assumption is included for the more general cases when the equal density assumption is relaxed.

5.5 THE HIDDEN SURFACE ALGORITHM

5.5.1 Perspective Projection

As usual, and as explained in chapters 2 and 3, we are looking from a point in 3-space, the viewpoint, through the perspective plane, to the scene. The image on the perspective plane of any point on the scene is the intersection of the plane and a straight line between the viewpoint and that point in the scene. If the projection is isometric or orthogonal, then the viewpoint is at infinity and the projection lines are parallel.

5.5.2 Schematic Algorithm

```
PROC Hiddensphere(SS)
   Project(SS);
   Normalize(SS);
   Sort(SS by $d_i$);
   G = {g} <- Calcgrid(SS);
   FOR i TO |G| DO
      H <- {h | h=g and there is a circle with centre in
            g that intersects $c_i$};
      U <- {u | u=c such that c has centre in some h};
      A <- {a | a is a arc resulting from intersecting
            some c with circles in U};
      A2 <- Ø;
      FOR p to |A| DO
         b <- true;
         FOR l TO |U| DO
            IF Contains($u_l$, $a_p$) THEN b <- FALSE;
         ENDDO;
         IF b THEN Plot($a_p$);
         IF b THEN A2 <- Union(A2,$a_p$);
      ENDDO;
      Shade(A2);
   ENDFOR;
ENDPROC;
```

<u>Notes</u>:

1. Hiddensphere takes the set of circles as argument.

2. Project(SS) projects the spheres according to the viewpoint and perspective plane.

3. Normalize(SS) scales and shifts the projection so that it fills a one by one square.

4. Sort(SS) sorts the projected circles by the distance of the original spheres from the viewpoint. The spheres are renumbered according to this order for future simplicitly.

5. Calcgrid(SS) calculates a grid of NG by NG cells for some NG dependent on SS in a way to be determined later. It returns G={g} the set of grid cells.

14. Contains(u_1, a_p) returns true or false depending on whether the circle u_1 contains the arc a_p. This is a fast operation since a_p cannot cut u_1; it is either all inside or all outside. So just pick a point on it and see if that point is closer than r than the centre of a_p.

16. Plot(a_p) draws the arc a_p. Since the circle, c, that a_p came from is known, the arc can be drawn in such a way as to indicate this. For example if the scene is a molecular model, circles corresponding to different types of atoms might be dotted or dashed instead of being drawn solidly.

17. Shade(A) shades the area enclosed by the arcs of A.

5.6 STATISTICAL ANALYSIS

A circle, c_i, is intersected by all other circles c_j such that $|d_i-d_j| < 2r$. The problem of finding them seems similar on the surface to the nearest neighbour algorithms of Rabin [1976] and Yuval [1975]. However there is little relation since we want all the points within a given distance, not the closest point.

<u>Lemma</u>: If there are N spheres packed with constant density, then the scene has size $\Theta(N^{1/3})$.

<u>Proof</u>: If the scene has linear size x then its volume is x^3. The total volume of the spheres is $\Theta(N)$ which is Θ(volume of the scene by definition), so $x=\Theta(N^{1/3})$.

<u>Lemma</u>: If the scene is projected to a size of one, then $r=\Theta(N^{-1/3})$.

<u>Proof</u>: Obvious.

<u>Theorem</u>: The greatest number of arcs that N circles can cut each other into is 2N(N-1).

<u>Proof</u>: Each pair, A & B, of circles can cut each other in two places so however many arcs A had before B cut it, it will have two more after. Each time A is cut, one more arc is created. The same goes for B. Thus there are up to 4 new arcs per pair of circles and C(N,2) = (N choose 2) pairs for a total for 2N(N-1). (This proof is just of an upper bound, but the bound is actually achieved.)

<u>Theorem</u>: The expected number of pairs of N circles that will actually intersect (of the C(N,2) possible) is $\Theta(N^2r^2)$.

<u>Proof</u>: A given circle, A, will be intersected by any other circle closer than 2r. This covers an area of $4pi*r^2$ (of a total area of one). Thus a given pair of circles has a

probability $\Theta(r^2)$ of intersecting, for a total of $\Theta(N^2r^2)$.

Corollary: The number of circles expected to intersect a given circle is $\Theta(N*r^2)$.

Given NG, the number of grid cells on a side of the screen, the side of a grid cell is thus 1/NG. So the radius, r, of a circle is r*NG times the size of a grid cell. Thus one circle covers $\Theta(1+r^2NG^2)$ cells. This is $|H|$ in line 8 above. It can be kept constant if

$$NG = O(1/r).$$

Now there are N circles distributed among NG^2 cells for an average of n/NG^2 circles centred in each cell. Thus in line 8 above,

$$|U| = |H|\ N/NG^2$$
$$= (1+r^2NG^2)N/NG^2$$

This is minimized when NG = Omega(1/r). Thus combining this with the previous result,

$$NG = \Theta(1/r)$$

is optimal and gives

$$|U| = \Theta(N*r^2)$$

Now line 9 above takes time $|U|\log|U|$ every time it is executed and it is excuted with $\Theta(N^2r^2)$ arcs so its total time is

$$T = \Theta(N^3r^4)\log(N^3r^4).$$

Since this is the slowest step in the algorithm, it is the time for the whole algorithm.

Now under the assumption of constant density packing,

$$r=N^{-1/3}$$

so $$T = \Theta(N^{5/3}\log(N))$$
$$= \Theta(X^{5/4}\log(X))$$

where X is the numberof circle intersections.

5.7 CHANGING THE ASSUMPTIONS

5.7.1 Different Sized Circles

Up to now the circles have been restricted to be the same size. This means that the spheres must be the same size and the projection isometric. However nothing in the algorithm restricts them to be so. The problem is that with varying size circles, the size is correlated with the distance from the viewpoint and thus with the probability that the circle is (partly) hidden. So the statistics become much messier. There is no guarantee that the algorithm will still run as fast because it depends on the farther spheres being mostly hidden. If the spheres are all the same size but a true perspective projection is used, the farther circles will be smaller. Thus they will be even more likely to be hidden and the algorithm shouldn't be slowed down. But if the original spheres are different sized, things become very messy.

Also if the circle size varies, there is no longer such a natural grid size. Above the grid cell size was chosen to optimize the execution time of the algorithm. This might not always be possible with varying size circles. Also there would no longer be a fixed neighbourhood of cells whose circles and whose circles alone could overlap a given cell. Probably cells would have to contain pointers to the neighbouring circles that overlapped them and when a new big circle was processed it would have to be added to the lists of all the cells it covered.

5.7.2 Ball And Stick Models

"Ball and stick" models, where the sticks are cylinders connecting the spheres present further problems. In the special case where the sticks are lines of zero width the extension to the algorithm is easier since the sticks can not hide spheres but can only be hidden. But if the sticks have a finite width, they can hide spheres and each other. Even the small arc of a circle at a stick´s end where it meets a sphere may be partially hidden.

5.8 SUMMARY

Thus it is possible to solve this special case of the hidden spheres fairly quickly using the technique of the variable grid. It would be preferable, nevertheless, to bring the time down from $\Theta(N^{5/3}\log(N))$ to $\Theta(N^{4/3}\log(N))$ since then it would be $\Theta(X*\log(X))$ where X is the number of intersections.

CHAPTER 6

CONVERTING VECTOR PLOTTER COMMANDS FOR RASTER DEVICES

6.1 INTRODUCTION

Many graphics plotters and display devices, both hard copy and CRT, such as Tektronix, Calcomp and Milgo are vector plotters. That is they draw edges between 2 given points, although possibly in small increments. This is usually the more natural and intuitive way of plotting. However other display devices, such as Evans & Sutherland PS-3, Ramtek, Gould, Xerox and Versatec are raster. They cover the screen in order from top to bottom with horizontal scan lines like a TV set. Orr [1978] contains a good summary of the various graphic display devices. Raster display devices are becoming increasingly important because they are better suited to shading, can use existing TV technology and are supplied by torrents of raster data from sources like Landsat. Although it is the prevailing opinion, as stated by Negroponte [1977], that raster graphics will soon supplant vector graphics, the latter will survive for a few years yet, if only because of the investment in existing equipment. Thus there is a need to convert between these two totally different methods.

This need probably exists in the short and mid term only, since there is a prevailing trend to identify special purpose functions that are used often and to implement them in special hardware. At the rate the cost of hardware is dropping, it will soon be cheaper to implement a conversion chip and display buffer right in the raster device than to design an efficient algorithm to run on the host computer. Nevertheless, even if such a raster plotter should be announced tomorrow, there would remain large numbers of raster plotters without such aids for several years to come. Thus it is worthwhile to develop these algorithms.

In this chapter, I analyze such algorithms for converting commands intended for a vector plotter so that they can drive a raster display. I compare a variety of different algorithms of varying complexity. Although there are many existing heuristic algorithms, some of which have been published, Jordan [1973] and Barret [1974], no systematic analysis has been published.

These algorithms assume a picture buffer of bits, one per pixel, into which the edges are written. However, since usually there is not enough internal memory to store the whole buffer, it must be split into strips and only one strip kept in memory at a time. Then there arises the problem of whether to read each edge once while reading and setting all the strips it falls in or whether to keep each strip in core once while reading the edge file several times. A mixed strategy can also be used. The edges can be kept whole or split. If they are split, they may be split wherever they cross into a new strip or they may be split into separate pixels. Curved edges might be handled separately from straight edges, or they might not. In many plots it is desirable to shade in areas. This is easy on raster devices but must be done by rows of closely spaced parallel lines on vector plotters. These crosshatch lines have different statistics from normal lines in an average plot since there are more of them and they are longer. Thus the algorithms must be designed accordingly.

6.2 ASSUMPTIONS

Assume that the screen is one by one.

Let

P = # bits on 1 side of the raster page.

N = # plotter edges to draw.

L = total edge length, page widths.

B = # bits precision of the vector plotter, that is the number of bits needed to express one coordinate of a point.

M = amount of internal memory available for data arrays, in bits.

Q = average projected vertical length of an edge.

T = time, or equivalently cost, for an algorithm to run.

S = # strips the raster screen is split into. These are horizontal strips of height $\frac{1}{S}$ running the whole width of the screen. S depends on the actual algorithm used. The strips will be described in more detail in the section on algorithms.

Sample values might be:

$P = 2000$ (Gould 5200 plotter/printer)

$N = 10000$

$L = 100$

$B = \log P = 11$

$M = 1{,}000{,}000$ (= 30K words on PDP-10).

6.2.1 Notes

1. These sample values depend greatly on the type of the plot; and particularly on whether it is just a line drawing or has crosshatch shading also. If so, then the total edge length, L, is much greater. These values of N and L are chosen to approximate a plot that is a 100 by 100 grid covering the screen. Each line running across the plot is not one but 100 edges. This appears to be a sufficiently complex plot to test the algorithms.

2. To make the numbers of bits easier to grasp, I shall assume a 36 bit machine with 1KW = 1024 * 36 = 36864 bits. This number of bits per word of course doesn´t affect the calculations which are all reducible to numbers of bits.

3. Simplifying assumption: Draw the edges 1 horizontal raster bit wide if they are inclined at less then 45 degrees to the horizontal and 1 vertical raster bit wide otherwise. That is, an edge that is inclined at an angle of less than 45 degrees will cause only one bit to be set in each column of raster bits that it passes through. In each row of raster bits, it will cause one or more bits to be set. Make no attempt here to give different slopes equal visual density, that is equal density of raster dots per unit length measured along the edge instead of along an axis. This should be done for aesthetic purposes, but is only a constant factor harder and the methods are well known.

Thus # raster bits to set =ØL*P = 2ØØ,ØØØ.

4. LOG is to base 2 and LN to base e = 2.71828... Pi = 3.14159...

5. Definition: <u>Edge</u>: An edge between two endpoints as drawn by a plotter or simulated on a raster display.
Definition: <u>Line</u>: A raster scan line.

6. Assume the edge angles of inclination are uniformly distributed. This makes $Q = \frac{2L}{pi}$. The most likely alternative is that the edges are half horizontal and half vertical but never oblique. Then $Q = \frac{L}{2}$. This assumption only affects costs by a small constant and is only relevant insofar as it might change a breakeven point between two algorithms. But even here, the two algorithms would have to be quite close in cost before this differentiated between them.

7. Cost is generally dominated by the amount of I/O and everything not mentioned as a parameter above, such as amount of temporary disk space and programmer time, is free. Another significant factor that is not considered is the number of different devices available for temporary storage. If there are more disks available, then there is less thrashing. Another factor is the relative cost of reading blocks sequentially versus reading them randomly. It is significantly cheaper to read ten consecutive blocks than to read ten blocks scattered randomly over the disk.

8. P has other reasonable values:

    5ØØ for a TV screen,

    1ØØØ for a Tektronix 4Ø1Ø,

    4ØØØ for a Tektronix 4Ø14.

6.3 ALGORITHMS FOR STRAIGHT EDGES

These algorithms illustrate different aspects of the tradeoff between internal storage used and amount of I/O. A given amount of memory can be used for different purposes. The two opposite extremes for this are:

1. Store a page of raster bits in memory. Read the edges one by one and for each edge set the corresponding bits. Display the page.

2. Store all the edges in memory. Calculate a raster scan line by checking all the edges and setting the bits in the line for any edges that intersect it. Write the line and calculate the next one. Display

the lines in order on the device.

Since there is rarely enough memory for either limiting approach, the following algorithms generally follow middle paths. Since in general it is necessary to use all the edges to calculate all the lines, two different approaches are possible: to iterate through the edges in order, or to iterate through the scan lines in order. We consider various refinements of these two basic approaches.

These are all essentially sorting methods: the difference lies in what is being sorted in what order. This conclusion is similar to that which Sutherland, Sproull & Schumacker [1974b] have found for hidden surface algorithms.

6.3.1 Iteration On The Edges

Define the display screen with a bit buffer of P^2 bits. If $M < P^2$ then split the buffer into horizontal strips of M bits. Have one strip in core at a time, and calculate which bits are set in it by one of the following methods:

6.3.1.1 Six Possible Algorithms -

1. Read the whole edge file for each strip. As each edge is read, calculate how much of the edge, if any, falls within the strip, calculate which bits of the strip to set, given the edge's slope, and set them. When all the edges have been processed, write the strip out, initialize the memory for another strip and read the edge file again.

2. This algorithm is the same as algorithm 1, except that after it is determined how much of an edge falls within the current strip, write out the remaining pieces, if any, to another file. There will be either zero or one pieces, depending on the relation between the edge and the strip. If the edge is totally within the strip then there will be no pieces left over. If it is either completely outside the strip or partly within the strip and partly within the next strip, then there will be one piece. The most common case occurs when the edge falls completely outside the strip. Then when processing the next strip, read this file, which should be shorter than the original edge file, and write another even shorter file of pieces to be read for the next strip, and so on.

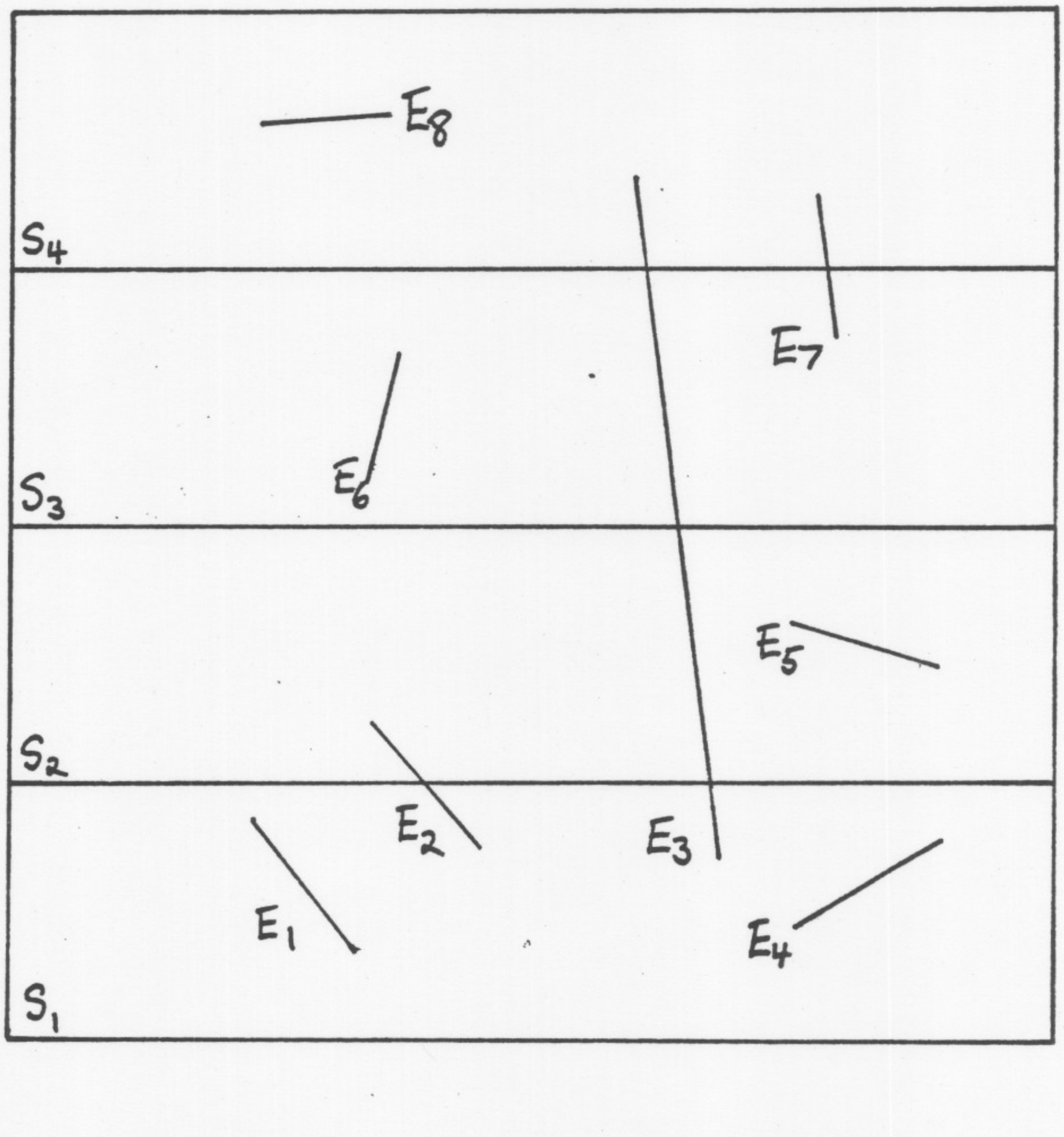

Figure 6-~~1~~: Dividing the screen into strips

In Figure 6-1, the screen is divided into four strips. Edges 1 and 4 are completely used up when strip 1 is calculated and so are not written. The parts of E_2 and E_3 not in S_1 are written. E_5, E_6, E_7, & E_8 are written unchanged.

3. Presort the edges by the lower Y endpoint coordinate of each edge. Then when processing a given strip, read the edge file from the start, ignoring edges not at least partly in the current strip, until the last edge that can possibly be in the strip is read. This will be the edge before the first edge whose lower Y endpoint is in the next higher strip. Don′t read the rest of the file. Thus when the first strip is processed, not much of the file is read. For each successive strip, more is read.

4. This is similar to algorithm 3, but does less reading of the edge file. When the file is being read to do strip #i, look at the edges and remember the location in the file of the first edge that will fall within strip #i+1 or a higher strip. Then when strip #i+1 is being formed, start reading the edge file at that point. If the edge file is on tape, do a fast forward, counting past blocks, to that point. If

the file is on disk, do a random access to that point. In either case, continue reading the file sequentially from there.

There is one problem with this method: one long edge at the beginning of the file that crosses all the strips forces the whole file to be read for each strip. There are two solutions to this problem:

1. Perform a preprocessing phase that splits all the very long edges into manageable pieces. It is not necessary to split every edge that falls into two strips, only the "long" ones. Exactly how long this is would depend on implementation details and is a simple optimization problem.

2. Instead of only remembering the first edge that falls within a later strip, chain together all these edges, or else remember a list of them, possibly by writing this to a separate file. Thus if the marked edges are far apart, a series of random accesses can be done without reading a long string of useless edges between them.

Again, the breakeven point between reading two edges by random access versus reading all the edges between sequentially depends on the relative costs of sequential and random access. In the limiting case of writing out edges that will be useful in the future, this method tends to algorithm 2, except that this algorithm sorts the edges first.

5. Initially read all the edges and write several files, one for each strip of the raster screen, each containing the pieces of the edges that fall within that strip. Edges that cross a strip boundary are cut and the pieces written to two or more files. To prevent disk thrashing, the different files should probably be on different devices. Then allocate the whole memory to the first strip, initialize it, and read the first edge file and set the appropriate bits in the memory. When the first strip has been calculated, write it and calculate the second strip, and so on.

6. Perform this algorithm like the above one but write all the pieces to one file, each piece tagged with the strip it goes in. Then sort the file by strip number. Continue as with algorithm 5, calculating one strip at a time, only here

reading all the edge pieces from one file instead of from different files.

6.3.1.2 Analysis Of The Algorithms -

Since external sorting is an integral part of all these algorithms we need to know its performance. Knuth [1973] sect 5.4.6 lists several algorithms. Since we are presumably using disk, not tape I/O, only the number of passes through the file is material. The exact figure is impossible to calculate since it depends on too many variable or unknown factors. However, all the algorithms have

$$P = \# \text{ of passes} = c_0 \log\left(\frac{N*B}{M} + c_1 \right)$$

That is, the number of passes depends on the number of times bigger the file is than the workspace and not on the number of records. Assuming that the file buffer space doesn't come out of M, or at least is a negligibly small fraction of it,

$c_0 = 1$

$c_1 = 0$ (assuming 3 way merging).

Thus we can approximate the number of passes by

$$P = \log(N) - \log(M) + \log(B)$$

and if the file is 2BN bits long,

$$\begin{aligned} T &= 2\ B\ N\ P \\ &= 2\ B\ N\ (\log(N) - \log(M) + \log(B)) \end{aligned}$$

This is in units where the cost of reading and writing 1 bit is 1. Now assume that the memory is being used to store a raster strip. Then

$$\begin{aligned} S &= \#\ \text{strips} \\ &= \text{ceil}(P^2/M)^* \\ &= 4 \text{ with the sample values.} \end{aligned}$$

Now the probability that an edge crosses a strip boundary is

$$\begin{aligned} &= \text{prob its centre is within } \frac{Q}{2} \text{ of a boundary.} \\ &\acute{=} \frac{1}{(S-1)\ Q} \\ &\acute{=} 0.0022 \text{ for the sample.} \end{aligned}$$

* ceil(x) is the smallest integer not smaller than x.

Alternately, assume that the memory is being used to store edges. Then the number of edges that can be in core at once

$$= \frac{M}{2B}$$
$$= 44{,}000$$

Now it may be possible to store the edges more efficiently and so pack more into memory. For instance, instead of storing both endpoints, use the fact that the average edge is 20 raster pixels long and store the X and Y increments instead of the second endpoint. Delta X and delta Y can be stored in about 5 bits each instead of the 11 bits needed by each component of the second endpoint. This reduces the space needed for an edge from 44 bits to 32 bits and raises the number of edges that can be stored in memory from 44,000 to 60,000. We could even go all the way and make a preprocessing pass through the edge file for the sole purpose of gathering statistics on the edges in order to design an optimally tuned Hamming code. However here the time to decode the edge´s coordinates might become significant.

Algorithm 1 reads the whole edge file S times. Assuming the average edge is small enough that relatively few cross a strip boundary (for the sample this is 0.2%) and assuming the edges are evenly distributed, pass #i of algorithm 2 reads $\frac{S-i+1}{S}$ of the edges and writes $\frac{S-i}{S}$ of them so the whole file is read $\frac{(S+1)}{2}$ times and written $\frac{(S-1)}{2}$ times for a total of S I/O operations on each edge. This is exactly the same cost as for algorithm 1 and so algorithm 2 need not be considered further.

These algorithms could be differentiated by various secondary cost factors, however. For instance, some operating systems might find it cheaper to write a block than to read one since they can buffer it for a longer time while queuing up several writing requests before handling them together. In this case, algorithm 2 would be cheaper since it does writes where algorithm 1 does reads.

The cost for 1 or 2 is:

$$\begin{aligned} C_1 &= \text{total number of bits read or written} \\ &= 2\ B\ N\ S \\ &= 880{,}000 \quad \text{for the sample.} \end{aligned}$$

Algorithm 3 is like 2 except that only reading is done, but the edges must be presorted. For the sample statistics, where internal sorting is sufficient, the cost of sorting is a*N*log(N), where a is the cost of sorting 1 record internally. So

$$C_3 = a*N*\log(N) + 2B*N*(S+1)/2$$
$$= 90{,}000a + 550{,}000$$

This is the I/O required for 55ØK bits. But reading the edge file once takes 2B*N = 22ØK bits so this requires the equivalent of reading the edge file 2 1/2 times. In the general case however, external sorting would be necessary, so

$$C_3 = \text{cost of sorting} + \text{cost of reading}$$
$$= 2*B*N*(\log(N) - \log(M) + \log(B)) + 2*B*N\frac{(S+1)}{2}$$
$$= 2*B*N*(\log(N) - \log(M) + \log(B) + \frac{(S+1)}{2}$$

This method tends to be better than algorithm 1 when N is small and S is larger.

Algorithm 4 cuts the extra edge file reading down to essentially nothing if the edges are short. However the simple algorithm becomes much worse if there are a few long edges. The methods of handling this require either preprocessing the edge file and splitting the long edges, or writing another file listing the long edges, or at least keeping some data in memory on the long edges. Here the optimum choice is dictated by the frequency of the long edges.

Algorithm 5 is not recommended because of disk thrashing and because every strip needs an output buffer in memory. In primitive languages like most implementations of Fortran, buffer space and user arrays cannot be (legally) equivalenced so the buffer space must remain allocated even when it is no longer needed. Further there are limits on how many open files there can be. Nevertheless in some cases, where the number of strips is small, this might be useful. Especially for a very large complicated plot it might be worthwhile to dedicate S output devices and use this method.

Since as was shown before, few edges cross a strip boundary, in algorithm 6, there are not many more pieces of edges than edges themselves. Thus sorting the pieces takes no more time than sorting the edges themselves. After that the edge file need be read just once. Indeed, since the sorting need only be done by strip number. So once all pieces for strip 1 are before any for strip 2 etc., the sorting can be stopped since the order of edges within a strip is immaterial. Although sorting N records completely takes $\Theta(N*\log(N))$ time, sorting to within S strips has an information theoretic lower bound of $\Theta(N*\log(S))$ time. Ignoring this improvement, the cost is

$$C_6 = 2*B*N\ (\log(N) - \log(M) + \log(B))$$

the same as the sorting cost in C_3. Note that this sort, like in algorithm 3, can be internal if the number of edges is small enough. For small N it only reads the edge file twice while algorithm 1 reads it S times. For any value of N, algorithm 3 reads the file as much as algorithm 6 to sort the edges and then reads it $\frac{(S+1)}{2}$ times more to do the plot. So algorithm 3 is uniformly worse than algorithm 6.

Thus algorithm 6 is the best in this class of algorithms that iterate over the edges. For very large N, algorithm 1, which reads the edge file S times will eventually be better than algorithm 6 which reads it $\Theta(N*\log(N))$ times. For the sample values, the changeover is above N = 5,000,000. How far above depends on statistics of the external sorting algorithm which are too difficult to calculate, among other reasons, because only an incomplete sort is being done. Thus for all reasonable plots, algorithm 6 is best, especially since at the breakover point there are so many edges that the plot is solid black.

6.3.2 Iteration On The Scan Lines

6.3.2.1 Algorithm 7 -

Assume that the scan lines are horizontal and run from $y=\frac{1}{P}$ at the bottom in increments of $\frac{1}{P}$ to y=1 at the top. Define an active edge relative to a given scan line to be an edge crossing the scan line. Let

V = average # of active edges

= # edges * component of average edge´s length perpendicular to scan line

= N*Q

D = # raster bits to set

= L*P

Sample: V = 64

D = 200,000.

If there are few enough edges that they can all be in core at once then store the edges sorted by their lower Y coordinate in a linked list in core. Label the sorted edges E_1, E_2, ... Let the lower Y coordinate of E_i be l_i and the higher be h_i. For the current scan line at any time, if it is #k, its equation is $y=\frac{k}{P}$. Allocate it an array of P bits, initially zero. Compare it against edges E_1 to E_i where i is the smallest integer such that $l_i > \frac{k}{P}$. No further edges can intersect the scan line. For E_1 to E_i check where they intersect the scan line and set the appropriate bit. Further, if for any E_j, $1<=j<=i$, $h_j <= \frac{k}{P}$, then delete E_j by linking together E_{j-1} and E_{j+1}.

6.3.2.2 Analysis -

This algorithm is useful because it only compares a scan line against the line´s active edges. An edge doesn´t start to be tested until the current scan line passes through it and it is deleted as soon as the current scan line rises above it. The problem of finding the intersection of the scan line and the edge can be speeded by storing with each active edge both where the last scan line intersected it and its slope. Then the next intersection can be obtained by 1 addition. This algorithm is similar to Watkins′ hidden surface algorithm, [1970] which also compares edges against scan lines.

Because of the link fields, the intersections and slopes stored with each edge, far fewer edges can be stored in core. Once external sorting is needed, the algorithm runs much slower because it is impossible to effectively delete an edge by linking the edge before to the edge after. Since the deleted edge is still physically in the file, it still must be read which is most of the cost. While the file could be randomly accessed to avoid this, the algorithm would nevertheless run slowly since the input could no longer be buffered.

6.3.3 Sorting The Raster Bits

6.3.3.1 Algorithm 8 -

Scan through the edges calculating all raster bits they set. Write this list of pixel numbers to a file and sort them. Then the actual scan lines can be formed by reading the file once.

6.3.3.2 Analysis -

The edges will cause to be set

$$\begin{aligned} N &= L*P \text{ raster bits} \\ &= 200{,}000. \end{aligned}$$

u = # raster bits that can be stored in core at any time

$$\begin{aligned} &= \frac{M}{2 \log(P)} \\ &= 45{,}454 \end{aligned}$$

$$\begin{aligned} d &= \text{cost of I/O for 1 raster bit} \\ &= \text{\# data bits to describe 1 raster bit} \\ &= 2 \log(P) \\ &= 22 \end{aligned}$$

$$\begin{aligned} C &= \text{cost of this sort} \\ &= dN*(\log(N) - \log(u) -.7) \\ &= 3{,}500{,}000 \end{aligned}$$

i.e. the cost of I/O for 95K words. Letting only L, the total length, vary,

$$C = 44000\ L\ (\log(L) - 3.8)$$

However the above results are for randomly ordered raster bits. Since the average edge length is 20 raster bits, presorting the edges before splitting them into raster bits will reduce the sorting cost greatly. However once the edges are presorted, this method becomes similar to the others, only slower because the others treat the edges as edges and don´t split them up. When the edges are split into pixels, the coherence information is lost. This algorithm is worthless but it is surprising how little I/O it takes. Evidently the coherence information is not as valuable as we might have thought. The idea of writing and sorting individual bits might be useful somewhere else, perhaps for objects that rasterize into very complicated bit patterns.

6.4 PLOTTING CURVED LINES

This case is impossible to analyze without having more statistics on the likely plots but algorithm 6, dividing the screen into strips and cutting the edges where they cross the boundaries and sorting the pieces, should work efficiently, provided we can intersect the boundary lines between the strips against the curved edges quickly. This intersection is simple for such common curves as conic sections and cubic splines. Curves that are too complicated to split can be stored with every strip they pass through. Then when the raster bits are being calculated for some strip, any bits outside that strip can be ignored. This takes more time to calculate the bits but saves time splitting the curves. As usual, the breakeven point depends on various constants of the implementation such as instruction times. If as before the edges are short, not many will cross and for those that don´t cross a simple bounds check will suffice.

6.5 PLOTTING SHADED REGIONS

6.5.1 Shading By Crosshatch Lines

It is rather a waste to crosshatch shade regions on a raster plotter when complete halftone facilities are available but as this will happen when vector plotter designed plots are converted, it should be considered. The difference between a normal plot and one with a lot of crosshatching is that in the latter there are many more edges and they are longer. As for how the time depends on the number of edges, N, algorithms 1-3 are $\Theta(N)$ and algorithm 6 is $\Theta(N*\log(N))$. The edge length doesn't affect the algorithms until a significant number of the edges cross a strip boundary, in which case all the algorithms run slower. The exact amount is impossible to calculate since it is totally dependent on the type of plot; e.g. bar graph, geographic map, etc. It might be desirable for a sophisticated algorithm to make one pass through the edge file only to gather statistics which it would use to fine-tune itself. In the limit where the plot is totally covered with adjacent rasters, and the edges are perpendicular to the strips (which is the worst case), every edge crosses 4 strips in the sample case. Then algorithm 1 reads the edge file the same number of times while algorithm 6 reads it $4*\log(4) = 5.5$ times as much. However, this just moves the crossover point down to somewhere above N = 140,000 which is still a large plot.

Algorithm 7 can be applied to edges that are crosshatch lines as well as to normal edges. Algorithm 8 will run very much slower for polygons since there are so many more pixels to write and sort. Nevertheless, for sufficiently complex regions, this method might have a place.

6.5.2 Raster Shading

Raster shading means to store the polygon per se and to calculate which bits to set when the strip buffer of raster line is being calculated. That is instead of calculating the shading lines and using them as the entities in place of the polygons. In this case, this same algorithms as before are optimal. If the regions are small not many will be cut by the strip boundaries, while if the regions are big enough that many do cross, unless they are long and thin there cannot be very many of them so any method would work. In algorithm 7, the polygons can be handled similarly to the edges. The polygons can be sorted by minimum Y value, and so on. This method strongly resembles Watkins hidden surface algorithm [1970].

When a region is being shaded on a strip in memory, any shading algorithm - halftone, crosshatch, repeated symbols, or whatever, can be used since the buffer can be filled with any desired pattern of bits.

6.6 SUMMARY

Thus the most powerful method (among those considered) of converting vector plotter commands to a raster plotter is to divide the screen into strips, each strip being the biggest that will fit into memory, read and split the edges where they cross the strip boundaries, sort them by strip number and overwrite them on the strips in memory in turn.

For large complicated plots containing diverse information such as straight lines, curves, and shaded regions, the optimum strategy might well be to use different algorithms on the different parts of the plot and then to combine the results later, possibly by reading the bit patterns resulting from the different parts, ORing them, and plotting the result.

CHAPTER 7

SUMMARY & FUTURE DIRECTIONS

7.1 SUMMARY

In this thesis, we have considered various aspects of the combinatorics of hidden surface algorithms, especially as they apply to object space algorithms. We have seen that object space algorithms are not necessarily as slow as they have been generally considered to be and even for general input scenes can run in time linear in the number of edge intersections, provided that is Omega(N*log(N)). In two special cases considered, algorithms have been demonstrated that run in time Θ(N*log(N)) and $\Theta(N^{5/3}\log(N))$ where N is the number of edges or circles in the input scene. Thus the lack of attention that has been focussed on object space algorithms in the last few years has been unjustified.

In spite of the rapid decrease in the cost of raster display devices, object space algorithms retain some advantages. Principally, they produce output that has meaning. That is, they calculate lists of visible lines and polygons. Instead of just having a set of pixels of certain colours, we have a database containing elements with meanings. Thus the output can be further processed. The only way that raster output can be further processed is to perform picture processing pattern recognition on it.

This distinction between object and image space algorithms is similar to a contest currently being waged in geography between raster techniques and vector techniques for storing cartographic and thematic data. The logical operations "and" and "or" can be performed faster on raster data and the data itself is supplied in raster form in torrents from sources such as the Landsat satellites. However the vector representation is much more compact and more complex operations such as the polygon overlay problem, as shown in White [1977], are easier with the vector form. When viewed one way, the two methods are at opposite ends of a continuum. Raster data has no meaning at all. Thus it is so voluminous that it has to be compressed if it is to be stored in a reasonable space. Compression is the act of finding meaning in the data and using it to code more efficiently. Vector encoding is the ultimate compression that extracts all the meaning from the data.

7.2 FUTURE DIRECTIONS

7.2.1 <u>More Special Cases</u>

There are other important special cases to be considered.

7.2.1.1 Movies -

When a movie is produced, the scene changes very little from frame to frame, yet most of the hidden surface calculations have to be repeated. Some algorithms such as Schumacker´s flight simulation system [1969] take advantage of the fact that the scene remains fixed and that only the viewpoint changes, but this is only one aspect of what might be done. What about small changes to the scene? If the effect on the result is localized, the calculations should be localized, also.

7.2.1.2 Very Large Scenes -

Another special case is when the scene is very large. Here, even an algorithm running in time linear in the size of the input scene would run too slowly, so a hierarchical method is necessary to quickly exclude large classes of obviously invisible data. Some work has been done in this, such as by Clark [1976b], but not much. An efficient algorithm in this class could be used in a project such as the Defense Mapping Agency´s ARTINS. This is an attempt to fairly completely model an area of a few square miles while including topographic data, thematic data, roads, symbols, etc., etc. If hidden surface scenes are ever to be generated in real time, as for instance would be seen during a fly-by by a low altitude fast aircraft, much more efficient algorithms will be needed.

Another problem here is that of automatic scaling. That is, if we are observing a tree from a mile away, we usually don´t want to see every leaf in the plot; a general green blur is acceptable. On the other hand, if we are ten feet away, we want to see all 1Ø,ØØØ leaves. This type of operation of displaying only the required accuracy is currently done better by image space algorithms. However, an object space algorithm with a hierarchical database would be worth investigating. One problem here is that although each individual leaf is invisibly small, the ensemble of all the leaves is quite visible. This effect must be calculated.

This project contains many other related hidden surface problems. For instance it is desired to represent an object such as a church accurately if it is big enough to be visible. Otherwise, it is desired to plot only a cross. How should the choice be made efficiently? Also, assume that the whole scene has a conceptually small change such as a snowfall. This raises all the object tops a few inches, and turns them white and fluffy. Is it necessary to recalculate the whole scene because of this small change?

7.2.1.3 Inhomogeneities Within The Scenes -

Input scenes are not homogeneous, even though they are often treated that way for simplicity. Faces may have texture such as lettering that it is better to store explicitly instead of as a set of even smaller faces. The scene may have a background such as the sky with clouds that is it useful to handle separately. Both these special cases have been handled separately by existing algorithms such as Weiler [1977]; but surely there are others, and it might be possible to develop a classification of these different types of scene components and form a general theory.

7.2.1.4 Computer Aided Design -

The art of designing curved shapes such as machine parts and ship hulls interactively by computer is growing fast. The user is generally not a computer scientist and only wishes to see the results of his efforts displayed as quickly as possible. So fast hidden surface algorithms capable of handling curved surfaces are necessary. It is desirable that the scene have some meaning so that the user can point to a point on the screen with a light pen or tablet and the computer will know what object he is referring to.

7.2.1.5 Summary -

The point is that there has not been published any general systematic analysis of the hidden surface problem from a theoretical viewpoint. The only attempt to compare the different algorithms and to develop a general theory is by Sutherland, Sproull & Schumacker [1974b], and while this gives some previously unknown general principles (the centrality of sorting, and the importance of coherence), surely more can be done. Not much work has been done on the hidden surface algorithms themselves recently since attention has concentrated more on the shading which dominates the hidden surface calculation time by far.

7.3 INTERACTIONS WITH THE REST OF COMPUTER SCIENCE

The hidden surface problem is not clean cut; it blends continuously into much of the rest of computer science.

7.3.1 Computational Geometry

For instance, there surely would be useful cross-fertilization between this and the work of Shamos [1975a], [1975c], [1976a], [1976b], [1977b] on computational geometry*. For example, consider his algorithm to intersect

* There are three different meanings for computational geometry. The meaning in this context is the application of analysis of algorithms and computational complexity techniques to algorithms in geometry such as finding closest points, finding convex hulls, intersecting polyhedra, etc. Classical geometers were satisfied with construction techniques that gave the correct answer and were unconcerned with efficiency, rating a technique by its elegance instead.

two convex polygons in linear time. If this could be extended to convex polyhedra and even better concave polyhedra, it would be directly useful.

7.3.2 Probabilistic Algorithms

Also the work by Rabin [1976] in probabilistic algorithms might have applications. This is because the goal in computer graphics is to produce realistic looking pictures and a great increase in overall realism would probably be acceptable at the cost of occassional minor errors. After all, this the way the human eye and mind operate: Various optimizations and shortcuts are taken that improve overall vision but from time to time cause optical illusions.

7.3.3 Relational Databases

Another bond between computer graphics and the rest of computer science occurs in the field of databases. As was mentioned in Chapter 3, various aspects of the hidden surface problem can be viewed as relational database questions. One of these is that of determining which of a set $SE=\{E_i\}$ of edges intersect. This is equivalent to finding records with common keys in certain fields. The main difference is that here we are manipulating infinite sets of records with a given key (all the points on an edge). However even in certain finite databases with millions of records, it might be useful to perform the analysis under the assumption that there are an infinite number.

7.3.4 Special Hardware

Even though hidden surface algorithms perform large amounts of calculations, these calculations are very regular. Thus they are suited for arrays of processors. Some work, for example by Fuchs [1977b], has been done in patching together the output from various processors, but more can certainly be done. Since processors are getting cheaper, this area will become more important. Of course, there have been special purpose hidden surface calculation machines such as by Evans & Sutherland, and Schumacker, for years.

7.3.5 Summary

Thus there are many promising interdisiplinary relations possible between hidden surface algorithms in computer graphics and the rest of computer science that should keep researchers occupied for years.