Visible Surface Solutions Using Plane Sweep

Brian L. Plante

ABSTRACT

This essay discusses the algorithms to be used for implementing a plane sweep algorithm to solve the visible surface problem in object space. An algorithm is presented based on the work of Nurmi and the MV-trees described by Swart and Ladner. This results in a solution for the visible surface problem that has $O((n+k)\log n)$ time complexity and $O((n+k)\log n)$ space complexity for scenes having n edges and k intersections of edges.

Acknowledgements

I would like to thank my supervisor Kelly Booth for his help and encouragement throughout this work.

I would also like to thank my student reader Lyn Bartram.

The members of the Computer Graphic Laboratory have all given me some help in my time here.

And I would like to thank all of my friends both old and new who helped in ways too numerous to mention.

Introduction

1.1. Visible Surface Problem

The *visible surface problem* is to render the surfaces that are visible in a scene. The solution to this problem is to determine which parts of the scene can be seen from a specified viewport, and must be rendered, and which parts cannot be seen from the viewport, and must be omitted from rendering. The only types of scenes which will be considered are scenes made up of plane polygons that are opaque. Finding the visible surfaces also finds the visible lines, so that the *visible line problem* is solved as a special case.

The algorithms described here assume that appropriate perspective transformations have been applied to all objects in a scene so that all points having the same (x,y) coordinates lie on the same line of sight and thus a simple z–comparison is sufficient to resolve visibility.

1.2. Image space vs. object space

There are two main divisions of visible surface algorithms determined by the precision to which the computation is done. An algorithm can be performed in *image space*, where the result is calculated to (near) the resolution of the device used to display the output. Alternatively, the computations can be performed in *object space*, usually to the precision of the hardware floating point instructions. A comparison of visible surface algorithms is made by Mathies [Mat] and by Sutherland, Sproull and Schumacker [SSS]. Descriptions of standard algorithms and transformations are given in the textbooks by Newman and Sproull [NeS] and Foley and Van Dam [FVD].

Image space algorithms usually only look at the visibility problem for regions of fixed size and shape (typically one pixel). Even when there are visibility changes within these regions, the internal structure is largely ignored. Such algorithms are often implemented as *scanline algorithms*, where the visibility at each pixel is calculated in the same order as the scanline of a raster output device. A typical scanline algorithm is Watkins's algorithm [Wat, Mor]. The output from an image space algorithm is for a fixed size of display; to change the display size typically involves re-solving the visible surface problem.

In object space algorithms the visibility problem is solved exactly. In practice, the result is computed to machine precision and the resulting set of non–overlapping polygons are passed to a renderer. This allows for a solution that can be stored and rendered on different devices of varying resolution without having to re-solve the visibility problem. The object space algorithms have the disadvantage that they look at everything in a scene, even if it will be too small to affect the final picture in any way. For example, a very thin rectangle may only cover a small fraction of a pixel in its smaller dimension, but an object space algorithm will still check the effect it has on the other elements of the scene and break up those objects to reflect the presence of the thin rectangle.

In Figure 1.1 the thin rectangle will contribute an insignificant amount to the pixels it covers, but it will still cause the other elements of the scene to be broken up, since an object space algorithm solves the visible surface problems exactly.

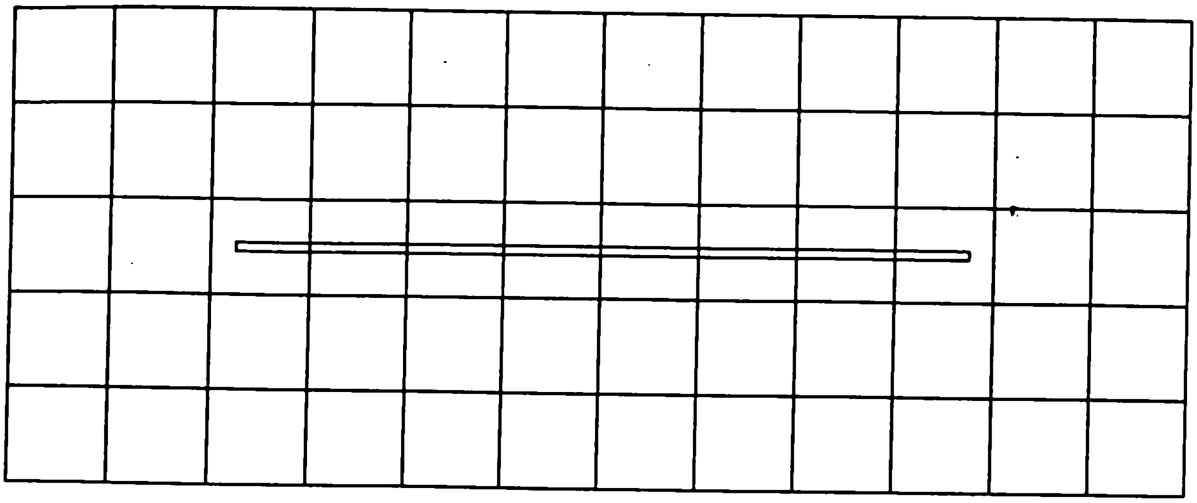


Figure 1.1

1.3. Polygonal Representation

The form of input for a scene is assumed to be a collection of planar polygons. Other surfaces can be approximated by polygons. It is assumed that this has been done prior to solving the visible surface problem if more complicated objects are being rendered.

If polygonal information is maintained properly, texturing and shading can be done by referring to the original polygons, rather than to the individual pieces of the polygon that remain after the visible surface calculation. This involves keeping together all of the pieces and holes of a polygon that may be generated by the visible surface algorithm so that the rendering algorithm can process the complete polygon at one time. The reason for this will be explained in Section 4.3.

Texturing and shading will be accomplished by using a *master polygon*, which is an outline of the complete polygon. All of the pieces and holes of the master polygon (the *subservient polygons*) are maintained in a linked list which has as its head the record for the master polygon. The visible surface algorithm accepts this form of polygonal representation for input as well as for output so that a scene can be modified and have the visible surface problem solved again.

1.4. Scanline algorithms

Scanline algorithms use a fixed increment (a scanline) to step down in the y direction. Any crossings of edges on the scanline are processed for changes in visibility, as are any edges that start or end. Watkins's algorithm is an example of a scanline algorithm used to solve the visible surface problem [Wat]. In Figure 1.2, the places where a scanline algorithm will stop are shown. Each scanline is a fixed distance from the previous scanline, so the algorithm checks for intersections of edges at places where the picture has not changed its visibility since the previous scanline. A pseudo-code version of a typical scanline algorithm is included here as a comparison with the plane sweep algorithms introduced later.

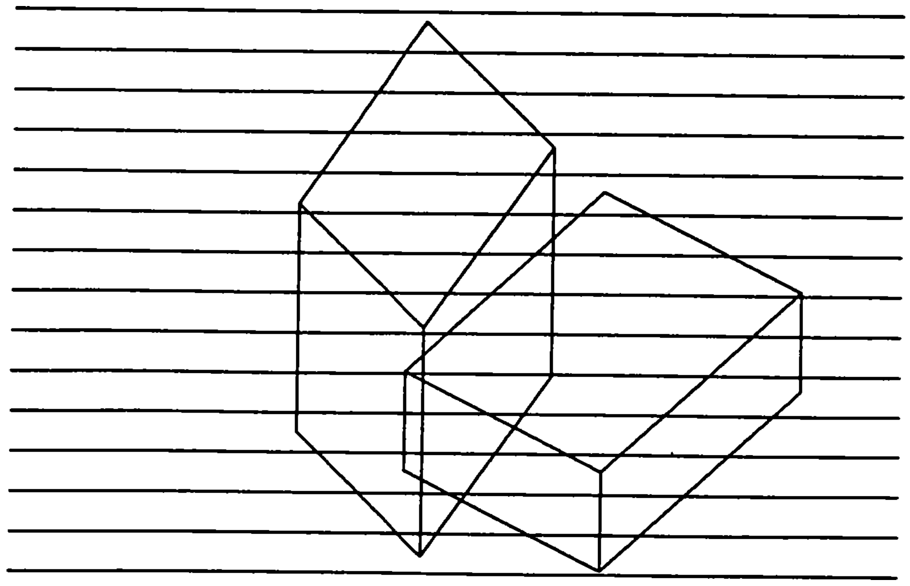


Figure 1.2

1.5. Psuedo–code for scanline algorithm

```
Edge_List ← bucket sort edges by minimum $y$-value
Active_Edges ← NIL
for each scanline do
    for each edge in Edge_List entering in this scanline do
        search Active_Edges and put edge data in segment block
        mark associated polygon of segment block as changing
    end for
    for each polygon that is changing do
        check all segment blocks & pack segment blocks if necessary
    end for
    while not at end of scan line
        get next sample span
        repeat
            compare depths of all segments in the sample span
            update Active_Edges to maintain sort in $x$-order
            mark any polygons exiting on next scanline as changing
        until span does not need subdividing to resolve
        if single intersection in the span then
            process the intersection & update Active_Edges
        endif
        output visible segments
    end while
end for
```

1.6. Plane Sweep Algorithms

The *plane sweep algorithms* are the object space equivalents of scanline algorithms for image space. Instead of using the fixed increment of a single scanline in image space, the sweep line moves by a variable increment to the "interesting" points. These points are the vertices of the polygons, the intersections of edges, and the ends of lines generated by inter–penetrating polygons. Each of these points may cause changes to the visible picture and thus must be examined by a correct visible surface algorithm. The points where a plane sweep will stop to check visibility changes are illustrated in Figure 1.3.

1.5. Psuedo–code for scanline algorithm

```
Edge_List ← bucket sort edges by minimum $y$-value
Active_Edges ← NIL
for each scanline do
        for each edge in Edge_List entering in this scanline do
                search Active_Edges and put edge data in segment block
                mark associated polygon of segment block as changing
        end for
        for each polygon that is changing do
                check all segment blocks & pack segment blocks if necessary
        end for
        while not at end of scan line
                get next sample span
                repeat
                        compare depths of all segments in the sample span
                        update Active_Edges to maintain sort in $x$-order
                        mark any polygons exiting on next scanline as changing
                until span does not need subdividing to resolve
                if single intersection in the span then
                        process the intersection & update Active_Edges
                endif
                output visible segments
        end while
end for
```

1.6. Plane Sweep Algorithms

The *plane sweep algorithms* are the object space equivalents of scanline algorithms for image space. Instead of using the fixed increment of a single scanline in image space, the sweep line moves by a variable increment to the "interesting" points. These points are the vertices of the polygons, the intersections of edges, and the ends of lines generated by inter–penetrating polygons. Each of these points may cause changes to the visible picture and thus must be examined by a correct visible surface algorithm. The points where a plane sweep will stop to check visibility changes are illustrated in Figure 1.3.

An efficient plane sweep algorithm attempts to examine only these points. One problem that a standard plane sweep does not address is finding the inter–penetrations of polygons. Since the edges of inter–penetrating polygons do not necessarily intersect on the projection plane, the plane sweep must be augmented to find these inter–penetrations.

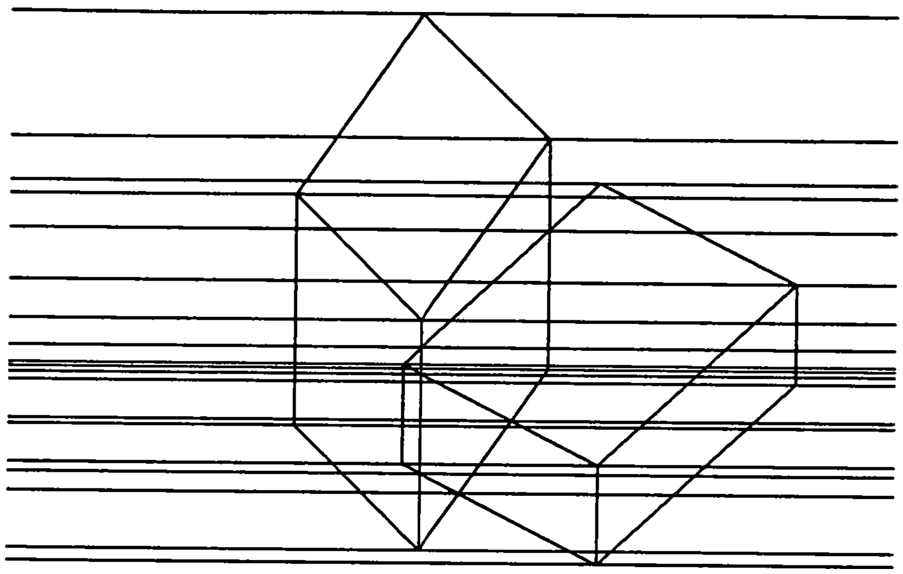


Figure 1.3

The Plane Sweep Algorithm

2.1. Finding Intersections

A plane sweep algorithm was first applied to the problem of finding the intersections of line segments in the plane by Shamos and Huey [ShH]. Their algorithm solved the problem of whether any two line segments on a plane intersected. Their algorithm ran in $O(n \log n)$ time and used $O(n)$ space for scenes with n edges.

The Shamos–Huey algorithm was generalized to find all k intersections of a collection of line segments by Bentley and Ottmann [BeO]. Their modified algorithm runs in $O((n+k) \log n)$ time and used $O(n+k)$ space. The key difference with the Bentley–Ottmann algorithm is that the ordering of line segments as they intersect the sweep line is updated at intersection points as well as at end points. By switching two line segments found to intersect, the ordering of line segments intersecting the sweep line can be maintained so that all of the intersections are reported.

Bentley and Ottmann's algorithm works by sweeping a horizontal *sweep line* through the plane from top to bottom. A priority queue of events is used to determine the next y-value at which to stop. This priority queue initially contains the end points of all of the line segments and is ordered by the y-value of the end points. As the sweep line progresses downward the priority queue contains both end points and pending intersections of line segments. The priority queue is typically implemented as a heap in order to allow for $O(log\ n)$ insertion and deletemin operations (see [AHU] for further discussion of priority queues).

Another structure, the *active edge list*, is used to maintain the line segments which are currently cut by the horizontal sweep line. The active edge list is ordered by x-value on the sweep line. This structure is usually kept using a balanced tree structure, such as AVL trees (see [AHU]), so that insertion and deletion can be done in $O(log\ n)$ time. This also ensures that finding the predecessor and successor can be done on $O(log\ n)$ time. Often extra pointers are used to link the entries so that the latter two operations can be done in constant time.

Brown improved the space bound to $O(n)$ by observing that only the next intersection for a line segment must be kept in the priority queue [Bro]. This is because intersections of line segments not currently adjacent in the sweep line order will be found when the line segments are adjacent, which must occur before the intersection point.

Both the Shamos–Huey and Bentley–Ottmann algorithms rely on the fact that for an intersection to occur, the two intersecting line segments must first become adjacent in the active edge list. In order to find all of the intersections of the line segments, only line segments that are adjacent in the active edge list must be tested for intersection. The only times that the adjacency can change is when a new line segment starts, an old line segments ends, or an intersection of two line segments occurs on the sweep line. These are the only times the active edge list must be updated. On each update at most two tests for new intersections need be made involving the line segments whose status has changed.

When an intersection is found, it is added to the priority queue according to its y-value so that the updating of the active edge list will be done at an appropriate time when the sweep line reaches that y-value.

At each of the events in the priority queue, testing for intersections between the adjacent line segments must be done according to the type of entry.

When a line segment **E** starts, it is inserted into the active edge list. The line segment which precedes **E** and the line segment **E** are tested to determine if they intersect. If an intersection is found it is added to the priority queue of events according to its y-value. A similar test is performed between **E** and the line segment that succeeds **E** in the active edge list.

When a line segment **E** ends, it is deleted from the active edge list. The line segment that preceded **E** and the line segment that succeeded **E** are tested to determine if they intersect. If an intersection point is found it is added to the priority queue of events according to its y-value.

When an intersection point of two line segments **E1** and **E2** is reached, the line segments must be switched in the active edge list. Assuming that **E1** preceded **E2** before the intersection, **E1** will succeed **E2** after the intersection. **E1** must be tested with the line segment which succeeds it after the switch, and **E2** must be tested with the line segment which precedes it after the switch. Any intersections found are added to the priority queue of events according to its y-value.

When an intersection is added to the priority queue, duplicate points must be rejected. If these points are not rejected, then a test must be made when the priority queue is processed so that the same intersection is not reported multiple times. In Figure 2.1, we see that line segment A is inserted into the active edge list, then line segment B is inserted. At this time the intersection point between A and B is found. When line segment C is deleted, A and B are again adjacent, so the intersection point is recomputed. If the insertion into the priority queue does not check for duplicates, there will be wasted space maintaining the duplicate entries.

The places where a sweep line will stop are shown in Figure 1.3. Unlike an image space algorithm, where the scan lines are a fixed distance apart, the plane sweep increments are variable, with a higher concentration in the areas where events are happening.

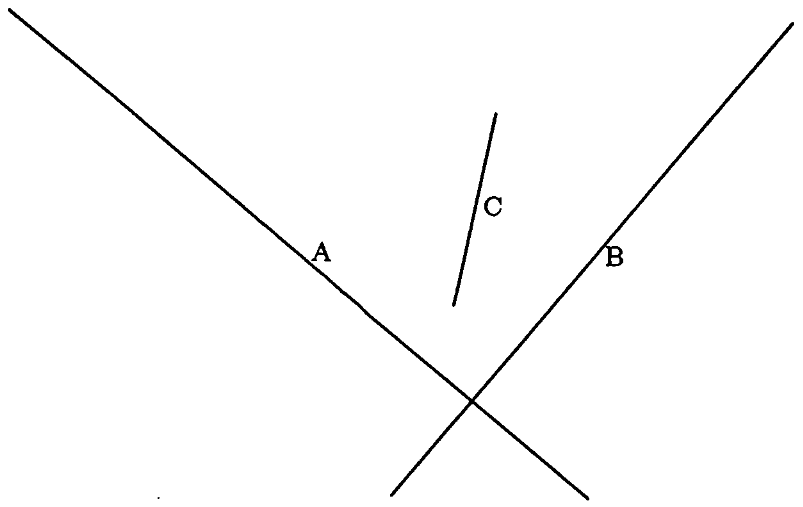


Figure 2.1

2.1.1. Psuedo-code for the Bentley–Ottmann algorithm

```
Initialize a priority queue Q with the end points sorted in $x$.
Initialize the active edge list R to be empty.
While the priority queue is not empty do
        SP ← deletemin(Q)
        if SP is a upper vertex of a line segment E
                insert E into R
                if E intersects the previous or next line segment in R then
                        add the intersection point or points to Q
        else if SP is a lower vertex of an line segment E
                if the previous and next line segment of E intersect then
                        add the intersection point to Q
                delete E from R
        else /* SP is the intersection of E1 and E2  */
                output the intersection of E1 and E2 at SP
                switch E1 and E2 in R
                if the left segment and the line segment before it in R intersect
                        add the intersection to Q
                if the right segment and the line segment after it in R intersect
                        add the intersection to Q
        endif
endwhile
```

2.2. Using plane sweep for visible surface problems

A scene is described by a collection of polygons, which may contain holes or which may consist of more than one piece. These polygons are projected onto a viewing plane, usually using a perspective transformation (see [FVD] for a discussion on perspective transformations). If polygons do not inter–penetrate, visibility only changes at the vertices of the polygons making up the scene and at the intersection points of edges of the polygons as they project onto the viewing plane. This is the same coherence property that is used in scanline algorithms to eliminate z-depth tests at most pixels on a scanline.

In order to determine the changes in visibility at each of the sweep points, depth and polygon information must be kept in addition to the active edge list. The polygonal information is kept implicitly in the active edge list, since the edges of the polygon that intersect the sweep line are all in the active edge list. This is accomplished by keeping track of which polygon an edge belongs to. Consecutive edges in the active edge list correspond to an interval of the sweep line that one polygon currently *covers*. A polygon covers an interval when it is the visible polygon for that interval.

Unfortunately, the covering polygon need not contain either of the two edges for an interval, but may be a completely unrelated polygon that is "uncovered" as the sweep line progresses down the plane. For each interval of the sweep line covered by a polygon, depth information must be kept for visibility testing. The method of storing the active edge list and the depth information can alter the time and space requirements of an algorithm. Various plane sweep algorithms will be discussed, including the worst case running times for each.

2.2.1. Work of Sechrest and Greenberg

One of the earliest uses of a plane sweep to solve the visible surface problem was by Sechrest and Greenberg [SeG]. In their algorithm, the active edge list was implemented as a linear linked list, rather than as a balanced tree structure. The depth information was also implemented as a linked list, so that the algorithm appears to have a running time of $O((n+k)\ n^2)$. For worst case processing, the use of linear linked lists is not appropriate, although for many practical scenes they are acceptable.

One major part of Sechrest and Greenburg's algorithm is the reconstruction of the visible surfaces in a manner which keeps all of the pieces and holes of the polygons together so that subsequent rendering of the scene can be done on all of the parts of the polygon. The advantages of this approach will be discussed in Section 4.3.

2.2.2. Work of Ottmann and Widmayer

Ottmann and Widmayer present a solution to the visible line problem that extends a solution for iso–oriented rectangles to non iso–oriented polygons [OWi]. A set of *iso—oriented rectangles* is a set of rectangles whose sides are parallel to orthogonal (vertical and horizontal) coordinate axes. The algorithm proceeds by creating the notion of a *zigzag* as an analogue of the side to a rectangle. This is necessary because the sides of rectangles allow the active edge list to be kept efficiently in a *segment tree* when a plane sweep is used for iso–oriented rectangles. Zigzags provide a similar order relation for the edges of polygons, and are used to define a semi–dynamic segment tree to maintain the active edges. The intervals stored at the internal nodes of the segment tree are used to store the polygonal depth information necessary for visibility tests. The active edge list for the plane sweep is also represented by this structure, as the zigzag edges define the ends of intervals of a polygon.

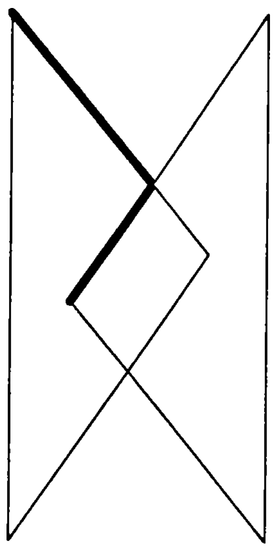


Figure 2.2a

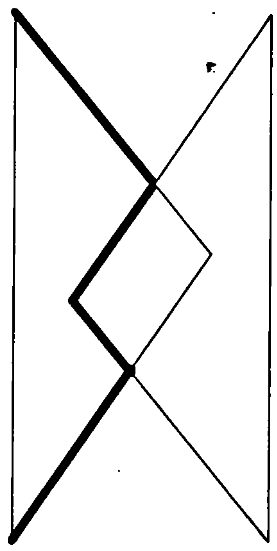


Figure 2.2b

The zigzag of an edge *s* is the polygonal line starting at the top end point of *s* that follows *s* until another edge *t* is intersected (see Figure 2.2a). After the intersection, the zigzag follows *t* in a "downward" direction until *t* intersects another edge. This continues until the bottom end point of a segment is met. This defines a one–to–one relationship between segments and zigzags. No two different points on a zigzag have the same *y* value and two distinct zigzags never cross each other, although zigzags do have intersection points in common.

A zigzag *za* is to the *left* of a zigzag *zb* iff *za* and *zb* have points *pa* and *pb* with a common *y*-value, such that the *x*-value of *pa* is less than the *x*-value of *pb*. The transitive closure of *left* will be denoted as $left^{+}$. If two zigzags do not have any *y*-values in common then they are not related by $left^{+}$. For all pairs of zigzags *za* and *zb* having no points with the same *y*-value, *za* is *above* *zb* iff all *y*-values of points of *za* are greater than all *y*-values of *zb*. A total order $left^{+} \mid above$ can now be defined between any two zigzags *za* and *zb*. If *za* and *zb* are related in $left^+$, then they are related in $left^+ \mid above$ in the same way, otherwise they are related in $left^+ \mid above$ in the same way as they are related in *above*.

The input to a scene is made up of polygons, so zigzags are modified to continue on from the bottom of an edge to the upper point of the next edge of the polygon sharing the vertex (See Figure 2.2b). In a scene of p convex polygons, there will be $2p$ modified zigzags. If the input polygons are not convex, there may be $O(n)$ modified zigzags. This defines an ordering between edges on the sweep line, by using the ordering of the zigzag that the edge is currently part of. This ordering can be computed in $O((n+k) \log n)$ time and $O(n)$ space using a plane sweep.

A segment tree is then built from the m zigzags found in a scene, and each zigzag is associated with the top-most edge in the zigzag. In order to use the segment tree, we define a dynamic *rank* array which is indexed by the n edges. For an edge s the array is defined as:

$$rank(s) = \begin{cases} i, & \text{if } s \text{ is on the } i\text{-th zigzag in the current order.} \\ undefined, & \text{otherwise} \end{cases}$$

As well, an array *current* is maintained, indexed by the ranks, which contains the edge having a given rank. These two arrays are updated as the sweep line progresses down the scene. A segment tree is a minimal-height binary tree with m leaves labelled from 1 to m representing the m ranks. Insertion or deletion of an interval $[a,b]$ is accomplished by updating the node-lists of each node in the segment tree to "cover" the interval.

This allows the visibility problem to be solved in $O(n \log^2 n + k \log n)$ time and $O(n \log n)$ space.

2.2.3. Work of Ottmann, Widmayer and Wood

Ottmann, Widmayer and Wood give an algorithm which eliminates the need for a prescan of the input to define an order relation. This is done using a fully dynamic segment tree instead of a semi–dynamic structure [OWW]. The dynamic segment tree is implemented by using a tree of bounded balance (weight balanced) instead of an optimal binary tree as in the semi–dynamic structure. For a discussion of trees of bounded balance, see Nievergelt and Reingold [NiR]. Using a dynamic segment tree, we can then use the x ordering of the active edges on the horizontal sweep line, which eliminates the need for a prescan to determine the order relation. At each node of the segment tree, the depth information of the polygons covering that interval is kept in an AVL tree, ordered by z-depth.

This algorithm has $O(n \log^2 n + k \log n)$ time and $O(n \log n)$ space complexity in the worst case, the same as the previous algorithm.

2.2.4. Work of Nurmi

An $O((n+k) \log n)$ time and space algorithm for the visible line problem is given in Nurmi [Nur]. His algorithm is based on the algorithm of Ottmann, Widmayer and Wood, but uses a more efficient data structure introduced by Swart and Ladner. This data structure accounts for the time improvement in Nurmi's algorithm, at the expense of increased space usage. Nurmi's algorithm will be discussed in detail in Chapter 3.

2.2.5. Work of Schmitt

An outline of a visible surface algorithm using a plane sweep is contained in Schmitt [Sch]. His algorithm handles polygons made up of the edges bounding them and is composed of four phases. The first phase builds a *connection graph* consisting of all of the endpoints of the line segments making up the polygons, all intersections points of these line segments, and *drain points.* The drain points are introduced in order to eliminate all left corners that have no connection to objects left of it. They are used to ensure that the connection graph is connected, and to clarify inclusion relations during transversal. Figure 2.3 illustrates a simple scene, its drain points and the corresponding connection graph.

The connection graph is a graph where the vertices are all points in the plane which are end points of line segments, all intersection points of line segments, and all drain points. Two vertices in the connection graph are connected by an edge iff the two vertices are neighbours on a specific line segment or a vertex needs a connection with another line segment via a drain point (a *drain edge*).

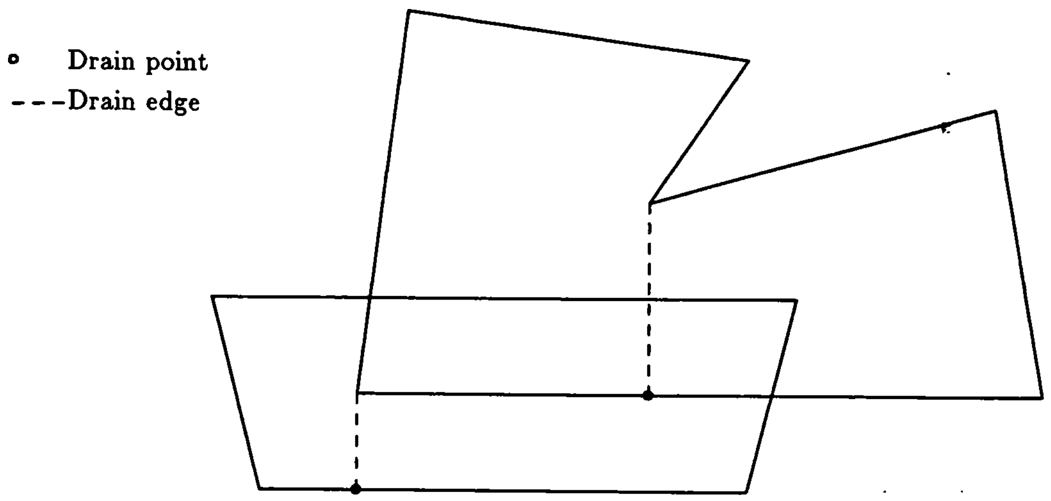
The second phase of Schmitt's algorithm traverses the connection graph and determines the inclusion status of the edges. The inclusion status of an edge e is defined by

$R_0(e)$ = set of polygons that contain e

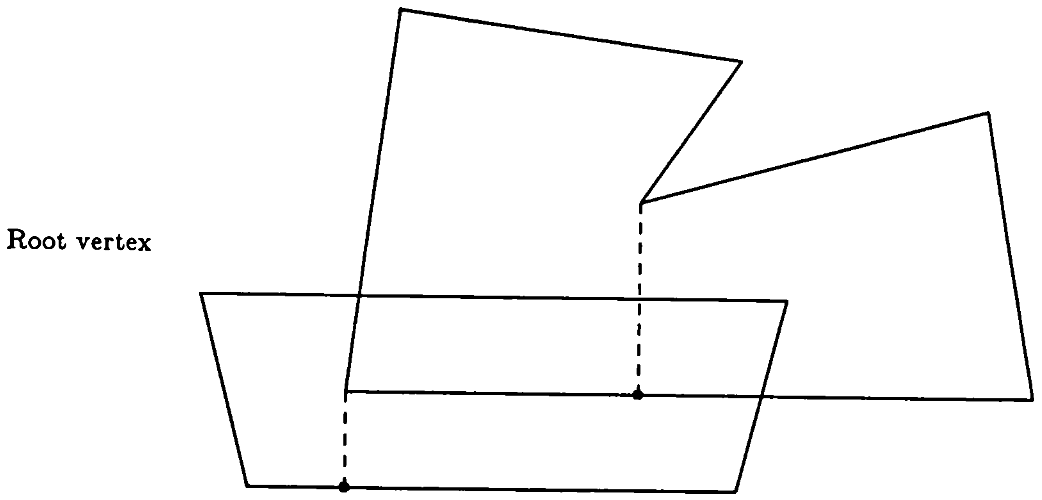
$R_{-1}(e)$ = set of polygons that e is on the lower boundary

$R_{+1}(e)$ = set of polygons for which e is on the upper boundary

The space requirement to store this information is $O(n)$. The graph can be transversed in $O(n+k)$ time.



Scene



Connection graph

Figure 2.3

All of this information is used by the third phase of Schmitt's algorithm as it determines the visible edges. During the transversal of the connection graph in the second phase, the visibility of the edges is determined from the inclusion status of the edges. This is done by checking for changes to the inclusion status that affect an edge as the graph is being transversed. If the edge is visible, it is kept for use by the fourth phase.

The fourth phase examines the visible edges of the connection graph and extracts the visible surfaces. This stage uses information about which surfaces the edges belong to on each side of the visible edge. The visibility of these polygons determines the surfaces which must be reported, since one or both polygons sharing the edge may be visible.

This algorithm is claimed to run in $O(r + (n+k) \log n)$ time, where r is the number of pairs (polygon and line segment) having the line segment completely within the interior of the polygon, n is the number of line segments, and k is the number of intersections of line segments on the projection plane. The space required is $O(n+k)$.

Schmitt's algorithm was not chosen for implementation due to the complexity of manipulating the connection graph and extracting the visible surfaces from it, as well as the high space requirement to store the connection graph.

2.2.6. Work of Devai

Using *line arrangements*, Devai has developed an algorithm which is $O(n^2)$ [Dev]. The first step of the algorithm is to extend all of the edges of polygons to infinite length lines. The graph **G** corresponding to the plane subdivisions induced by these lines is then computed. In addition to the intersections at finite points, it is assumed that all half-lines of the subdivision meet at a single point at infinity. **G** is computed iteratively, by adding each line in turn to the graph and then finding the intersection points of the new line with each of the lines already in the graph.

A list of the intersections on each line is kept in sorted order. This is used to determine the visible portions of each line. All of the polygons which intersect each line are put onto a list for each line. Then for each polygon intersecting a line, the visibility of each portion of the line is calculated.

This algorithm runs in time $O(n^2)$ and requires $O(n^2)$ space. It does not depend on the number of intersections present in the scene, since the lines are extended to infinity, and all of the intersections are found. Since the term of k in the previous algorithms can be $\Omega(n^2)$, those algorithms can require $O(n^2 \log n)$ time for the worst scenes, which is more than required by Devai's algorithm. For less complex pictures where the number of intersections is less then $\Omega(n / \log n)$, the previous algorithms will require less work than Devai's algorithm.

2.2.7. Work of McKenna

McKenna has shown that visible surface processing can be accomplished in $O(n^2)$ time and space [McK]. As with Devai, line arrangements are used to build a graph of intersections with all lines extended to infinity. Instead of processing each line separately, the second stage of McKenna's algorithm sweeps a "bendable" topologically vertical line from left to right through the arrangement. A vertically ordered list of the $n+1$ current regions crossed by the sweep line is maintained. This corresponds to the active edge list of a regular plane sweep. For each region the boundary segments above and below the region are kept. When boundaries meet at the right endpoint of the region they are added to an unordered queue of completable regions.

For each of the current regions an ordered doubly linked list is maintained of the polygons that contain the region, ordered by their z-value so that the closest face is the first element of the list. The heads of the lists are kept in an array indexed by faces and current regions, enabling the list to be accessed in constant time. The frontmost face for each region is then kept for processing by the next stage of the algorithm.

These frontmost faces are then used to determine visibility by removing all of the arrangement segments whose incident regions have common frontmost faces. The corresponding vertices are also removed. This leaves a tiling of the plane by the visible polygons.

This algorithm runs in $O(n^2)$ time and uses $O(n^2)$ space to solve the visible surface problem.

2.3. Worst case results

Devai has shown that the visible line problem is $\theta(n^2)$, by giving an $O(n^2)$ algorithm and an example requiring $\Omega(n^2)$ steps to determine the visible lines [Dev]. Further work by McKenna has shown that the visible surface problem is also $\theta(n^2)$ [McK]. It had been conjectured that the visible surface problem was more difficult than the visible line problem. Schmitt [Sch] conjectured that the visible surface problem would take $\Omega(n^2 \log n)$ time to solve in the worst case.

It is an open question whether these results can be improved upon so that the order statistics include k. This would result in an algorithm which would still be $O(n^2)$ in the worst case, but it would run faster when there are few intersections between edges. Since it would take $O(n \log n)$ to determine that there are no intersections between edges, it would be necessary to have a term of $n \log n$ in the running time. Therefore an "optimal" algorithm might have a running time of $O(n \log n + k)$.

The next chapter discusses the algorithm of Nurmi in more detail. It was chosen for implementation because the data structures used in the algorithm are conceptually simple compared to the other algorithms. The time bound is the best for the algorithms which are intersection sensitive. It was felt that for practical scenes it would run faster than McKenna's algorithm.

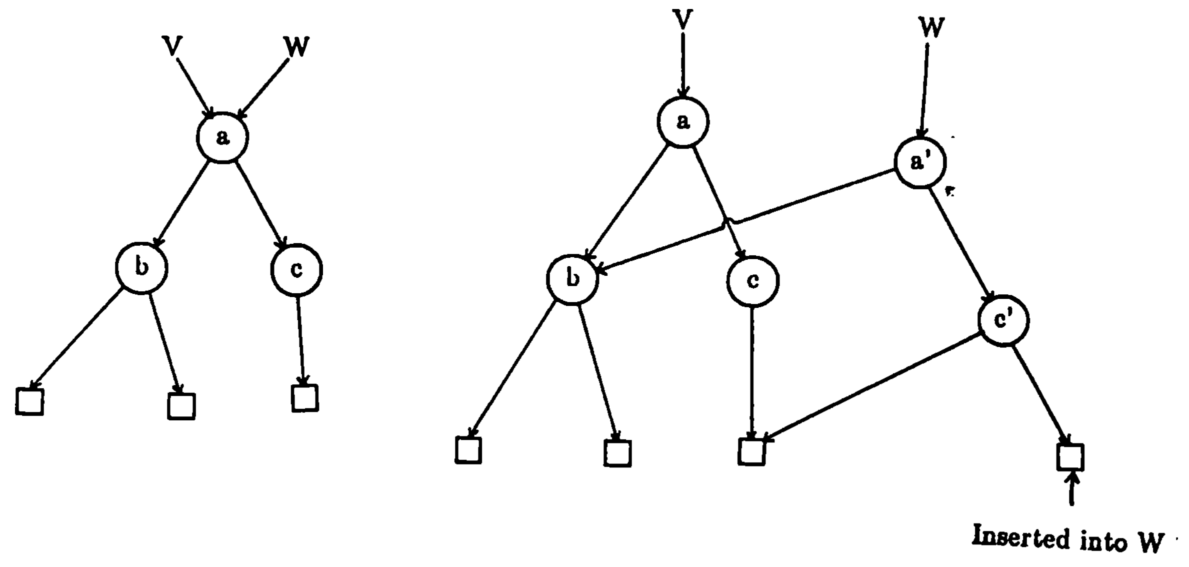
Algorithm Description

3.1. MV–trees

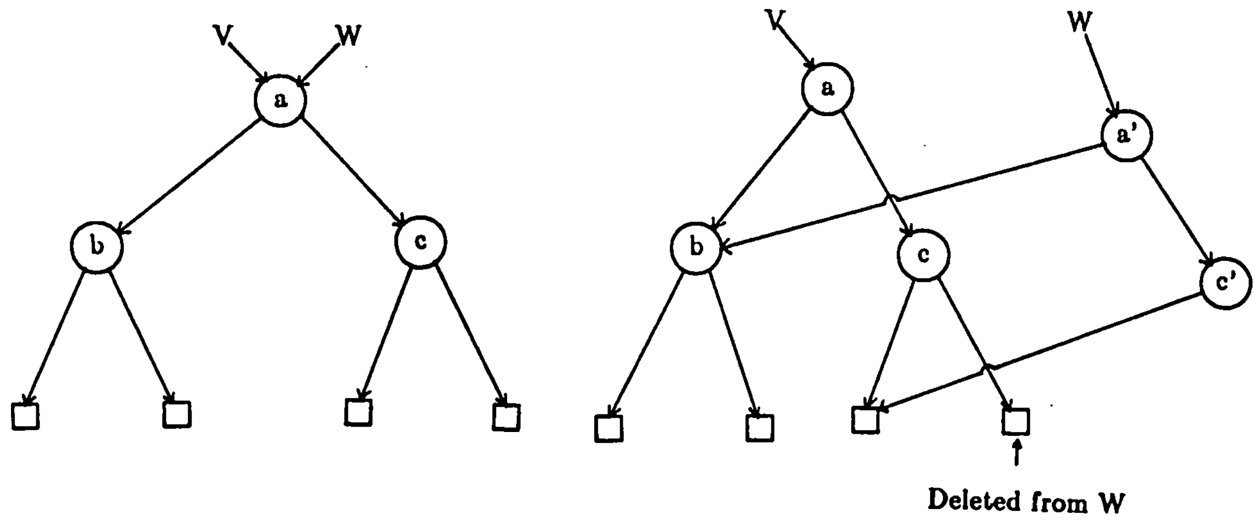
MV–trees were introduced in Swartz and Ladner [SwL], to allow trees to be copied in $O(1)$ time and to allow insertion, deletion and searches to be done in $O(\log n)$ time. These are based on balanced binary trees that rely on balancing actions on the path of insertion or deletion. Examples of algorithms which maintain balanced trees in this manner are dichromatic trees, 2-3 trees, and AVL-trees [AHU].

In order to accomplish constant time copying of trees, only the pointer to the root of the tree is copied. On insertion of a node, each node on the path to the new node is copied, so that only the one tree is affected. Similarly, on deletion of a node, the nodes on the path to the deleted node are copied, so again only the one tree is affected. Usage counts for the nodes are maintained to reclaim storage which is no longer referenced by any of the trees. Since the standard insertion and deletion algorithms were originally $O(\log n)$, and we have only added a constant amount of work to be done at each step, the new tree algorithms remain $O(\log n)$. For each insertion and deletion there will be $O(\log n)$ additional space used compared to the original tree. The additional space is less than copying the entire tree for a copy operation, but the total space used by MV–trees is currently an open problem. The changes in an MV–tree for insertion and deletion are shown in Figure 3.1.

MV–trees are used to maintain the polygonal depth information for each of the intervals in the active edge structure. This allows the algorithm to simply copy a pointer in order to copy the currently active polygons for the interval it is being inserted into, without copying the entire tree of polygons. Insertion and



Insertion in an MV–tree



Deletion in an MV–tree

Figure 3.1

deletion of polygons in this structure can be done in logarithmic time. At each step there are at most two of these operations to be performed.

3.2. Nurmi's Algorithm

The input to the algorithm is a set of clipped polygons, which have been projected onto a plane. The projection will normally be done by using a perspective transformation, although an orthographic projection could also be used. 3-D clipping in normalized coordinates ensures that there will only be positive z-coordinates, ranging from 0 to 1. The input polygons may be concave, may have holes, and may even be non–connected, so that the output polygons of the algorithm could serve as input. This was done so that small changes in a scene could be made without re–solving the visibility of the entire scene, but only re–solving the portions of the scene which have changed.

The output of the algorithm will be a set of non–overlapping polygons which completely "tile" the clipping window. These polygons will maintain information on the original "master" polygon they are derived from, so that each polygon may be processed in its entirety for shading and texturing purposes.

A plane sweep algorithm is used to determine the intersections of the edges of the polygons. The active edge structure is implemented as an AVL tree and the nodes of the structure are used both for determining the intersection points and for maintaining the intervals of the sweep line. The operations of search, insert and delete can all be performed in $O(\log n)$ time for AVL trees.

For each interval of the sweep line, a structure is kept with the faces that cover the interval. These structures are implemented as MV–trees, using a base tree type of AVL trees; these will be referred to as *node—lists*. The ordering of faces is according to z-depth. The operations of search, insert and delete can be performed in $O(log\ n)$ time as with normal AVL trees. Since these are

MV–trees a copy operation will take $O(1)$ time.

The sweep line stops at each vertex of the input polygons and at any intersections of edges found during the sweep. At each of these points, the data structures are updated and visibility tests are performed if necessary. The update operations maintain the data structures to ensure that the node–list of each active edge contains the face which covers the interval from that edge to the next edge on its right side along the sweep line. As well the visible portions of the polygons in the scene are being updated as the sweep progresses. By definition, the right–most edge's node–list is empty.

Shadow edges are used as duplicate edges for the visible edges. This is done to enable quick testing of whether an edge is visible and to keep each edge in only one polygon. A shadow edge are always oriented in the opposite direction from its associated edge. This ensures that any visible polygons constructed will have the correct orientation. A polygon that has its edges in clockwise order is visible in the interior and a polygon that has its edges in counter–clockwise order defines a hole.

The update operations and visibility tests to be done at each point vary depending on the type of point encountered. The operators succ() and prev() give the successive and previous active edge on the sweep line, for an edge. In each case below edge e1 preceded e2 in the active edge list before the vertex was processed.

Case 1: *Top vertex* - insert the two edges (e1, e2) from the vertex into the active edge structure. The node–list for the first edge is a copy of the node–list of prev(e1) with the polygon of e1 either deleted or added from the list, depending on whether e1 is part of a hole or not. The node–list for e2 is the same as the node–list for prev(e1). Check for an intersection of e1 and prev(e1) and add it to the priority queue if it exists. Check for an intersection of e2 and

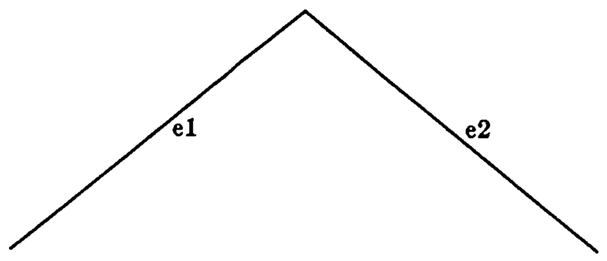


Figure 3.2

succ(e2) and add it to the priority queue if it exists. If the polygon of these edges is at the top of the node–list of e1, start a new visible polygon and create a shadow edge for the previously visible polygon.

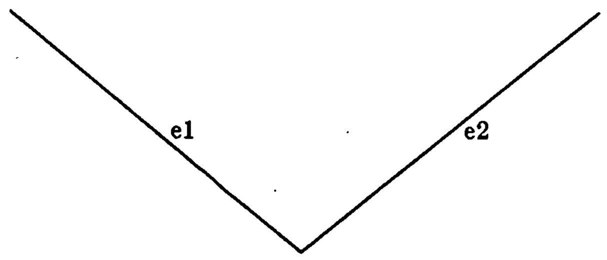


Figure 3.3

Case 2: *Bottom vertex* - delete the two edges (e1,e2) from the active edge structure. Check for an intersection between prev(e1) and succ(e2) and add it to the priority queue if it exists. If shadow edges are present for e1 and e2, join the shadow edges of e1 and e2. These two shadow edges will be part of the same master polygon, but they might not form a closed polygon.

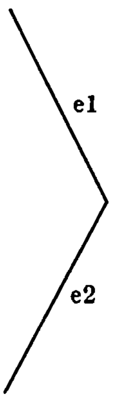


Figure 3.4

Case 3: *Side vertex* - replace edge e1 with the new edge e2 in the active edge structure, and the edge e2 gets edge e1's node–list unchanged. The edge e2 keeps the same visibility as edge e1 had. Check for an intersection of e2 and prev(e2) and add it to the priority queue if it exists. Check for an intersection of e2 and succ(e2) and add it to the priority queue if it exists. If e1 has a shadow edge then a shadow edge must be created for e2. This shadow edge is joined with the shadow edge of e1, in the opposite direction that e1 and e2 are joined.

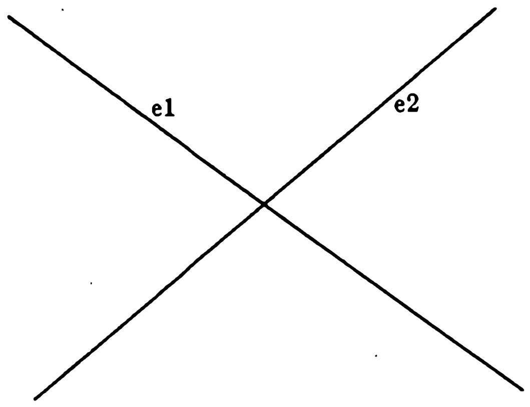


Figure 3.5

Case 4: *Intersection point* - switch the two edges (e1, e2) in the active edge structure. If the interior of e1's polygon lies to the left of the edge then add the polygon to e2's node–list otherwise delete the polygon from e2's node–list. If the interior of e2's polygon lies on the right of the edge then add the polygon to e1's node–list otherwise delete the polygon from e1's node–list. If both edges are invisible then they will continue to be invisible. If both edges are visible then the closer edge at the intersection point continues to be visible and the other edge becomes invisible. If one edge is visible then it continues to be visible and the other edge becomes visible if its face is at the top of its node–list at the intersection point. This is explained in more detail in Section 3.4.

Each case involves at most two updating operations to the structures storing the active edges and the node–lists. Each of these operations takes $O(log\ n)$ time and the priority queue updates also take $O(log\ n)$. Since the sweep line visits $O(n+k)$ points (where k is the number of edge intersections), the algorithm requires $O((n+k)\ \log\ n)$ time.

The $O(1)$ copying time for MV–trees is necessary so that the trees do not need to be copied each time a node–list is copied. If standard AVL trees are used, it will require $O(n)$ steps to make a copy of the tree. This will affect the time bound on the algorithm so that it no longer runs in $O((n+k) \log n)$ time.

The space requirements for the algorithm cannot exceed the time bounds so the space required is at most $O((n+k) \log n)$. The possible increase in space from $O(n+k)$ is caused by the MV–trees which trade off time for space through the copying operation. The exact space bounds of MV–trees is not known at the current time, and remains an open problem.

3.3. Psuedo code for Nurmi's algorithm

In the following, **E1** is the edge *above* the sweep point **SP**, and **E2** is the edge *below* the sweep point. At an intersection, **E1** is the leftmost edge. The function prev(**E**) returns the previous edge in the active edge list and the function succ(**E**) returns the next edge.

```
for each polygon P insert all the vertices into priority queue Q
initialize active edge list R to the background polygon's edges
while the priority queue is not empty do
    SP ← deletemin(Q)
    if SP is a side vertex then
        replace E1 with E2 in R
        if E2 and prev(E2) intersect then add the intersection to Q
        if E2 and succ(E2) intersect then add the intersection to Q
        if E1 has a shadow edge then
            continue the shadow edge at SP with E2's shadow edge
        endif
    elseif SP is a top vertex then
        insert E1 and E2 into R
        if E1 and prev(E1) intersect then add the intersection to Q
        if E2 and succ(E2) intersect then add the intersection to Q
        if SP's polygon is frontmost then
            create shadow edges for E1 and E2
            create a hole in the previously visible polygon
        endif
    elseif SP is a bottom vertex then
        if prev(E1) and succ(E2) intersect then add the intersection to Q
        delete E1 and E2 from R
        if E1 has a shadow edge then
            merge the shadow edges of E1 and E2
        endif
    else { SP is an intersection }
        if at least one of E1 or E2 is visible
            determine visibility after the intersection,
                and update shadow edges as appropriate.
        endif
        switch E1 and E2 in R
        if E2 and prev(E2) intersect then add the intersection to Q
        if E1 and succ(E1) intersect then add the intersection to Q
    endif
endwhile
```

3.4. Updating at intersections

During the sweep, visible edges have a "shadow" edge associated with them to define the border between two visible polygons. These are represented by dotted lines in the diagrams. These shadow edges are used to define holes and visible pieces of polygons to ensure that the final polygons are a set of non–overlapping polygons that completely tile the window. These edges are updated as necessary at each sweep point, and their existence on an edge defines the edge to be visible. This gives a fast method for determining if the edges at an intersection point are invisible. If they are both invisible before the intersection they will continue invisible after the intersection and the only processing to be done is to update the active edge list and to check for new intersections.

If at least one of the edges coming into an intersection is visible, then updates to the visibility status of the edges may be necessary. There are eight cases to consider at an intersection point, depending on the visibility of each edge coming into the intersection point, and which side of the edge the polygon is on. These cases are illustrated in Figure 3.6. Each of these cases will be discussed in turn.

3.4.1. Case 1

We have a shadow edge for P2 along **e1**, which must be joined to **e2**, and P2 is possibly closed off. The shadow edge of **e2** for P3 continues along **e1**, replacing the old shadow edge. **e2** will no longer have a shadow edge since it is no longer visible.

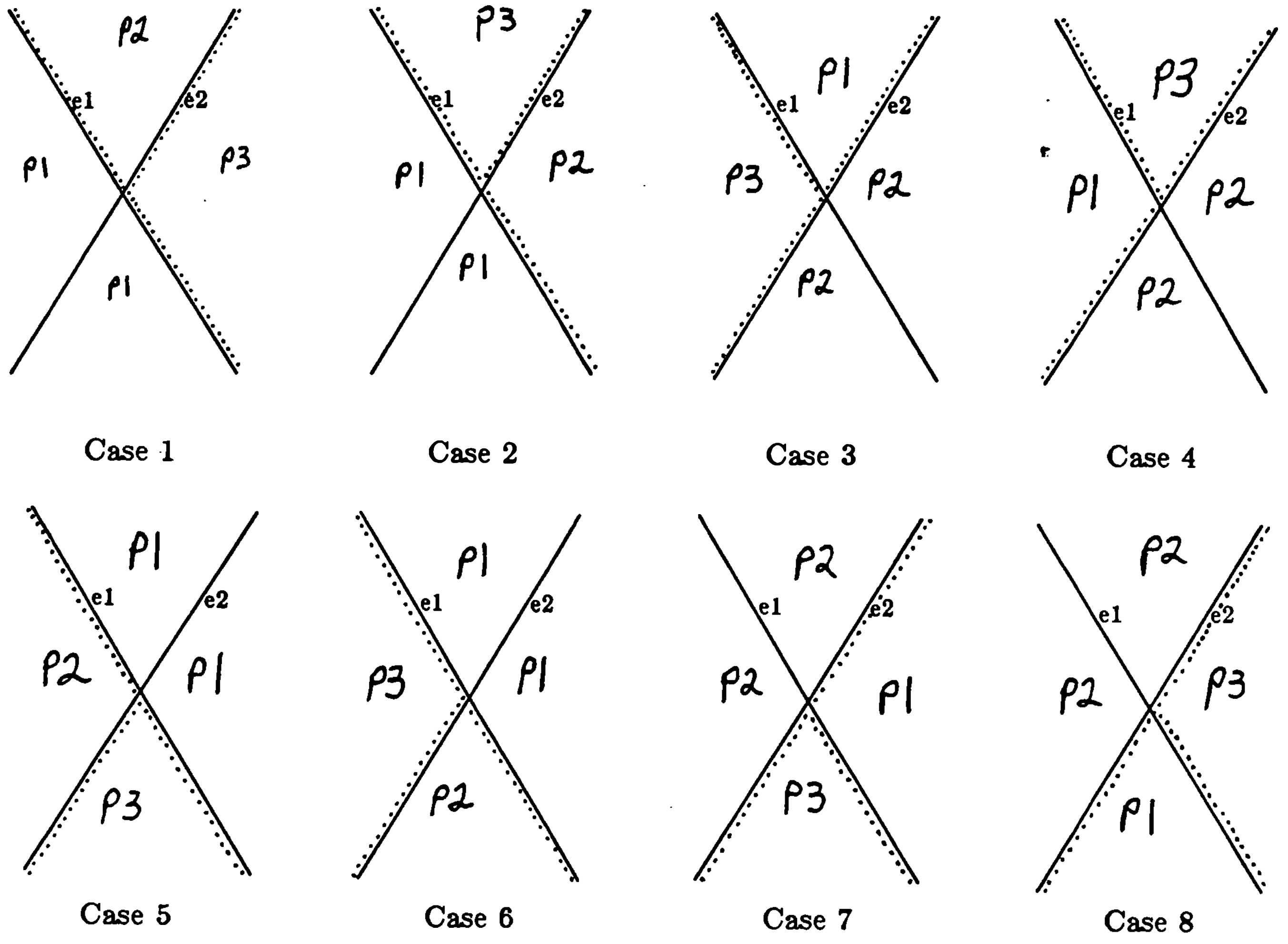


Figure 3.6

3.4.2. Case 2

The shadow edges of **e1** and **e2** are joined and P3 is possibly closed off. **e1** gets a new shadow edge for P2 and **e2** loses its shadow edge.

3.4.3. Case 3

The shadow edge of **e2** is joined with **e1** and P1 is possibly closed off. The shadow edge of **e1** for P3 continues as a shadow edge of **e2** and **e1** loses its shadow edge.

3.4.4. Case 4

The shadow edges of **e1** and **e2** are joined and P3 is possibly closed off. **e2** gets a new shadow edge for P1 and **e1** loses its shadow edge.

3.4.5. Case 5

The shadow edge of **e1** for P2 continues as **e2**. Both **e1** and **e2** get shadow edges for P3.

3.4.6. Case 6

The shadow edge of **e1** for P3 continues as the shadow edge of **e2**. **e1** gets a shadow edge for P2.

3.4.7. Case 7

The shadow edge of **e2** for P1 continues as **e1**. Both **e1** and **e2** get shadow edges for P3.

3.4.8. Case 8

The shadow edge of **e2** for P3 continues as the shadow edge of **e1**. **e2** gets a new shadow edge for P1.

Modifying the GR package interface

4.1. Overview

The GR package evolved from a set of programming assignments given in the introductory course on computer graphics. These assignments were originally in FORTRAN, and were changed to Pascal in 1979. The change of language allowed sophisticated data structures to be used in the package, and improved student productivity. The research package used by the Computer Graphics Laboratory (CGL) is an implementation of these assignments in the C language. The major functional difference between the two packages is that a larger number of input and output devices are supported by the research version of the package.

The GR package on CGL was modified to accommodate the plane sweep algorithm for solving the visible surface problem. A detailed description of the GR package is contained in [Lea]. Only an overview will be presented here. The GR package is split into five levels, each with its own function.

4.1.1. Level 5

This level of the package is the *application library.* This level is used to read in scene files and to display the scene file by walking the *directed acyclic graph* which represents the scene file within this level and contains nodes for modeling transformations (rotate, scale, translate) and modeling primitives (polygons and text - only polygons are considered here). A routine called ReadScene is used to read in a file containing a description of the graph. This routine is used to input static scenes to the package.

4.1.2. Level 4

This level forms the graphics interface module for the GR package. A number of graphics primitives are provided by this level to enable a user to do drawing, colouring, and geometric transformations. Routines to handle the perspective transformation are also included. Input to this level is specified as non–homogeneous three–dimensional coordinates (x,y,z). The output of this level is specified as homogeneous coordinates (x,y,z,w).

4.1.3. Level 3

This level of the package looks after clipping its input to a rectangular volume, so that $-1 \leq x \leq 1$, $-1 \leq y \leq 1$ and $0 \leq z \leq 1$. Both line clipping and polygon clipping are supported. Conversion from homogeneous coordinates back to non–homogeneous normalized device coordinates is also performed at this level.

4.1.4. Level 2

This level maps the (x,y,z) from floating point normalized device coordinates to *virtual device coordinates*, which are 15–bit unsigned integers. A viewport may be defined to be a portion of these *virtual device coordinates*, so that the picture is mapped to a portion of the virtual screen.

4.1.5. Level 1

This level contains all of the device–dependent code for input and output devices. Coordinates are translated to/from the actual device coordinates by the level 1 routines. A number of devices are supported by level 1 of the package, including the Ikonas frame buffer, the Tektronix 4027 and 4010 display terminals, the Hewlett Packard 2648A terminal, and a Summagraphics Bitpad One tablet.

4.2. Plane Sweep

The plane sweep algorithm fits between levels two and three of the GR package, making it effectively level 2.5. The current implementation modifies the standard package so that all of the Level 3 routines now call Level 2.5 routines instead of Level 2 routines for output related tasks that could affect the plane sweep. The master polygon and all of its pieces and holes are clipped for input into the sweep. Polygon clipping is used rather than line clipping because the plane sweep assumes that its input is a collection of polygons. The calculations for the sweep are carried out in floating point arithmetic and the polygons are then either written to a file for subsequent processing or passed down to level 2, where the coordinates are converted to integer virtual device coordinates. The output file is passed through a conversion program to convert the polygons into the format acceptable to the standard GR ReadScene routine.

4.3. Master Polygons

Every input polygon is either a *master polygon* or is associated with a master polygon. The master polygons are processed with their subservient polygons through the transformations of level 4 of the GR package and through the clipping in level 3. Currently this relationship does not continue to the lower levels for rendering, but the software could be extended to maintain the polygon together for processing at the rendering stage.

The master polygons keep all of the information on the polygon, such as normals at vertices, the plane equation, and any texture mapping parameters. Each of the subservient polygons keeps a pointer to the master polygon and the master polygon keeps a pointer to a linked list of its subservient polygons. Each master polygon must have at least one subservient polygon defining a potentially visible portion, or else the whole polygon is considered to be visible. In order to allow holes to be represented, a hole is transversed in the opposite direction to that of the master polygon, so that the order of vertices for both holes and regular polygons determines which side of an edge is on the interior of the polygon.

The master polygon concept is used to ensure that shading and texturing can be done correctly. Linear interpolation is the usual method for determining normals or shading values at points other than a vertex. This is not rotationally independent, so the order in which operations are performed can give different values for the same point. This can cause shading values for pieces of a polygon which abut to be different. A similar problem occurs with textures not lining up properly for the various pieces of a polygon.

4.4. ReadScene Changes

The format of the ReadScene was modified to allow master polygons to be distinguished from the subservient polygons. If a polygon is subservient to another polygon, then the number of that polygon is added as an additional field on the scene node description. If this field is missing or is zero, then the polygon is assumed to be a master polygon, so Poly in Level 4 of the GR package was modified to create a subservient polygon consisting the the entire master polygon if there are no subservient polygons given as input. By implementing the input in this manner, old definitions can be read in, and all of the input polygons will be considered as master polygons. The ReadScene files will be upward compatible, but files output from the plane sweep must have the master polygons deleted before the files can be used by the standard GR package.

Conclusions

5.1. Current Implementation

The integration of a plane sweep algorithm as a new level of the GR package is in progress. The output from the algorithm may be run through a conversion program to create a new ReadScene file which can be read in by the standard package or the new version. If this file is to be used with the standard package, a second step is necessary in order to remove the master polygons from the file, and to interpolate the normals at the vertices of the subservient polygons so that the standard package has the information necessary to render the scene. This allows the scene to be rendered using the standard package without modifying the lower levels of the package to accommodate the master polygon representation.

A background polygon is defined to cover the window to ensure that there is a polygon covering the whole window. This ensures that the window can be "tiled" with no holes. This polygon is positioned at maximum z-depth so that other polygons in the scene will always obscure it. The input polygons to the plane sweep have the vertices of their edges put into the priority queue after the master polygon and its subservients have been clipped. When all of the polygons have been processed the plane sweep is started.

During the plane sweep the visible polygons intersecting the sweep line are kept as an oriented collection of edges. When these polygons are closed they are added to a linear list of visible polygons. These are the polygons which are output as the tiling of the plane. All of the master polygons are output, together with the visible portions of each polygon. This output file is processed by a conversion program to create a new ReadScene file.

5.2. Possible extensions

The lower levels of the GR package should be modified to process the master polygons and the subservients together. This would be done in conjunction with rewriting the rendering software to take advantage of having all the parts of a polygon available, so that shading and texturing can be done correctly on the visible portions of the scene.

Currently the information on polygon colours is not retained by the plane sweep. The colour nodes read by ReadScene should be included in the output file generated by the plane sweep.

5.3. Applications

This software will be used to generate images for use by the Department of Psychology for cognitive experiments. These experiments consist of static images which will be used on the DY–4 workstations and the Amiga personal computer. Because the plane sweep solves the visible surface problem independent of the resolution of the display, the same scene files can be used for both display systems.

The output from the plane sweep has the advantage of being compact compared to a raster image. This allows it to be sent to the display systems quickly. Usually the rendering speed is limited by the display system capabilities.

References

[BeO] Bentley, J. L., Ottmann, T. A., *Algorithms for reporting and counting geometric intersections*, IEEE Transactions on Computers C-28 (1979) pp. 643–647.

[Bro] Brown, K. Q., *Comments on "Algorithms for reporting and counting geometric intersections"*, IEEE Transactions on Computers C-30 (1981) pp. 147–148.

[Dev] Devai, F., *Quadratic bounds for hidden-line elimination*, Proceedings of 2nd Symposium on Computational Geometry (1986) pp. 269–275.

[FVD] Foley, J. D., Van Dam, A., *Fundamentals of interactive computer graphics*, Addison–Wesley (1982).

[Mat] Mathies, L., *Scanning algorithms for computer graphics*, Masters Essay, University of Waterloo (1981).

[McK] McKenna, M., *Worst-case optimal hidden—surface removal*, John Hopkins University Report JHU/EECS-86/05 (1986).

[Mor] Morgan, M. F., *Graphics support for cognitive research in human factors*, Masters Essay, University of Waterloo (1985).

[NeS] Newman, W. M., Sproull R. F., *Principles of interactive computer graphics*, McGraw–Hill (1979).

[NiP] Nievergelt, J., Preparata, F. P., *Plane-Sweep algorithms for intersecting geometric figures*, CACM Vol 25 No. 10 (Oct 1982).

[NiR] Nievergelt, J., Reingold, E. M., *Binary search trees of bounded balance*, SIAM Journal on Computing (1973) pp. 33–43.

[Nur] Nurmi, O., *A fast line-sweep algorithm for hidden line elimination*, BIT Vol 25, No. 3 (1985).

[OWi] Ottmann, T. and Widmayer, P., *Solving visibility problems using skeleton structures*, MFCS (1984).

[OWW] Ottmann, T. and Widmayer, P., Wood, D., *A Worst-case efficient algorithm for hidden-line elimination*, Intern. J. Computer Math., Vol 18 (1985).

[OWo] Ottmann, T., Wood, D., *Space economical plane-sweep algorithms*, University of Waterloo Technical Report CS-84-32.

[SeG] Sechrest, S., Greenberg, D., *A Visible polygon reconstruction algorithm*, TOG Vol 1, No 1 (Jan 1982).

[Sch] Schmitt, A., *Time and space bounds for hidden line and hidden surface algorithms*, Eurographics'81.

[ShH] Shamos, M. I., Hoey, D., *Geometric intersection problems*, Proceedings of the 17th Annual IEEE Symposium on Foundations of Computer Science (1976).

[SSS] Sutherland, I. E., Sproull, R. F., Schumacker, R. A., *A characterization of ten hidden—surface algorithms*, Computing Surveys, Vol 6, No 1 (March 1974).

[SwL] Swart, G., Ladner, R, *Efficient algorithms for reporting intersections*, University of Washington Technical Report 83-07-03.

[Wat] Watkins, G. S., *A real time visible surface algorithm*, University of Utah Technical Report UTEC-CSc-70-101.