# Exploiting scanline coherence for front-to-back visible surface determination

Sharat Chandran
Santosh Chandrachood
{sharat,santosh}@cse.iitb.ernet.in

## 1 Background

Visible surface determination, also known as hidden surface removal, is a very popular problem. In its most general form, it involves computing the scene visible from a given viewpoint when given a set of objects in three dimensional space. Often, the objects are broken up into polygonal faces before fed into this part of the rendering pipeline, and are considered opaque. The faces determine the complexity of the procedure.

Preceding this module are steps like perspective and clipping transformations which operate on one face at a time, and are independent of other faces. Unfortunately, to determine whether a face is visible or not, all or many other faces need to be compared against for detection of partial or full obscurity. This makes the cost of the visible surface determination problem expensive, and it comes as no surprise that this problem has been well studied.

The problem of hidden surface removal may be solved in *object space*. In this case, one tries to describe the visible scene without regard to a specific output device (or its resolution), on which it is to be displayed. Formally stated, one tries to calculate a subdivision of the viewing plane, into maximally connected regions, in each of which (some portion of) a *single* object can be seen.

Alternatively, the problem may be solved in *image space*, i.e., one tries to get the image correct on the specific output device under consideration, without actually bothering to geometrically describe the visible scene. For instance, on a pixel based device, one only calculates, for each pixel, the object, if any, visible at that pixel.

In practice, for a large class of situations, image space algorithms turn out to be superior in execution time, although object space algorithms often tend to be intellectually stimulating. This should not come as a surprise to computer science theoreticians — bucketing based sorting methods are faster than 'optimal' heapsort and more commonly used in compiler design, for instance. It is image space algorithms that we are interested in, specifically; however, note that the inherent solution paradigm between the two methods are often interchangeable.

### 1.1 Using depth sorting to display the visible scene

In considering image space algorithms, we turn quickly to the topic of this paper, and classify methods as either front-to-back or back-to-front.

*A face $f_1$ is said to be nearer (or less deep) than a face $f_2$ if no ray joining a point on $f_1$ to the viewpoint intersects $f_2$.*

Suppose one has sorted the faces of the input polygons by nondecreasing depth[1] from the viewpoint. Then one can use either of the following approaches.

### 1.1.1 Back to front display

Also known as the *painter's algorithm*, here, one paints the faces on the screen in back to front order, i.e. the furthest face first. Every successive (and hence nearer) face thus *overwrites* the faces painted before it, wherever its projection intersects the projections of the latter. Hence the final image obtained is correct.

The advantage of this method is its simplicity. On the other hand, notice that for those pixels which belong to the projections of more than one face, this method paints them many times, whereas one would want them to be painted only once – with the colour of the nearest face containing it. This constitutes the major disadvantage, since realistic computation of the colour is expensive.

### 1.1.2 Front to back display

Here one paints the faces in front to back order, i.e., the nearest face first. A face is painted however, only in the regions of the screen which are as yet untouched by any (previously painted) face. Thus each pixel is painted at most once, unlike in the back to front approach. On the other hand, in its crudest form, painting each face would involve checking *every* pixel contained in its projection to see if it is already lit or unlit. The algorithm of Gordon and Chen [GC91] attempts to reduce this overhead.

## 1.2 Moving and multiple viewpoints

We have not yet considered an important variation in hidden surface removal that is central to our interest. This deals with the situation when the 'given' viewpoint is constantly changing. A typical example would be in quickly rendering the scene from a moving aeroplane. In such cases it is useful to have a method to quickly determine the front to back order.

Binary Space Partitioning (BSP) is a method to preprocess a given set of (polygonal) faces, so that when given any position of the viewpoint, one can obtain the depth sorted order of the polygons quickly. This method suitably splits the input polygons (say there are $n$ of them), to obtain a new set of $n'$ (smaller sized) polygons. Simultaneously, it builds a binary tree called the BSP tree, each node of which is a polygon belonging to the latter set. Now, given any arbitrary viewpoint, one can obtain the depth sorted order of the (new) polygons in $O(n')$ time, by performing a (modified) in-order traversal of the BSP tree. For details about the BSP tree method, the reader is referred to [GC91]and [PY89]. (Also, in many cases, $n^{'} = O(n)$.)

## 1.3 This article

We shall assume throughout this article that perspective transformations, clipping, and other standard preliminary steps have already been done. Further, depth ordering of the resulting polygons has been obtained as mentioned above. The real bottleneck then, is the display.

[1]With the above definition of depth, it is possible that both, $f_1$ and $f_2$ are 'nearer' than the other. In this case, either order is permissible. Also, one can have a 'cyclic overlap,' wherein no depth order is possible. Both situations are not of concern here, and are handled by the techniques mentioned later.

Gordon and Chen describe a method for the *front-to-back* display of faces and establish the competitive nature of their technique. The efficiency is achieved because, simply speaking, a group of visible pixels is painted together; if a pixel belonging to some face is visible, it is very likely that the pixel adjacent to it is also visible. There are, of course, a few pixels on a single scanline that do not have this property.

We further develop the theme of [GC91]. We argue that if we have, inductively, solved the visible surface problem on a single scan line, it is unlikely that the next scan line is likely to be too different; this facilitates the solution. Can we efficiently determine how different the next scan line is? We answer this affirmatively in Section 2.2, and the results of our approach are demonstrated in Section 3.

# 2 Algorithm descriptions

In this section, we first formulate more precisely the idea of front-to-back scan-line coherence for visible surface determination. In order to do this, it becomes helpful to first describe the earlier method of [GC91]. Later, we spell out our algorithm in complete detail.

## 2.1 The method of Gordon and Chen

The authors preprocess the polygons into a BSP tree, so that given a viewpoint, the polygons can be obtained in front-to-back order in linear time.

The data structure used in [GC91] is a "dynamic screen." If the screen has $N$ scanlines of $M$ pixels each, the dynamic screen is an array of pointers $DS[1, \ldots, N]$, where each $DS[i]$ points to a linked list of contiguous *segments* of unlit pixels on scanline $i$ (Fig.1). These segments are referred to as *image segments*.

Each node of the list has three fields: $Min$ and $Max$ for the left and right abscissae of the segment it represents on the screen, and $Next$, a pointer to the next list element. Initially each $DS[i]$ contains just one element representing the entire unlit scanline.

The polygons are added front-to-back to an initially blank screen. When each polygon $P$ is scan converted, one obtains the list of *object segments* on each scanline $i$ (i.e. the regions covered by $P$ on scanline $i$). This list is then 'merged' with the list $DS[i]$ to give the visible portions of $P$ on scanline $i$. $DS[i]$ is simultaneously updated. Fig.2 illustrates an example.

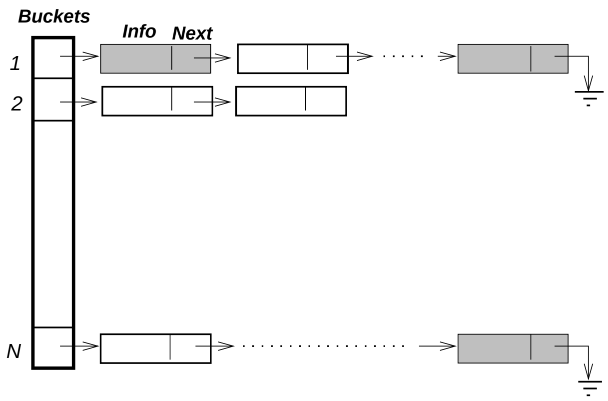


Figure 1: The Dynamic Screen

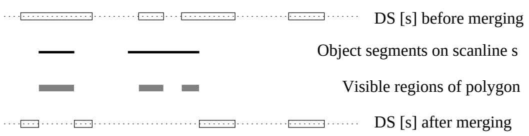


Figure 2: The result of a merge operation

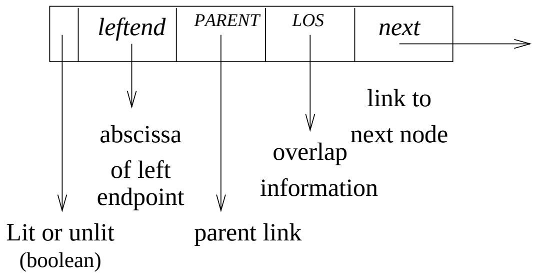


Figure 4: Structure of each node

## 2.2 Motivation behind our method

The algorithm of [GC91] speeds up front-to-back hidden surface removal in image space by considering several pixels (on a scan line) together. This algorithm does not however take any advantage of the scan line coherence of (the projections of) the input polygons.

Consider Fig. 3. Both lit and unlit segments (corresponding to lit and unlit pixels) have been shown.

*Let us say that two segments (on different scan lines)* **overlap** *if their extents overlap.* Thus $A_k$ and $B_3$ (on scan lines $i$ and $i+1$ respectively) overlap; so do $A_k$ and $B_5$. But $A_k$ and $B_1$ do not overlap.

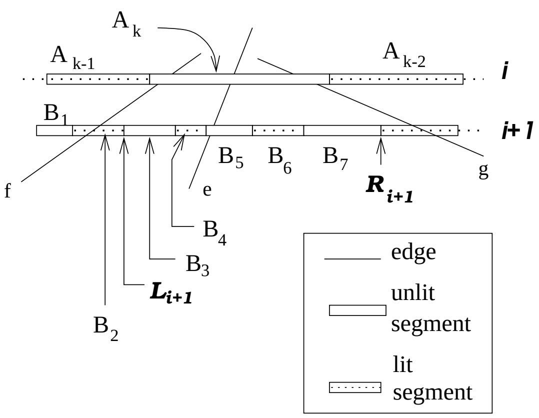


Figure 3: Scanline Coherence

The projected edge $e$ of some face intersects scan line $i$ in the (unlit) segment $A_k$, at the point $e_i$. Typically, we can expect the intersection point $e_{i+1}$ of $e$ with the scan line $i+1$ to lie in one of the segments (lit or unlit) of the scan line $i+1$ that overlap with the segment $A_k$, i.e., $e_{i+1}$ is between $L_{i+1}$ and $R_{i+1}$. Thus, if we maintain, along with each segment (lit or unlit), information as to which segments on the next scan line 'overlap' with it, then we are able to quickly (say, in one step) locate the segments containing the intersection points of an object with the next scan line, from the corresponding information on the current scan line.

More formally, on a given scanline, let $N$, $K$, and $P$ be the number of image segments, number of object segments and number of image segments affected (i.e. deleted or shrunk) by the rendering of this polygon; then, whereas [GC91] incurs a cost of $O(N+K+P)$ per polygon per scanline, we aim at $O(K+P)$ only.

The idea, of course, fails if the edge is too oblique, e.g., edges $f$ and $g$ in Fig. 3. We need to ensure that even in such a case, our method does at most as much work as the original algorithm.

## 2.3 Changes in the data structure

It becomes more convenient to maintain both the lit and unlit segments in the Dynamic Screen. Accordingly, the contents of of each node (representing a segment) are as in Fig.4.

```
1.  for scanline s = topmost ordinate to bottommost ordinate of P
2.  begin
3.      for each object segment obj on scanline s
4.      begin
5.          l := Locate left endpoint x_l, of obj in DS[s] ;
6.          Delete (portions of) all segments between l and the right
               endpoint of obj. When deleting an unlit segment,
               paint s in that region with the colour of P
7.          Update overlap links between scanlines s − 1 and s
8.          Update overlap links between scanline s and s + 1
9.      end
10. end
```

Figure 5: Steps in painting a polygon $P$

The $LOS$ (*L*eftmost *O*verlapping *S*egment) field points to the leftmost of the nodes (representing lit or unlit segments) on the scanline $i+1$ that overlap with the node (segment) represented by this node. If the $LOS$ field of a set of nodes on scanline $i-1$ points to a node, then the $PARENT$ field of that node points to the leftmost of such segments; otherwise it is $NULL$.

Henceforth a 'segment' will be understood to mean such a node (representing a lit or unlit image segment), unless otherwise mentioned.

## 2.4 The algorithm

Painting a polygon $P$ involves the steps of Fig.5. The steps are described in greater detail in the following subsections.

It is step 5 of Fig.5 where we expect to do better than [GC91] because of the $LOS$ links. (The right endpoint of *obj* is reached as we walk along and delete segments on scanline $s$, starting from $l$. This is step 6 of Fig.5. Since we have to ultimately paint the scanline in this region, this step is unavoidable.) The overheads we incur over [GC91] are

1. steps 7 and 8,

2. we have twice as many segments as [GC91] in the Dynamic Screen at any point of time.

### 2.4.1 Step 5: locating the left endpoint

As in Fig.5, $l$ is the segment containing $x_l$. Using the argument of section 2.2, we perform the steps of Fig. 6. If *obj* is the leftmost object segment of polygon $P$ on this scanline, then $LasstVisited$ is the header node $DS[i]$; otherwise $LastVisited$ is the last visited segment on scanline $s$ (while processing the previous object segment). If the edge under consideration entered the scan conversion on scanline $s$ itself, then *hint* is $NULL$; otherwise *hint* $= LOS(l')$ where $l'$ is the segment on scanline $s-1$, which intersected that same edge. *hint* is remembered while performing step 5 on scanline $s-1$.

```
ptr ← LastVisited ;
if ( (hint ≠ NULL) and (hint.leftend > LastVisited.leftend)
            and (hint.leftend ≤ x))
        /* then we can skip a few segments */
        ptr ← hint ;

/* travel rightwards */
while ( (ptr.NEXT ≠ NULL) and (ptr.RIGHTEND ≤ x) )
        ptr ← ptr.NEXT ;

/* ptr now points to the segment containing x */
```

Figure 6: Locating a left endpoint

#### 2.4.2 Step 6: Updating segment endpoints in $DS[s]$

This step is similar to what [GC91] would do. The precise steps are given in Appendix 4.

#### 2.4.3 Step 7: Updating $LOS$ and $PARENT$ links between scanlines $s-1$ and $s$

While performing step 6, we use the $PARENT$ fields to find which is the leftmost segment on scanline $s-1$, whose left endpoint falls in the region affected by step 6. Starting from this segment, we travel rightwards along scanline $s-1$, and update the $LOS$ and $PARENT$ links as we proceed. The details are given in Appendix 4.

#### 2.4.4 Step 8: Updating $LOS$ and $PARENT$ links between scanlines $s$ and $s+1$

Starting from the segment (on scanline $s+1$), which is the $LOS$ of the leftmost affected segment (on scanline $s$) in step 6, we travel right. We update the $PARENT$ and $LOS$ links as we proceed. The details are given in Appendix 4.

### 2.5 A logarithmic method

The method of Section 2.4.1 found the correct image segment in time $O(N)$ where $N$ is the number of image segments on the scanline. We can bring this down to $O(\log N)$ by changing the way we maintain the overlap information. The details of this method are given in Appendix B.

## 3 Experimental results

To quantitatively estimate the improvement of our method over that of [GC91], we implemented both methods. We tested both the programs with inputs consisting of various numbers of triangles. The triangles were input in front to back order, so that the ordering step (which is common to both) could be eliminated. The tests were performed on a DEC Alpha 130 workstation, running OSF/1 v2.0.

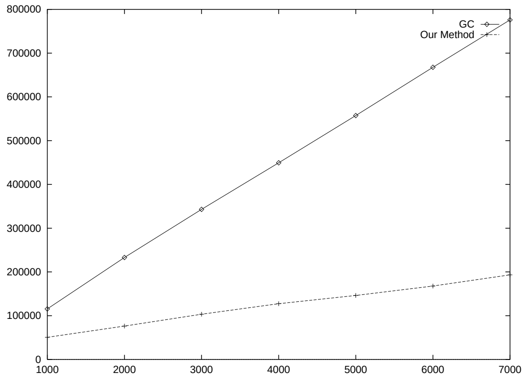


Figure 8: No. of Accesses(size =100)

The triangles were generated, at random, by a program which took as input parameters, the number of polygons required and a number $maxsize$. For each triangle, the program first generated a point within the screen, at random. Then taking this as the origin, it generated the coordinates of the three corners of the triangle as integers within the range $[-maxsize, maxsize]$.

The timings of the two programs were competitive for small number of triangles as the dynamic screen is less fragmented, and the gain in step 5 (Fig. 5 ) is marginal to overcome the overheads incurred in steps 7 and 8 (Fig. 5 ). The gain over the method of [GC91] occurs as the number of polygons increses. A substantial gain begins as the number of polygons exceeds 1000. As expected our method traveses far less no. of segments than [GC91]as shown in Figures(7,8,9). The table 1 shows the time gain over [GC91].

It can be seen that the method gains as the number of polygons and the size of polygons increase. The number of object segments to be scanconverted increases with the increase in the number and the size of polygons, thus increasing the number of segments to be traversed. Moreever, the gain due to the increase in the size of polygons is attributed to the fact that as the size of polygons increases, the likelihood of the next object segment to be scanconverted lying in the already painted region of dynamic screen increases, in turn eliminating the overheads in steps 7 and 8 (as the dynamic screen is left unaltered).

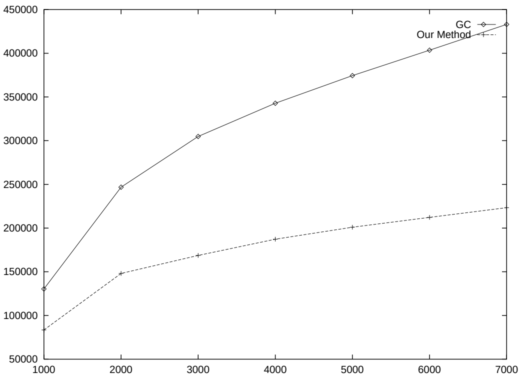


Figure 7: No. of Accesses(size = 25)

# 4 Conclusion

An important problem in computer graphics is the dynamic determination of visible scenes when the viewer is moving. If the objects can be modeled as a set of polygonal faces in three-dimensional space, as is usually the case, a useful data structure is the BSP tree which allows to arrange the faces of the polygons (or their appropriate parts) in order of sight.

When presented with this ordered list of polygonal faces, the practitioner has a choice of painting the polygons in either back-to-front order, or in the reverse order. The former

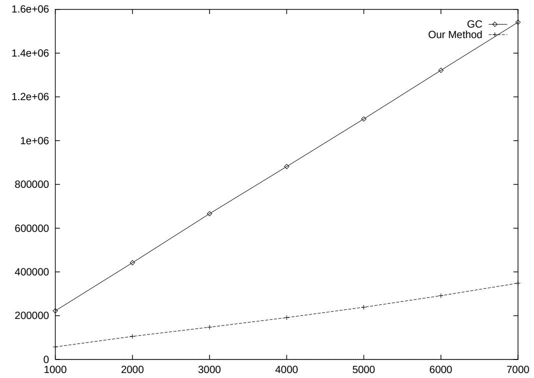


Figure 9: No. of Accesses(size =225)

| Polygons | Size: 25 | | Size: 100 | | Size: 225 | |
|---|---|---|---|---|---|---|
| | GC[91] | Our Method | GC[91] | Our Method | GC[91] | Our Method |
| 1000 | 1.63 | 1.37 | 2.34 | 1.56 | 3.74 | 2.40 |
| 2000 | 3.02 | 2.60 | 4.59 | 3.33 | 7.48 | 4.78 |
| 3000 | 4.44 | 3.71 | 6.95 | 4.68 | 11.24 | 7.37 |
| 4000 | 5.46 | 4.85 | 9.42 | 6.35 | 15.02 | 9.14 |
| 5000 | 6.89 | 5.95 | 11.51 | 7.69 | 18.58 | 11.02 |
| 6000 | 8.17 | 6.95 | 13.77 | 9.23 | 22.1 | 13.17 |
| 7000 | 9.42 | 8.10 | 16.15 | 10.74 | 25.79 | 15.35 |

Table 1: Experimental Results

method is simple since we know a face yet to be painted can be painted over the faces already painted.

However, it was observed that when the number of polygons increase, it is a waste to paint the same area multiple times. A front-to-back order was suggested where we must now take care not to paint in areas that has already been painted. The method in [GC91] relied on the good likelihood that a set of contiguous pixels were likely to have been painted *along a scan-line.*

In this article, we have shown that it is also very likely that two adjacent scan lines are similar in the sense of being painted, and developed a scan coherent front-to-back visible surface algorithm. We quantitatively measured this likelihood in terms of the number of accesses to segments.

We implemeted our ideas on various inputs, we found that although two adjacent scanlines *are* similar, it costs us to determine precisely when they are, and when they aren't. However, the overheads in determining the scanline coherence are overcome as the number and the size of the polygons increase which in turn increase the number of object segments to be scanconverted, and our mehtod improves substantially over the method of [GC91] .

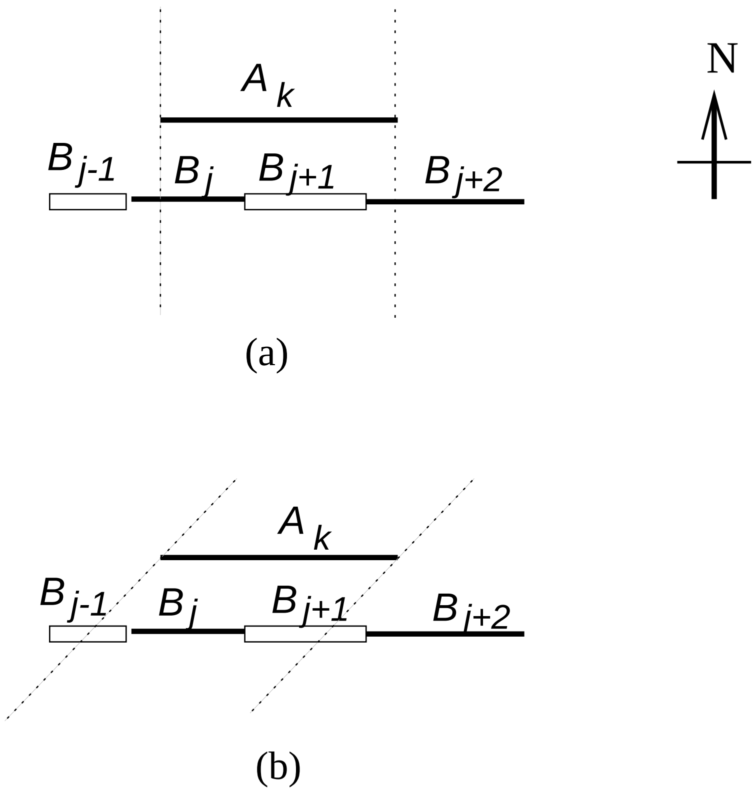


Figure 10: Updating endpoints of segments on a scanline

## Appendix A: Details of steps of Figure 5

### Step 6: Updating segment endpoints in $DS[s]$

This step is similar to what [GC91] would do. Fig.10 illustrates the following description. If *obj* is entirely contained in $l$ (the segment returned by step 5), and $l$ is unlit, we split $l$ (Fig.10.(a)). If *obj* is not entirely contained in $l$, we

1. shrink $l$, if $l$ is unlit, by updating $(l.next).leftend$ to $x_l$. Paint in the region shrunk.
2. delete all segments from $l.next$ onwards until we reach $r$, the segment containing the right endpoint $x_r$ of *obj*. $r$ is not deleted. For each unlit segment deleted, paint the screen.
3. shrink $r$, if $r$ is unlit, by updating $r.leftend$ to $x_r$. Paint in the region shrunk.
4. if the above operation leaves two consecutive lit segments viz. either $l$ or $l.next$, and $r$, then delete $r$.

### Step 7: Updating $LOS$ and $PARENT$ links between scanlines $s-1$ and $s$

Only one segment on scanline $s$ spans the region between $x_l$ and $x_r$ after step 6. Let this be $SpanObj$.

To know the segments on scanline $s-1$ whose $LOS$ fields need to be updated, we use a variable $par_{obj}$, initialised to $NULL$. Whenever we shrink or delete a segment $\sigma$ in step 6, we check if $PARENT(\sigma)$ is non-$NULL$. If yes, and if $par_{obj}$ is still $NULL$, we set $par_{obj}$ to $PARENT(\sigma)$.

If $par_{obj}$ is $NULL$, no $LOS$ links need to be updated. Otherwise we travel rightwards from $par_{obj}$. For all segments with left ends between $x_l$ and $x_r$, we set the $LOS$ fields to $SpanObj$. Simultaneously if the leftmost of such segments is to the left of the current $PARENT(SpanObj)$ we must set $PARENT(SpanObj)$ to it. If step 6 involved splitting $l$, then we must continue along scanline $s-1$ to update the links with *SpanObj.next* also.

### Step 8: Updating $LOS$ and $PARENT$ links between scanlines $s$ and $s+1$

This is symmetrical to step 7. We travel rightwards from $LOS(l)$ till we reach a segment $child_{obj}$ containing $x_l$. This segment is the new value of $LOS(SpanObj)$. Also, if $SpanObj$ is to the left of the current $PARENT(child_{obj})$, then we set $PARENT(child_{obj})$ to $SpanObj$. For the segments following $child_{obj}$, until we reach a segment whose right end lies beyond $SpanObj$, we set the $PARENT$ field to $NULL$. We continue along scanline $s+1$ to update the links with *SpanObj.next* also.

## Appendix B: Logarithmic method

We maintain a B$^+$-tree index on the endpoints of the segments of each scan line. There is one such B$^+$-tree per scan line.

### Overlap information

In Fig. 3, instead of the $LOS$ field, we maintain, from $A_k$, a link to the lowest common ancestor (in the B$^+$-tree of scan line $i+1$) of the endpoints of segments $B_j$, $\ldots$ $B_{j+m}$. Let us call this link $AOS(A_k)$ (Ancestor of Overlapped Segments). We also maintain, with $A_k$, the abscissae of the left endpoints of $B_j$ and $B_{j+m+1}$. Call this the $ROS(A_k)$ (Range of Overlapped Segments).

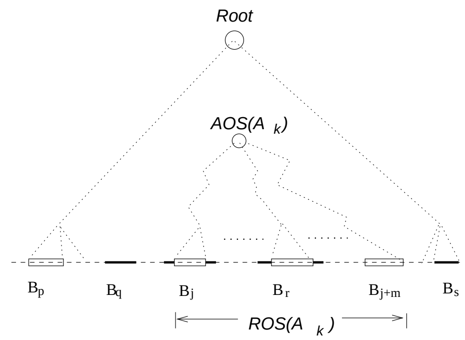


Figure 11: AOS and ROS

### Locating intersection points

To locate $f_{i+1}$ : if $f_{i+1} \in ROS(A_k)$ then start at $AOS(A_k)$, else start from the root of the B$^+$-tree of scan line $i+1$. Now search down the B$^+$-tree.

### Steps of the algorithm

The *update_endpoints*() routine consists of deleting endpoints from the B$^+$-tree and inserting new endpoints, apart from changing the values of existing endpoints. The overlap information for a segment $A_k$ is computed by travelling down the $B^+$-tree of the next scan line simultaneously for both endpoints of $A_k$ until we fork. That point is the new value of $AOS(A_k)$.

Apart from the differences outlined above, this method is similar to the linear method, and all the steps outlined there apply here too.

### Analysis

We perform a bounded number of operations on the $B^+$-tree for every endpoint of an active object segment (say $K$ in all), and for every affected image segment (say $P$ in all). Since each operation on the $B^+$-tree costs $O(\log N)$, the total cost is $O((K+P)\log N)$.

## References


[GC91] Dan Gordon and Shuhong Chen. Front-to-back display of BSP trees. *IEEE Computer Graphics and Applications*, 11(5):79–85, September 1991.

[PY89] M. Paterson and F. Yao. Binary partitions with applications to Hidden surface removal and Solid Modelling. In *Proc. of the fifth annual symp. on Comp. Geom.*, 1989.