The Visual Computer

The scanline principle: efficient conversion of display algorithms into scanline mode

Ella Barkan[1], Dan Gordon[2]

[1] IBM Haifa Research Lab, Matam Technology Center, Haifa 31905, Israel
[2] Dept. of Computer Science, The Technion–Israel Inst. of Technology, Haifa 32000, Israel
(on leave from Haifa University)
e-mail: ella@haifa.vnet.ibm.com, dgordon@cs.technion.ac.il

The scanline principle is a general technique for efficiently converting any display algorithm that is based on polygon scan conversion into scanline mode, i.e., the image is produced in scanline order with required memory proportional to one scanline. Based on critical-points scan conversion, the technique reduces the Z-buffer or its variants to one scanline. Current scanline depth buffers are inefficient in both time and space. The scanline principle can also transform list-priority methods, such as BSP trees, into scanline mode. The scanline mode enables efficient supersampling and averaging, and low latency in image generation, compression and transmission.

Key words: Hidden surface removal – Scanline – Scan conversion – Z-buffer – BSP trees – Critical-points

1 Introduction

Hidden surface removal is one of the oldest problems in computer graphics, and the literature abounds with many solutions (Foley et al. 1990). In choosing an algorithm for this problem, there are many considerations and probably no "best" method; one usually chooses the optimal method for a given application. Some of the more obvious points to consider are: time and space efficiency, robustness (ability to handle numerical inaccuracies after transformations), object intersections, types of objects that can be displayed, image quality, transparent objects, and available hardware.

One useful method for many applications is image generation using *scanline mode*, where the final image is produced one scanline at a time in consecutive scanline order. Implicit in this definition is the assumption that the space requirement for the image is proportional to one scanline and not to the entire screen. Scanline mode has many useful applications such as generating a supersampled image with a low memory overhead, transmitting the image to a remote screen in scanline order while it is being generated, and functioning as input to many image-compression methods. The last two properties make scanline mode particularly useful for web-based applications, since they provide a low latency method for generating, compressing, and transmitting images over limited bandwidth channels.

In this paper we present an efficient general technique for converting *any* display algorithm based on the separate scan conversion of every polygon into an algorithm that operates in scanline mode. We call this general method the *scanline principle*. When applied to the Z-buffer algorithm, this technique produces an algorithm with a depth buffer of just one scanline, which we call the *S-buffer*. Scanline depth buffers have been used before, but their implementations for non-convex polygons, using the standard scan conversion algorithm (Foley et al. 1990; Sect. 3.6), are very inefficient in both time and space. The principle can also be applied to any of the Z-buffer's variants such as the A-buffer (Carpenter 1984), or the handling of transparent objects. The S-buffer is also useful in a client-server environment, has some virtual reality applications, and is easily parallelizable (see Sect. 5.6).

Another application of the scanline principle is the transformation of list-priority algorithms (Foley et al. 1990; Sect. 15.5) into scanline mode. Two well-known examples are the *depth sort* algorithm (Newell et al. 1992) and the display of BSP trees

(Fuchs et al. 1980). BSP trees encode the geometric order of the polygons in a given scene, and they are widely used in applications, such as virtual walkthroughs, where the viewpoint changes frequently but the objects in the environment are fixed relative to each other. The scanline principle can be applied to list-priority algorithms to produce the display in either the standard back-to-front order, or the more efficient "output-sensitive" front-to-back order (Gordon and Chen 1991).

For some applications, the scanline mode has one disadvantage: it requires all the objects to be in memory while they are being displayed, so it is incompatible with image generation in online mode. By *online mode* we mean that an object is displayed as soon as it becomes available and no further memory is required for it. This is not a real problem for interactive applications where the objects are usually available in memory all the time in order to enable object manipulation or a change of viewpoint.

The Z-buffer is probably the simplest hidden surface algorithm and its hardware implementation is now commonplace. It is naturally robust because objects may intersect, and it also operates in online mode. The Z-buffer's main disadvantages are its size and the fact that the image is not produced in scanline order. The size of a simple Z-buffer is not a large problem, because memory is relatively cheap; but for supersampling or for some of the variants of the Z-buffer the large memory is a drawback. It is also a problem if the image is viewed on a remote client and the server has to handle many clients viewing unrelated scenes. In contrast, the S-buffer works in scanline mode and has the robustness of the Z-buffer without the large memory.

Our technique for the efficient conversion of any algorithm (based on polygon scan conversion) into scanline mode relies on the "critical-points" scan conversion technique (Gordon 1983a; Gordon et al. 1994). This method, called CP for short, is based on the preliminary identification and sorting of all vertices of a polygon which are local minima with respect to y (the "critical points"). The critical points and the original polygon data are used in place of the edge table in the usual method (Foley et al. 1990; Sect. 3.6) which contains an entry for every edge. At the sweep stage, insertions and deletions into the active edge table (AET) are done only at local extrema, as opposed to the standard method which inserts and eventually deletes every edge. By using CP, we can efficiently "dovetail" the scan conversion of many polygons so that the image is obtained in scanline order. For polygons with many vertices and few critical points, the S-buffer performed better than the standard software and hardware on medium-powered workstations.

The paper is organized into seven sections. Section 2 provides some background on scanline algorithms and depth buffers, and places our work in relation to previous work. Section 3 presents CP in a form suitable for use by the scanline principle. Section 4 describes the scanline principle and its application to the Z-buffer and BSP trees, and Sect. 5 discusses several further extensions and applications. The last two sections conclude with qualitative and quantitative comparisons.

2 Relation to previous work

2.1 Scanline algorithms

Scanline algorithms use scanline coherence to achieve hidden surface removal. (See (Foley et al. 1990; Sect. 15.6), who also write about the history of the method and some enhancements. Another important work is by Séquin and Wensley (Séquin and Wensley 1985), who improve on Hamlin and Gear's "Cross" algorithm (Hamlin and Gear 1977), to obtain a scanline object-space algorithm.) These scanline algorithms share three common features:

1. All the active edges of all the polygons are held in one single list (the AET).
2. In passing from one scanline to the next, the projections of the edges on the screen may intersect. This requires reordering the AET at every scanline, and for dense scenes, a lot of work is wasted on sorting the edges of hidden polygons.
3. Intersection of the objects in 3D is not allowed. In principle, intersections can be handled, but at a great increase in the computation time.

The third restriction is a limitation even when objects are not supposed to intersect, because translations and rotations can produce inaccuracies that cause an actual intersection. Séquin and Wensley (Séquin and Wensley 1985), in describing the Berkeley UNIGRAFIX system, remark that their system needs prefiltering programs which remove face intersections and give their system the necessary robustness.

One of the earliest scanline algorithms is due to Watkins (see Watkins 1970 or its description in (Newman and Sproull 1973; Sect. 14.4 and Appendix VII)). It can handle intersecting objects and is designed so that it can also be implemented in hardware. It is based on subdividing a scanline into "simple" parts, and also uses scanline coherence for greater efficiency.

To demonstrate the scanline principle, we will consider two methods based on polygon scan conversion: the Z-buffer and BSP trees (other list-priority algorithms are also suitable). Both methods are robust and they handle each polygon separately; so there is no need to reorder edges provided care is taken in inserting the edges into the AET (see Sect. 3). By applying our technique to the Z-buffer or BSP tree algorithms, we obtain the following advantages:

1. Scanline mode
2. Robustness – there is no limitation on object intersections
3. Very small memory (compared to the Z-buffer)
4. Arbitrary polygons
5. Reordering of edges is not required
6. Simplicity

2.2 Scanline depth buffers

Scanline depth buffers are not new. Myers (Myers 1975) uses a scanline Z-buffer for *triangles*,[1] and G. Elber (personal communication 1994) notes that using a scanline Z-buffer for *convex* polygons is a known practice. Crocker (Crocker 1984) uses a scanline depth buffer in conjunction with a standard scanline algorithm in order to achieve "invisibility coherence" – a concept useful for speeding up the scanline algorithm. However, Crocker's scanline depth buffer is not directly used for the actual hidden surface removal.

Rogers (Rogers 1985) extends Myers' method, but his method is inefficient since it is based on the regular scan conversion method and some details of implementation are unclear when the method is applied to convoluted polygons. Hill (Hill 1995) considers the Z-buffer algorithm from a functional programming perspective. Using the concept of "lazy evaluation", he obtains an algorithm that uses only a scanline depth buffer; however, he notes that his method is inefficient since it does not use coherence relations in the evaluations.

[1] Foley et al. (Foley et al. 1990, p. 684) mention Myers' result, but do not write that it applies only to triangles.

Newman and Sproull (Newman and Sproull 1979, Sect. 24.4) describe a scanline depth buffer method which is a straightforward application of the regular scan conversion. The obvious drawback of this method is the necessity of using an edge table for every polygon. For the sake of completeness, we describe below an adaptation which uses a single edge table for all the polygons. Compared to using CP, this is still inefficient in both time and space due to the extra data for all the edges on the edge table as well as the insertion or deletion of every edge into its polygon's AET. The precise differences are detailed in Sect. 2.3.

For a scanline depth buffer using regular scan conversion the edges of all the polygons are first inserted into a single edge table (ET), with each edge inserted at the beginning of its list. This is done one polygon at a time, resulting in an ET where the entries of each polygon are grouped together for every scanline. Next, for each scanline, the entries are sorted among themselves using a list-sorting algorithm.

At the sweep stage, every polygon will have its own separate AET and a list of the currently active polygons is maintained. The following is done for every scanline:

1. Old edges are removed from the AET's of the active polygons.
2. New edges are added from the ET. Since entries in the ET are grouped by polygons and sorted by x, every polygon's AET is simply merged with its entries in the ET (the control of this operation is driven by the ET). New polygons may now become active. If a polygon's AET is empty at this stage, it is removed from the list of active polygons.
3. The scanline depth buffer is initialized to $-\infty$.
4. For every active polygon, pixels are filled in between alternate AET elements and a check made against the current depth value for every pixel.

2.3 Previous work using critical points

Scan conversion using critical points first appeared in (Gordon 1983a), and was used in (Gordon et al. 1994) for medical visualization. The technique can be used instead of the standard method (Foley et al. 1990, Sect. 3.6.3) for any application, including scanline hidden surface removal with one AET for all the polygons, as in (Foley et al. 1990, Sect. 15.6). Local minima were also used by Sechrest and Greenberg (Sechrest and Greenberg 1982), independently of (Gordon 1983a), to obtain visible surfaces at object resolution. However, their technique is not a scan conversion algorithm, and it requires rather complex data structures. Local minima were also used in computational geometry (Fournier and Montuno 1984; Hertel and Mehlhorn 1985), with results similar to (Hertel and Mehlhorn 1985) obtained independently in (Gordon 1983b).

If n denotes the total number of vertices, c the number of critical points, h the number of scanlines, and n_i the number of edges starting from scanline i, the precise differences between CP and the standard scan conversion are (Gordon et al. 1994):

1. In addition to the original input, the standard algorithm requires $O(n+h)$ extra space, while CP requires only $O(c)$.
2. The time required by CP is only $O(c^2)$ (not counting the time to fill pixels, which is common to both methods), while the time for the regular method is $O(hc + \sum_{i=1}^{h} (n_i \log n_i))$. The c^2 can be reduced to $c \log c$, but this requires a more complex data structure for the AET, and the overhead is unjustified for most applications.

3 CP: Critical points scan conversion

The following description of CP is detailed because we aimed at self-containment, and because seemingly minor changes in the code are problematic. It also differs somewhat from (Gordon et al. 1994) due to the required dovetail process. Assume that a polygon P with n vertices in 2D is given by an array of the coordinates of its vertices in cyclic order – $(X[0], Y[0]), \ldots ,(X[n-1], Y[n-1])$ (a bidirectional cyclic linked list can also be used). In the following, all index calculations are carried out modulo n, i.e., $n = 0 \pmod n$. The polygon edge connecting $(X[i], Y[i])$ with $(X[i \pm 1], Y[i \pm 1])$ is denoted as EDGE$[i, i \pm 1]$.

3.1 Outline of CP

CP first determines the set CR of critical points, sorted by increasing values of y. From every critical point, moving along the list of vertices in either direction can only lead upwards. CR, together with the arrays X and Y, is used instead of the standard edge table in the sweep stage. Logically, P's boundary is made up of monotonic *sections*; two sections start at every critical point, and two sections terminate at a local maximum. Note that this step is carried out in object space, so it can be done as a device-independent preprocessing step.

At the sweep stage, the AET is started from the lowest critical point and advances one scanline at every stage. Every element of the AET describes the intersection of the current scanline with P's edges, and the AET is ordered by increasing values of x. The key idea behind CP is that the elements of the AET are used for entire monotonic sections along P's boundary; i.e., they remain active for as long as the section they describe continues to rise. (In the standard approach, all edges are inserted and eventually deleted from the AET.) At every new scanline, the following occurs:

1. From every element on the AET, the monotonic section is followed upwards until it either meets the new scanline or turns down before reaching it. In the first case, the information in the element is updated (it may now describe a different edge along the same section), in the second case, the element is deleted from the AET.
2. If the polygon is not simple, i.e., edges may intersect, then the AET is reordered. For simple polygons, this step is not necessary.
3. If there are new critical points between the old scanline and the new, the monotonic sections starting from them are followed up until they intersect the new scanline, and new elements are added to the AET. (It is possible, though, that a new monotonic section turns down before reaching the new scanline; in that case, the section is abandoned.)
4. Screen pixels are filled in between alternate pairs of the AET's elements in the usual manner.

The algorithm terminates when the AET is empty. Figure 1 shows a polygon, its critical points, and the structure of the AET at consecutive scanlines.

3.2 Determining the critical points

The procedure DETCR determines the set CR of critical points. Care is needed here because two or more consecutive vertices may have the same y value.

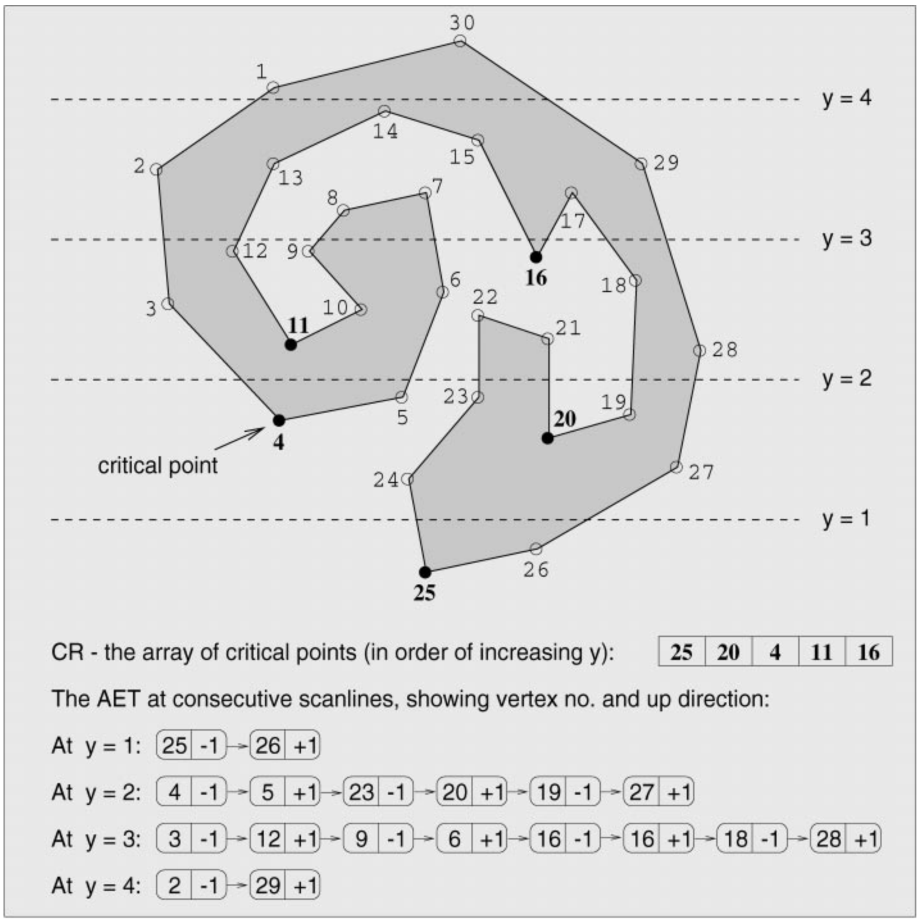
30
1
y = 4
14
2
15
29
13
8
7
17
12
9
y = 3
16
18
3
6
22
10
11
21
28
y = 2
23
19
5
20
4
critical point
24
27
y = 1
26
25
CR - the array of critical points (in order of increasing y): 25 | 20 | 4 | 11 | 16
The AET at consecutive scanlines, showing vertex no. and up direction:
At y = 1: 25 -1 → 26 +1
At y = 2: 4 -1 → 5 +1 → 23 -1 → 20 +1 → 19 -1 → 27 +1
At y = 3: 3 -1 → 12 +1 → 9 -1 → 6 +1 → 16 -1 → 16 +1 → 18 -1 → 28 +1
At y = 4: 2 -1 → 29 +1

Fig. 1. A polygon with its critical points, showing the AET at consecutive scanlines

A secondary comparison on x does not help because horizontal edges may overlap (either due to unusual input or because a planar polygon is viewed horizontally). Our solution is to loop around the polygon with a boolean flag, which is set *True* whenever y properly decreases. When y properly increases (and the flag is *True*), we have a critical point. It may even be necessary to loop almost twice around the polygon (in case the the flag is *True* when we come back around to the first vertex). Pseudocode for DETCR appears in Appendix A.

We henceforth assume that CR is an array of the indices of the critical points, sorted by y, and that c is the number of critical points. This means that the critical points, in order of increasing y, are the vertices $(X[\mathrm{CR}[1]], Y[\mathrm{CR}[1]]), \ldots, (X[\mathrm{CR}[c]], Y[\mathrm{CR}[c]])$. If $c = 0$, then the polygon is degenerate and there is nothing to scan convert.

3.3 The AET: Its structure and update

Throughout the following, we assume that $y = H$ is the current scanline, and that H is a globally known variable. Each element K of the AET has the following fields:

```
K.IND - Index of the highest vertex on the
        section whose y-value is <= H,
        i.e., Y[K.IND] <= H and
        Y[K.IND + K.DIR] > H.
K.DIR - Direction of advance along the arrays X
        and Y leading upwards along the
        monotonic section; K.DIR = +1 or
        K.DIR = -1. This means that
        Y[K.IND + K.DIR] > Y[K.IND].
K.XC  - x-coordinate of the intersection of
        the scanline y = H with the current edge
        EDGE[K.IND, K.IND + K.DIR]. This field
        must be in floating point.
K.SLP - The inverse of the slope of the current
```

```
      edge: (X[K.IND + K.DIR] - X[K.IND]) /
      (Y[K.IND + K.DIR] - Y[K.IND])
      This is also a floating point field.
K.NXT - Pointer to the next element of the AET.
```

For reasons of efficiency, we assume AET itself is also a structure of the same type as above. This simplifies the coding and makes deletions possible in $O(1)$ time without the necessity of a doubly linked list for the AET (the DELETE procedure gets a pointer to the predecessor of the element to be deleted).
The purpose of the procedure MOVEUP is to ascend upwards along a monotonic section from a given vertex until the section meets a given scanline. If the section turns down before reaching the scanline, an indication is given. Pseudocode for MOVEUP can be found in Appendix A.

3.4 The complete program CP

The complete program CP for a single polygon is presented below. We assume the existence of the following procedures:

```
1. INSERT(J, DI, XX, SL): Creates a new element
         K with K.IND = J, K.DIR = DI, K.XC = XX,
         and K.SLP = SL; the procedure inserts
         K into the AET so that the AET remains
         sorted by XC. In case of equality,
         a secondary comparison is done on the
         field SL. This eliminates the need
         for sorting if there are no edge
         intersections.
2. DELETE(K): Deletes an element K from the AET.
         This can be done in O(1) time provided
         the actual parameter is a pointer to
         K's predecessor.
3. FILL: Procedure to fill in between
         alternate pairs of AET elements. In
         keeping with standard practice, if K1
         and K2 are consecutive AET elements
         which delineate a span of pixels to
         fill, then FILL paints in all
         pixels(I,H) such that
         K1.XC <= I < K2.XC, where y = H is the
         current scanline. The pixels are
         painted in provided they are
         inside the viewport
```

The following is the complete critical points program. The main work is done by the procedure POLINE, which updates the AET towards a new scanline and renders a single scanline of the polygon. POLINE is detailed in Appendix A.

```
Program CP (Critical Points)
/***************************
P is the polygon to be scan-converted, X and Y
are its coordinate arays, CR is the array of the
indices of P's critical points, and c is the
number of critical points.
*/
{ DETCR; if ( c = 0 ) return;
  /* if no crit. pt. then polygon is degenerate
  and ignored */
  AET = empty; /* initialize AET to empty */
  LS = y-value of lowest scanline in viewport;
  HS = y-value of highest scanline in viewport;
  H = max( LS, ceiling(Y[CR[1]]) );
  /* first relevant scanline */
  ic = 1; /* index (in array CR) of
  current crit. pt. */
  do { POLINE; H = H + 1; }
  until ( AET is empty or H > HS )
} /* end of CP */
```

Several extensions of CP are possible (see Gordon et al. 1994):

1. Self-intersecting polygons: since edges may intersect, sorting should be added to POLINE (see comment in POLINE in Appendix A)
2. Polygons with holes: the holes should also be in cyclic order
3. Scan converting several polygons simultaneously: the AET holds all the edges, and sorting is necessary since edges may intersect
4. Hatching in vector graphics: H advances by $\Delta H \neq 1$ and a minor change is needed in MOVEUP
5. Special requirements, such as might be required by medical applications (Gordon et al. 1994).

4 Converting algorithms to scanline order

In this section we describe the general scanline principle and give two important examples of its application: a single scanline depth buffer, and the scanline display of BSP trees (which can also work for any list-priority algorithm (Foley et al., Sect. 15.5).

4.1 The general scanline principle

We now assume that we have a set of polygons PSET. For each polygon P in PSET, we denote by $P.*$ its associated data and procedures, e.g., $P.X$, $P.n$, *P*.DETCR, etc. *P*'s procedures now operate exclusively on *P*'s data. *P* also has any other data relevant to the application, such as 3D data and lighting information. In addition, *P*.FILL is modified according to the particular algorithm that is being processed. The general algorithm is specified below. The scanline generation of the image is obtained by an outer loop on the scanlines and an inner loop on the polygons which calls on *P*.POLINE for every polygon P. $P.POLINE$ renders a single scanline of polygon P and the algorithm uses it in a dovetail fashion on all the polygons. Details of implementation will vary with the particular algorithm being modified.

```
Outline of the General Scanline Algorithm
/***********************************/
{ Determine order in which to display polygons
  /* if relevant to algorithm */
  for (every polygon P) /* initialize P */
     { P.DETCR; P.AET = empty; P.ic = 1; }
  for (every scanline H)
  /* outer loop on scanlines */
     { initialize the scanline;
     /* this depends on the algorithm */
       for (every P intersected by H,
       in the order required by the algorithm)
          P.POLINE;
          /* inner loop on polygons */
     }
}
```

4.2 The S-buffer: a scanline depth buffer

The application of the principle to the Z-buffer produces an algorithm which requires a depth buffer of one scanline. We use a 1D array S whose size is equal to the number of pixels in a scanline. Initialializing a scanline simply means setting all values of S to infinity. *P*.FILL is modified so that a pixel is set only if its depth value is less than the corresponding value in S (in which case S is also updated). No particular order between the polygons is required here, but a rough front-to-back order could be more efficient since fewer pixels would have to be set.

If the scene contains many small polygons, we can improve the algorithm's efficiency by only working at each scanline with the polygons intersected by the scanline. We create a polygon table (PT), with a list for every scanline. Every polygon P is added to the list at line $ceiling(P.Y[P.\mathrm{CR}[1]])$ – this is the first scanline at which the polygon becomes active (the list is not ordered, and each polygon is added to just one list). Also, we maintain an active polygon table (APT), which is a list of all the polygons intersected by the current scanline. A polygon P is added to the APT when the current scanline reaches its position in PT, and deleted from the APT when *P*.AET becomes empty. The detailed S-buffer algorithm is presented in Appendix A.

4.3 Displaying BSP trees in scanline order

Back-to-front display of BSP trees is simple. Once the viewpoint is determined, we create a list of the polygons in back-to-front order relative to the viewpoint – this is the first step in the general algorithm. This step uses the standard method of traversing a BSP tree: starting from the root, recursively traverse the far subtree of the node, then the node itself, and then the near subtree. Let $P_1, \ldots, P_n$ denote the sequence of polygons in the back-to-front order. *P*.FILL is unmodified, i.e., pixels are filled between alternate pairs of elements on *P*.AET. In this application, the inner loop of the general scanline algorithm follows the order of the polygons according to the back-to-front list.

As in the S-buffer, when there are many small polygons, it is more efficient to maintain and use an auxiliary data structure of the active polygons (or APT) at every scanline. Initially, a PT is set up as in the S-buffer. At every scanline, the polygons starting at that scanline are inserted into the APT, and when the scanline passes beyond an active polygon, it is removed from the APT. At every scanline, the APT is traversed in back-to-front order, and one scanline of each active polygon is displayed. Thus, it should be possible to traverse the APT in back-to-front order, i.e., in the same order as $P_1, \ldots, P_n$. Note that the additional data structures take up $O(n)$ space.

We present two choices for the APT. One option is to maintain it as a balanced binary search tree, ordered by the polygons' *index* in the above-mentioned back-to-front order: $P_1, \ldots, P_n$. Inorder traversal of the tree provides the required back-to-front order, and each insertion or deletion takes time $O(\log m)$, where m is the number of elements in the APT. Alternately, the active polygons can be maintained as a doubly linked list, in the same order as the large list. When a new polygon P_i has to be inserted into the APT, the (large) back-to-front list is followed from P_i's position until another active polygon P_j is met. P_i is then inserted into the APT before P_j. If there is no active polygon after P_i, then P_i is added at the end of the APT. Other details of implementation are straightforward. The time to search for a new element's position in the APT takes $O(n/m)$ time on average, where n is the number of polygons and m is the average size of the APT.

As mentioned, the display of BSP trees can be sped up considerably by processing them in *front-to-back* order relative to the viewpoint. However, this requires a data structure that maintains a linked list for every scanline which is a run-length encoding of the pixels that have not been set (see Gordon and Chen 1991 for details). By applying the scanline principle to this algorithm we obtain scanline order and save considerable space since only one list is held in memory at any given time (instead of all the lists for all the scanlines). The following details should be added to the general technique:

- Create a list of the polygons in front-to-back order.
- Initializing a scanline requires the setting of the unset pixel list to empty (only one list is required for the current scanline).
- P.FILL is modified to operate as in the original front-to-back algorithm: a merge-type operation is performed between P.AET and the list of unset pixels; this operation fills the visible parts of P and updates the list.
- An APT, as described above, can also be used.

5 Extensions and applications

This section describes some applications of the scanline mode in general, as well as various extensions of the S-buffer.

5.1 Applications of the scanline mode

1. *Antialiasing by supersampling and averaging.* By generating a few $(2-3)$ lines of a supersampled image, we can do an averaging operation to a obtain an antialiased image one scanline at a time. The memory required by a scanline algorithm is proportional to one scanline of the supersampled image. Clearly, at any given time, only a few lines of the supersampled image need to be kept in memory. Note that in order to get supersampling with an ordinary Z-buffer, we either need a much larger buffer than even the regular one, or we need to use an accumulation buffer which adds several subpixel translations of the viewing matrix together.
2. *Transmission of an image.* The scanline mode enables the transmission of the final image one scanline at a time over any kind of channel without the need to maintain a full image in memory.
3. *Image compression.* Most image compression algorithms require only a few scanlines at a time in memory, for example, run-length encoding, GIF (based on Lempel-Ziv compression), and JPEG (requires eight scanlines at a time). Generating the image in scanline mode is therefore ideal for this purpose.
4. *Web-based applications.* The scanline mode is ideal for graphics over the internet. Besides the small memory requirement, it provides a low-latency input into an image-compression algorithm, from which the compressed image can then be transmitted efficiently over narrow bandwidth channels.
5. *Large plotters.* These may have as many as $10^4 \times 10^4$ pixels.

5.2 Scanline A-buffer and transparency

The A-buffer (Carpenter 1984) is an antialiasing technique that extends the Z-buffer by adding a bit-mask to every pixel. The S-buffer can be extended in the same way, but in this case the bit-masks are added only to the pixels of one scanline. Other than that, the scanline A-buffer would work in the same way as the original full-size A-buffer. Note that the bit-masks are initialized to zero at each new scanline.

In (Carpenter 1984), Carpenter also proposed handling transparent objects by using a sorted list of semi-transparent surface fragments for each pixel. However, the memory requirements for this could be extensive. Fournier and Fussell (Fournier and Fussell 1988) show how a two-pass frame buffer can render total transparencies. (See also (Foley et al. 1990, Sect. 16.5).) However, these methods do not operate in scanline mode, and some of them require the polygons to be given in a certain order.

Transparency is solved easily by extending the S-buffer in a manner similar to Carpenter's suggestion (Carpenter 1984), but carried out in scanline mode. For every pixel in the current scanline, we keep a linked list of all the polygons that are transparent and not blocked by an opaque polygon. The last element in the list is the closest opaque polygon. The list is ordered by increasing z-values. Updating the list is done in an obvious manner and we omit details. The S-buffer is initialized by putting a "background" element on the list with the properties "opaque" and z-value ∞.

5.3 Shadow polygons

Clearly, the S-buffer can be used to generate light buffers which are used for rendering shadows (Foley et al. 1990, Sect. 16.4). Another approach to handling shadows (Appel 1968; Bouknight and Kelly 1970) is to create shadow polygons. For every polygon P, we create a list of all its shadow polygons – these are polygons in the same plane as P, obtained as shadows of other polygons in the scene (provided they are likely to intersect P). These shadow polygons can be processed together with the regular polygons in scanline order. The S-buffer can be extended to handle the shadow polygons by considering all the shadow polygons of a polygon P together with P so that P.AET now includes edges of P's shadow polygons. When rendering P, we know for every pixel exactly which shadow polygons cover the pixel (by maintaining the usual in/out bit vector for every polygon). With this information, the pixel can be rendered correctly. Note that sorting is now necessary in POLINE since some of P's shadow polygons may intersect P or each other.

5.4 Server environment

Assume that the user is visualizing a scene on a remote screen that obtains its image from a server. In such a situation, one would want to minimize the memory requirements from the server, so a software Z-buffer is undesirable. A hardware Z-buffer, if available on the server, could be used, but there could be competing demands for it from other clients, assuming they are all viewing unrelated scenes. As mentioned above, the scanline mode is another desirable property, so the S-buffer is particularly suitable for this situation. Recent practice with graphics over the internet calls for more reliance on software techniques (e.g., by using Java applets) and less use of hardware-based features.

A related situation occurs when several users wish to view the same scene, but they may have screens of different resolutions. A simple solution is for the server to generate the scene at the highest required resolution and do a separate averaging operation for each of the different resolutions (similarly to antialiasing). Alternatively, if a user has a device with some processing power, the averaging operation can be done locally. Resampling at different resolutions is a common technique, but the scanline principle allows us to do it in scanline mode with a low overhead.

5.5 Virtual reality

The S-buffer can be very useful in a multi-user interactive virtual reality environment. Assume that several users wish to share the same virtual environment, observe it from different viewpoints, and interact with the environment and each other. This situation is best handled by a single server which maintains the modifiable database and the virtual positions of the users in the environment. The S-buffer algorithm is suitable for this situation. All the polygons' vertices are common data for all the users, and for each separate user the server needs to hold the sorted lists of critical points (which may depend on the viewpoint), and maintain one S-buffer and AET's for the observable polygons. Each user will have a different view matrix, but there is no need to maintain for him the transformed coordinates of the scene's vertices; instead, the user's matrix can be applied to each vertex of the common data whenever required.

For certain applications, it may be possible to precompute the critical points of some of the polygons and use them for all users. Consider a virtual museum in which the user is restricted so that he cannot tilt his head. Then for some of the vertical polygons, the critical points will always belong to some restricted set. For example, for walls and picture frames, the critical points will always be along the bottom edge. Due to changing perspectives, the choice of which points to use from the restricted set is view-dependent, so it has to be determined at run time.

We should note also that BSP trees are widely used for certain virtual reality applications. The scanline rendering of BSP trees enables the servers to generate, compress, and transmit images with a low overhead.

5.6 Parallel rendering

The S-buffer is suitable for an image-driven parallel processing environment, where each processor is responsible for a span of consecutive scanlines. Determining the critical points of all the polygons is clearly parallelizable, since each polygon is handled separately. At the sweep stage, each processor handles only those polygons that intersect its span of scanlines. The processor sets the variables LS and HS (see Appendix A) to the limits of its span, and then executes the S-buffer algorithm.

6 Evaluation of the S-buffer

There are various ways to compare different algorithms. Qualitative comparisons are concerned with the different properties (e.g., scanline mode). Quantitative comparisons are concerned with time and space comparisons – both theoretical and actual run-times. We compare the following three methods: regular Z-buffer, scanline S-buffer, and the standard scanline coherence algorithm (Foley et al., Sect. 15.6), which we refer to as Scan. We also made run-time comparisons between the S-buffer and state-of-the-art tools on typical workstations.

6.1 Qualitative comparisons

Table 1 summarizes the qualitative differences between the Z-buffer, the S-buffer, and Scan. Positive entries in the table are emphasized by italics. The entry "prohibitive" indicates that while the possibility exists, it is only at a great expense in terms of space or time (see Sects. 2.1, 5.1, and 5.2). Note that with regard to object types, the standard scanline method has been extended to quadrics by Pavlidis (Pavlidis 1985). The S-buffer can probably be extended in a similar manner, though we have not pursued this possibility. With regard to hardware implementation, Scan is not the hardware-oriented Watkins algorithm mentioned in Sect. 2.1.

Table 1. Qualitative differences between the Z-buffer, S-buffer, and Scan

Property	Z-buffer	S-buffer	Scan
Depth buffer	Large	*Small*	*None*
Scanline mode	No	*Yes*	*Yes*
Intersecting objects	*Yes*	*Yes*	Prohibitive
No edge sorting	*Yes*	*Yes*	No
Supersampling	Prohibitive	*Yes*	*Yes*
Transparency	Prohibitive	*Yes*	*Yes*
Online mode	*Yes*	No	No
Hardware implementation	*Yes*	Possible	No
Object types	*All*	Polygons (extensions?)	Polygons and extensions

6.2 Time and space comparisons

In this Section we present the results of time and space comparisons between the various algorithms. The analysis is straightforward, and the analysis of CP can be found in (Gordon et al. 1994). For the Z-buffer, we assume that CP is used for scan conversion. We use the following notations:

P_i	$=$ polygon i
p	$=$ number of polygons
n_i	$=$ number of vertices of polygon i
n	$= \sum_{i=1}^{p} n_i =$ total number of vertices
c_i	$=$ number of critical points of P_i
c	$= \sum_{i=1}^{p} c_i =$ total number of critical points
A_i	$=$ area (number of pixels) covered by P_i
A	$=$ area covered by all polygons (every pixel counted just once)
r	$=$ horizontal (and vertical) resolution (number of pixels in scanline)
S	$=$ total time to sort all edges in Scan

Table 2 summarizes the time and space requirements of the three algorithms. As can be seen, the time differences between the depth methods and the Scan algorithm depend on many factors, and there are situations where depth methods are faster and some where Scan is faster. For a detailed study of the actual differences in run-time between the Z-buffer and Scan, see (Slater et al. 1992). Note, however, that their implementation of the scanline algorithm uses the standard scan conversion, whereas

Table 2. Time and space requirements of the Z-buffer, S-buffer, and Scan

	Z-buffer	S-buffer	Scan
Space requirement	$\theta(n+r^2)$	$\theta(n+r)$	$\theta(n)$
Time requirement	$\theta(n+\sum_{i=1}^{p}(c_i^2+A_i))$		$\theta(n+c^2+A+S)$

the time of $\theta(n + c^2 + A + S)$ assumes the use of CP. As noted earlier, all occurrences of c^2 could be replaced by $c \log c$, but this would be impractical for most applications, since the inherent overhead of the more complex data structures (particularly for small c) would increase the actual execution times.

6.3 Implementation and run-time results

From a user's point of view, if the qualitative differences between the various algorithms are not important, it makes sense to compare the S-buffer (or any other algorithm) with the standard tools available on his workstation. With this in mind, we implemented the S-buffer on Silicon Graphics workstations and compared it with the OpenGL routines for hidden surface removal, using the GLU library tools for the non-convex polygons. The methods were compared on two different machines: a low-end Indy workstation with 32 MB memory and 4600PC processor running at 100 Mhz (without a hardware Z-buffer), and an Indigo2 High-Impact with 64 MB memory, hardware Z-buffer, and R4400 processor running at 250 Mhz.

Experiments were run on three different types of polygons: convex, non-convex with few $(1-5)$ critical points, and non-convex with many critical points ($c = n/2$ – the worst possible case for CP). Each data set contained 1000 polygons, with the number of vertices (n) varying from 100 to 1000. Such a large number of vertices appear in applications where the polygons are approximations to smooth contours, obtained, for example, by CAD outputs of curved contours or by freehand drawings.

We refer to the OpenGL implementation on Silicon Graphics workstations as SGI. The run-times of the two methods are presented in Table 3 and Table 4

Table 3. S-buffer and SGI run-times (in seconds) for 1000 polygons (Indy)

No. of vertices	Convex polygons		Non-convex with $c = 1$–5		Non-convex with $c = n/2$	
	S-buffer	SGI	S-buffer	SGI	S-buffer	SGI
100	16.55	16.34	17.37	15.17	73.98	40.75
200	17.03	27.47	18.17	26.61	133.77	68.15
300	17.90	36.17	18.11	37.54	182.91	84.74
400	18.43	45.90	18.68	48.35	266.24	109.17
500	18.77	52.76	19.37	57.71	339.97	126.29
600	18.01	59.05	19.46	69.57	414.95	156.72
700	18.43	66.23	19.24	77.83	549.43	176.41
800	19.06	73.83	20.18	89.24	655.13	208.86
900	18.77	81.16	19.68	94.01	785.77	224.91
1000	20.58	89.03	20.34	108.21	914.32	243.29

Table 4. S-buffer and SGI run-times (in seconds) for 1000 polygons (Indigo2)

No. of vertices	Convex polygons		Non-convex with $c = 1$–5		Non-convex with $c = n/2$	
	S-buffer	SGI	S-buffer	SGI	S-buffer	SGI
100	4.62	0.58	5.06	5.92	25.25	7.95
200	5.10	0.99	5.70	11.71	50.97	17.89
300	5.38	1.36	5.71	18.72	75.93	29.99
400	6.03	1.77	5.97	23.85	117.47	44.55
500	5.95	2.15	6.29	29.82	155.55	82.89
600	5.65	2.48	6.24	36.70	190.49	78.60
700	5.87	2.87	6.40	42.19	239.24	105.15
800	6.29	3.25	6.64	48.29	284.88	129.87
900	6.04	3.64	6.58	54.45	322.89	150.27
1000	6.46	4.04	6.59	61.18	363.02	196.81

Table 5. Run-time ratios of the S-buffer over SGI standard methods (S/SGI) for $n = 100–1000$

	Convex polygons	Non-convex with $c = 1–5$	Non-convex with $c = n/2$
Indy	1.01 – 0.23	1.14 – 0.18	1.81 – 3.75
Indigo2	8.10 – 1.59	0.85 – 0.10	3.17 – 1.84

for the Indy and the Indigo2, respectively. The results are summarized in Table 5, which shows the ratios of the S-buffer run-times over the SGI run-times (denoted S/SGI). The left value in each column is S/SGI for $n = 100$, and the right value is for $n = 1000$. As expected, in some cases the S-buffer did worse than SGI and in some cases it did better, even on the Indigo2 with the hardware Z-buffer.

Since $c = n/2$ is the worst case for CP, the third column provides an upper limit on the performance of the S-buffer relative to SGI. For convex polygons, the S-buffer fares better than SGI on the Indy, but worse than SGI on the Indigo2. This is due to the hardware Z-buffer of the Indigo2. For non-convex polygons, however, the SGI methods first triangulate the polygons using the auxiliary library routines of the GLU. This computational step, for non-convex polygons with few critical points, results in a better performance of the S-buffer over the SGI. As can be seen from Table 3 and Table 4, there is only a negligble difference between the S-buffer's performance on convex and on non-convex polygons with few critical points. However, the SGI methods perform better in the S-buffer's worst case ($c = n/2$).

Generally, the results indicate that the S-buffer performs best when n is large, and can even beat the expensive hardware. Large values for n are typically obtained when the polygon is an approximation to a smooth contour: n increases with the accuracy of the approximation, but the number of critical points remains fixed, since they depend only on the smooth contour. It can also be seen from Table 3 and Table 4 that the run-times of the S-buffer increase very slowly with n when the number of critical points is small.

In practice, the scanline principle has proved to be simple and robust. The S-buffer was implemented in an undergraduate first course on graphics, and the scanline display of BSP trees was used for supersampling and antialiasing in undergraduate projects.

7 Conclusions

We have presented a simple method for the efficient conversion of any hidden surface algorithm into scanline mode, provided the algorithm is based on polygon scan conversion. Scanline mode is a property with many useful applications, such as supersampling and averaging with a low overhead and remote visualization over the web, since it enables the efficient generation, compression, and transmission of rendered images. The scanline principle is based on the "critical points" method of polygon scan conversion, and therefore produces more efficient scanline algorithms than those that would be obtained by using the standard scan conversion.

When applied to the Z-buffer algorithm, the scanline principle converts it into the S-buffer, which has the robustness of the Z-buffer but not its memory requirement. This property makes it extremely attractive in situations where space is at a premium. Extensions of the S-buffer include efficient rendering of transparent objects, and using the A-buffer in a scanline mode. Furthermore, since the algorithm is based on critical-points scan conversion, it performs particularly well on polygons with many vertices (but without too many critical points), even outdoing expensive hardware. Such an application occurs when the polygons are approximations to smooth contours. Other extensions include parallel rendering – a useful property as multi-processor workstations become commonplace. The S-buffer is also suitable for interactive multiple-user virtual reality environments due to the low memory overhead per user.

Another important application of the scanline principle is the conversion of list-priority algorithms, such as the display of BSP trees, into scanline mode, in either the usual back-to-front order, or the more efficient front-to-back order. This application is useful when BSP trees are used for scene representation, as in virtual walkthroughs and similar setups.

```
Procedure DETCR
/**************
CAND is a boolean flag that indicates whether the current vertex
is a candidate for a critical point. CAND is set True whenever y
decreases but is unchanged when the y-values are equal. Whenever
y increases and CAND is True, we have a critical point.
*/
{ CAND = False; set CR to empty;
  for ( i = 0 to n-1 )
       { if ( Y[i] > Y[i+1] ) CAND = True; /* i+1 is a candidate */
         else if ( Y[i] < Y[i+1] and CAND ) /* i is a critical pt. */
                   { add i to CR; CAND = False; }
       }
  if (CAND = True) /* vertex with index 0 is a candidate */
     for ( i = 0 to n-2 )
          if ( Y[i] > Y[i+1] ) break; /* no crit. pt. - exit from loop */
          else if ( Y[i] < Y[i+1] ) /* i is a crit. pt. */
                    { add i to CR; break; }
  Sort CR by increasing values of y;
} /* end of DETCR */

Procedure MOVEUP(J, DI, XX, SL)
/*******************************
J is the current index from which to start the search. J will (possibly)
change to become the index of the highest vertex with y <= H. y = H is
assumed to be the globally known current scanline. If the polygon section
turns down before reaching H, J is set to -1. DI = +1 or -1 and indicates
the direction of advance on the polygon arrays X and Y. XX and SL are re-
spectively the x-coordinate of the intersection with y = H and the inverse
slope of EDGE[J, J+DI]. SL is used to compute the intersection efficiently
by adding SL to XX, but if the edge changes, or if this is the first call to
MOVEUP with a new monotonic section (in which case SL is initially set to 0),
then XX and SL are computed in the usual manner. MOVEUP assumes Y[J] <= H.
*/
{ JOLD = J; /* save original index */
  J1 = J + DI /* index of next vertex in direction DI */
  while ( Y[J1] <= H )
      { if ( Y[J1] < Y[J] ) /* polygon section turns down */
             { J = -1; return; }
        else { J = J1; J1 = J + DI; } /* moveup */
        /* note that MOVEUP continues along horizontal edges until the polygon
           section either rises and meets y=H, or turns down. Any changes in
           the inequalities in the code will produce unexpected results.
        */
      } /* end of while reached - intersection found; compute new
           intersection and (possibly) new slope */
  if ( SL = 0 or J != JOLD ) /* new vertex */
     { SL = (X[J+DI] - X[J]) / (Y[J+DI] - Y[J]); /* new slope */
       XX = X[J] + SL*(H - Y[J]); /* new intersection */
     }
  else XX = XX + SL; /* same edge - calculate XX more efficiently */
  return;
} /* end of MOVEUP */
```

```
Procedure POLINE
/***************
y = H is the globally-known current scanline. POLINE updates the AET
to "meet" the current scanline and calls FILL to fill the pixels.
Variable ic is initialized in the main program and updated in POLINE.
*/
{ /* update or delete current elements on the AET: */
  for(each element K on the AET)
      { MOVEUP(K.IND, K.DIR, K.XC, K.SLP);
        if (K.IND = -1) DELETE(K);
      }
  /* if there are intersecting edges, sort AET here */
  /* add new elements to AET from the critical points: */
  while ( ic <= c and Y[CR[ic]] <= H )
      { /* follow both directions from the crit. pt.: */
        for(DI = -1 to DI = +1 step 2)
            { J = CR[ic]; SL = 0.0;
              MOVEUP(J, DI, XX, SL);
              if ( J > -1 ) INSERT(J, DI, XX, SL);
            }
        ic = ic + 1; /* advance to next crit. pt. */
      }
  FILL; /* fill in pixels between alternate pairs of AET elements
} /* end of POLINE */

Procedure S-buffer
/*****************
PT is an array of lists (of polygons) whose size is equal
to the number of scanlines. APT is a list of polygons.
*/
{ /* initialization stage: */
  LS = y-value of lowest scanline in viewport;
  HS = y-value of highest scanline in viewport;
  /* initialize PT: */
  for ( line = LS to HS ) initialize PT[line] to empty;
  for(every polygon P in PSET)
      { P.DETCR; /* determine P's crit. pts. */
        if ( P.c = 0 ) continue;
        /* if P is degenerate, eliminate it from further consideration */
        line = ceiling( P.Y[P.CR[1]] );
        if ( line > HS ) continue; /* P is above viewport, ignore it */
        if ( line < LS )
           if ( P is completely below LS ) continue;
              /* P is below viewport, ignore it */
           else line = LS;
        add P to PT[line];
        P.AET = empty;
        P.ic = 1; /* index of P's first crit. pt. */
      }
  /* if the polygons are expected to occupy just a restricted
     number of scanlines, LS and HS can be modified according
     to the lowest and highest y-values of the polygons.
  */
  /* sweep stage: */
  for( H = LS to HS )
      { /* outer loop on the scanlines */
        initialize S (the scanline S-buffer) to infinity;
        set all pixels in scanline to background color;
        add PT[H] to APT; /* add new active polygons */
        for(every P in APT)
            { /* inner loop on the polygons */
              P.POLINE;
              if ( P.AET is empty ) remove P from APT;
            } /* end of polygon loop */
      } /* end of scanline loop */
} /* end of S-buffer */
```

Acknowledgements. The authors wish to thank G. Elber for his comment about scanline Z-buffers for convex polygons; this led to using CP for arbitrary polygons. Thanks are also due to the anonymous referees whose comments have greatly improved the usefulness, completeness and presentation of the paper. This research is a part of the first author's MA thesis, carried out under the second author's supervision at the University of Haifa.

Appendix A. detailed algorithms

Note that all index calculations are carried out modulo n, where n is the number of vertices of the polygon.

References

1. Appel A (1968) Some techniques for shading machine renderings of solids. In: Proceedings of the Spring Joint Computer Conference, Annual Conference. Thompson Book Co., Washington, DC, pp 37–45
2. Bouknight WJ, Kelly KC (1970) An algorithm for producing half-tone computer graphics presentations with shadows and movable light sources. In: Proceedings of the Spring Joint Computer Conference, AFIPS Press, Montvale, NJ, pp 1–10
3. Carpenter L (1984) The A-buffer, an antialiased hidden surface method. Comput Graph (Proc. ACM SIGGRAPH Conf.) 18:103–108
4. Crocker GA (1984) Invisibility coherence for faster scanline hidden surface algorithms. Comput Graph (Proc. ACM SIGGRAPH Conf.) 18:95–102
5. Foley JD, van Dam A, Feiner SK, Hughes J (1990) Computer graphics: principles and practice. Addison-Wesley, Reading, Mass., 2nd edition
6. Fournier A, Fussell D (1988) On the power of the frame buffer. ACM Trans Graph 7:103–128
7. Fournier A, Montuno DY (1984) Triangulating simple polygons and equivalent problems. ACM Trans Graph 3:153–174
8. Fuchs H, Kedem ZM, Naylor BF (1980) On visible surface determination by a priori tree structures. Comput Graph (Proc. ACM SIGGRAPH Conf.) 14:124–133
9. G Hamlin Jr, Gear CW (1977) Raster-scan hidden surface algorithm techniques. Comput Graph (Proc. ACM SIGGRAPH Conf.) 11:206–213
10. Gordon D (1983a) The critical points method in computational geometry. Technical report, Department of Computer Studies, University of Haifa, Haifa, Israel
11. Gordon D (1983b) Scan conversion based on a concept of critical points. Technical Report CSD83-1, Department of Computer Studies, University of Haifa, Haifa, Israel
12. Gordon D, Chen S (1991) Front-to-back display of BSP trees. IEEE Comput Graph Appl 11:79–85
13. Gordon D, Peterson MA, Reynolds RA (1994) Fast polygon scan conversion with medical applications. IEEE Comput Graph Appl 14:20–27
14. Hertel S, Mehlhorn K (1985) Fast triangulation of the plane with respect to simple polygons. Information and Control 64:52–76
15. Hill S (1995) The lazy z-buffer. Information Processing Lett 55:65–70
16. Myers AJ (1975) An efficient visible surface program (report to the National Science Foundation). Technical Report DCR 74-00768 A01, Computer Graphics Research Group, Ohio State University, Columbus, Ohio
17. Newell ME, Newell RG, Sancha TL (1972) A solution to the hidden surface problem. In: ACM National Conference Proceedings, Annual Conference. ACM, New York, pp 236–243
18. Newman WM, Sproull RF (1973) Principles of Interactive Computer Graphics (1st edn). McGraw-Hill, New York
19. Newman WM, Sproull RF (1979) Principles of Interactive Computer Graphics (2nd edn). McGraw-Hill, New York
20. Pavlidis T (1985) Scan conversion of regions bounded by parabolic splines. IEEE Comput Graph Appl 5:47–53
21. Rogers DF (1985) Procedural Elements for Computer Graphics. McGraw-Hill, New York
22. Sechrest S, Greenberg DP (1982) A visible polygon reconstruction algorithm. ACM Trans Graph 1:25–42
23. Séquin CH, Wensley P (1985) Visible feature return at object resolution. IEEE Comput Graph Appl 5:37–50
24. Slater M, Drake K, Davison A, Kordakis E, Billyard A, Miranda E (1992) A statistical comparison of two hidden surface techniques: the scan-line and Z-buffer algorithms. Comput Graph Forum 11:131–138
25. Watkins GS (1970) A real-time visible surface algorithm. PhD thesis, Technical Report UTEC-CSc-70-101, NTIS AD-762 004, Computer Science Department, University of Utah, Salt Lake City, Utah

ELLA BARKAN received B.A. and M.A. degrees in Mathematics with Computer Science from the University of Haifa in 1994 and 1998, respectively. Her research and professional interests include computer graphics, computer vision, and image processing. Since 1997, she has been a research staff member of the IBM Haifa Research Laboratory, which is affiliated with the IBM Research Division. Currently, she is engaged in advanced research and development within the image processing group of the multimedia department.

DAN GORDON received B.Sc. and M.Sc. degrees in mathematics from the Hebrew University of Jerusalem, and the D.Sc. degree in mathematics from the Technion – Israel Institute of Technology. He is a senior lecturer of computer science at the University of Haifa, currently on sabbatical at the Technion. His research interests include computer graphics, as well as parallel techniques in medical image reconstruction and reconfigurable processor arrays. Dan Gordon is a member of the ACM and ACM-SIGGRAPH.