Volume 20 (2001), number 4 pp. 257–269 COMPUTER GRAPHICS *forum*

CP3: Robust, Output-sensitive Display of Convex Polyhedra in Scanline Mode

Ella Barkan[1] and Dan Gordon[2]

[1]IBM Haifa Research Laboratory, Matam Technology Center, Haifa 31905, Israel
[2]Department of Computer Science, University of Haifa, Haifa 31905, Israel
ella@haifa.vnet.ibm.com, gordon@cs.haifa.ac.il

Abstract

A new technique is developed for displaying disjoint convex polyhedra. The method has the following properties: It is output-sensitive, displays the objects in scanline mode, and it is naturally robust. There is no complex data structure uniting the different polyhedra, so dynamic insertions and deletions are simple. Its robustnes is based on a novel method of comparing depths by representative "axes" of objects instead of surfaces. The method is based on two extensions of the "critical-points" method for polygon scan conversion: One extension allows the efficient display of planar graphs in scanline mode, and another extension is into the third dimension. Test runs indicate that it compares extremely favorably with other methods that operate in scanline mode, as well as with standard software and hardware techniques of medium-level workstations.

Keywords: Convex Polyhedra, Scanline Mode, Robustness, Output-sensitive, Critical Points, Scan Conversion, Planar Graph

1. Introduction

Computer graphics abounds with many different display methods which differ from each other in various characteristics and suitability of application. The following are some of the properties that need to be considered when choosing a display method:

- *Speed of display*.
- *Quality of display*.
- *Robustness*: Some display methods are sensitive to minor intersections, resulting in various display artifacts. Such intersections can occur as a result of numerical inaccuracies. The ability of an algorithm to overcome such problems is known as robustness.
- *Scanline mode*: This means that the image is obtained in scanline order and the memory required by the algorithm (exclusive of the scene data) is proportional to one scanline. This property has many applications, such as efficiency of supersampling and antialiasing, generating input to various image-compression methods, etc.
- *Dynamic changes*: Some display methods (such as BSP-trees) rely on data structures which incorporate all the objects in the scene. This makes it difficult to make dynamic insertions and deletions of objects from the scene.
- *Generality of input*: This refers to the types of objects being displayed and to any restrictions placed on them, such as non-intersection.
- *Output-sensitivity*: Informally, this term refers to the property that most of the display time is proportional to the complexity of the final image, and only a small part of the time depends on the complexity of the scene (the 3D data). This property is particularly important when complex rendering is involved.

In this paper we present a new display method for disjoint convex polyhedra that has the following properties: It operates in scanline mode, it is output-sensitive, and it is robust. The algorithm does not require any data structures to unify all the polyhedra, and so the data can be modified easily (provided the new data satisfies the constraints). Its

robustness enables it to overcome minor intersections caused by numerical inaccuracies. The algorithm can also produce a Z-buffer of the image; this is useful for creating shadow Z-buffers. By "disjoint convex polyhedra" we mean convex polyhedra with disjoint interiors, since we allow the objects to touch at their surface.

Convex polyhedra occur naturally in many applications. Complex objects are often built up by adjoining convex objects, or they may be the result of some partitioning method that was applied to non-convex polyhedra for some purpose such as collision detection — see for example Chazelle *et al.* [1] and the many references therein. A totally different setting might be the (virtual reality) display of a multitude of convex objects such as a pile of fruit or spheres: The robustness of the new algorithm is particularly suitable for polyhedral approximations to closely packed curved objects, since the approximations may intersect. Its output-sensitivity makes it efficient for displaying complex shading and textures. Another potential area of application is the visualization of complex molecules.

The "built-in" robustness of the new method is due to the fact that relative depths of objects are determined by *representative axes* of the objects and not by their surfaces. To the best of our knowledge, this is a new approach to the problem of hidden surface removal. The new method is based on two extensions of the "critical points" principle [2–4]: One extension allows the scan conversion of planar straight line graphs, and another extension uses critical points of 3D objects in two principal directions. Test runs indicate that the new approach compares extremely favorably with other scanline methods, and, for many-faceted objects, it can even compete with standard software and hardware tools of medium-level workstations.

The rest of the paper is organized as follows: Section 2 provides some background and places the new method in relation to other methods. Section 3 describes the extension of the critical points method to planar graphs, Section 4 describes the new algorithm, **CP3** (3-dimensional critical points), and Section 5 concludes with results and a discussion.

2. Relation to Previous Work

Scanline algorithms for hidden surface removal have received a great deal of attention — see for example Foley *et al.* [5], where the basic scanline algorithm is presented. Any algorithm can produce the image in scanline mode if it uses scratch memory of the same size as the screen, so we take the term "scanline algorithm" to mean that the memory required by the algorithm, exclusive of the scene data, is proportional to one scanline. Such algorithms offer many advantages over other techniques. For example, they enable efficient antialiasing by supersampling and averaging, since only a few scanlines need to be kept in memory at any one time. Another application is to provide input to many image compression algorithms, which usually operate on a small fixed number of scanlines. This property is useful over the web, since it enables the efficient generation, compression and transmission of images over limited bandwidth channels.

Most of these methods use various forms of coherence in order to speed up rendering. One of the earliest methods is Hamlin and Gear's "cross" algorithm [6]. Séquin and Wensley [7] extended this method, and their work describes the Berkeley UNIGRAFIX system. Atherton [8] extends the scanline algorithm to handle constructive solid geometry (CSG) objects, and Patel and Hubbold [9] present a scanline algorithm for CSG objects including polygons, spheres and swept surfaces. Their method is also capable of handling transparencies.

The *scanline principle* [4] is a recently introduced general technique for the efficient conversion of any display algorithm into scanline mode, provided the algorithm is based on polygon scan conversion. This method enables the efficient generation of a scanline depth buffer, as well as various extensions of the Z-buffer, such as the A-buffer [10]. Previous versions of scanline depth buffers were inefficient. Another application of this principle is the scanline display of BSP trees [11], using either the standard back-to-front order of display, or the output-sensitive front-to-back order [12]. The scanline principle is also based on the critical points method of polygon scan conversion, which is outlined below in Section 3.1.

A common feature to most of the scanline algorithms (with the exception of those generated by the scanline principle) is the use of a single AET (active edge table), which is the list of edges intersected by the current scanline. As the active scanline advances, edges may intersect each other. This requires sorting the AET at every scanline, and for dense scenes, it means that a lot of time is wasted on sorting edges of invisible polygons. Among the coherence properties in common use is *object coherence*: When the end of a visible polygon is reached and the adjacent polygon (on the object) is front-facing, then it is taken as the currently visible polygon without any depth calculations. Only when the end of an object is reached, is it necessary to search the potentially visible polygons for the one that is closest. However, this coherence does not eliminate the need for sorting edge intersections.

As noted, the issue of robustness is well-known for the problems it presents to designers of geometric algorithms. A depth buffer overcomes these problems, but at the price of not being output sensitive. Séquin and Wensley [7] write that the problem of robustness requires special treatment, and the UNIGRAFIX system has added features that detect and remove object intersections. Chazelle *et al.* [1] write in their conclusions: "Robustness is (as always) a thorny problem."

The standard scanline method of hidden surface removal [5], which is generally applied to non-intersecting objects, can cause anomalous display if there are minor intersections.

Compared to the above-mentioned methods and others, the new method has the following features:

- The critical points principle is extended to planar graphs and to the horizontal direction (in addition to the vertical direction).
- A separate AET is used for every object, so no sorting of edges is necessary.
- Time spent on hidden objects is kept to a minimum by updating only their silhouettes. This makes the method output-sensitive to a large degree.
- When the end of an object is reached during scanline processing, the next visible object is immediately available from the run-time data structures (whose expected size is logarithmic in the number of objects).
- No complex data structures are used to unify the different objects in the scene, so dynamic modification of the scene is simple.
- The new method of depth comparisons make the algorithm naturally robust.

3. CPG: Critical Points Scan Conversion for Planar Graphs

The critical points method of scan conversion for polygons has been described before [2–4]. Its basic features are a replacement of the standard edge table by a sorted list of the so-called "critical points", which are simply local minima of the polygon with respect to y. In this section we first present the outline of **CP** for polygons, and then describe its extension to planar graphs.

Clearly, a planar graph could be scan converted simply by scan converting all its polygons separately, but this would be much less efficient. Note that this can be done in scanline mode, provided a single AET is used for all the polygons.

3.1. Outline of CP for polygons

Assume that a polygon P is given by a circular array (or a circular doubly-linked list) of its vertices in cyclic order, $V = (v[1], \ldots, v[n])$. **CP** first determines the set CR of critical points, sorted by increasing values of y. From every critical point, moving along V in either direction can only lead upwards. CR, together with V, is used in the sweep stage instead of the standard edge table. 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.

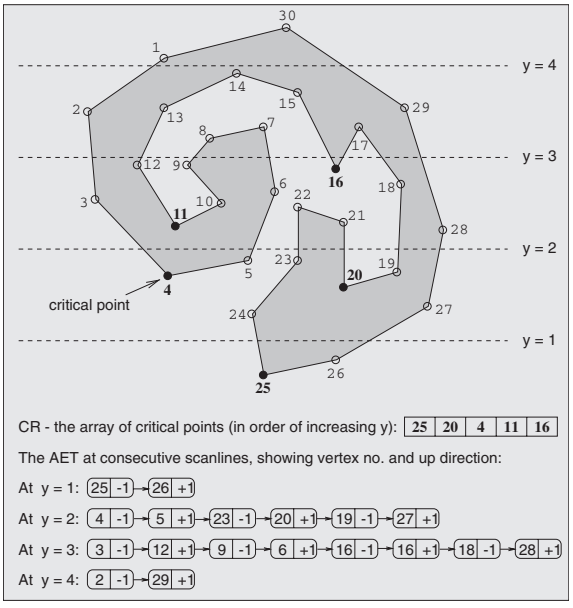

Figure 1: *A polygon with its critical points, showing the AET at consecutive scanlines.*

Figure 1 shows a polygon, its critical points, and the structure of the AET at consecutive scanlines. Shown in the figure is the following information: The index of the closest vertex below the scanline, and the "up" direction, ± 1, which indicates the direction of advance along V in order to get to the next highest vertex (we consider $v[n+1] = v[1]$ and $v[0] = v[n]$). As can be seen, whenever the scanline passes a critical point, two new AET elements are added, and they stay on the AET until the polygon section turns down.

At the sweep stage, the AET (list of active edges) 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:

- 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), and in the second case, the element is deleted from the AET.
- If the polygon is not simple, i.e. edges may intersect,

then the AET is reordered. For simple polygons, this step is not necessary.

- 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.)
- 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.

3.2. Description of CPG

Assume that $G = (V, E)$ is a planar straight line graph. We define a vertex $v \in V$ as a *critical vertex* if, for every u adjacent to v, $(v \cdot y < u \cdot y)$ **or** $(v \cdot y = u \cdot y$ **&** $v \cdot x < u \cdot x)$, where $v \cdot x$ and $v \cdot y$ denote the x and y coordinates of v, respectively.

Our first step, as in **CP** for polygons, is to determine the set CV of all critical vertices, and sort it by y. At the sweep stage, an active edge table, denoted by AET is initialized to empty. At every scanline, pixels are filled between adjacent AET elements, and the AET is updated to meet the next scanline as follows:

If an edge that is currently on the AET intersects the next scanline, its information is simply updated. Otherwise, it is deleted from the AET, and all upwards paths from its upper vertex are followed until all the intersections with $y = H$ are found. This is done by a depth-first-search from the upper vertex — see Figure 2.

The two-dimensional nature of the problem is exploited for increased efficiency as follows: the position of the deleted edge on the AET is maintained. From every vertex that we need to search, the upwards-leading edges are searched in cyclic order from left to right (this is done either by sorting the edges by their inverse slope, or by using the fact that all neighbors of a vertex are in cyclic order). Every edge that is found to intersect $y = H$ is inserted into the AET according the held position, and the position is updated. The depth-first-search strategy, combined with the left-to-right cyclic order, ensures that new edges will be inserted into the AET in the correct positions, and each insertion takes time $O(1)$.

In the example shown in Figure 2, the scanline advances from $y = H1$ to $y = H2$. We see three deleted edges, the leftmost one leading to vertex number 1. A search is initiated from vertex 1 upwards to find all edge intersections with the line $y = H2$, provided these intersections were not found before. Vertex 1 is marked so that no further searches will be initiated from it. The leftmost upwards edge is

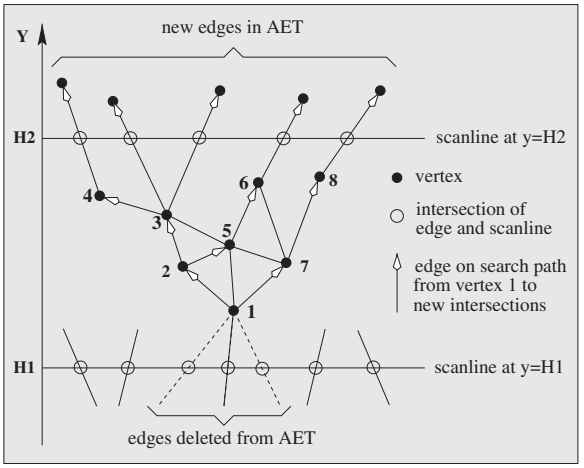

Figure 2: *Critical points for planar graphs: updating the AET from scanline $y = H1$ to $y = H2$.*

$edge(1, 2)$. Since this edge does not meet the new scanline, we mark vertex 2, and recursively continue the search from vertex 2. The recursive procedure produces a depth-first-search, leading to vertex 3, then to 4. From 4, one edge is found to intersect $y = H2$, and the search continues from the other upwards edges from vertex 3. After 3 is exhausted, the search continues along the next edge from 2: $edge(2, 5)$, leading to 6 and to one more edge intersection. Continuing from vertex 1, we see that $edge(1, 5)$ is *not* followed because 5 was marked when it was reached from 2. This leaves only $edge(1, 7)$, leading upwards only to vertex 8 because 5 and 6 are already marked.

The detailed algorithm is presented in Appendix A.

3.3. Analysis of CPG

The time analysis of **CPG** is straightforward. Let n be the total number of vertices, m the number of edges, c the number of critical vertices, and d_i the degree of vertex i. The time to determine and sort the critical vertices is $O(\sum_{i=1}^{n} d_i + c \log c)$. The time to sort the neighbors of vertex i in cyclic order is $O(d_i \log d_i)$. Adding new elements to the AET from a single critical vertex takes time $O(m)$, because a search is made just for the first (leftmost) edge from the critical vertex. Hence, the total extra time for new vertices is $O(mc)$. Note that in theory, we could use some well-balanced data structure for the AET, reducing the time for new vertices to $O(c \log m + \sum_{i=1}^{c} d_i)$, assuming the first c vertices are the critical ones. However, in most practical situations, the overhead of such structures would not be worth the effort.

Every edge is inserted into the AET in time $O(1)$ (except for the first edge from a critical vertex as noted above), and the time for this is $O(m)$. Assuming that the number

of scanlines is h, we need to traverse the AET h times, resulting in time $O(hm)$. Summing up all the times and noting that $\log c$ is bounded by $\log m$ and $m = \Theta(n)$ in a planar graph, we obtain

$$O\left(\sum_{i=1}^{n} d_i \log d_i + n(c+h)\right)$$

when the AET is a simple list, and

$$O\left(\sum_{i=1}^{n} d_i \log d_i + c \log n + hn\right)$$

when the AET is a well-balanced data structure.

In most practical cases, the vertices are of bounded degree, so $\sum_{i=1}^{n} d_i \log d_i = \Theta(n)$, and the times are $O(n(c+h))$ and $O(c \log n + hn)$ for the two choices of the AET, respectively.

4. CP3: 3-dimensional Critical Points Algorithm

4.1. Outline

Assume that we have a set of convex polyhedra. The projection of the visible edges of a convex polyhedron on the screen is a planar straight line graph, with just one critical vertex, and can be rendered by the method **CPG** described in the previous section. **CP3** extends **CPG** by adding geometric data structures to facilitate hidden surface removal. **CP3** operates in scanline mode, hence its outermost external loop is on the scanlines. We use the term "scanplane" to refer to the plane containing a scanline and parallel to the screen z-axis. In moving from one scanplane to the next, we maintain a data structure, called the AOL — "active object list", which is the list of objects intersected by the scanplane. As the scanplane advances, the information in the AOL is updated, and, possibly, new objects are added. The addition of new objects to the AOL is done by a preliminary step of determining and sorting all the vertices of the objects which are minimal with respect to y, called the "y-critical points".

Figures 3 and 4 demonstrate the vertical and horizontal sweeps, respectively. The vertical sweep starts from the lowest y-critical point and advances upwards until all the objects have been processed. At every scanline, the scanplane intersects the objects on the AOL, and the intersections form a set of convex polygons — see Figure 4. Each polygon's minimal vertex with respect to x is called the object's *x-critical point* (equality is resolved by a secondary comparison with respect to z). The objects on the AOL are ordered according to the x-values of their x-critical points. Clearly, this requires reordering the AOL at every new scanline, but the change in order is small because of coherence from one scanline to the next. Note that the x-critical points of the convex polygons are usually not vertices of the 3-dimensional objects. We refer to the straight

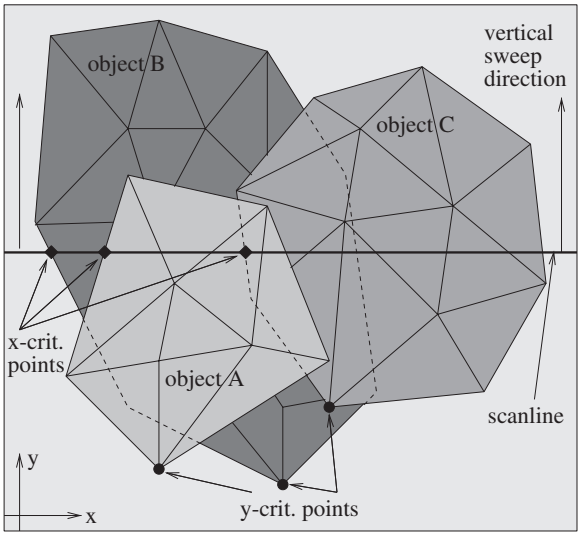

Figure 3: *Vertical sweep of* **CP3**.

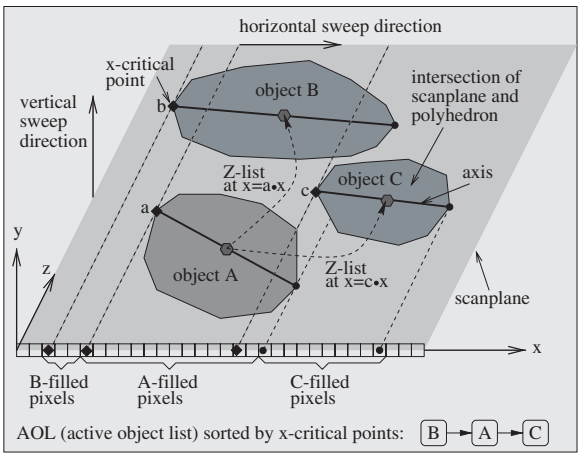

Figure 4: *Vertical and horizontal sweeps of* **CP3***, showing three consecutive positions of the horizontal sweep line.*

line segment joining a polygon's x-critical point with its x-maximal point as the *axis* of the object. These axes are shown in Figure 4.

The algorithm is output-sensitive because when an object is completely hidden from view, we advance in the vertical sweep only along the object's silhouette, formed by the outer edges of the object's graph. When an object that was initially hidden becomes visible or partly visible at a scanline, all the current intersections of the scanline and the edges are found by executing the Meet procedure of **CPG** — see Appendix A. Meet is executed either from the object's y-critical point or from the last scanline at which the object was visible, to the current scanline.

At every scanline, processing starts from the first object on the AOL, which is B in the example shown in Figures 3 and 4. The object (at the scanline) is rendered according to the **CPG** algorithm, until the next object on the AOL is met. This object is A in the example. The process continues until the entire scanline has been processed.

At every position of the scanplane, the visible surfaces are determined by a "critical points" method similar to **CP**, except that the sweep is horizontal (in the x-direction). We denote the horizontal line passing through a pixel (x, y) and parallel to the z-axis as the (x, y)-*line*. Shown in Figure 4 is a scanplane and its intersection with three objects, A, B, C, whose order in the AOL is B, A, C, according to the x-values of their respective x-critical points b, a, c. During the horizontal scan, we maintain a dynamically changing linked list of objects, called the *Z-list*, such that at every pixel (x, y), the Z-list contains exactly those objects whose intersection with the current scanplane satisfies the following conditions:

(1) The (x, y)-line cuts through all the objects on the Z-list.
(2) The objects on the Z-list are ordered by increasing z-values of their intersections with the (x, y)-line.
(3) If A appears before B on the Z-list, then A's maximal x-value is *less than* B's maximal x-value.

Conditions 1 and 2 mean that at every pixel, we need to render the object that is currently first on the Z-list. Condition 3 ensures that there is no object on the Z-list that will be totally hidden by any other object that is currently on the Z-list. This property has the effect of reducing the size of the Z-list. Another way to see it is that the maximal vertices (with respect to x) of the objects on the Z-list are in increasing x-values. These conditions are maintained during the dynamic insertions (and deletions) to the Z-list. Figure 5 shows a typical situation, where an (x, y)-line intersects a set of axes, but only the ones satisfying conditions 1–3 are on the Z-list.

The Z-list is initialized with the first object of the AOL, which is B in the example of Figure 4. Pixels are filled by performing **CPG** on the first object in the Z-list (B), until we reach the projection of the x-critical point of the next object on the AOL (A in the example). An "attempt-to-insert" is performed on A: A is compared with the objects currently on the Z-list, and if it is not completely hidden by them, it is added to the Z-list so that the above conditions are maintained. In the example, A becomes the first object on the Z-list, which is marked in Figure 4 as the "Z-list at $x = a \cdot x$".

Again, pixels are filled by performing **CPG** on the first object in the Z-list (A), until the projection of c, which is the x-critical point of the next object on the AOL, C. C is now inserted into the Z-list, and this causes B to be removed, because C's extent in the x-direction goes beyond

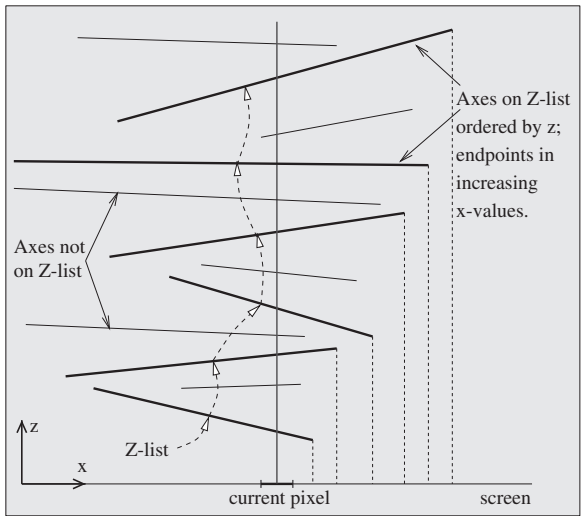

Figure 5: *Only potentially visible axes are on the current Z-list.*

B's extent. The resulting Z-list is denoted "Z-list at $x = c \cdot x$" in the figure. Pixels are still filled from object A, since it is still first on the Z-list. When the end of A is reached, it is removed from the Z-list, and rendering continues from the first element on the Z-list, which is now C.

The insertion of objects into the Z-list is done on the basis of each polygon's axis (defined previously, — see Figure 4). The position of a new object in the Z-list is determined by comparing the z-value of its x-critical point with the z-values of the intersections of the current axes with the (x, y)-line. Once this position is determined, two further operations are performed:

First, the object's axis is compared with the axis of the object preceding it on the Z-list, and if the new object's axis is "hidden", it is not inserted. Second, if the object is inserted, its axis is compared with the axes of the objects following it on the list, and if it is found to hide any of them, then these objects are removed from the Z-list. In the example shown in the figures, this is the reason B is removed when C is inserted. The search for objects to be removed starts from the new object's successor on the Z-list, and it is terminated as soon as one object is found to extend beyond the new object's axis (because the right ends of all the axes are in increasing values of x).

Since **CP3** supplies a depth value for every pixel, it can be used to produce shadow Z-buffers (at any desired resolution) for shadow generation. This option was not implemented in the current version. A detailed description of CP3 is presented in Appendix B.

4.2. Robustness

CP3 compares relative depths of objects by using their axes instead of the usual method of comparing surfaces. An important result of this is the automatic robustness of the algorithm: The surfaces of close objects may even intersect each other, but as long as the axes do not intersect, the image will be displayed as if there was no intersection. Even if an axis of one object penetrates the surface of another object, the algorithm will perform correctly (i.e. as if there was no such intersection).

However, there may be cases when two axes of adjacent objects share the same endpoint, or they may be very close. As a result of numerical inaccuracies, these axes may intersect. We shall show that a simple extra check can be used to handle this problem. There are basically four types of cases where such minor intersections can occur, shown in Figure 6 as cases a, b, c and d. Cases a and b occur when the left endpoint of a new axis is very close to the right endpoint of an axis that is already on the Z-list. Cases c and d occur when two left endpoints are very close and the axes intersect due to numerical inaccuracies. The actual problem that results in cases c and d depends on the relative distance of the right endpoints of the old and new axes. In Figure 6, the dotted part of the new axis indicates that it may be longer than the old, and the undotted part indicates that it may be shorter.

In cases a and b, the error in the resulting image will be very small: In case a the old axis is removed from the Z-list slightly prematurely, and in case b, a small part of the old axis may be displayed when it should have been removed by the new axis. Cases c and d are the real potential problems, because not only will the intersection cause a problem for one of the two axes involved (as specified in Figure 6), but the order on the Z-list will be wrong, causing additional problems for new elements.

The algorithm can be made to operate in an "extra-safe" mode which will handle these problems: When comparing the depth values of (the left endpoint of) a new axis with the depth of an axis currently on the Z-list, a check is made whether the absolute value of the difference is less than some threshold value. If so, the relative positions of the axes are determined according to the *slopes* of the axes relative to the current scanline: The axis with the smaller slope should be considered as being in front of the other one.

The slope comparisons will have the following results in the four cases shown in Figure 6: In case a, the new axis will be placed after the old one on the Z-list; in case b, the new axis will be placed before the old one, and cause the old one to be removed. In case c the slope comparison will consider the new axis to be in front of the old, and in case d, the old axis will be considered as being in front of the new. Clearly, this approach will correct the problem of minor axis intersections.

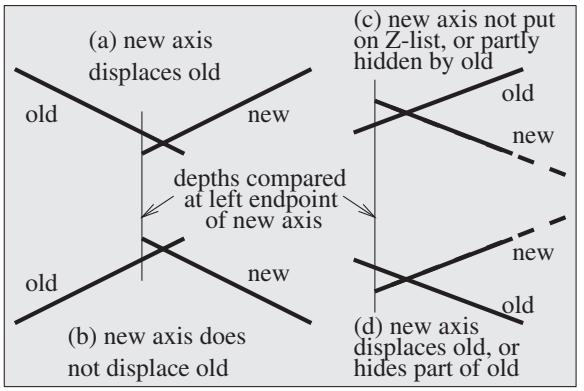

Figure 6: *Potential problems of minor axis intersection; solved by slope comparisons.*

4.3. Performance analysis

The main purpose of the data structures and algorithm of **CP3** is to ensure that only visible portions of the objects will be rendered. It is clear from the description **CP3** that this condition is satisfied. For each visible (or partly visible) object, the display time is determined by the operation of the **CPG** algorithm on that object, and that analysis was presented in Section 3.3. If an object is totally invisible, only its silhouette is updated at each scanline, and the time for this is simply $O(h)$, where h is the number of scanlines.

We therefore consider in this section only the operations required to update the data. Assume that we have n objects in the scene. We shall assume that the objects are evenly distributed in all directions, and let $k = n^{1/3}$. The analysis presented below can be easily modified for other assumptions. Under our assumption, a scanplane cuts through $O(k^2)$ objects, so the expected size of the AOL is $O(k^2)$. There are two alternatives to organizing the objects in preparation for their insertion into the AOL during the vertical sweep:

(1) Sort all the objects by their y-critical points, in time $O(n \log n)$.
(2) Insert the objects into buckets, with one bucket per scanline. Each object is inserted into the bucket corresponding to the first scanline it intersects, i.e. the bucket corresponding to its y-critical point. The objects in every bucket are sorted by increasing x-values of their y-critical points. The expected time to sort one bucket is $O((n/h)\log(n/h))$, and the time for all the buckets is $O(n \log(n/h))$

Every object gets inserted once into the AOL (and deleted once). Under option 1, the total time for insertions is clearly $O(nk^2) = O(n \times n^{2/3})$. Under option 2, the AOL

Table 1: *Run times (in seconds) for flat- and Gouraud-shaded polygons*

No. of polygons	5120		20,480		81,920		327,680	
CP3	0.49	1.36	0.77	1.70	1.41	2.63	3.65	5.21
Scan	2.87	3.38	11.44	12.62	54.31	72.19	329.15	n/a
S-buffer	1.16	2.10	1.83	3.73	4.21	8.23	12.6	23.48
SGI (non-scanline, hardware-assisted)	0.03	0.07	0.24	0.37	1.53	1.60	4.73	6.51

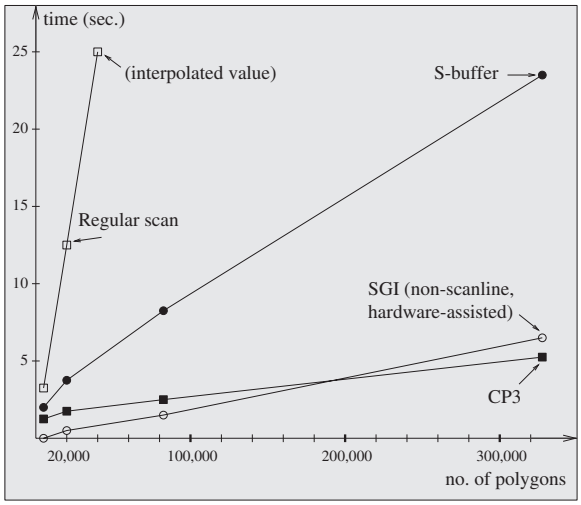

Figure 7: *Plot of run times for Gouraud-shaded polygons.*

is merged at every scanline with the current bucket in time $O(k^2 + n/h)$, and the total for h scanlines is $O(hk^2 + n)$.

Summing up the times under both options, we get:

Option 1: $O(n \times n^{2/3})$,
Option 2: $O(h \times n^{2/3} + n \log(n/h))$.

Another operation related to the AOL is keeping it sorted at every scanline. In most normal scenes, we can assume scanline coherence with respect to the AOL, i.e. only very few elements on the AOL change their position from one scanline to the next. Under this assumption, just $\Theta(k^2) = \Theta(n^{2/3})$ steps at every scanline are necessary to traverse the AOL and exchange the position of a small number of elements (we assume the AOL is a bidirectional list). Even the worst case for this operation is $\Theta(n^{2/3} \log n)$ per scanline, and $\Theta(hn^{2/3} \log n)$ for h scanlines; we shall see below that this is no worse than the expected time for operations on the Z-list.

We now consider the operations related to the Z-list. Consider a line parallel to the z-axis through the center of some pixel. Under our previous assumptions, such a line would pass through an average number of k objects. However, the objects on the Z-list are restricted so that their right endpoints are in increasing order of x. It follows that the expected size of the Z-list is $O(\log k)$ (which is also $O(\log n)$). This is due to the well-known fact that the expected size of a monotonic subsequence of a random sequence is logarithmic in the size of the sequence. Hence, the expected time to insert k^2 objects into the Z-list is $O(k^2 \log k)$, and the time for h scanlines is $O(hk^2 \log k) = O(hn^{2/3} \log n)$.

From the above, we see that the total time for the AOL and Z-list operations is:

Option 1: $O((n + h \log n)n^{2/3})$,
Option 2: $O(hn^{2/3} \log n + n \log(n/h))$.

The factor $hn^{2/3} \log n$, which is due to the Z-list, is common to both options. From this we see that regardless of the relative size of n and h, option 2 is always preferable.

5. Results and Conclusions

5.1. Test results

In order to test our results, we set up a scene of convex polyhedra of random size organized as follows: A cube containing the entire scene was randomly split in two by a plane perpendicular to the x-axis. Each half was then randomly split in two by a plane perpendicular to the y-axis, and then each quadrant was randomly split in two by a plane perpendicular to the z-axis. This process continued until the required number of boxes was generated. Within each box, we then generated a convex polyhedron by starting with an icosahedron and refining the mesh by subdividing each triangle into four triangles, up to a predetermined level.

We compared **CP3** with two other scanline algorithms: One is the standard method [5] with a single AET for all the edges, with object coherence as described in Section 2. The other is a scanline depth buffer derived from **CP**, called the

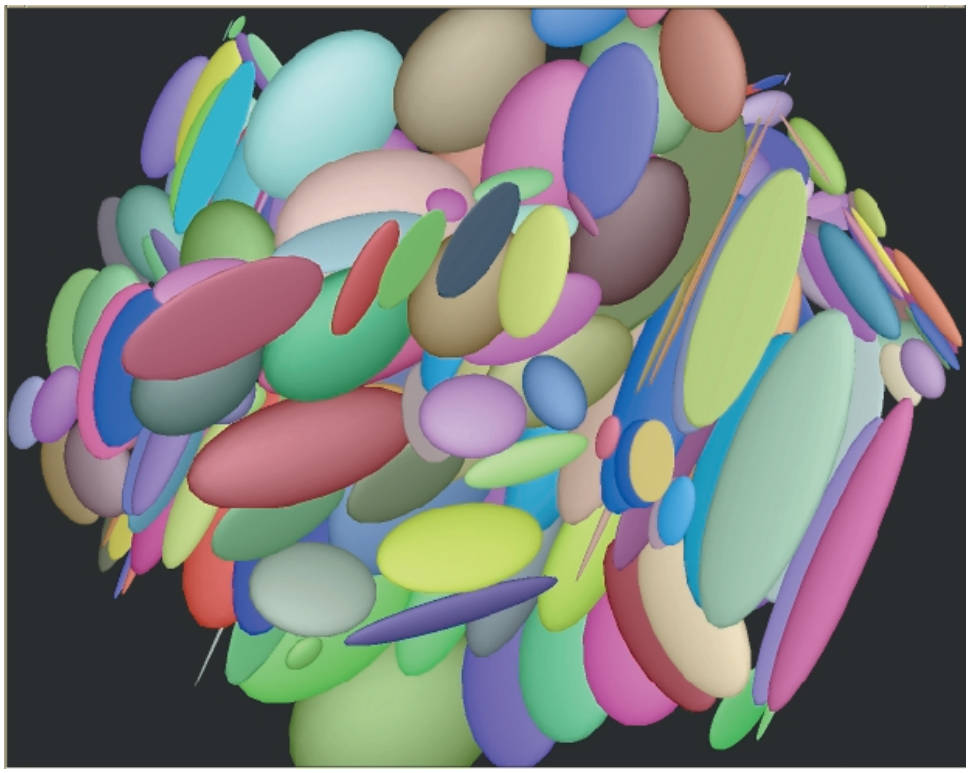

Figure 8: *Gouraud-shaded test image:* 256 *objects, each with* 320 *triangles.*

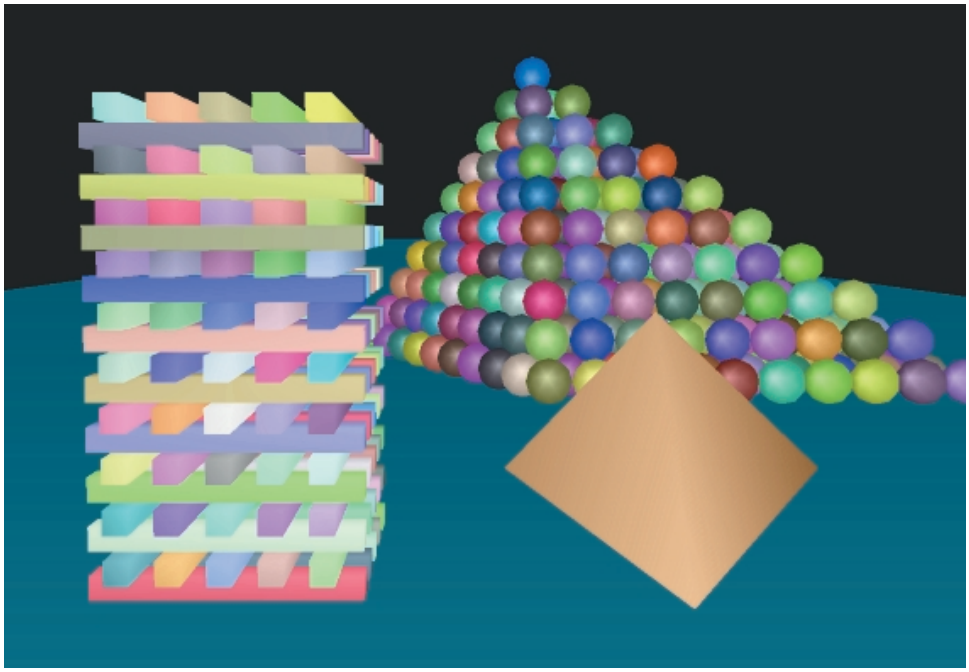

Figure 9: *Gouraud-shaded scene.*

S-buffer [4]. We also present the results using the Silicon Graphics standard hardware and software of the Indigo2 high impact workstation, with 128 MB memory, hardware Z-buffer and 4400 processor running at 250 MHz. However, it should be noted that the SGI results are *not* produced in scanline order.

The four methods, which we refer to as **CP3**, Scan, S-buffer and SGI, were tested with up to 256 objects, with the number of polygons per object varying from 20 to 1280. Table 1 show the results for 256 objects, with flat-shaded and Gouraud-shaded polygons. Figure 8 shows the scene created with 256 Gouraud-shaded objects, each one composed of 320 triangles — a total of 81,920 triangles. Figure 9 shows a Gouraud-shaded scene rendered entirely by CP3.

As expected, the output-sensitive **CP3** produced markedly better results than the other scanline methods, especially when the number of polygons increases. This is even more pronounced when Gouraud shading is used, and the effect can be expected to increase with more complex rendering methods, such as Phong shading or texture mapping. For the case of many polygons and complex shading, **CP3** even outperforms the hardware of the Indigo2. The results for Gouraud shading are presented graphically in Figure 7.

5.2. Conclusions and further research

We have presented a new output-sensitive method for the scanline display of disjoint convex polyhedra. The method is robust, due to the fact that depth comparisons are based on the axes of the objects. There is no data structure unifying the different objects, so dynamic additions and deletions of objects involve minimal overhead. **CP3** uses a new critical points algorithm for scan converting graphs, and is thus particularly useful for rendering curved objects approximated by convex polyhedra. Naturally, this rendering technique can be combined with other methods, such as the Z-buffer, simply by filling up the depth buffer with the depth information obtained from **CP3**. Shadow Z-buffers can also be created and used by **CP3**.

Future research on **CP3** can be expected to proceed in several directions:

- Extension to non-convex objects (without partitioning into convex parts).
- Allowing intersections between objects. The axis technique will not be suitable for this, but **CPG**, the graph scan conversion, can be used. A further direction is the display of Boolean operations between objects, as in CSG (constructive solid geometry).
- Changing level of detail: In order to allow display with a coarser level of detail, the graph scan conversion algorithm should be modified so that fewer edges are considered, with the silhouette edges representing the least amount of required detail.

Acknowledgements

The authors are grateful to the anonymous referees whose comments and suggestions have helped to improve the presentation and quality of this paper. This research is part of the first author's M.A. thesis carried out under the second author's supervision at the University of Haifa.

References

1. B. Chazelle, D. B. Dobkin, N. Shouraboura and A. Tal. Strategies for polyhedral surface decomposition: an experimental study. *Computational Geometry: Theory and Applications*, 7(4–5):327–342, 1997.
2. D. Gordon. Scan conversion based on a concept of critical points. Technical Report CSD83-1. Department of Computer Studies, University of Haifa, May 1983.
3. D. Gordon, M. A. Peterson and R. A. Reynolds. Fast polygon scan conversion with medical applications. *IEEE Computer Graphics & Applications*, 14(6):20–27, November 1994.
4. E. Barkan and D. Gordon. The scanline principle: Efficient conversion of display algorithms into scanline mode. *The Visual Computer*, 15(5):249–264, 1999.
5. J. D. Foley, A. van Dam, S. K. Feiner and J. Hughes. *Computer Graphics: Principles and Practice*. 2nd edn. Addison-Wesley, Reading, MA, 1990.
6. G. Hamlin Jr. and C. W. Gear. Raster-scan hidden surface algorithm techniques. *Computer Graphics (Proc. ACM SIGGRAPH Conf.)*, 11(2):206–213, July 1977.
7. C. Séquin and P. Wensley. Visible feature return at object resolution. *IEEE Computer Graphics & Applications*, 5(5):37–50, May 1985.
8. P. R. Atherton. A scan-line hidden surface removal procedure for constructive solid geometry. *Computer Graphics (Proc. ACM SIGGRAPH Conf.)*, 17(3):73–82, July 1983.
9. M. Patel and R. J. Hubbold. A scanline method for solid model display. *Computer Graphics Forum*, 6(2):141–150, March 1987.
10. L. Carpenter. The A-buffer, an antialiased hidden surface method. *Computer Graphics (Proc. ACM SIGGRAPH Conf.)*, 18(3):103–108, July 1984.
11. H. Fuchs, Z. M. Kedem and B. F. Naylor. On visible surface determination by a priori tree structures. *Computer Graphics (Proc. ACM SIGGRAPH Conf.)*, 14(3):124–133, July 1980.

12. D. Gordon and S. Chen. Front-to-back display of BSP trees. *IEEE Computer Graphics & Applications*, 11(5):79–85, September 1991.

Appendix A: Critical Points for Planar Straight Line Graphs

Note that text following the symbol # is a comment.

```
PROGRAM CPG
# Program to scan convert a planar graph.  pos
# indicates a position on the AET.  We assume
# that the AET has a dummy element at its start.
#
{ Determine the set CV of all critical vertices;
  sort CV by y;
  Initialize AET to empty;
  unmark every vertex in V;
  v = lowest critical vertex;
  H = v.y; # y-coordinate of v.
  H = min(H, lowest scanline in viewport);
  do{ Meet(AET, H); # Update AET to "meet" y=H.
      CPG_Fill;  H = H + 1;
    }
  until(AET is empty OR H > highest scanline)
}

PROCEDURE CPG_Fill
# Routine to fill pixels between every
# two adjacent elements of the AET.

PROCEDURE Meet(AET, H)
# Update the AET to "meet" the scanline y=H.
{ for (every edge e in AET)
    { v = e's upper vertex;
      if (v.y > H)
        update the data in AET's list element;
      else  # e terminates at or before y=H.
        { pos = e's predecessor on the AET;
          Delete e from the AET;
          if (v is unmarked)
            { mark v;  Search(v, H, pos);
              # Recursive search - see below.
            }
        }
    }
  # Add new elements to AET from crit. vertices:
  for (every unmarked v in CV with
       v.y <= H, in ascending order)
    { mark v; Search(v, H, 0); }
} # End of Meet.

PROCEDURE Search(v, H, pos)
# This function searches upwards from a vertex v
# until it finds all edges intersecting y=H which
# lie on an upward path from v. These edges are
# inserted into the AET starting at position pos.
# if pos = 0, the insertion routine will search
# the # AET from the beginning for the correct
# position. On return, Search updates the parame-
# ter pos to # the position (on the AET) of the
# last inserted edge. Search assumes v.y <= H.

{ Let v.up = {w | (v,w) is an edge &
                  v.y <= w.y & w is unmarked};
  Sort v.up by Inv_Slope(v,w);
    # sort in cyclic clockwise order.
  for (every w in v.up in sorted order)
    { if (w.y > H) Insert(v, w, pos);
      else { mark w; Search(w, H, pos); }
    }
} # End of Search.

FUNCTION Inv_Slope(v, w)
# Computes the inverse slope of edge directed
# edge (v,w).  Inv_Slope assumes v.y < w.y
{ return (w.x-v.x)/(w.y-v.y); }

PROCEDURE Insert(v, w, pos)
# This function assumes that the edge (v,w) inter-
# sects y=H and inserts the intersection into the
# AET at position pos. The parameter pos is then
# updated, so that when it is used again, the new
# edge will be the successor of (v,w) on the AET.
# If pos = 0, the position of the element in the
# AET is searched from the beginning, but pos is
# also updated, so it can be used for the next
# insertion.
```

Appendix B: Detailed Algorithms for CP3

```
PROGRAM CP3
# 3D critical points algorithm for display of
# disjoint convex polyhedra in scanline mode.
{ determine and sort the y-critical points;
  determine LS = lowest scanline;
  determine HS = highest scanline;
  AOL.Set_Empty(); # Initialize the AOL to empty.
  for (H = LS; H <= HS; H++) # rendering of one
    CP3_Line(H); # scanline of output image.
}

PROCEDURE CP3_Line(H)
# Variables: current: object currently filled.
#    currx: x-value of first unfilled pixel.
#    nxt_obj: next object on the AOL.
#
{ # Update silhouette of objects
  # on AOL to scanline y=H:
  AOL.Sil_Update(H);
  # Sort AOL by values of updated x-crit. points:
```

```
  AOL.Sort_by_Xcritical();
  # Add new objects to AOL from list
  # of y-critical points:
  AOL.Add_New(H);
  if (AOL.Empty()) # Continue to next scanline.
    return;
  # Initialize Z_List to hold first object on AOL:
  Z_List.Init(AOL.First());
  nxt_obj = AOL.Next_Object(); # next obj. on AOL
  # Determine current x from which to fill pixels:
  currx = AOL.First().xcrit;
  while ( not(Z_List.Empty()) )
    { current = Z_List.First();
      # current object to be filled.
      if (current was not updated)
        current.AET_Update(H);
      if ( nxt_obj == nil OR
           current.xmax < nxt_obj.xcrit )
        { # Fill current object to its end:
          current.CPG_Fill(currx, current.xmax,H);
          currx = current.xmax; # Update currx.
          Z_List.Delete_First();
            # Remove current from Z_List.
          if (Z_List.Empty())
            { if (nxt_obj==nil) # No more objects.
                return;
              else # Next object enters Z_List.
                { Z_List.Add(nxt_obj);
                  currx = nxt_obj.xcrit;
                    # Update currx.
                  nxt_obj = AOL.Next_Object();
                    # New next object.
                }
            }
        }
       else # overlap of current and nxt_obj.
        { # Fill current object to start of
          # nxt_obj using FILL routine of CPG:
          current.CPG_Fill(currx,
                           nxt_obj.xcrit, H);
          # Update currx to first unfilled pixel:
          currx = nxt_obj.xcrit;
          Z_List.Add(nxt_obj);
            # Add nxt_obj to Z_List.
          nxt_obj = AOL.Next_Object();
            # New next object.
        }
    }
}

PROCEDURE Z_List.Add(new_obj)
# Routine to add new object to Z_List.
# Z_List is linked list of objects such that:
# 1. The line through current pixel cuts all
#    objects on Z_list in order of increasing Z.
# 2. If A appears before B on the Z_list
#    then A.xmax < B.xmax
# Comparisons are made by passing a line thru
# new_obj's x-critical point and evaluating its
# intersections with axes of objects on # Z_List.
# Variables:
#   x_pos: x-value of new_obj's x-critical point.
#   curr: current obj. on Z-List to start search.
#   pred: predecessor of current (Z-list has
#         a dummy element at beginning).
#   cur_Z: depth of curr's axis at x_pos.
#   new_Z: depth of new_obj's x-critical pt.

{ if (Z_List.Empty())
    { # Initialize Z_List with new_obj.
      Z_List.Init(new_obj);  return; }
  x_pos = new_obj.xcrit;
    # x-value at which depth comparisons are made.
  curr = Z_List.First(); # First object in Z_List.
  pred = predecessor of curr on Z_List;
    # (Dummy before first element of Z_List)
  new_Z = new_obj.Z(x_pos);
    # Depth of new_obj's x-critical point.
  # Traverse Z_List for place of new_obj:
  while (Z_List not yet traversed)
    { cur_Z = curr.Z(x_pos);
      if (new_Z < cur_Z)
        { # insert new object before current:
          Z_List.Insert(new_obj, pred);
          # Objects hidden by new object
          # will be removed from Z_List:
          while(new_obj.xmax >= curr.xmax)
            { Z_List.RemoveCur(pred);
              curr = Z_List.Current(pred); }
          return;
        }
      else # new_obj's depth (at x_pos) is
           # greater than current's depth.
        # If new_obj is completely hidden by
        # current it will not be added to Z_list.
        if ( curr.xmax >= new_obj.xmax ) return;
        else # Continue search on the list.
          { pred = curr;
            curr = Z_List.Next(pred); }
    }
  # New object will be added as last on Z_List.
  if (pred.xmax < new_obj.xmax)
    Z_List.Insert(new_obj, pred);
}

PROCEDURE Z_List.Init(Object)
# Routine to initialize the Z_List to contain
# the object Object (and no other objects).

FUNCTION Z_List.First()
# Returns first object in the Z_List.
```

```
# If Z_List is empty, returns nil.

PROCEDURE Z_List.Delete_First()
# Routine to remove first object from the Z_List.

FUNCTION Z_List.Empty()
# Returns True if Z_List is empty, else False.

PROCEDURE Object.AET_Update(H)
# Routine to update object's data (including
# its AET) to meet the scanline y=H.

PROCEDURE Object.CPG_Fill(xfirst, xlast, H)
# Routine to fill all the pixels with integer
# coordinates (i,H) that satisfy the condition:
# xfirst <= i < xlast This routine is a
# restricted form of the CPG Fill routine.

PROCEDURE AOL.Sil_Update(H)
# Routine to update the silhouettes of all the
# objects that are in the Active Objects List.

PROCEDURE AOL.Sort_by_Xcritical()
# Routine to sort all the objects in the
# AOL by their x-critical points.

PROCEDURE AOL.Add_New(H)
# Routine to add new objects to AOL, at scanline
# y=H, from the list of y-critical points.

FUNCTION AOL.Empty()
# Returns True if AOL is empty, False otherwise.

FUNCTION AOL.First()
# Routine that returns first element from AOL.
# If AOL is empty, returns nil.

FUNCTION AOL.Next()
# Routine that returns next element from AOL.
# If AOL is empty, returns nil.
```