# CHAPTER 10

# Hidden Surfaces

Opaque objects that are closer to the eye and in the line of sight of other objects will block those objects or portions of those objects from view. In fact, some surfaces of these opaque objects themselves are not visible because they are eclipsed by the objects' visible parts. The surfaces that are blocked or hidden from view must be "removed" in order to construct a realistic view of the 3D scene (see Fig. 1-3 where only three of the six faces of the cube are shown). The identification and removal of these surfaces is called the *hidden-surface problem*. The solution involves the determination of the closest visible surface along each projection line (Sec. 10.1).

There are many different hidden-surface algorithms. Each can be characterized as either an *image-space method*, in which the pixel grid is used to guide the computational activities that determine visibility at the pixel level (Secs. 10.2, 10.5, and 10.6), or an *object-space method*, in which surface visibility is determined using continuous models in the object space (or its transformation) without involving pixel-based operations (Secs. 10.3 and 10.4).

Notice that the hidden-surface problem has ties to the calculation of shadows. If we place a light source, such as a bulb, at the viewpoint, all surfaces that are visible from the viewpoint are lit directly by the light source and all surfaces that are hidden from the viewpoint are in the shadow of some opaque objects blocking the light.

## 10.1 DEPTH COMPARISONS

We assume that all coordinates $(x, y, z)$ are described in the normalized viewing coordinate system (Chap. 8).

Any hidden-surface algorithm must determine which edges and surfaces are visible either from the center of projection for perspective projections or along the direction of projection for parallel projections.

The question of visibility reduces to this: given two points $P_1(x_1, y_1, z_1)$ and $P_2(x_2, y_2, z_2)$, does either point obscure the other? This is answered in two steps:

1. Are $P_1$ and $P_2$ on the same projection line?
2. If not, neither point obscures the other. If so, a depth comparison tells us which point is in front of the other.

For an orthographic parallel projection onto the $xy$ plane, $P_1$ and $P_2$ are on the same projector if $x_1 = x_2$ and $y_1 = y_2$. In this case, depth comparison reduces to comparing $z_1$ and $z_2$. If $z_1 < z_2$, then $P_1$ obscures $P_2$ [see Fig. 10-1($a$)].

For a perspective projection [see Fig. 10-1($b$)], the calculations are more complex (Prob. 10.1). However, this complication can be avoided by transforming all three-dimensional objects so that parallel

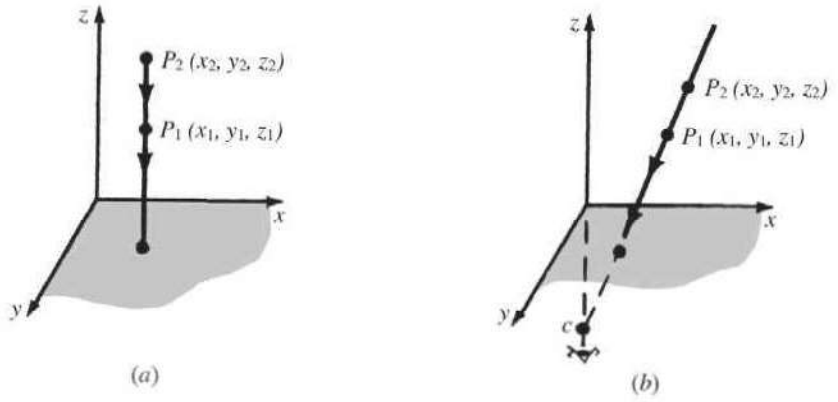



**Fig. 10-1**

projection of the transformed object is equivalent to a perspective projection of the original object (see Fig. 10-2). This is done with the use of the *perspective to parallel transform* $T_p$ (Prob. 10.2).

If the original object lies in the normalized perspective view volume (Chap. 8), the *normalized perspective to parallel transform*

$$NT_p = \begin{pmatrix} \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & \frac{1}{1-z_f} & \frac{-z_f}{1-z_f} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

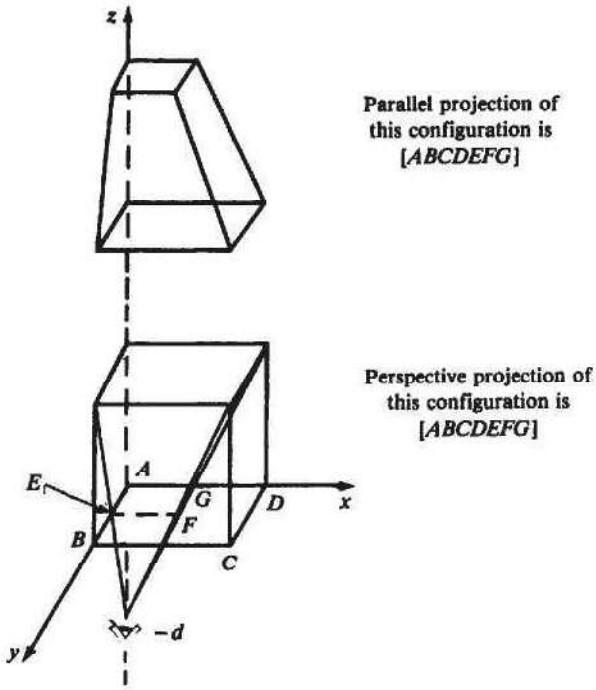



**Fig. 10-2**

(where $z_f$ is the location of the front clipping plane of the normalized perspective view volume) transforms the normalized perspective view volume into the unit cube bounded by $0 \le x \le 1$, $0 \le y \le 1$, $0 \le z \le 1$ (Prob. 10.3). We call this cube the *normalized display space*. A critical fact is that the normalized perspective to parallel transform preserves lines, planes, and depth relationships.

If our display device has display coordinates $H \times V$, application of the scaling matrix

$$S_{H,V,1} = \begin{pmatrix} H & 0 & 0 & 0 \\ 0 & V & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

transforms the normalized display space $0 \le x \le 1$, $0 \le y \le 1$, $0 \le z \le 1$ onto the region $0 \le x \le H$, $0 \le y \le V$, $0 \le z \le 1$. We cal this region the *display space*. The *display transform* $DT_p$:

$$DT_p = S_{H,V,1} \cdot NT_p$$

transforms the normalized perspective view volume onto the display space.

Clipping must be done against the normalized perspective view volume prior to applying the transform $NT_p$. An alternative to this is to combine $NT_p$ with the normalizing transformation $N_{\text{per}}$ (Chap. 8), forming the single transformation $NT'_p = NT_p \cdot N_{\text{per}}$. Then clipping is done in homogeneous coordinate space. This method for performing clipping is not covered in this book.

We now describe several algorithms for removing hidden surfaces from scenes containing objects defined with planar (i.e., flat), polygonal faces. We assume that the displays transform $DT_p$ has been applied (if a perspective projection is being used), so that we always deal with parallel projections in display space.

## 10.2 *Z*-BUFFER ALGORITHM

We say that a point in display space is "seen" from pixel $(x, y)$ if the projection of the point is scan-converted to this pixel (Chap. 3). The $Z$-buffer algorithm essentially keeps track of the smallest $z$ coordinate (also called the *depth value*) of those points which are seen from pixel $(x, y)$. These $Z$ values are stored in what is called the $Z$ buffer.

Let $Z_{\text{buf}}(x, y)$ denote the current depth value that is stored in the $Z$ buffer at pixel $(x, y)$. We work with the (already) projected polygons $P$ of the scene to be rendered.

The $Z$-buffer algorithm consists of the following steps.

1. Initialize the screen to a background color. Initialize the $Z$ buffer to the depth of the back clipping plane. That is, set

$$Z_{\text{buf}}(x, y) = Z_{\text{back}}, \qquad \text{for every pixel } (x, y)$$

2. Scan-convert each (projected) polygon $P$ in the scene (Chap. 3) and during this scan-conversion process, for each pixel $(x, y)$ that lies inside the polygon:

   (*a*) Calculate $Z(x, y)$, the depth of the polygon at pixel $(x, y)$.

   (*b*) If $Z(x, y) < Z_{\text{buf}}(x, y)$, set $Z_{\text{buf}}(x, y) = Z(x, y)$ and set the pixel value at $(x, y)$ to the color of the polygon $P$ at $(x, y)$. In Fig. 10-3, points $P_1$ and $P_2$ are both scan-converted to pixel $(x, y)$; however, since $z_1 < z_2$, $P_1$ will obscure $P_2$ and the $P_1$ $z$ value, $z_1$, will be stored in the $Z$ buffer.

Although the $Z$-buffer algorithm requires $Z$-buffer memory storage proportional to the number of pixels on the screen, it does not require additional memory for storing all the objects comprising the scene. In fact, since the algorithm processes polygons one at a time, the total number of objects in a scene can be arbitrarily large.

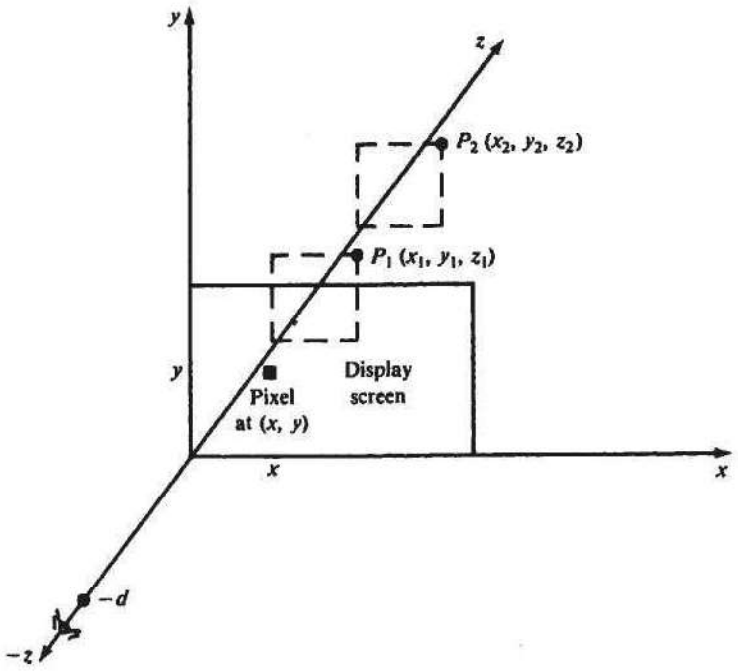



Fig. 10-3

## 10.3 BACK-FACE REMOVAL

Object surfaces that are orientated away from the viewer are called *back-faces*. The back-faces of an opaque polyhedron are completely blocked by the polyhedron itself and hidden from view. We can therefore identify and remove these back-faces based solely on their orientation without further processing (projection and scan-conversion) and without regard to other surfaces and objects in the scene.

Let $\mathbf{N} = (A, B, C)$ be the normal vector of a planar polygonal face, with $\mathbf{N}$ pointing in the direction the polygon is facing. Since the direction of viewing is the direction of the positive $z$ axis (see Fig. 10-3), the polygon is facing away from the viewer when $C > 0$ (the angle between $\mathbf{N}$ and the $z$ axis is less than 90°). The polygon is also classified as a back-face when $C = 0$, since in this case it is parallel to the line of sight and its projection is either hidden or overlapped by the edge(s) of some visible polygon(s).

Although this method identifies and removes back-faces quickly it does not handle polygons that face the viewer but are hidden (partially or completely) behind other surfaces. It can be used as a preprocessing step for other algorithms.

## 10.4 THE PAINTER'S ALGORITHM

Also called the *depth sort* or *priority algorithm*, the painter's algorithm processes polygons as if they were being painted onto the view plane in the order of their distance from the viewer. More distance polygons are painted first. Nearer polygons are painted on or over more distance polygons, partially or totally obscuring them from view. The key to implementing this concept is to find a priority ordering of the polygons in order to determine which polygons are to be painted (i.e., scan-converted) first.

Any attempt at a priority ordering based on depth sorting alone results in ambiguities that must be resolved in order to correctly assign priorities. For example, when two polygons overlap, how do we decide which one obscures the other? (See Fig. 10-4.)

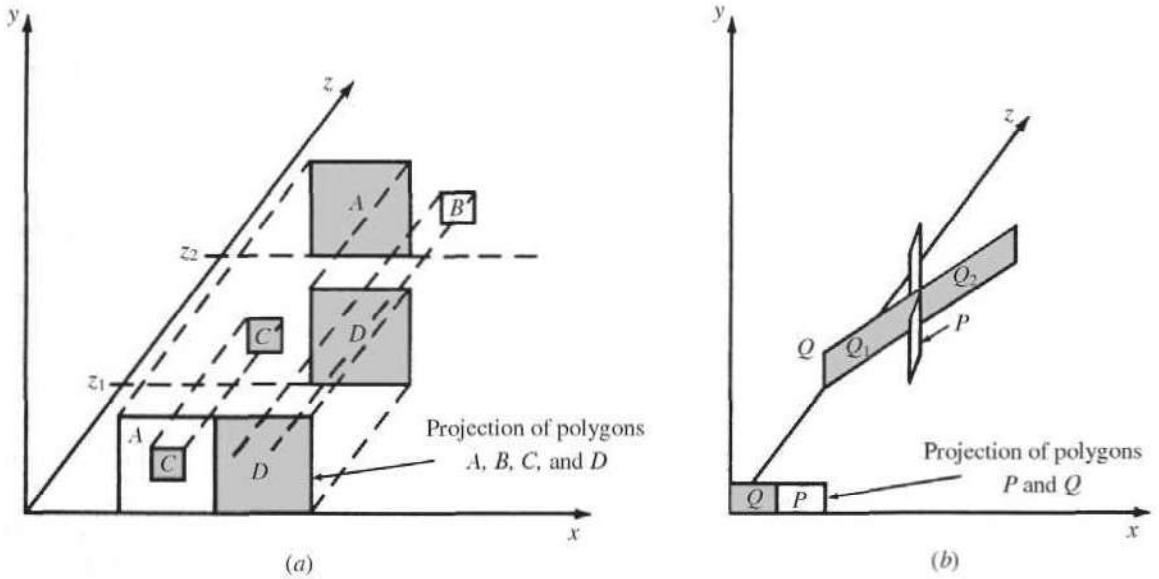



**Fig. 10-4** Projection of opaque polygons.

### Assigning Priorities

We assign priorities to polygons by determining if a given polygon $P$ obscures other polygons. If the answer is no, then $P$ should be painted first. Hence the key test is to determine whether polygon $P$ does *not* obscure polygon $Q$.

The $z$ extent of a polygon is the region between the planes $z = z_{\min}$ and $z = z_{\max}$ (Fig. 10-5). Here, $z_{\min}$ is the smallest of the $z$ coordinates of all the polygon's vertices, and $z_{\max}$ is the largest.

Similar definitions hold for the $x$ and $y$ extents of a polygon. The intersection of the $x$, $y$, and $z$ extents is called the *extent*, or bounding box, of the polygon.

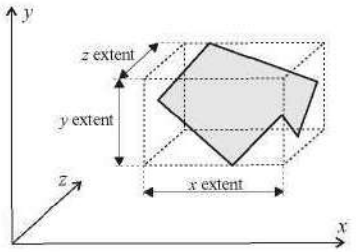



**Fig. 10-5**

### Testing Whether *P* Obscures *Q*

Polygon $P$ does not obscure polygon $Q$ if any one of the following tests, applied in sequential order, is true.

Test 0: the $z$ extents of $P$ and $Q$ do not overlap and $z_{Q_{\max}}$ of $Q$ is smaller than $z_{P_{\min}}$ of $P$. Refer to Fig. 10-6.

Test 1: the $y$ extents of $P$ and $Q$ do not overlap. Refer to Fig. 10-7.

Test 2: the $x$ extents of $P$ and $Q$ do not overlap.

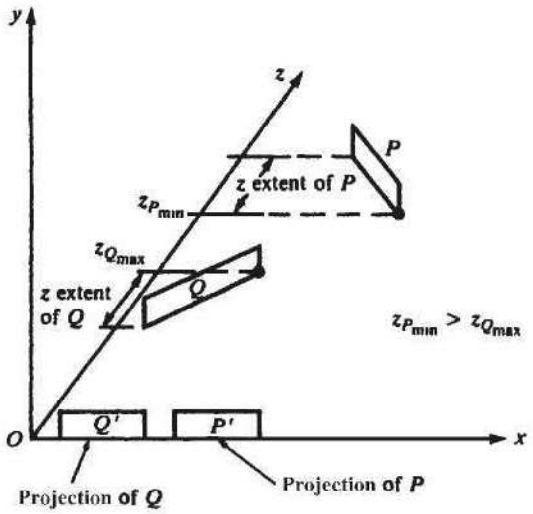



Fig. 10-6

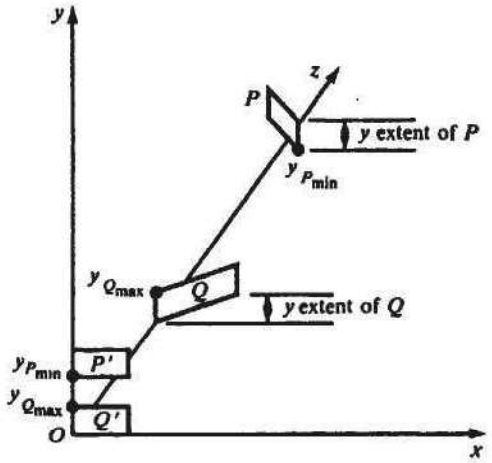



Fig. 10-7

Test 3: all the vertices of $P$ lie on that side of the plane containing $Q$ which is farthest from the viewpoint. Refer to Fig. 10-8.

Test 4: all the vertices of $Q$ lie on that side of the plane containing $P$ which is closest to the viewpoint. Refer to Fig. 10-9.

Test 5: the projections of the polygons $P$ and $Q$ onto the view plane do not overlap. This is checked by comparing each edge of one polygon against each edge of the other polygon to search for intersections.

## The Algorithm

1. Sort all polygons into a polygon list according to $z_{max}$ (the largest $z$ coordinate of each polygon's vertices). Starting from the end of the list, assign priorities for each polygon $P$, in order, as described in steps 2 and 3 (below).

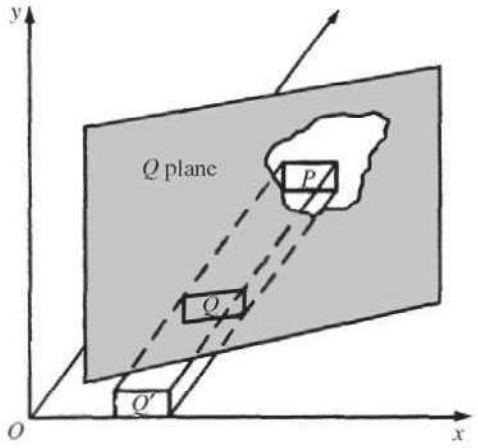



**Fig. 10-8**

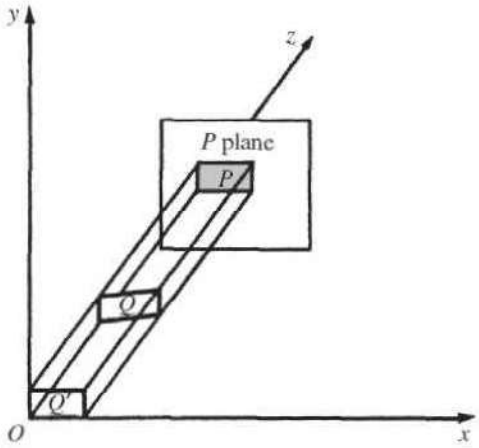



**Fig. 10-9**

2. Find all polygons $Q$ (preceding $P$) in the polygon list whose $z$ extents overlap that of $P$ (test 0).
3. For each $Q$, perform tests 1 through 5 until true.
   (*a*) If every $Q$ passes, scan-convert polygon $P$.
   (*b*) If false for some $Q$, swap $P$ and $Q$ on the list. Tag $Q$ as swapped. If $Q$ has already been tagged, use the plane containing polygon $P$ to divide polygon $Q$ into two polygons. $Q_1$ and $Q_2$ [see Fig. 10-4(*b*)]. The polygon-clipping techniques described in Chap. 5 can be used to perform the division. Remove $Q$ from the list and place $Q_1$ and $Q_2$ on the list, in sorted order.

Sometimes the polygons are subdivided into triangles before processing, thus reducing the computational effort for polygon subdivision in step 3.

## 10.5 SCAN-LINE ALGORITHM

A scan-line algorithm consists essentially of two nested loops, an $x$-scan loop nested within a $y$-scan loop.

### *y* Scan

For each $y$ value, say, $y = \alpha$, intersect the polygons to be rendered with the scan plane $y = \alpha$. This scan plane is parallel to the $xz$ plane, and the resulting intersections are line segments in this plane (see Fig. 10-10).

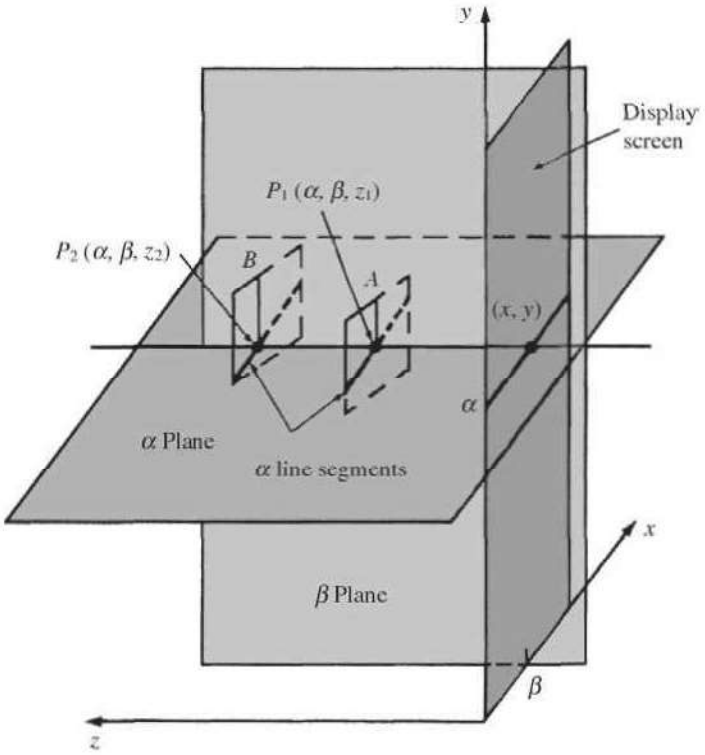



**Fig. 10-10**

### *x* Scan

1. For each $x$ value, say, $x = \beta$, intersect the line segments found above with the $x$-scan line $x = \beta$ lying on the $y$-scan plane. This intersection results in a set of points that lies on the $x$-scan line.
2. Sort these points with respect to their $z$ coordinates. The point $(x, y, z)$ with the smallest $z$ value is visible, and the color of the polygon containing this point is the color set at the pixel corresponding to this point.

In order to reduce the amount of calculation in each scan-line loop, we try to take advantage of relationships and dependencies, called *coherences*, between different elements that comprise a scene.

### Types of Coherence

1. *Scan-line coherence.* If a pixel on a scan line lies within a polygon, pixels near it will most likely lie within the polygon.
2. *Edge coherence.* If an edge of a polygon intersects a given scan line, it will mot likely intersect scan lines near the given one.

3. *Area coherence.* A small area of an image will most likely lie within a single polygon.
4. *Spatial coherence.* Certain properties of an object can be determined by examining the *extent* of the object, that is, a geometric figure which circumscribes the given object. Usually the extent is a rectangle or rectangle solid (also called a *bounding box*).

Scan-line coherence and edge coherence are both used to advantage in scan-converting polygons (Chap. 3).

Spatial coherence is often used as a preprocessing step. For example, when determining whether polygons intersect, we can eliminate those polygons that don't intersect by finding the rectangular extent of each polygon and checking whether the extents intersect—a much simpler problem (see Fig. 10-11). [*Note*: In Fig. 10-11 objects $A$ and $B$ do not intersect; however, objects $A$ and $C$, and $B$ and $C$, do intersect. In preprocessing, corner points would be compared to determine whether there is an intersection. For example, the edge of object $A$ is at coordinate $P_3 = (6, 4)$ and the edge of object $B$ is at coordinate $P_4 = (7, 3)$.] Of course, even if the extents intersect, this does not guarantee that the polygons intersect. See Fig. 10-12 and note that the extents of $A'$ and $B'$ overlap even though the polygons do not.

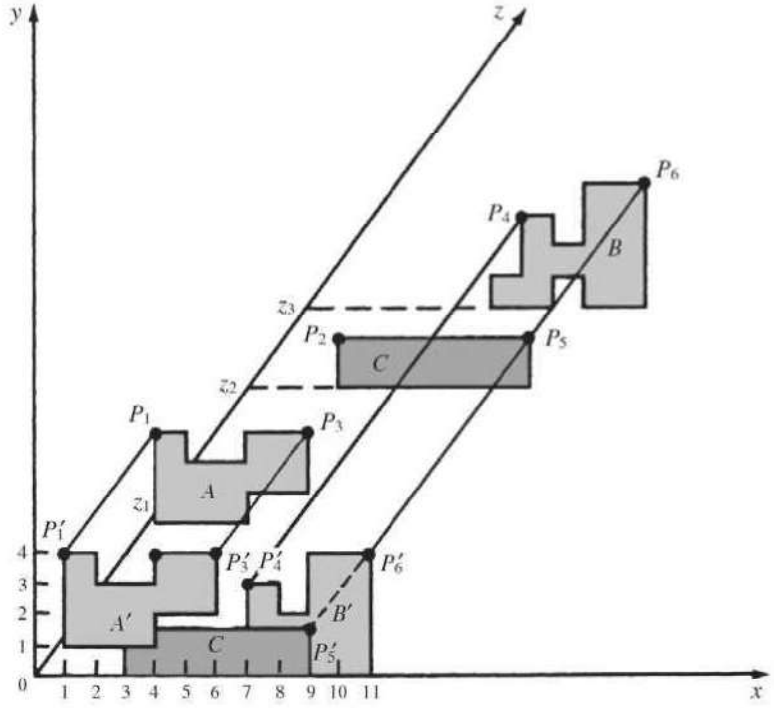



**Fig. 10-11**

Coherences can simplify calculations by making them incremental, as opposed to absolute. This is illustrated in Prob. 10.13.

### A Scan-line Algorithm

In the following algorithm, scan line and edge coherence are used to enhance the processing done in the $y$-scan loop as follows. Since the $y$-scan loop constructs a list of potentially visible line segments, instead of reconstructing this list each time the $y$-scan line changes (absolute calculation), we keep the list and update it according to how it has changed (incremental calculation). This processing is facilitated by

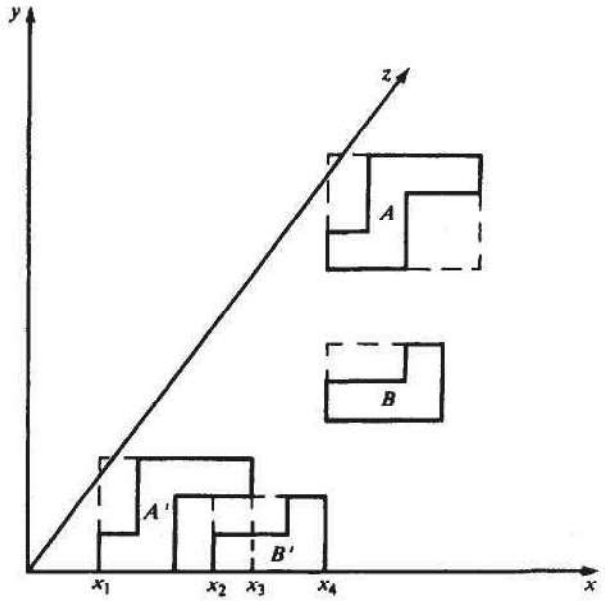



**Fig. 10-12**

the use of what is called the *edge list*, and its efficient construction and maintenance is at the heart of the algorithm (see Chap. 3, Sec. 3.7, under "A Scan-line Algorithm").

The following data structures are created:

1. *The edge list*—contains all nonhorizontal edges (horizontal edges are automatically displayed) of the projections of the polygons in the scene. The edges are sorted by the edge's smaller $y$ coordinate ($y_{\min}$). Each edge entry in the edge list also contains:
   (*a*) The $x$ coordinate of the end of the edge with the smaller $y$ coordinate.
   (*b*) The $y$ coordinate of the edge's other end ($y_{\max}$).
   (*c*) The increment $\Delta x = 1/m$.
   (*d*) A pointer indicating the polygon to which the edge belongs.
2. *The polygon list*—for each polygon, contains
   (*a*) The equation of the plane within which the polygon lies—used for depth determination, i.e., to find the $z$ value at pixel $(x, y)$.
   (*b*) An IN/OUT flag, initialized to OUT (this flag is set depending on whether a given scan line is in or out of the polygon).
   (*c*) Color information for the polygon.

The algorithm proceeds as follows.

I. *Initialization*.
   (*a*) Initialize each screen pixel to a background color.
   (*b*) Set $y$ to the smallest $y_{\min}$ value in the edge list.

   Repeat steps II and III (below) until no further processing can be performed.

II. *y-scan loop*. Activate edges whose $y_{\min}$ is equal to $y$. Sort active edges in order of increasing $x$.

III. *x-scan loop*. Process, from left to right, each active edge as follows:

(*a*) Invert the IN/OUT flag of the polygon in the polygon list which contains the edge. Count the number of active polygons whose IN/OUT flag is set to IN. If this number is 1, only one polygon is visible. All pixel values from this edge and up to the next edge are set to the color of the polygon. If this number is greater than 1, determine the visible polygon by the smallest $z$ value of each polygon at the pixel under consideration. These $z$ values are found from the equation of the plane containing the polygon. The pixels from this edge and up to the next edge are set to the color of this polygon, unless the polygon becomes obscured by another before the next edge is reached, in which case we set the remaining pixels to the color of the obscuring polygon. If this number is 0, pixels from this edge and up to the next one are left unchanged.

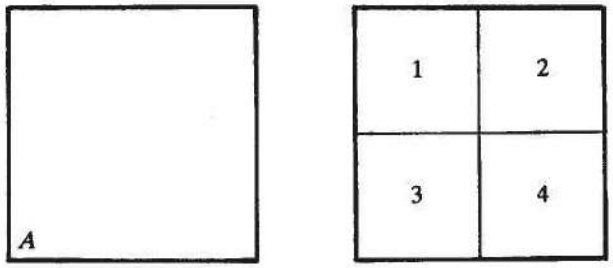
(*b*) When the last active edge is processed, we then proceed as follows:

1. Remove those edges for which the value of $y_{\max}$ equals the present scan-line value $y$. If no edges remain, the algorithm has finished.
2. For each remaining active edge, in order, replace $x$ by $x + 1/m$. This is the edge intersection with the next scan line $y + 1$ (see Prob. 10.13).
3. Increment $y$ to $y + 1$, the next scan line, and repeat step II.

## 10.6 SUBDIVISION ALGORITHM

The subdivision algorithm is a recursive procedure based on a two-step strategy that first decides which projected polygons overlap a given area $A$ on the screen and are therefore potentially visible in that area. Second, in each area these polygons are further tested to determine which ones will be visible within this area and should therefore be displayed. If a visibility decision cannot be made, this screen area, usually a rectangular window, is further subdivided either until a visibility decision can be made, or until the screen area is a single pixel.

Starting with the full screen as the initial area, the algorithm divides an area at each stage into four smaller areas, thereby generating a quad tree (see Fig. 10-13).




(*a*) Initial area. (*b*) First subdivision.

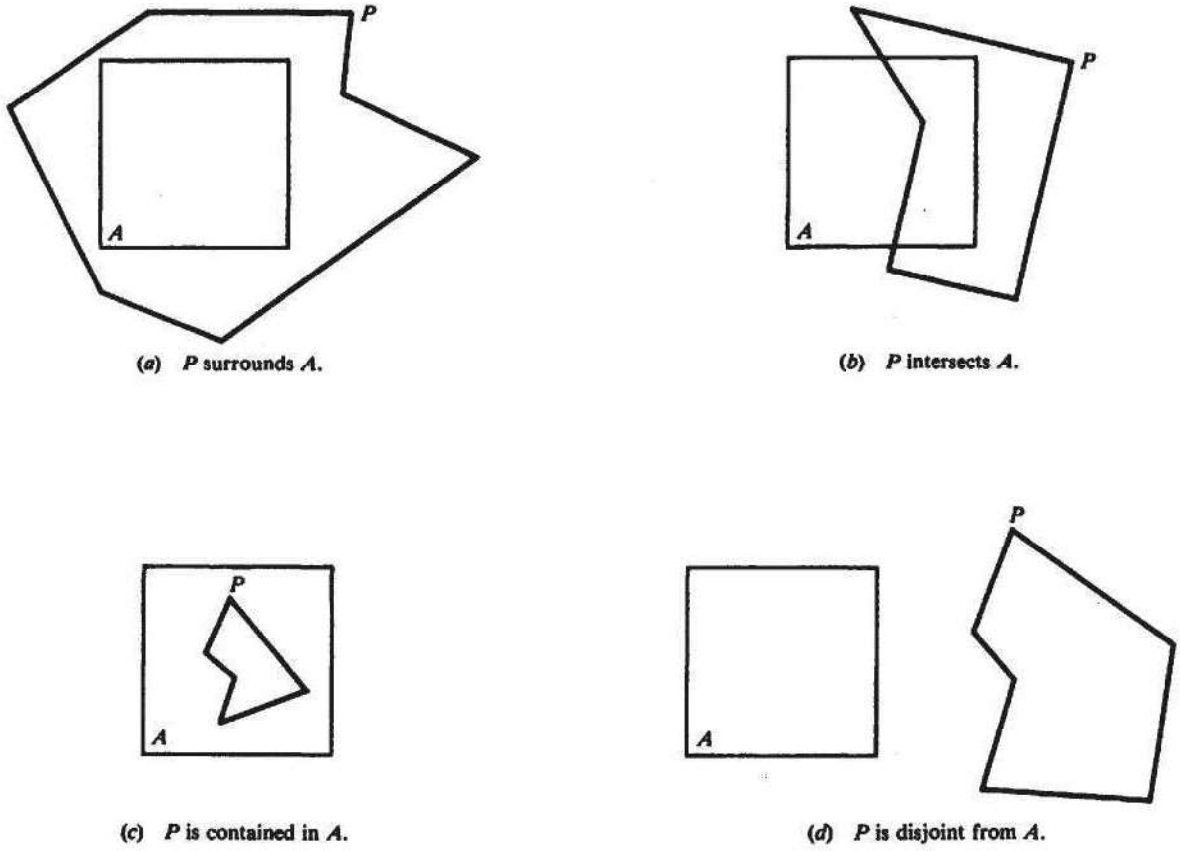
| | | | |
|---|---|---|---|
| 1 | 2 | 1 | 2 |
| 3 | 4 | 3 | 4 |
| 1 | 2 | 1 | 2 |
| 3 | 4 | 3 | 4 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 1 | 2 | 1 | 2 | 1 | 2 |
| 3 | 4 | 3 | 4 | 3 | 4 | 3 | 4 |
| 1 | 2 | 1 | 2 | 1 | 2 | 1 | 2 |
| 3 | 4 | 3 | 4 | 3 | 4 | 3 | 4 |
| 1 | 2 | 1 | 2 | 1 | 2 | 1 | 2 |
| 3 | 4 | 3 | 4 | 3 | 4 | 3 | 4 |
| 1 | 2 | 1 | 2 | 1 | 2 | 1 | 2 |
| 3 | 4 | 3 | 4 | 3 | 4 | 3 | 4 |

(*c*) Second subdivision. (*d*) Third subdivision.

Fig. 10-13

The processing exploits area coherence by classifying polygons $P$ with respect to a given screen area $A$ into the following categories: (1) *surrounding polygon*—polygon that completely contains the area [Fig. 10-14(*a*)], (2) *interesecting polygon*—polygon that intersects the area [Fig. 10-14(*b*)], (3) *contained polygon*—polygon that is completely contained within the area [Fig. 10-14(*c*)], and (4) *disjoint polygon*—polygon that is disjoint from the area [Fig. 10-14(*d*)].




(*a*) *P* surrounds *A*. (*b*) *P* intersects *A*.

(*c*) *P* is contained in *A*. (*d*) *P* is disjoint from *A*.

**Fig. 10-14**

The classification of the polygons within a picture is the main computational expense of the algorithm and is analogous to the clipping algorithms discussed in Chap. 5. With the use of one of these clipping algorithms, a polygon in category 2 (intersecting polygon) can be clipped into a contained polygon and a disjoint polygon (see Fig. 10-15). Therefore, we could proceed as if category 2 were eliminated.

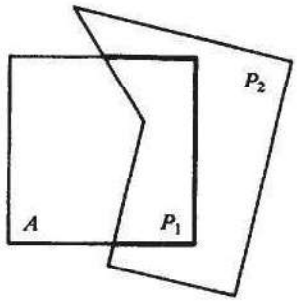



**Fig. 10-15**

For a given screen area, we keep a potentially visible polygons list (PVPL), those in categories 1, 2, and 3. (Disjoint polygons are clearly not visible.) Also, note that on subdivision of a screen area, surrounding and disjoint polygons remain surrounding and disjoint polygons of the newly formed areas. Therefore, only contained and intersecting polygons need to be reclassified.

### Removing Polygons Hidden by a Surrounding Polygon

The key to efficient visibility computation lies in the fact that a polygon is not visible if it is in back of a surrounding polygon. Therefore, it can be removed from the PVPL. To facilitate processing, this list is sorted by $z_{\min}$, the smallest $z$ coordinate of the polygon within this area. In addition, for each surrounding polygon $S$, we also record its largest $z$ coordinate, $z_{S_{\max}}$.

If, for a polygon $P$ on the list, $z_{P_{\min}} > z_{S_{\max}}$ (for a surrounding polygon $S$), then $P$ is hidden by $S$ and thus is not visible. In addition, all other polygons after $P$ on the list will also be hidden by $S$, so we can remove these polygons from the PVPL.

### Subdivision Algorithm

1. Initialize the area to be the whole screen.
2. Create a PVPL with respect to an area, sorted on $z_{\min}$ (the smallest $z$ coordinate of the polygon within the area). Place the polygons in their appropriate categories. Remove polygons hidden by a surrounding polygon and remove disjoint polygons.
3. Perform the visibility decision tests:
   (*a*) If the list is empty, set all pixels to the background color.
   (*b*) If there is exactly one polygon in the list and it is a classified as intersecting (category 2) or contained (category 3), color (scan-convert) the polygon, and color the remaining area to the background color.
   (*c*) If there is exactly one polygon on the list and it is a surrounding one, color the area the color of the surrounding polygon.
   (*d*) If the area is the pixel $(x, y)$, and neither *a*, *b*, nor *c* applies, compute the $z$ coordinate $z(x, y)$ at pixel $(x, y)$ of all polygons on the PVPL. The pixel is then set to the color of the polygon with the smallest $z$ coordinate.
4. If none of the above cases has occurred, subdivide the screen area into fourths. For each area, go to step 2.

## 10.7 HIDDEN-LINE ELIMINATION

Although there are special-purpose hidden-line algorithms, each of the above algorithms can be modified to eliminate hidden lines or edges. This is especially useful for wireframe polygonal models where the polygons are unfilled. The idea is to use a color rule which fills all the polygons with the background color—say, black—and the edges and lines a different color—say, white. The use of a hidden-surface algorithm now becomes a hidden-line algorithm.

## 10.8 THE RENDERING OF MATHEMATICAL SURFACES

In plotting a mathematical surface described by an equation $z = F(x, y)$, where $x_{\min} \le x \le x_{\max}$ and $y_{\min} \le y \le y_{\max}$, we could use any of the hidden-surface algorithms so far described. However, these general algorithms are inefficient when compared to specialized algorithms that take advantage of the structure of this type of surface.

The mathematical surface is rendered as a wireframe model by drawing both the *x-constant curves* $z = F(\text{const}, y)$ and the *y-constant curves* $z = F(x, \text{const})$ (see Fig. 10-16). Each such curve is rendered as

a polyline, where the illusion of smoothness is achieved by using a fine resolution (i.e., short line segments) in drawing the polyline (Chap. 3).

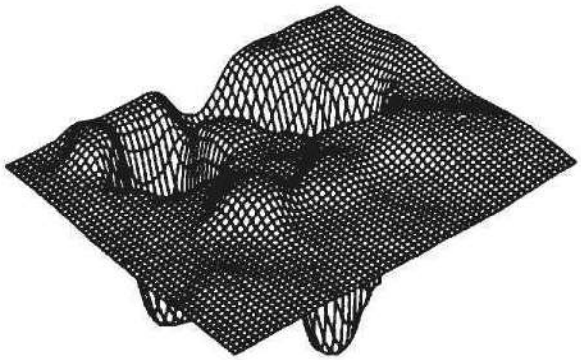


**Fig. 10-16**

Choose an $M \times N$ plotting resolution

$$x_{\min} \leq x_1 < x_2 < \cdots < x_M \leq x_{\max} \qquad \text{and} \qquad y_{\min} \leq y_1 < y_2 < \cdots < y_N \leq y_{\max}$$

The corresponding $z$ values are $z_{ij} = F(x_i, y_j)$. An $x$-constant polyline, say $x = x_j$, has vertices

$$P_1(x_j, y_1), \ldots, P_N(x_j, y_N)$$

Similarly, the $y = y_k$ polyline has vertices

$$Q_1(x_1, y_k), \ldots, Q_M(x_M, y_k)$$

Choosing a view plane and a center of projection or viewpoint $C(a, b, c)$, we create a perspective view of the surface onto this view plane by using the transformations developed in Chap. 7. So a point $[x, y, F(x, y)]$ on the surface projects to a point $(p, q)$ in view plane coordinates. By applying an appropriate 2D viewing transformation (Chap. 5), we can suppose that $p$ and $q$ line within the horizontal and vertical plotting dimensions of the plotting device, say, $H \times V$ pixels.

### The Perimeter Method for Rendering the Surface

Each plotted $x$ and $y$ constant polyline outlines a polygonal region on the plotting screen (Fig. 10-17).

The algorithm is based on the following observations: (1) *ordering*—the $x$- and $y$-constant curves (i.e., polylines) are drawn in order starting with the one closest to the viewpoint and (2) *visibility*—we draw only that part of the polyline that is outside the perimeter of all previously drawn regions (Fig. 10-18). One implementation of the visibility condition uses a min–max array $A$, of length $H$ (that of the plotting device), which contains, at each horizontal pixel position $i$, the maximum (and/or minimum) vertical pixel value

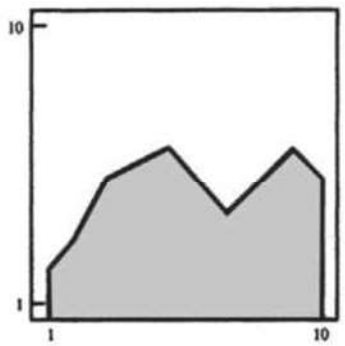



**Fig. 10-17**

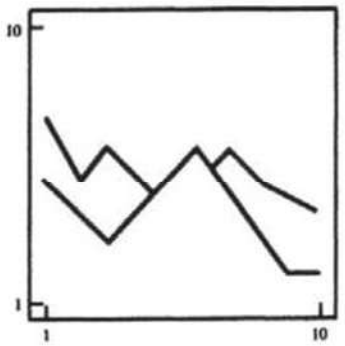



**Fig. 10-18**

drawn thus far at $i$; that is $A(i) = \overset{\max}{\min}$ (vertical pixel values drawn so far at $i$) (Fig. 10-19). Selection of the max results in a drawing of the top of the surface. The min is used to render the bottom of the surface, and the max and min yields both top and bottom.

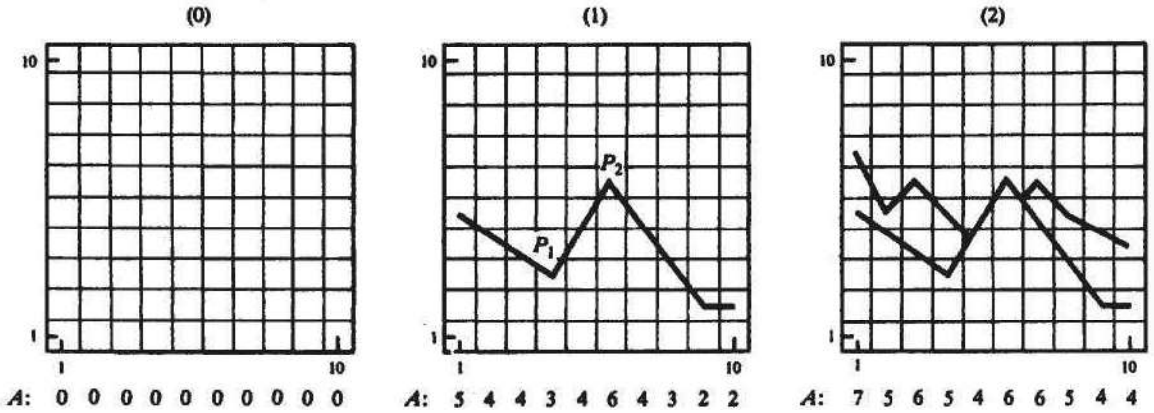



**Fig. 10-19** Updating min–max array $A$ (using maximum values only).

### The Visibility Test

Suppose that $(p', q')$ is the pixel that corresponds to the point $(p, q)$. Then this pixel is *visible* if either

$$q' > A(p') \qquad \text{or} \qquad q' < A(p')$$

where $A$ is the min–max array.

The visibility criteria for a line segment are: (1) the line segment is visible if both its endpoints are visible; (2) the line segment is invisible if both its endpoints are not visible; and (3) if only one endpoint is visible, the min–max array is tested to find the visible part of the line.

Drawing the $x$- or $y$-constant polylines thus consists of testing for the visibility of each line segment and updating the min–max array as necessary. Since a line segment will, in general, span several horizontal pixel positions (see segment $\overline{P_1P_2}$ in Fig. 10-19), the computation of $A(i)$ for these intermediate pixels is found by using the slope of the line segment or by using Bresenham's method described in Chap. 3.

### The Wright Algorithm for Rendering Mathematical Surfaces

The drawing of the surface $z = F(x, y)$ proceeds as follows.

1. To perform initialization, determine whether the viewpoint is closer to the $x$ or the $y$ axis. Suppose that it is closer to the $x$ axis. We next locate the $x$-constant curve that is closest to the viewpoint at $x = 1$.
   (*a*) Initialize the min–max array to some base value, say, zero.
   (*b*) Start with the $x$-constant curve found above.
2. Repeat the following steps using the visibility test for drawing line segments and updating the min–max array each time a line segment is drawn:
   (*a*) Draw the $x$-constant polyline.
   (*b*) Draw those parts of each $y$-constant polyline that lie between the previously drawn $x$-constant polyline and the next one to be drawn.
   (*c*) Proceed, in the direction of increasing $x$, to the next $x$-constant polyline.

## Solved Problems

**10.1** Given points $P_1(1, 2, 0)$, $P_2(3, 6, 20)$, and $P_3(2, 4, 6)$ and a viewpoint $C(0, 0, -10)$, determine which points obscure the others when viewed from $C$.

**SOLUTION**

The line joining the viewpoint $C(0, 0, -10)$ and point $P_1(1, 2, 0)$ is (App. 2)

$$x = t \qquad y = 2t \qquad z = -10 + 10t$$

To determine whether $P_2(3, 6, 20)$ lies on this line, we see that $x = 3$ when $t = 3$, and then at $t = 3$, $x = 3$, $y = 6$, and $z = 20$. So $P_2$ lies on the projection line through $C$ and $P_1$.

Next we determine which point is in front with respect to $C$. Now $C$ occurs on the line at $t = 0$, $P_1$ occurs at $t = 1$, and $P_2$ occurs at $t = 3$. Thus comapring $t$ values, $P_1$ is in front of $P_2$ with respect to $C$; that is, $P_1$ obscures $P_2$.

We now determine whether $P_3(2, 4, 6)$ is on the line. Now $x = 2$ when $t = 2$ and then $y = 4$, and $z = 10$. Thus $P_3(2, 4, 6)$ is not on this projection line and so it nether obscures nor is obscured by $P_1$ and $P_2$.

**10.2** Construct the perspective to parallel transform $T_p$ which produces an object whose parallel projection onto the $xy$ plane yields the same image as the perspective projection of the original object onto the normalized view plane $z = c'_z/(c'_z + b)$ (Chap. 8, Prob. 8.6) with respect to the origin as the center of projection.

**SOLUTION**

The perspective projection onto the plane $z = c'_z/(c'_z + b)$ with respect to the origin is (Chap. 7, Prob. 7.4):

$$Per = \begin{pmatrix} z_v & 0 & 0 & 0 \\ 0 & z_v & 0 & 0 \\ 0 & 0 & z_v & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

where $z_v = c'_z/(c'_z + b)$. The perspective projection onto the view plane of a point $P(x, y, z)$ is the point

$$P'\left(\frac{z_v x}{z}, \frac{z_v y}{z}, z_v\right)$$

Define the perspective to parallel transform $T_p$ to be

$$T_p = \begin{pmatrix} z_v & 0 & 0 & 0 \\ 0 & z_v & 0 & 0 \\ 0 & 0 & \dfrac{1}{1 - z_f} & \dfrac{-z_f}{1 - z_f} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

(where $z = z_f$ is the location of the normalized front clipping plane; see Chap. 8, Prob. 8.6).

Now, applying the perspective to parallel transform $T_p$ to the point $P(x, y, z)$, we produce the point

$$Q'\left(\frac{z_v x}{z}, \frac{z_v y}{z}, \frac{z - z_f}{z(1 - z_f)}\right)$$

The parallel projection of $Q'$ onto the $xy$ plane produces the point

$$Q'\left(\frac{z_v x}{z}, \frac{z_v y}{z}, 0\right)$$

So $Q'$ and $P'$ produce the same projective image. Furthermore, $T_p$ transforms the normalized perspective view volume bounded by $x = z$, $x = -z$, $y = z$, $y = -z$, $z = z_f$, and $z = 1$ to the rectangular volume bounded by $x = z_v$, $x = -z_v$, $y = z_v$, $y = -z_v$, $z = 0$, and $z = 1$.

**10.3** Show that the normalized perspective to parallel transform $NT_p$ preserves the relationships of the original perspective transformation while transforming the normalized perspective view volume into the unit cube.

**SOLUTION**

From Prob. 10.2, the perspective to parallel transform $T_p$ transforms a point $P(x, y, z)$ to a point

$$Q\left(\frac{z_v x}{z}, \frac{z_v y}{z}, \frac{z - z_f}{z(1 - z_f)}\right)$$

The image under parallel projection of this point onto the $xy$ plane is

$$Q'\left(\frac{z_v x}{z}, \frac{z_v y}{z}, 0\right)$$

The factor $z_v$ can be set equal to 1 without changing the relation between points $Q$ and $Q'$.

The matrix that transforms $P(x, y, z)$ to the point $Q\left(\frac{x}{z}, \frac{y}{z}, \frac{z - z_f}{z(1 - z_f)}\right)$ is then

$$\bar{T}_p = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \dfrac{1}{1 - z_f} & \dfrac{-z_f}{1 - z_f} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

In addition, this matrix transforms the normalized perspective view volume to the rectangular view volume bounded by $x = 1$, $x = -1$, $y = 1$, $y = -1$, $z = 0$, and $z = 1$.

We next translate this view volume so that the corner point $(-1, -1, 0)$ translates to the origin. The translation matrix that does this is

$$T_{(1,1,0)} = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The new region is a volume bounded by $x = 0$, $x = 2$, $y = 0$, $y = 2$, $z = 0$, and $z = 1$.

Finally, we scale in the $x$ and $y$ direction by a factor $\frac{1}{2}$ so that the final view volume is the unit cube: $x = 0$, $x = 1$, $y = 0$, $y = 1$, $z = 0$, and $z = 1$. The scaling matrix is

$$S_{1/2,1/2,1} = \begin{pmatrix} \frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The final normalized perspective to parallel transform is

$$NT_p = S_{1/2,1/2,1} \cdot T_{(1,1,0)} \cdot \bar{T}_p = \begin{pmatrix} \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & \dfrac{1}{1 - z_f} & \dfrac{-z_f}{1 - z_f} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

**10.4** Why are hidden-surface algorithms needed?

**SOLUTION**

Hidden-surface algorithms are needed to determine which objects and surface will obscure those objects and surfaces that are in back of them, thus rendering a more realistic image.

**10.5** What two steps are required to determine whether any given points $P_1(x_1, y_1, z_1)$ obscures another point $P_2(x_2, y_2, z_2)$? (See Fig. 10-1.)

**SOLUTION**

It must be determined (1) whether the two points lie on the same projection line and (2) if they do, which point is in front of the other.

**10.6** Why is it easier to locate hidden surfaces when parallel projection is used?

**SOLUTION**

There are no vanishing points in parallel projection. As a result, any point $P(a, b, z)$ will line on the same projector as any other point having the same $x$ and $y$ coordinates $(a, b)$. Thus only the $z$ component must be compared to determine which point is closest to the viewer.

**10.7** How does the $Z$-buffer algorithm determine which surfaces are hidden?

**SOLUTION**

The $Z$-buffer algorithm sets up a two-dimensional array which is like the frame buffer; however the $Z$ buffer stores the depth value at each pixel rather than the color, which is stored in the frame buffer. By setting the initial values of the $Z$ buffer to some large number, usually the distance of back clipping plane, the problem of determining which surfaces are closer is reduced to simply comparing the present depth values stored in the $Z$ buffer at pixel $(x, y)$ with the newly calculated depth value at pixel $(x, y)$. If this new value is less than the present $Z$-buffer value (i.e., closer along the line of sight), this value replaces the present value and the pixel color is changed to the color of the new surface.

**10.8** Using a $2 \times 2$ pixel display, show how the $Z$-buffer algorithm would determine the color of each pixel for the given objects $A$ and $B$ in Fig. 10-20.

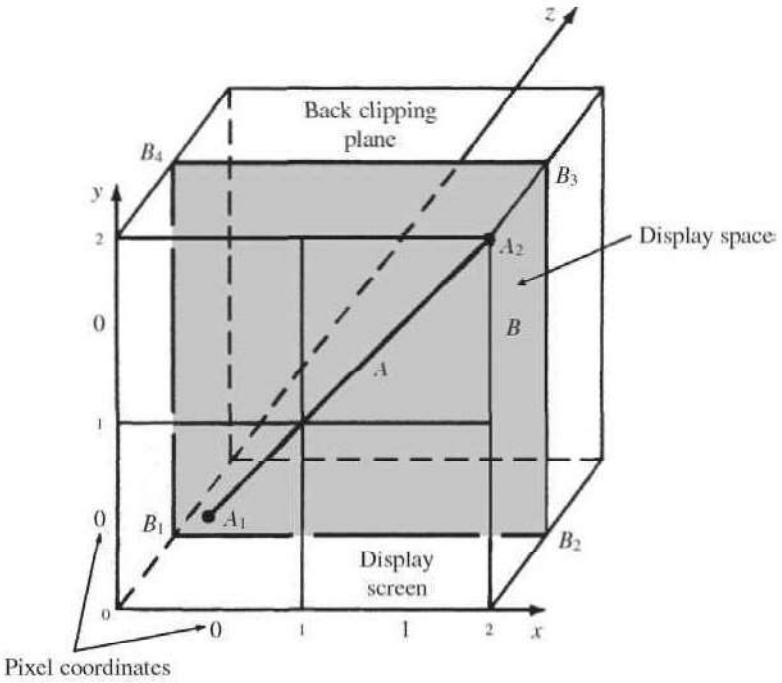



**Fig. 10-20**

**SOLUTION**

The display space for the $2 \times 2$ pixel display is the region $0 \le x \le 2$, $0 \le y \le 2$, and $0 \le z \le 1$. In Fig. 10-20, $A$ is the line with display space coordinates $A_1(\frac{1}{2}, \frac{1}{2}, 0)$ and $A_2(2, 2, 0)$; line $A$ is on the display screen in front of square $B$. $B$ is the square with display space coordinates $B_1(0, 0, \frac{1}{2})$, $B_2(2, 0, \frac{1}{2})$, $B_3(2, 2, \frac{1}{2})$, and $B_4(0, 2, \frac{1}{2})$. The displayed image of $A$ (after projection and scan conversion) would appear on a $2 \times 2$ pixel display as

| | | |
|---|---|---|
| 1 | $y$ | $a$ |
| 0 | $a$ | $y$ |
| | 0 | 1 |

where $a$ is the color of $A$ and $y$ is the background color. We have used the fact (Chap. 3, Sec. 3.1) that a point $(x, y)$ scan-converts to the pixel [Floor($x$), Floor($y$)]. (We assume for consistency that $2 \equiv 1.99\ldots$.) The displayed image of $B$ is

| | | |
|---|---|---|
| 1 | $b$ | $b$ |
| 0 | $b$ | $b$ |
| | 0 | 1 |

where $b$ is the color of $B$. We apply the $Z$-buffer algorithm to the picture composed of objects $A$ and $B$ as follows:

1. Perform initialization. The $Z$ buffer is set equal to the depth of the back clipping plane $z = 1$, and the frame buffer is initialized to a background color $y$.

Frame buffer $=$

| | | |
|---|---|---|
| 1 | $y$ | $y$ |
| 0 | $y$ | $y$ |
| | 0 | 1 |

$Z$ buffer $=$

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 0 | 1 | 1 |
| | 0 | 1 |

2. Apply the algorithm to object $A$.
   (*a*) The present $Z$-buffer value at pixel (0, 0) is that of the back clipping plane, i.e., $Z_{\text{buf}}(0, 0) = 1$. The depth vapue of $A$ at pixel (0, 0) is $z = 0$. Then $Z_{\text{buf}}(0, 0)$ is changed to 0 and pixel (0, 0) has the color of $A$.

Frame buffer $=$

| | | |
|---|---|---|
| 1 | $y$ | $y$ |
| 0 | $a$ | $y$ |
| | 0 | 1 |

$Z$ buffer $=$

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 0 | 0 | 1 |
| | 0 | 1 |

   (*b*) Object $A$ is not seen from pixel (1, 0), so the $Z$-buffer value is unchanged.

Frame $=$

| | | |
|---|---|---|
| 1 | $y$ | $y$ |
| 0 | $a$ | $y$ |
| | 0 | 1 |

$Z_{\text{buf}} =$

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 0 | 0 | 1 |
| | 0 | 1 |

   (*c*) Object $A$ is not seen from pixel (0, 1), so the $Z$-buffer value is unchanged.

Frame $=$

| | | |
|---|---|---|
| 1 | $y$ | $y$ |
| 0 | $a$ | $y$ |
| | 0 | 1 |

$Z_{\text{buf}} =$

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 0 | 0 | 1 |
| | 0 | 1 |

(*d*) The depth value of $A$ at pixel (1, 1) is 0. Since this is less than the present $Z$-buffer value of 1, pixel (1, 1) takes the color of $A$.

$$\text{Frame} = \begin{array}{c|cc} 1 & y & a \\ 0 & a & y \\ & 0 & 1 \end{array} \qquad Z_{\text{buf}} = \begin{array}{c|cc} 1 & 1 & 0 \\ 0 & 0 & 1 \\ & 0 & 1 \end{array}$$

3. Apply the algorithm to $B$.

(*a*) The depth value for $B$ at pixel (0, 0) is $\frac{1}{2}$, and $Z_{\text{buf}}(0, 0) = 0$. So the color of pixel (0, 0) is unchanged.

$$\text{Frame} = \begin{array}{c|cc} 1 & y & a \\ 0 & a & y \\ & 0 & 1 \end{array} \qquad Z_{\text{buf}} = \begin{array}{c|cc} 1 & 1 & 0 \\ 0 & 0 & 1 \\ & 0 & 1 \end{array}$$

(*b*) The depth value of $B$ at pixel (1, 0) is $\frac{1}{2}$. The present $Z$-buffer value is 1. So the $Z$-buffer value is set to $\frac{1}{2}$ and pixel (1, 0) takes the color of $B$.

$$\text{Frame} = \begin{array}{c|cc} 1 & y & a \\ 0 & a & b \\ & 0 & 1 \end{array} \qquad Z_{\text{buf}} = \begin{array}{c|cc} 1 & 1 & 0 \\ 0 & 0 & \frac{1}{2} \\ & 0 & 1 \end{array}$$

(*c*) The depth value of $B$ at pixel (0, 1) is $\frac{1}{2}$. The present $Z$-buffer value is 1, so the color at pixel (0, 1) is set to that of $B$, and the $Z$ buffer is updated.

$$\text{Frame} = \begin{array}{c|cc} 1 & b & a \\ 0 & a & b \\ & 0 & 1 \end{array} \qquad Z_{\text{buf}} = \begin{array}{c|cc} 1 & \frac{1}{2} & 0 \\ 0 & 0 & \frac{1}{2} \\ & 0 & 1 \end{array}$$

(*d*) The depth value of $B$ at pixel (1, 1) is $\frac{1}{2}$. The present $Z$-buffer value is 0. So the color at pixel (1, 1) remains unchanged.

$$\text{Frame} = \begin{array}{c|cc} 1 & b & a \\ 0 & a & b \\ & 0 & 1 \end{array} \qquad Z_{\text{buf}} = \begin{array}{c|cc} 1 & \frac{1}{2} & 0 \\ 0 & 0 & \frac{1}{2} \\ & 0 & 1 \end{array}$$

The final form of the $Z$ buffer indicates that line $A$ lines in front of $B$.

**10.9** What is the maximum number of objects that can be presented by using the $Z$-buffer algorithm?

**SOLUTION**

The total number of objects that can be handled by the $Z$-buffer algorithm is arbitrary because each object is processed one at a time.

**10.10** How does the basic scan-line method determine which surfaces are hidden?

**SOLUTION**

The basic scan-line method looks one at a time at each of the horizontal lines of pixels in the display area. For example, at the horizontal pixel line $y = \alpha$, the graphics data structure (consisting of all scan-converted polygons) is searched to find all polygons with any horizontal ($y$) pixel values equal to $\alpha$.