4.5 VISIBLE SURFACE DETECTION METHODES

A major consideration in the generation of realistic graphics displays is identifying those parts of a scene that are visible from a chosen viewing position. There are many approaches we can take to solve this problem, and numerous algorithms have been devised for efficient identification of visible objects for different types of applications. Some methods require more memory, some involve more processing time, and some apply only to special types of objects. Deciding upon a method for a particular application can depend on such factors as the complexity of the scene, type of objects to be displayed, available equipment, and whether static or animated displays are to be generated. The various algorithms are referred to as visible-surface detection methods. Sometimes these methods are also referred to as hidden-surface elimination methods, although there can be subtle differences between identifying visible surfaces and eliminating hidden surfaces. For wireframe displays, for example, we may not want to actually eliminate the hidden surfaces, but rather to display them with dashed boundaries or in some other way to retain information about their shape. In this chapter, we explore some of the most commonly used methods for detecting visible surfaces in a three-dimensional scene.

4.5.1 CLASSIFICATION OF VISIBLE-SURFACE DETECTION ALGORITHMS

Visible-surface detection algorithms are broadly classified according to whether they deal with object definitions directly or with their projected images. These two approaches are called object-space methods and image-space methods, respectively.

An object-space method compares objects and parts of objects to each other within the scene definition to determine which surfaces, as a whole, we should label as visible. In an image-space algorithm, visibility is decided point by point at each pixel position on the projection plane. Most visible-surface algorithms use image-space methods, although object space methods can be used effectively to locate visible surfaces in some cases. Line display algorithms, on the other hand, generally use object-space methods to identify visible lines in wireframe displays, but many image-space visible-surface algorithms can be adapted easily to visible-line detection.

Although there are major differences in the basic approach taken by the various visible-surface detection algorithms, most use sorting and coherence methods to improve performance. Sorting is used to facilitate depth comparisons by ordering the individual surfaces in a scene according to their distance from the view plane. Coherence methods are used to take advantage of regularities in a scene. An individual scan line can be expected to contain intervals (runs) of constant pixel intensities, and scan-line patterns often change little from one line to the next. Animation frames

contain changes only in the vicinity of moving objects. And constant relationships often can be established between objects and surfaces in a scene.

4.5.2 BACK-FACE DETECTION

A fast and simple object-space method for identifying the back faces of a polyhedron is based on the "inside-outside" tests. A point *(x*, y, z) is "inside" a polygon surface with plane parameters A, B, C, and D if When an inside point is along the line of sight to the surface, the polygon must be a back face (we are inside that face and cannot see the front of it from our viewing position).

We can simplify this test by considering the normal vector N to a polygon surface, which has Cartesian components *(A, B, C).* In general, if V is a vector in the viewing direction from the eye (or "camera") position, then this polygon is a back face if

$$\mathbf{V.N>0}$$

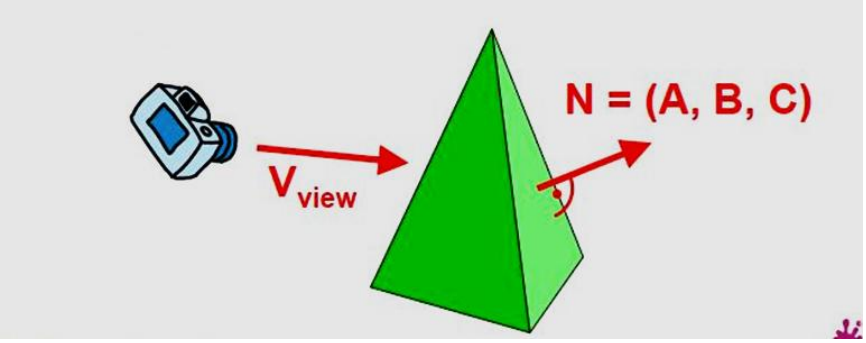
Furthermore, if object descriptions have been converted to projection coordinates and our viewing direction is parallel to the viewing z-axis, then **V = (0, 0, V_z)** and

$$\mathbf{V.N=V_ZC}$$

So that we only need to consider the sign of C, the ; component of the normal vector **N**

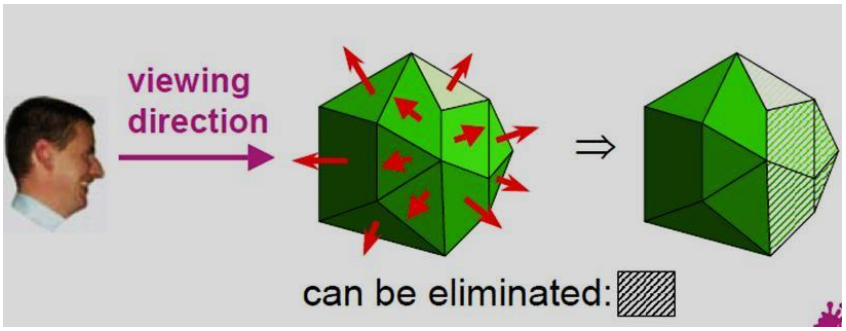
In a right-handed viewing system with viewing direction along the negative z_V axis, the polygon is a back face if C < 0. AIso, we cannot see any face whose normal has z component C = 0, since our viewing direction is grazing that polygon. Thus, in general, we can label any polygon as a back face if its normal vector has a z component value:

$$\mathbf{C<=0}$$



Similar methods can be used in packages that employ a left-handed viewing system. In these packages, plane parameters A, B, C and D can be calculated from polygon vertex coordinates specified in a clockwise direction (instead of the counterclockwise direction used in a right-handed system). Also, back faces have

normal vectors that point away from the viewing position and are identified by C >= 0 when the viewing direction is along the positive z_v axis. By examining parameter C for the different planes defining an object, we can immediately identify all the back faces.



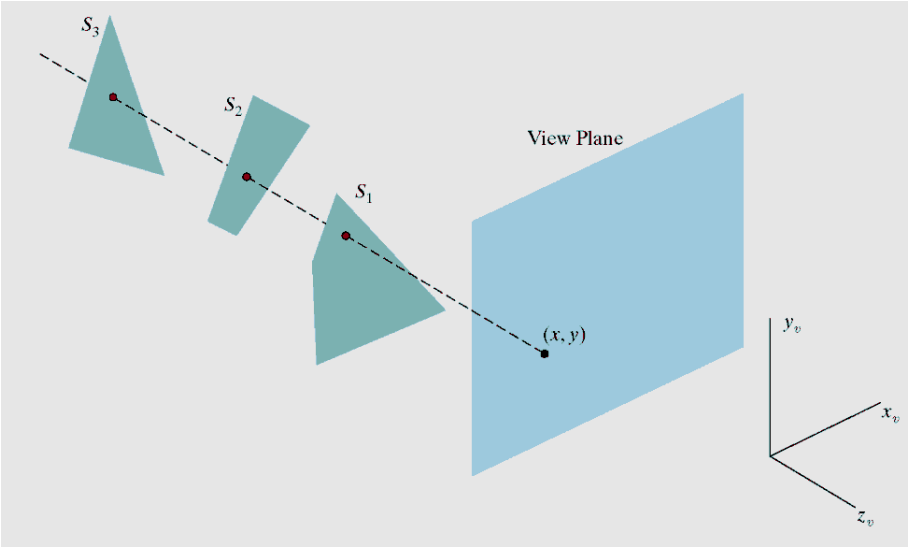
4.5.3 DEPTH-BUFFER METHOD

A commonly used image-space approach to detecting visible surfaces is the depth-buffer method, which compares surface depths at each pixel position on the projection plane. This procedure is also referred to as the z-buffer method, since object depth is usually measured from the view plane along the z axis of a viewing system. Each surface of a scene is processed separately, one point at a time across the surface. The method is usually applied to scenes containing only polygon surfaces, because depth values can be computed very quickly and the method is easy to implement. But the method can be applied to nonplanar surfaces With object descriptions converted to projection coordinates, each *(x,* y, *2)* position on a polygon surface corresponds to the orthographic projection point *(x,* y) on the view plane. Therefore, for each pixel position *(x,* y) on the view plane, object depths can be compared by comparing z values. **Figure: 4.12** shows three surfaces at varying distances along the orthographic projection line from position *(x,* y) in a view plane taken as the $x_v y_v$ plane. Surface S_1, is closest at this position, so its surface intensity value at (x, y) is saved.

We can implement the depth-buffer algorithm in normalized coordinates, so that *z* values range from 0 at the back clipping plane to Z_{max} at the front clipping plane. The value of Z_{max} can be set either to 1 (for a unit cube) or to the largest value that can be stored on the system.

As implied by the name of this method, two buffer areas are required. A depth buffer is used to store depth values for each (x, y) position as surfaces are processed, and the refresh buffer stores the intensity values for each position. Initially, all positions in the depth buffer are set to 0 (minimum depth), and the refresh buffer is initialized to the background intensity. Each surface listed in the polygon tables is then processed, one scan line at a time, calculating the depth (z value) at each (x, y) pixel position. The calculated depth is compared to the value previously stored in the depth buffer at that position. If the calculated depth is greater than the value stored in the depth buffer, the

new depth value is stored, and the surface intensity at that position is determined and in the same *xy* location in the refresh buffer.



(Figure 4 - 12)

We summarize the steps of a depth-buffer algorithm as follows:

1. Initialize the depth buffer and refresh buffer so that for all buffer positions (x, y),

$$\text{depth}(x, y) = 0, \qquad \text{refresh}(x, y) = I_{\text{backgnd}}$$

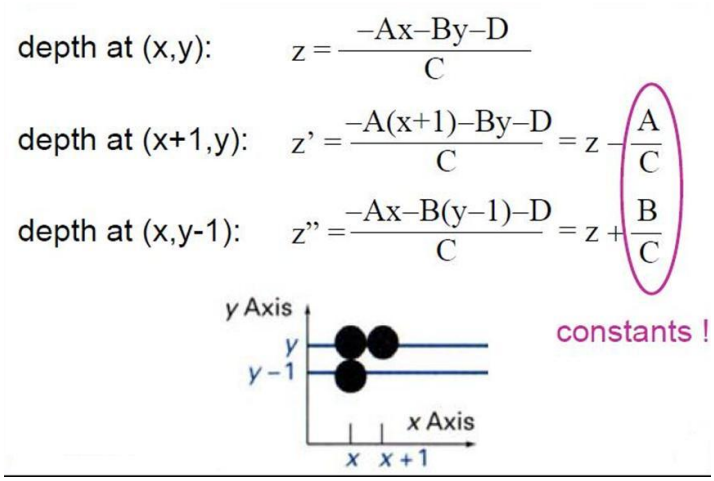
2. For each position on each polygon surface, compare depth values to previously stored values in the depth buffer to determine visibility.

   - Calculate the depth z for each (x, y) position on the polygon.
   - If $z > \text{depth}(x, y)$, then set

$$\text{depth}(x, y) = z, \qquad \text{refresh}(x, y) = I_{\text{surf}}(x,y)$$

where I_{backgnd} is the value for the background intensity, and $I_{\text{surf}}(x,y)$ is the projected intensity value for the surface at pixel position (x,y). After all surfaces have been processed, the depth buffer contains depth values for the visible surfaces and the refresh buffer contains the corresponding intensity values for those surfaces.

Depth values for a surface position *(x*, y) are calculated from the plane equation for each surface:

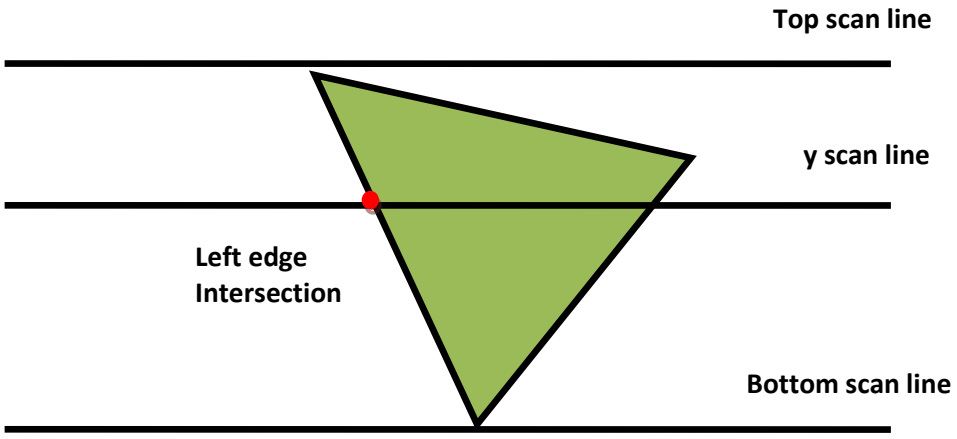


(Fig. 4-13)

For any scan line **(Fig. 4-13),** adjacent horizontal positions across the line differ by 1, and a vertical y value on an adjacent scan line differs by 1. If the depth of position *(x*, y) has been determined to be z, then the depth *z'* of the next position (x + 1, y) along the scan line is obtained from Equation of (x, y) as *Y - 1*

The ratio *-A/C* is constant for each surface, so succeeding depth values across a scan line are obtained from preceding values with a single addition. On each scan line, we start by calculating the depth on a left edge of the polygon that intersects that scan line **(Fig. 4-14).** Depth values at each successive position across the scan line are then calculated by the previous equations. We first determine the y-coordinate extents of each polygon, and process the surface from the topmost scan line to the bottom scan line, as shown in **Fig. 4-14.** Starting at a top vertex, we can recursively calculate x positions down a left edge of the polygon **as x' = *x* - l/m**, where *m* is the slope of the edge **(Fig. 4-15)**. Depth values down the edge are then obtained recursively as

$$z' = z + \frac{A/m + B}{C}$$

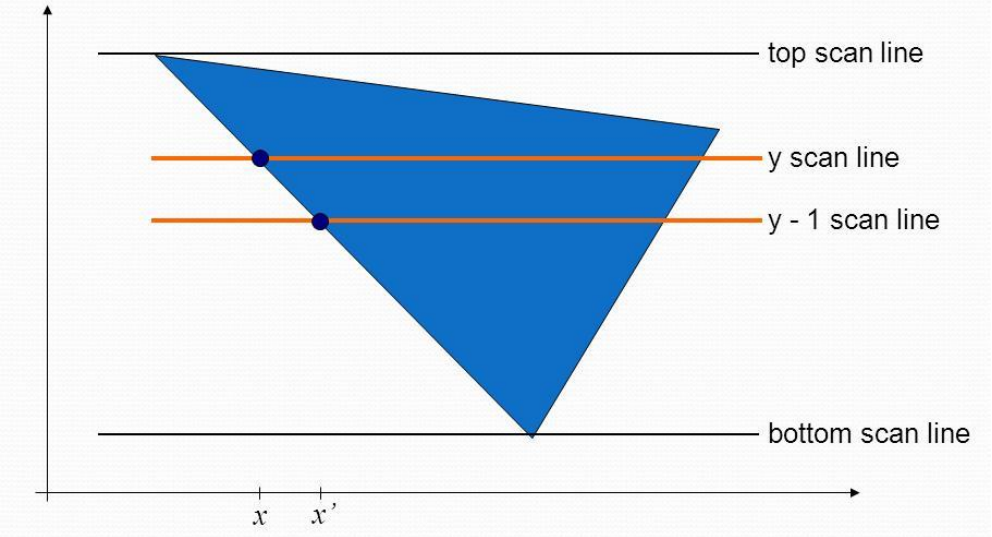


(Fig. 4-14)

If we are processing down a vertical edge, the slope is infinite and the recursive calculations reduce to

$$z' = z + \frac{B}{C}$$

An alternate approach is to use a midpoint method or Bresenham-type algorithm for determining x values on left edges for each scan line. Also the method can be applied to curved surfaces by determining depth and intensity values at each surface projection point.

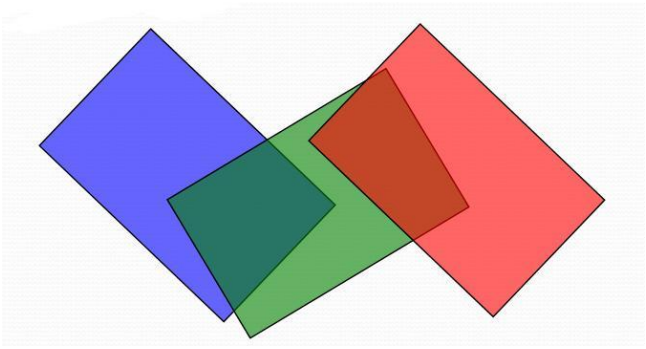


(Fig. 4-15)

For polygon surfaces, the depth-buffer method is very easy to implement, and it requires no sorting of the surfaces in a scene. But it does require the availability of a second buffer in addition to the refresh buffer. A system with a resolution of 1024 by 1024, for example, would require over a million positions in the depth buffer, with each position containing enough bits to represent the number of depth increments needed. One way to reduce storage requirements is to process one section of the scene at a time, using a smaller depth buffer. After each view section is processed, the buffer is reused for the next section.

4.5.4 A-BUFFER METHOD

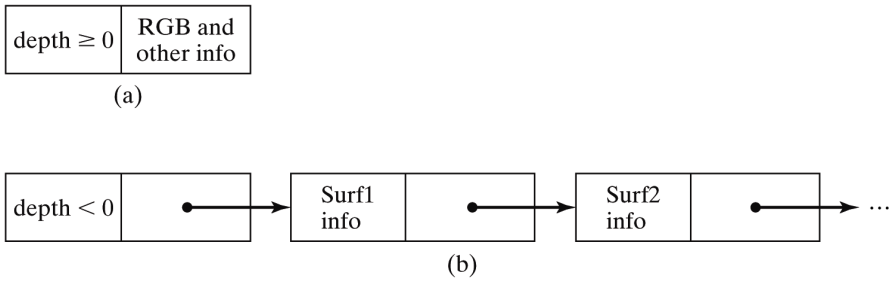
The A-buffer method is an extension of the depth-buffer method.The A-buffer method is visibility detection method developed at Lucas film Studios for the rendering system REYES (Renders Everything You Ever Saw) The A-buffer expands on the depth buffer method to allow transparencies The key data structure in the A-buffer is the *accumulation buffer.*



(Fig. 4-16)

Each position in the A-buffer has two fields

- **Depth field**- stores a positive or negative real number
- **Intensity field**--stores surface-intensity information or a pointer value



If depth is >= 0, the number stored at that position is the depth of a single surface overlapping the corresponding pixel area.

The intensity field then stores the RGB components of the surface color at that point and the percent of pixel coverage.

If depth < 0 this indicates multiple-surface contributions to the pixel intensity. The intensity field then stores a pointer to a linked List of surface data

Surface information in the A-buffer includes:

a. RGB intensity components

b. Opacity parameter

c. Depth

d. Percept of area coverage

e. Surface identifier

f. Other surface rendering parameters

The algorithm proceeds just like the depth buffer algorithm.The depth and opacity values are used to determine the final colour of a pixel

4.5.5 SCAN-LINE METHOD

An image space method for identifying visible surfaces. Computes and compares depth values along the various scan-lines for a scene
Two important tables are maintained:

- **The edge table**
- **The POLYGON table**

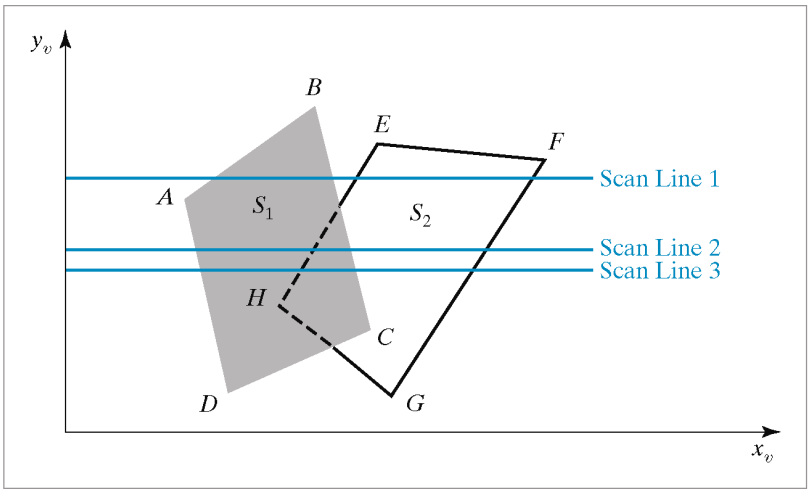
The **edge table** contains:

- Coordinate end points of each line in the scene
- The inverse slope of each line
- Pointers into the POLYGON table to connect edges to surfaces

The **POLYGON** table contains:

- The plane coefficients
- Surface material properties
- Other surface data
- Maybe pointers into the edge table

To facilitate the search for surfaces crossing a given scan-line an active list of edges is formed for each scan-line as it is processed. The active list stores only those edges that cross the scan-line in order of increasing x. Also a flag is set for each surface to indicate whether a position along a scan-line is either inside or outside the surface Pixel positions across each scan-line are processed from left to right. At the left intersection with a surface the surface flag is turned on. At the right intersection point the flag is turned off. We only need to perform depth calculations when more than one surface has its flag turned on at a certain scan-line position



(Fig. 4-17)

Figure 4-17 illustrates the scan-line method for locating visible portions of surfaces for pixel positions along the line. The active list for line 1 contains information from the edge table for edges *AB,* BC, *EH,* and *FG.* For positions along this scan line between edges *AB* and BC, only the flag for surface S_1 is on. Therefore no depth calculations are necessary, and intensity information for surface S_1, is entered from the polygon table into the refresh buffer. Similarly, between edges EH and FG, only the flag for surface S_2 is on. NO other positions along scan line 1 intersect surfaces, so the intensity values in the other areas are set to the background intensity. The background intensity can be loaded throughout the buffer in an initialization routine.

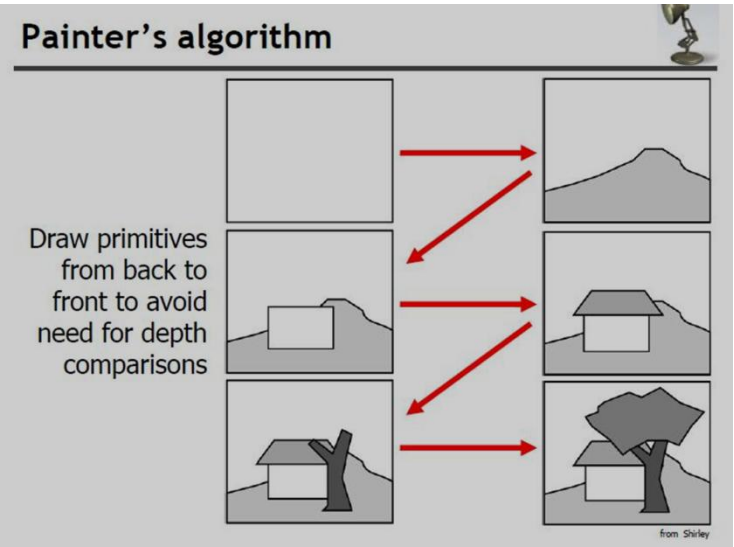
For scan lines 2 and 3 in **Fig. 4-17**, the active edge list contains edges *AD, EH, BC,* and *FG.* Along scan line 2 from edge *AD* to edge *EH,* only the flag for surface S_1 is on. But between edges EH and *BC,* the flags for both surfaces are on. In this interval, depth calculations must be made using the plane coefficients for the two surfaces. For this example, the depth of surface S_1 is assumed to be less than that of S_2, so intensities for surface S_1 are loaded into the refresh buffer until boundary BC is encountered. Then the flag for surface S_1 goes off, and intensities for surface *S2* are stored until edge FG is passed.

We can take advantage of-coherence along the scan lines as we pass from one scan line to the next. In **Fig. 4-17**, scan line 3 has the same active list of edges as scan line 2. Since no changes have occurred in line intersections, it is unnecessary again to make depth calculations between edges *EH* and *BC.* The two surfaces must be in the same orientation as determined on scan line 2, so the intensities for surface S_1 can be entered without further calculations.

4.5.6 DEPTH SORTING METHOD

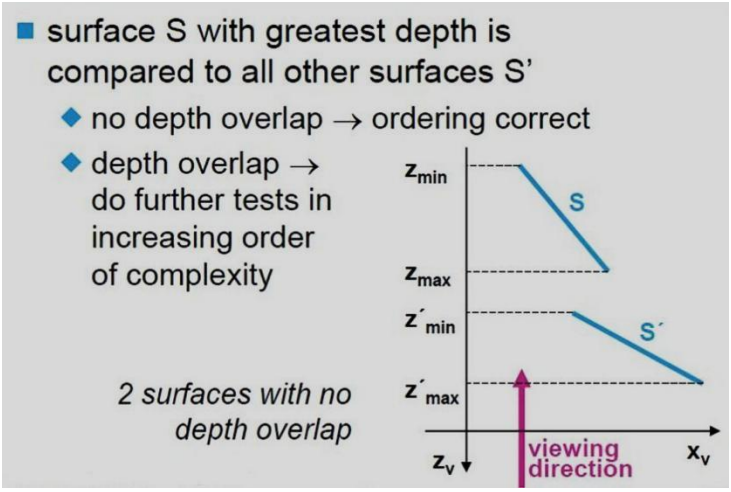
Using both image space and object-space operations. The depth-sorting method performs the following basic functions:

- ► Surfaces are sorted in order of decreasing depth.
- ► Surfaces are scan converted in order, starting with the surface of greatest depth.
- Sorting operations are carried out in both image and object space.
- The scan conversion of the polygon surfaces is performed in image space.
- This method for solving the hidden-surface problem is often referred to as the painter's algorithm



(Fig. 4-18)

First sort surfaces according to their distance from the view plane. The intensity values for the farthest surface are then entered into the refresh buffer. Taking each succeeding surface in turn (in decreasing depth order), we "paint" the surface intensities onto the frame buffer over the intensities of the previously processed surfaces.

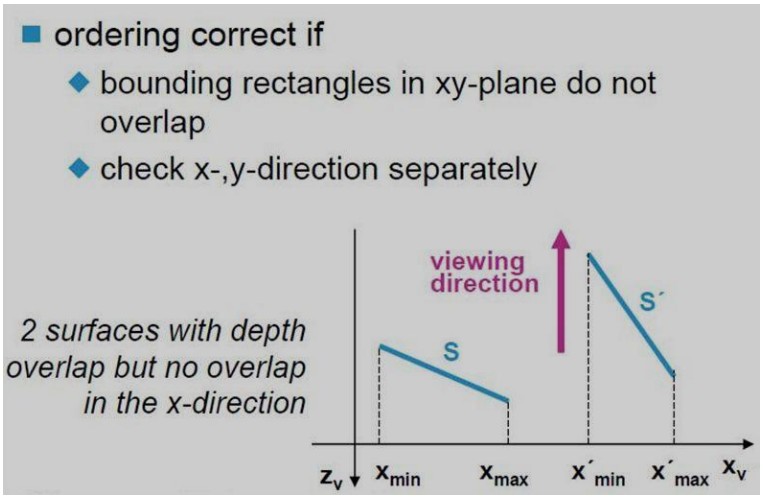


If a depth overlap is detected at any point in the list, we need to make some additional comparisons to determine whether any of the surfaces should be reordered.

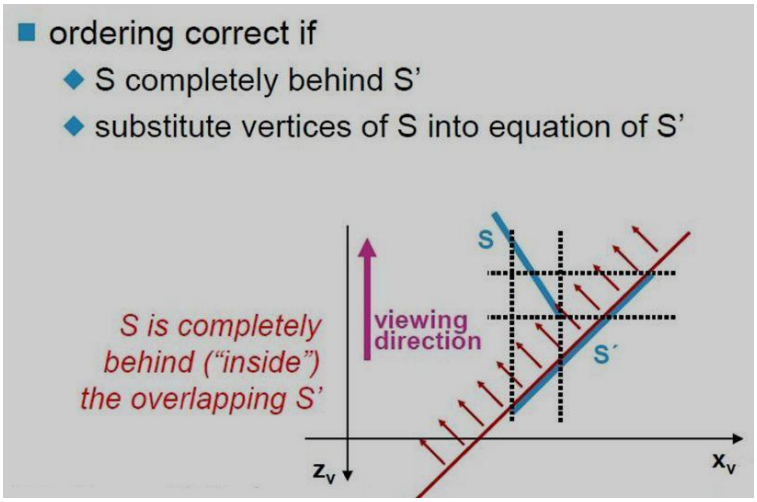
We make the following tests for each surface that overlaps with S. If any one of these tests is true, no reordering is necessary for that surface. The tests are listed in order of increasing difficulty.

1) The bounding rectangles in the xy plane for the two surfaces do not overlap
2) Surface S is completely behind the overlapping surface relative to the viewing position.
3) The overlapping surface is completely in front of S relative to the viewing position.

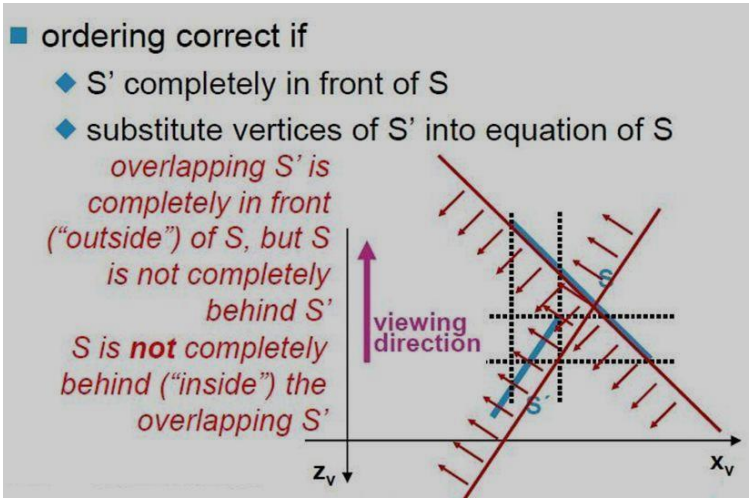
4) The projections of the two surfaces onto the view plane do not overlap.



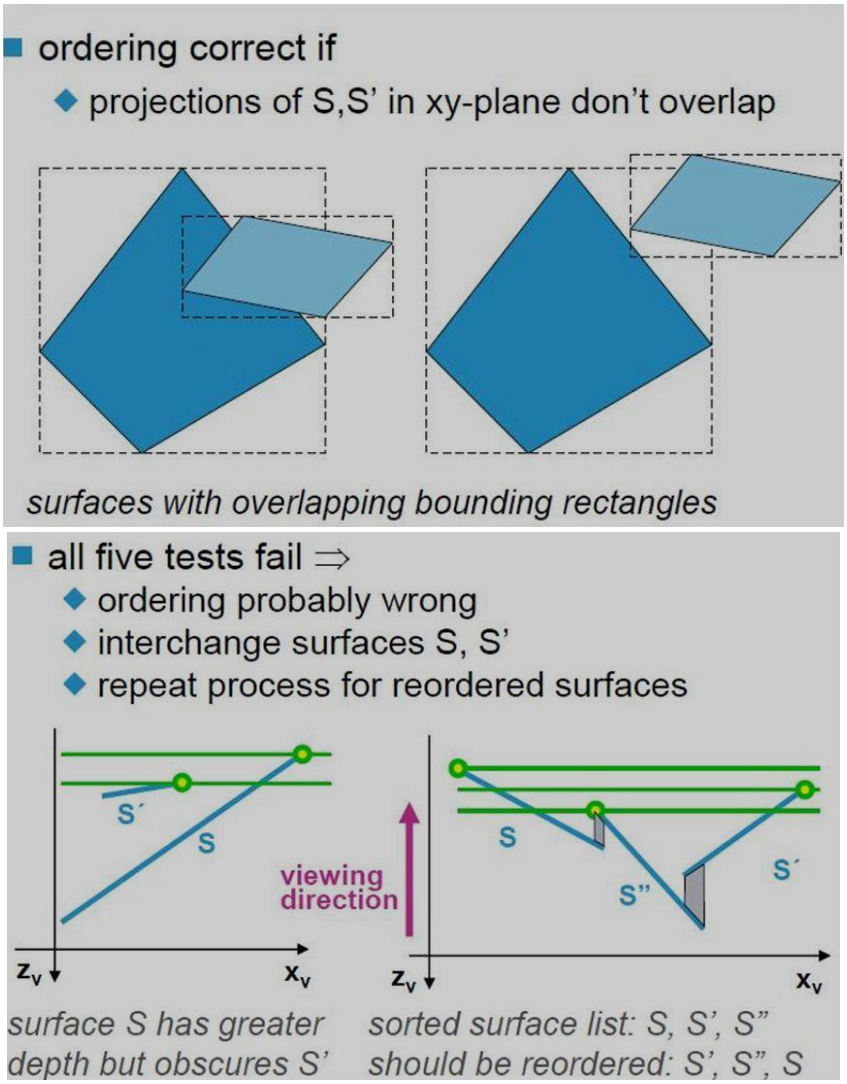
The coordinates for all vertices of S into the plane equation for the overlapping surface and check the sign of the result. If the plane equations are set up so that the outside of the surface is toward the viewing position, then *S* is behind S' if all vertices of *S* are "inside" S'.



S' is completely in front of S if all vertices of S are "outside" of *S'*.



If tests 1 through 3 have all failed, we try test 4 by checking for intersections between the bounding edges of the two surfaces using line equations in the xy plane.



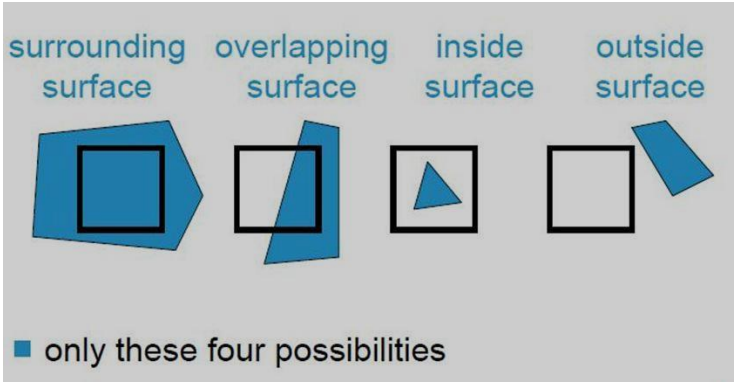
4.5.7 AREA-SUBDIVISION METHOD

The area-subdivision method takes advantage of area coherence in a scene by locating those view areas that represent part of a single surface. We apply this method by successively dividing the total viewing area into smaller and smaller rectangles until each small area is the projection of part of a single visible surface or no surface at all. We continue this process until the subdivisions are easily analyzed as belonging to a single surface or until they are reduced to the size of a single pixel.

An easy way to do this is to successively divide the area into four equal parts at each step. There are four possible relationships that a surface can have with a specified area boundary

- **Surrounding surface**-One that completely encloses the area.
- **Overlapping surface**-One that is partly inside and partly outside the area.

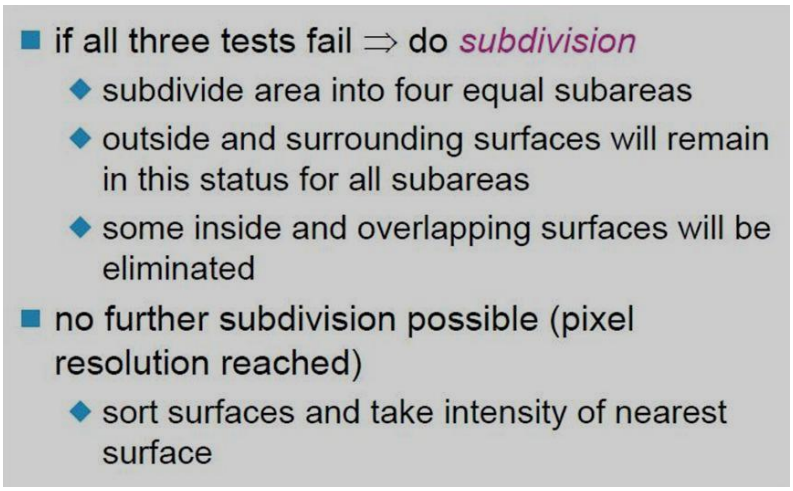
- **Inside surface**-One that is completely inside the area.
- **Outside surface**-One that is completely outside the area.



The tests for determining surface visibility within an area can be stated in terms of these four classifications.

No further subdivisions of a specified area are needed if one of the following conditions is true:

- All surfaces are outside surfaces with respect to the area.
- Only one inside, overlapping, or surrounding surface is in the area.
- A surrounding surface obscures all other surfaces within the area boundaries.



Example for **Area-Subdivision Method**