UNIT 2 VISIBLE-SURFACE DETECTION

Structure **Page Nos.**

2.0 INTRODUCTION

Given a set of 3-D objects and a viewing position. For the generation of realistic graphics display, we wish to determine which lines or surfaces of the objects are visible, either from the COP (for perspective projections) or along the direction of projection (for parallel projections), so that we can display only the visible lines or surfaces. For this, we need to conduct visibility tests. Visibility tests are conducted to determine the surface that is visible from a given viewpoint. This process is known as *visible-line* or *visible-surface determination*, or *hidden-line* or *hidden-surface elimination.*

To illustrate the concept for eliminating hidden-lines, edges or surfaces, consider a typical wire frame model of a cube (see *Figure 1*). A wire frame model represents a 3-D object as a line drawing of its edges. In *Figure 1(b)*, the dotted line shows the edges obscured by the top, left, front side of the cube. These lines are removed in *Figure 1(c)*, which results in a realistic view of the object. Depending on the specified viewing position, particular edges are eliminated in the graphics display. Similarly, *Figure 2(a)* represents more complex model and *Figure 2(b)* is a realistic view of the object, after removing hidden lines or edges.

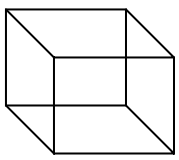


Figure1 (a)

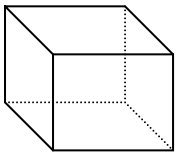


Figure1 (b)

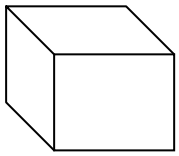


Figure 1(c)

.

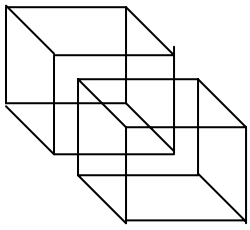


Figure 2(a)

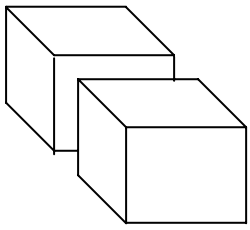


Figure 2(b)

There are numerous algorithms for 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. These requirements have encouraged the development of carefully structured visible surface algorithms.

There are two fundamental approaches for visible-surface determination, according to whether they deal with their projected images or with object definitions directly. These two approaches are called *image-space approach* and *object-space approach*, respectively. Object space methods are implemented in the physical coordinate system in which objects are defined whereas image space methods are implemented in screen coordinate system in which the objects are viewed.

In both cases, we can think of each object as comprising one or more polygons (or more complex surfaces). The first approach (image-space) determines which of n objects in the scene is visible at each pixel in the image. The pseudocode for this approach looks like as:

```
for(each pixel in the image)
{
   determine the object closest to the viewer that is passed by the projector
through the pixel;
    draw the pixel in the appropriate color;
}
```

This approach requires examining all the objects in the scene to determine which is closest to the viewer along the projector passing through the pixel. That is, in an image-space algorithm, the visibility is decided point by point at each pixel position on the projection plane. If the number of objects is 'n' and the pixels is 'p' then effort is proportional to n.p.

The second approach (object-space) compares all objects directly with each other within the scene definition and eliminates those objects or portion of objects that are not visible. In terms of pseudocode, we have:

```
for (each object in the world)
{
   determine those parts of the object whose view is unobstructed (not blocked)
by other
    parts of it or any other object;
    draw those parts in the appropriate color;
}
```

This approach compares each of the *n* objects to itself and to the other objects, and discarding invisible portions. Thus, the computational effort is proportional to n^2.

Image-space approaches require two buffers: one for storing the pixel intensities and another for updating the depth of the visible surfaces from the view plane.

In this unit, under the categories of image space approach, we will discuss two methods, namely, *Z-buffer (or Depth-buffer) method* and *Scan-line method*. Among all the algorithms for visible surface determination, the Depth-buffer is perhaps the simplest, and is the most widely used. *Z-buffer method*, detects the visible surfaces by comparing surface depths (*z*-values) at each pixel position on the projection plane. In *Scan-line method*, all polygon surfaces intersecting the scan-line are examined to determine which surfaces are visible on the basis of depth calculations from the view plane. For scenes with more than one thousand polygon surfaces, *Z*-buffer method is the best choice. This method has nearly constant processing time, independent of

number of surfaces in a scene. The performance of *Z*-buffer method is low for simple scenes and high with complex scenes. Scan-line methods are effectively used for scenes with up to thousand polygon surfaces.

The third approach often combines both object and image-space calculations. This approach utilizes depth for sorting (or reordering) of surfaces. They compare the depth of overlapping surfaces and identify one that is closer to the view-plane. The methods in this category also use image-space for conducting visibility tests.

Area-subdivision method is essentially an image-space method but uses object-space calculations for reordering of surfaces according to depth. The method makes use of area coherence in a scene by collecting those areas that form part of a single surface. In this method, we successively subdivide the total viewing area into small rectangles until each small area is the projection of part of a single visible surface or no surface at all.

2.1 OBJECTIVES

After going through this unit, you should be able to:

- understand the meaning of Visible-surface detection;
- distinguish between image-space and object-space approach for visible-surface determination;
- describe and develop the depth-buffer method for visible-surface determination;
- describe and develop the Scan-line method for visible-surface determination, and
- describe and develop the Area-Subdivision method for visible-surface determination.

2.2 VISIBLE-SURFACE DETECTION

As you know for the generation of realistic graphics display, hidden surfaces and hidden lines must be identified for elimination. For this purpose we need to conduct visibility tests. Visibility tests try to identify the visible surfaces or visible edges that are visible from a given viewpoint. Visibility tests are performed by making use of either i) *object-space* or ii) *image-space* or iii) both *object-space* and *image-spaces*.

Object-space approaches use the directions of a surface normal w.r.t. a viewing direction to detect a back face. *Image-space* approaches utilize two buffers: one for storing the pixel intensities and another for updating the depth of the visible surfaces from the view plane. A method, which uses both *object-space* and *image-space,* utilizes depth for sorting (or reordering) of surfaces. The methods in this category also use image-space for conducting visibility tests. While making visibility tests, coherency property is utilized to make the method very fast.

In this section, we will discuss three methods (or algorithms) for detecting visible surfaces:

- Depth-buffer method
- Scan-line method
- Area subdivision method

Depth-buffer method and Scan-line method come under the category of image-space, and area-subdivision method uses both object-space and image-space approach.

2.2.1 Depth-buffer (or z-buffer) Method

Depth-buffer method is a fast and simple technique for identifying visible-surfaces. This method 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. This algorithm compares surface depths at each pixel position (x,y) on the view plane. Here we are taking the following assumption:

- Plane of projection is z=0 plane
- Orthographic parallel projection.

For each pixel position (x,y) on the view plane, the surface with the smallest z-coordinate at that position is visible. For example, *Figure 3* shows three surfaces S1, S2, and S3, out of which surface S1 has the smallest z-value at (x,y) position. So surface S1 is visible at that position. So its surface intensity value at (x,y) is saved in the refresh-buffer.

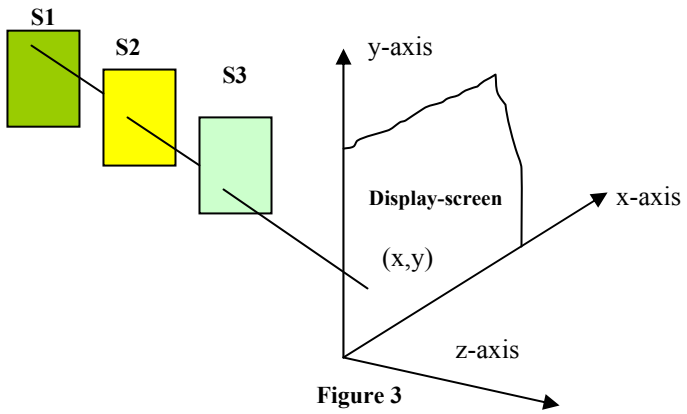


Figure 3

Here the projection is orthographic and the projection plane is taken as the xy-plane. So, each (x,y,z) position on the polygon surfaces corresponds to the orthographic projection point (x,y) on the projection plane. Therefore, for each pixel position (x,y) on the view plane, object depth can be compared by comparing z-values, as shown in *Figure 3*.

For implementing z-buffer algorithm two buffer areas (two 2-D arrays) are required.

1) Depth-buffer: **z-buffer**(i,j) , to store z-value, with least z, among the earlier z-values for each (x,y) position on the view plane.
2) Refresh-buffer: **COLOR**(i,j): for storing intensity values for each position.

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

Given: A list of polygons {P1,P2,…..,Pn}.
Step1: Initially all positions (x,y) in the depth-buffer are set to 1.0 (maximum depth) and the refresh-buffer is initialized to the background intensity i.e.,
z-buffer*(x,y):=1.0; and*
COLOR*(x,y):= Background color.*

Step2: For each position on each polygon surface (listed in the polygon table) is then processed (scan-converted), one scan line at a time. Calculating the depth (z-value) at each (x,y) pixel position. The calculated depth is then compared to the value previously stored in the depth buffer at that position to determine visibility.
a) If the calculated z-depth is less than the value stored in the depth-buffer, the new depth value is stored in the depth-buffer, and the surface intensity at

that position is determined and placed in the same (x,y) location in the refresh-buffer, i.e.,

If z-depth< ***z-buffer***(x,y), then set
z-buffer*(x,y)=z-depth;*
COLOR*(x,y)=*I_{surf}*(x,y);* // where $I_{surf}(x,y)$ is the projected intensity value of the polygon surface Pi 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.

In terms of pseudo code, we summarize the depth-buffer algorithm as follows:

Given: *A list of polygons {P1,P2,.....,Pn}*
Output: A COLOR array, which display the intensity of the visible polygon surfaces.

```
Initialize:
        z-buffer(x,y):=0; and
       COLOR(x,y):= Back-ground color.
Begin
    For (each polygon P in the polygon list) do {
     For (each pixel (x,y) that intersects P) do  {
        Calculate z-depth of P at (x,y)
         If (z-depth < z-buffer[x,y]) then  {
          z-buffer(x,y)=z-depth;
           COLOR(x,y)=Intensity of P at (x,y);
         }
      }
    }
display COLOR array.
```

Calculation of depth values, z, for a surface position (x,y):

We know that for any polygon faces, the equation of the plane is of the form:

A.x+B.y+C.z+D=0 ----------(1) , where A, B, C, D are known to us.

To calculate the depth value z, we have to solve the plane equation (1) for z:

z=(– A. x – B . y – D)/C --------(2)

Consider a polygon in *Figure 4* intersected by scan-lines at y and y – 1 on y-axis.

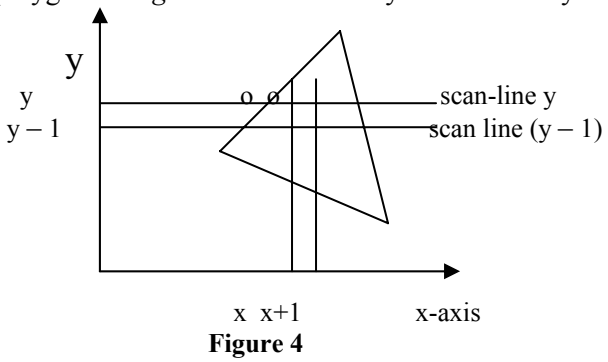


Figure 4

Now, if at position (x, y) equation (2) evaluates to depth z, then at next position (x+1,y) along the scan line, the depth z_H can be obtained as:

$z_H = [-A.(x+1) - B.y - D]/C$ ----------(3)

From equation (2) and (3), we have

$z - z_H = A/C$

$z_H = z - A/C$ ----------(4)

The ratio –A/C is constant for each surface. So we can obtain succeeding depth values across a scan-line from the preceding values by a single addition. On each scan-line, we start by calculating the depth on the left edge of the polygon that intersects that scan-line and then proceed to calculate the depth at each successive position across the scan -line by Equation-(4) till we reach the right edge of the polygon.

Similarly, if we are processing down, the vertical line x intersecting the (y–1)th scan-line at the point (x, y-1). Thus from Equation (2) the depth z_v is obtained as:

$z_v = [-A.x - B.(y-1) - D]/C$

$= ([-A.x - B.y - D]/C) + B/C$

$= z + B/C$ ----------(5)

Starting at the top vertex, we can recursively calculate the x position down the left edge of the polygon from the relation: x'=x-1/m, where m is the slope of the edge (see *Figure 5*). Using this x position, the depth z' at (x',y-1) on the (y-1) scan-line is obtained as:

$z' = [-A.x' - B.(y-1) - D]/C$

$= [-A.(x - 1/m) - B.(y-1) - D]/C$

$= z + (A/m + B)/C$ ----------(6)

Since $m = \infty$ for a vertical line, Equation (6) becomes equation (5).

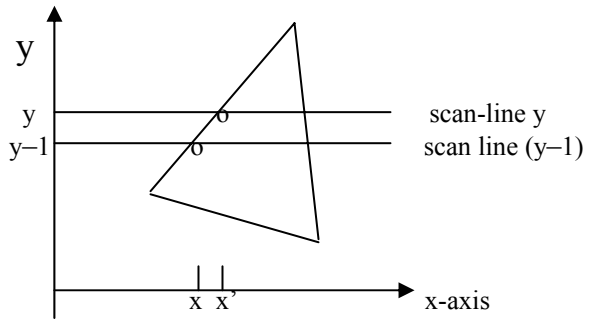


Figure 5: Intersection position on successive scan lines along a left polygon edge

Thus, if we are processing down, then we can obtain succeeding depth values across a scan-line from the preceding values by a single addition by using Equation (5), i.e., $z_v = z + B/C$.

Thus, the summary of the above calculations are as follows:

- You can obtain succeeding depth values across a scan-line from the preceding values by a single subtraction, i.e., z'=z–A/C .
- If we are processing down, then we can also obtain succeeding depth values across a scan-line from the preceding values by a single addition, i.e., z'= z+(A/m+B)/C . In other words, if we are processing up, then we can obtain succeeding depth values across a scan-line from the preceding values by a single subtraction, i.e., z'= z– (A/m+B)/C.

The following *Figure 6* summarizes the above calculations.

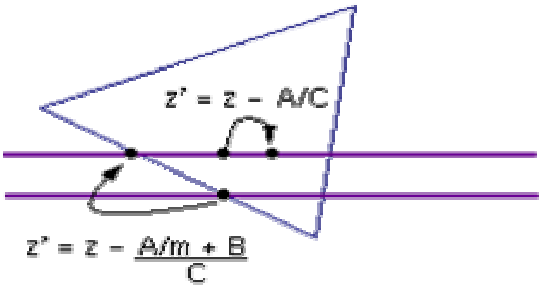


Figure 6: Successive depth values, when processing left to right or processing up across a scan-line

Advantages (z-buffer method):

1) The z-buffer method is easy to implement and it requires no sorting of surface in a scene.
2) In z-buffer algorithm, an arbitrary number of objects can be handled because each object is processed one at a time. The number of objects is limited only by the computer's memory to store the objects.
3) Simple hardware implementation.
4) Online algorithm (i.e., we dont need to load all polygons at once in order to run algorithm).

Disadvantages:

1) Doubles memory requirements (at least), one for z-buffer and one for refress-buffer.
2) Device dependent and memory intensive.
3) Wasted computation on drawing distant points that are drawn over with closer points that occupy the same pixel.
4) Spends time while rendering polygons that are not visible.
5) Requires re-calculations when changing the scale.

Example 1: How does the z-buffer algorithm determine which surfaces are hidden?

Solution: Z-buffer algorithm uses a two buffer area each of two-dimensional array, one z-buffer which stores the depth value at each pixel position (x,y), another frame-buffer which stores the intensity values of the visible surface. By setting 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, this value replaces the value stored in the z-buffer and the pixel color value is changed to the color of the new surface.

Example 2: What is the maximum number of objects that can be handled by the z-buffer algorithm?

Solution: In z-buffer algorithm, an arbitrary number of objects can be handled because each object is processed one at a time. The number of objects is limited only by the computer's memory to store the objects.

Example 3: What happens when two polygons have the same z value and the z-buffer algorithm is used?

Solution: z-buffer algorithms, changes colors at a pixel if $z(x,y)<z_{buf}(x,y)$, the first polygon surface will determine the color of the pixel.

Example 4: Assume that one allow 256 depth value level to be used. Approximately how many memory would a 512x512 pixel display require to store z-buffer?

Solution: A system that distinguishes 256 depth values would require one byte of memory (2^8=256) to represent z-value.

☞ Check Your Progress 1

1) z-buffer method use(s) …….. :
   a) Only object-space approach b) Only image-space approach c) both object-space & Image-space.

   ……

   ……

   ……

   ……

2) What happens when two polygons have the same z value and the z-buffer algorithm is used?

   ……

   ……

   ……

3) Assuming that one allows 2^{32} depth value levels to be used, how much memory would a 1024x768 pixel display require to stores the z-buffer?

   ……

   ……

   ……

2.2.2 Scan-Line method

In contrast to z-buffer method, where we consider one surface at a time, scan-line method deals with multiple surfaces. As it processes each scan-line at a time, all polygon intersected by that scan-line are examined to determine which surfaces are visible. The visibility test involves the comparison of depths of each overlapping surface to determine which one is closer to the view plane. If it is found so, then it is declared as a visible surface and the intensity values at the positions along the scan-line are entered into the refresh-buffer.

Assumptions:
1. Plane of projection is Z=0 plane.
2. Orthographic parallel projection.

3. Direction of projection, d = (0,0, –1)
4. Objects made up of polygon faces.

Scan-line algorithm solves the hidden- surface problem, one scan-line at a time, usually processing scan lines from the bottom to the top of the display.

The scan-line algorithm is a one-dimensional version of the depth –Buffer. We require two arrays, intensity [x] & depth [x] to hold values for a single scan-line.

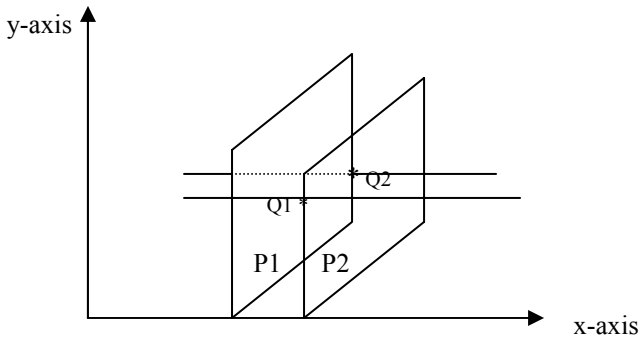


Figure 7

Here at $Q_1$and Q_2 both polygons are active (i.e., sharing).

Compare the z-values at Q_1 for both the planes (P_1 & P_2). Let $z_1^{(1)}$,$z_1^{(2)}$ be the z-value at Q_1 ,corresponding to P_1& P_2 polygon respectively.

Similarly
$z_2^{(1)}$, $z_2^{(2)}$ are the z-values at Q_2, corresponding to P_1 & P_2 polygon respectively.

Case1: $z_1^{(1)} < z_1^{(2)}$
$z_2^{(1)} < z_2^{(2)}$ ➔ Q1,Q2 is filled with the color of P2.

Case2: $z_1^{(2)} < z_1^{(1)}$
$z_2^{(2)} < z_2^{(1)}$ ➔ Q1,Q2 is filled with the color of P2.

Case3: Intersection is taking place.

In this case we have to go back pixel by pixel and determine which plane is closer. Then choose the color of the pixel.

Algorithm (scan-line):

For each scan line perform step (1) through step (3).

1) For all pixels on a scan-line, set
depth [x]=1.0 (max value) & Intensity [x] = background-color.

2) For each polygon in the scene, find all pixels on the current scan-line (say S1) that lies within the polygon. For each of these x-values:

   a) calculate the depth z of the polygon at (x,y)
   b) if z < depth [x], set depth [x]=z & intensity corresponding to the polygon's shading.

3) After all polygons have been considered, the values contained in the intensity array represent the solution and can be copied into a frame-buffer.

Advantages of Scan line Algorithm:

Here, every time, we are working with one-dimensional array, i.e., x[0…x_max] for color not a 2D-array as in depth buffer algorithm.

Example 5: Distinguish between z-buffer method and scan-line method. What are the visibility test made in these methods?

Solution: In z-buffer algorithm every pixel position on the projection plane is considered for determining the visibility of surfaces w. r. t. this pixel. On the other hand in scan-line method all surfaces intersected by a scan line are examined for visibility. The visibility test in z-buffer method involves the comparison of depths of surfaces w. r. t. a pixel on the projection plane. The surface closest to the pixel position is considered visible. The visibility test in scan-line method compares depth calculations for each overlapping surface to determine which surface is nearest to the view-plane so that it is declared as visible.

Example6: Given two triangles P with vertices P1(100,100,50), P2(50,50,50), P3(150,50,50) and q with vertices Q1(40,80,60), q2(70,70,50), Q3(10,75,70), determine which triangle should be painted first using the scan-line method.

Solution: In the scan-line method, two triangles P and Q are tested for overlap in xy-plane. Then they are tested for depth overlap. In this question, there is no overlap in the depth. But P and Q have overlap in xy-plane. So the Q is painted first followed by P.

☞ Check Your Progress 2

1) All the algorithm, which uses image-space approach, requires:
   a) One buffer-area b) two buffer-areas c) three-buffer areas

   ……

   ……

   ……

2) All the algorithm, which uses object-space approach, requires:
   a) One buffer-area b) two buffer-areas c) three- buffer areas

   ……

   ……

   ……

3) Scan line method deals with _____ surface(s) at a time for ascertaining visibility
   a) single b) two c) multiple d) 100

   ……

   ……

   ……

4) What are the relative merits of object-space methods and image-space methods?

   ……

   ……

   ……

2.2.3 Area-Subdivision method

This method is essentially an image-space method but uses object-space operations reordering (or sorting) of surfaces according to depth. This method takes advantage of area-coherence in a scene by locating those view areas that represent part of a single surface. In this method we successively subdivide the total viewing (screen) area, usually a rectangular window, into small rectangles until each small area is the projection of part of a single visible surface or no surface at all.

Assumptions:

- Plane of projection is z=0 plane
- Orthographic parallel projection
- Direction of projection d=(0,0,-1)
- Assume that the viewing (screen) area is a square
- Objects are made up of polygon faces.

To implement the area-subdivision method, we need to identify whether the area is part of a single surface or a complex surface by means of visibility tests. If the tests indicate that the view is sufficiently complex, we subdivide it. Next, we apply the tests to each of the smaller areas and then subdivide further if the tests indicate that the visibility of a single surface is still uncertain. 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.

Starting with the full screen as the initial area, the algorithm divides an area at each stage into 4 smaller area, as shown in *Figure 8*, which is similar to quad-tree approach.

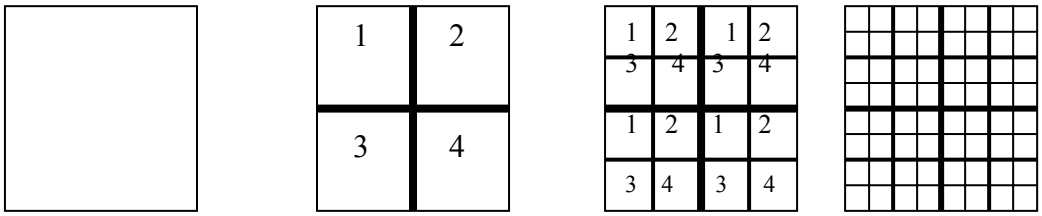


Figure 8 (a) Initial area **(b) 1st subdivision** **(c) 2nd Subdivision** **(d) 3rd Subdivision**

Test to determine the visibility of a single surface are made by comparing surfaces (i.e., polygons P) with respect to a given screen area A. There are 4 possibilities:

1) **Surrounding polygon:** Polygon that completely contains the area (*Figure 9(a)*).
2) **Intersecting (or overlapping) polygon:** Polygon that intersects the area (*Figure 9(b)*).
3) **Contained polygon:** polygon that is completely contained within the area (*Figure 9(c)*).
4) **Disjoint polygon:** Polygon that is completely outside the area (*Figure 9(d)*).

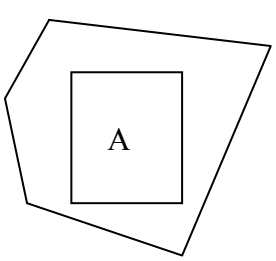


Figure 9(a)

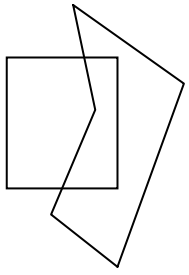


Figure 9(b)

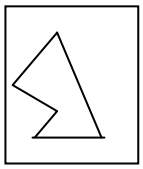


Figure 9(c)

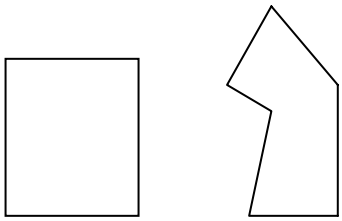


Figure 9(d)

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

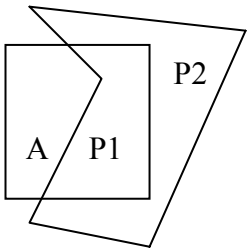


Figure 10

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

Case 1: All the polygons are disjoint from the area. In this case, the background color can be displayed in the area.

Case 2: Exactly one polygon faces, after projection, intersecting or contained in the square area. In this case the area is first filled with the background color, and then the part of the polygon contained in the area is scan converted.

Case 3: There is a single surrounding polygon, but no intersecting or contained polygons. In this case the area is filled with the color of the surrounding polygon.

Case 4: More than one polygon is intersecting, contained in, or surrounding the area, but one is a surrounding polygon that is in front of all the other polygons. Determining whether a surrounding polygon is in front is done by computing the z coordinates of the planes of all surrounding, intersecting and contained polygons at the four corners of the area; if there is a surrounding polygon whose four corner z coordinates are larger than one those of any of the other polygons, then the entire area can be filled with the color of this surrounding polygon.

To check whether the polygon is any one of these four cases, we have to perform the following test:

Test 1: For checking disjoint polygons (use **Min-max test**).
Suppose you have two polygons *P1* and *P2*. The given polygons *P1* and *P2* are disjoint if any of the following four conditions is satisfied (see *Figures-11(a) and 11(b)*): These four tests are called **Min-max test**.

i) $x^{(1)}_{max} < x^{(2)}_{min}$
ii) $x^{(2)}_{max} < x^{(1)}_{min}$
iii) $y^{(1)}_{max} < y^{(2)}_{min}$
iv) $y^{(2)}_{max} < y^{(1)}_{min}$

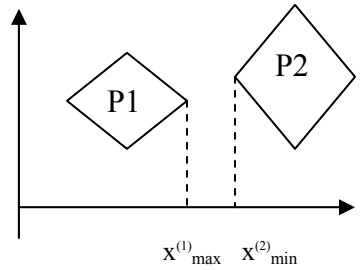


Figure 11(a)

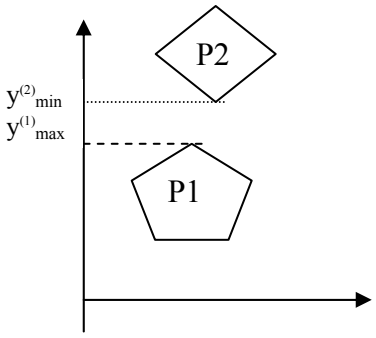


Figure 11(b)

Test 2: (Intersection Test)**:** If **Min-max test** fails then we go for intersection test. Here we take each edge one by one and see if it is intersecting. For example, see Figure 12, for each edge of *P1* we find the intersection of all edges of *P2*.

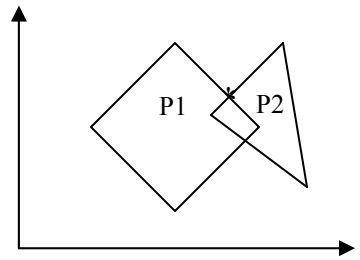


Figure 12

Test 3: (Containment test): If intersection test fails, then it can be either contained polygon or surrounding polygon. So we do the containment test. For this test we have the following three cases, shown in *Figures 13(a),(b) and (c)*.

a) P1 contains P2.
b) P2 contains P1.
c) P1 and P2 are disjoint.

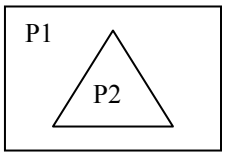
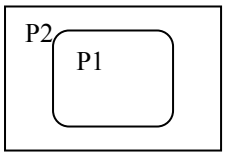
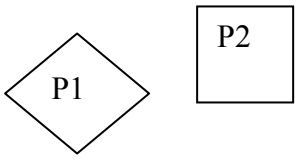


Figure 13(a): P1 contained P2



(b) P2 contained P1



(c) P1and P2 are disjoint

Case a: Verify a vertex point of P2 lies inside of P1. If the result is true, P2 is completely inside of P1.

Case b: If the result of case-a is not true, then verify whether P2 contains a vertex point of P1. If the result is true, then P2 contains P1.

Case c: If both **case-a** and **case-b** (containment test) failed then we conclude that P1 and P2 are disjoint.

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_{smax}.

If, for a polygon *P* on the list, $z_{pmin} > z_{smax}$ (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 w.r.t. an area, sorted on zmin (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 classified as intersecting (category 2) or contained (category 3), color (scan-converter) 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.

Example 7: Suppose there are three polygon surfaces P,Q, R with vertices given by:
P: P1(1,1,1), P2(4,5,2), P3(5,2,5)
Q: Q1(2,2,0.5), Q2(3,3,1.75), Q3(6,1,0.5)
R: R1(0.5,2,5.5), R2(2,5,3), R3(4,4,5)

Using the Area subdivision method, which of the three polygon surfaces P, Q, R obscures the remaining two surfaces? Assume z=0 is the projection plane.

Solution: Here, we have z=0 is the projection plane and P, Q, R are the 3-D planes. We apply first three visibility decision tests i.e. (a), (b) and (c), to check the bounding rectangles of all surfaces against the area boundaries in the xy-plane. Using test 4, we can determine whether the minimum depth of one of the surrounding surface S is closer to the view plane.

Example 8: What are the conditions to be satisfied, in Area-subdivision method, so that a surface not to be divided further?

Solution: In an area subdivision method, the given specified area IS not to be divided further, if the following four conditions are satisfied:

1) Surface must be outside the specified area.

2) There must be one overlapping surface or one inside surface.
3) One surrounding surface but not overlapping or no inside surface.
4) A surrounding surface/obscures all other surfaces with the specified area.

☞ Check Your Progress 3

1) Area- subdivision method uses:
a) Only image-space b) Only object-space c) both image and object space

……

……

……

2) What are the basic concepts of Area-subdivision method?

……

……

……

2.3 SUMMARY

- For displaying a realistic view of the given 3D-object, hidden surfaces and hidden lines must be identified for elimination.
- The process of identifying and removal of these hidden surfaces is called the *visible-line* or *visible-surface determination*, or *hidden-line* or *hidden-surface elimination.*
- To construct a realistic view of the given 3D object, it is necessary to determine which lines or surfaces of the objects are visible. For this, we need to conduct visibility tests.
- Visibility tests are conducted to determine the surface that is visible from a given viewpoint.
- There are two fundamental approaches for visible-surface determination, according to whether they deal with their projected images or with object definitions directly. These two approaches are called *image-space approach* and *object-space approach*, respectively.
- Object space methods are implemented in the physical coordinate system in which objects are defined whereas image space methods are implemented in screen coordinate system in which the objects are viewed.
- Image-space approach requires examining all the objects in the scene to determine which is closest to the viewer along the projector passing through the pixel. That is, the visibility is decided point by point at each pixel position on the projection plane. If the number of objects is ‘n’ and the pixels is ‘p’ then effort is proportional to **n.p.**

- Object-space approach compares all objects directly with each other within the scene definition and eliminates those objects or portion of objects that are not visible.

- Object-space approach compares each of the n objects to itself and to the other objects, discarding invisible portions. Thus the computational effort is proportional to $\boldsymbol{n^2}$**.**

- Under the category of *Image space approach*, we have two methods: 1) *Z-buffer* method and 2) *Scan-line* method.

- Among all the algorithms for visible surface determination, the Z-buffer is perhaps the simplest, and is the most widely used method.

- *Z-buffer* method detects the visible surfaces by comparing surface depths (z-values) at each pixel position on the projection plane.

- For implementing z-buffer algorithm, we require two buffer areas (two 2-D arrays): 1) **Depth-buffer[i,j]**, to store the depth-value of the visible surface for each pixel in the view plane, and 2) **Refresh-buffer[i,j],** to store the pixel intensities of the visible surfaces.

- In contrast to z-buffer method, *Scan-line method* deals with multiple surfaces. As it processes each scan-line at a time, all polygon intersected by that scan-line are examined to determine which surfaces are visible. The visibility test involves the comparison of depths of each overlapping surfaces to determine which one is closer to the view plane. If it is found so, then it is declared as a visible surface and the intensity values at the positions along the scan-line are entered into the refresh-buffer.

- *Area-subdivision method* is essentially an image-space method but uses object-space calculations for reordering of surfaces according to depth. The method makes use of area coherence in a scene by collecting those areas that form part of a single surface. In this method, we successively subdivide the total viewing area into small rectangles until each small area is the projection of part of a single visible surface or no surface at all.

2.4 SOLUTIONS/ANSWERS

Check Your Progress 1

1) b

2) z-buffer algorithms, changes colors at a pixel if $z(x,y)<z_{buf}(x,y)$, the first polygon surface (which is written) will determine the color of the pixel.

3) A system that distinguishes 2^{32} depth values would require four bytes of memory to represent each z value. Thus total memory needed= 4x1024x768=3032K

Check Your Progress 2

1) b

2) a

3) c

4) Image space approaches we determine which of the objects in the scene is visible, at each pixel, by comparing the z-value of each object. Object-space approach determines the visibility of each object in the scene. For this all objects are compared within scene definition.

   Image-space methods are implemented in screen coordinate system whereas Object-space methods are implemented in the physical coordinate system.

   Image-space approaches were developed for raster devices whereas object-space approaches were developed for vector graphics systems. In case of image-space approaches, the results are crude and limited by the resolution of the screen whereas in object-space approaches, we have very precise results (generally to the precision of a machine).

Check Your Progress 3

1) a

2) The area-subdivision algorithm works as follows:

   Step-1: A polygon is seen from within a given area of the display screen if the projection of that polygon overlaps the given area.

   Step-2 : Of all polygons that overlap a given screen area, the one that is visible in this area is the one in front of all the others.

   Step-3 : If we cannot decide which polygon is visible (in front of the others) from a given region, we subdivide the region into smaller regions until visibility decisions can be made (even if we subdivide the region up to the pixel level).