## 5.1 Hidden Surfaces and Hidden Surface Removal Approaches

In case of 3D viewing, additional effort should be made to specify the viewing direction. So major concern for realistic graphics is identifying those parts of a scene that are visible from a chosen viewing position

Several algorithms have been devised some require more memory , some involve more processing time and some apply only to special types of objects.

Various algorithms are referred to as "*Visible–Surface Detection*" also referred to as "*Hidden Surface Elimination Method*" algorithms broadly classified approaches are "*Object Space Methods ( OSM )*" and "*Image Space Method ( ISM )*"

*OSM* compares objects and parts of object to each other within the scene definition to determine which surfaces as a whole we should label as visible e.g. *Back Face Detection method*

*ISM* decides point by point at each pixel position and is most widely used.

### 5.1.1 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

$$Ax + By + Cz + D < 0$$

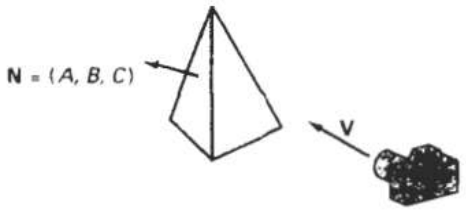
When an inside point is along the line of sight to the surface, the polygon must be a back face (we are inside that face and cannot see the front of it from our viewing position).

We can simplify this test by considering the normal vector N to a polygon surface, which has Cartesian components (A, B, C).

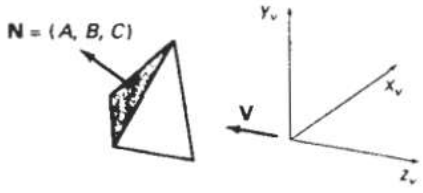
In general, if V is a vector in the viewing direction from the eye (or "camera") position, then this polygon is a back face if

$$V.N > 0$$ as *both vectors are point in the same direction, angle between them is 0*

If object descriptions have been converted to projection coordinates and our viewing direction is parallel to the viewing $z_v$ axis, then $V = (0, 0, V_z)$ and so that we only need to consider the sign of C, the z component of the normal vector N



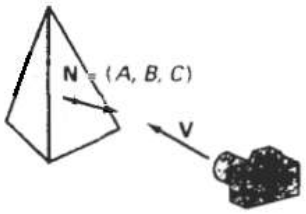

Vector **V** in the viewing direction and a back-face normal vector **N** of a polyhedron




A polygon surface with plane parameter $C < 0$ in a right-handed viewing coordinate system is identified as a back face when the viewing direction is along the negative $z_v$ axis.

If the object gets rotated towards the camera them both vectors start facing towards each other then

$$V.N < 0$$ as *vectors are point in opposite direction, angle between them is 180*



Vector **V** in the viewing direction and normal vector **N** of a polyhedron

In a right-handed viewing system with viewing direction along the negative $z_v$ axis, the polygon is a back face if C < 0. Also, 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

$$C <= 0$$

**Steps:**

1. Polygon vertices are labeled in clockwise direction
2. Compute the normal vector N to a polygon surface
3. Compute the viewing vector from the eye (or "camera") position V to the Polygon surface
4. Compute the dot product of the surface normal and the viewing vector V.N
5. Perform the Tests:

```
If    V.N  >  0
       The polygon is a back face
Else
       The polygon is visible from the viewing direction
```

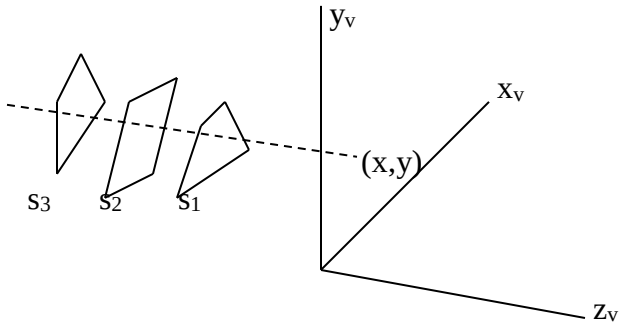
**Limitations:**

**1)** This method works fine for convex polyhedra, but not necessarily for concave polyhedra.
**2)** This method can only be used on solid objects modeled as a polygon mesh.

### 5.1.2 Z-Buffer/Depth Buffer Method

Commonly used *ISM*

Since object depth is usually measured from the view plane along the z axis of a viewing surface. Each surface of a scene is processed separately , one point at a time across the surface




Usually for polygon surface, because depth values can be computed very quickly and the method is easy to implement and can apply to non planar surfaces.

Implementing the algorithm in normalized coordinates , z value ranges from 0 at the back clipping plane to $z_{max}$ at the front clipping plane.
Algorithm

i. Initialize the *depth buffer* and *refresh buffer* so that for all buffer position(x,y)
depth(x,y) = 0 or ($z_{min}$) and refresh(x,y) = $I_{background}$
ii. 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
iii. If z > depth(x,y) then set
depth = z and refresh(x,y) = $I_{surface}$ (x,y)
where $I_{background}$ (x,y) is background intensity , $I_{surface}$ (x,y) is projected intensity value for 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 value for a surface position (x, y) is $z = (-Ax - By - D)/c$ ……(i)
Let depth z' at (x + 1 , y) $z' = -A(x+1) - By - D/c$ or $z' = z - A/c$……..(ii)

Now since (-A/c) is constant for each surface, so succeeding depth values across a scan-line are obtained from preceding values with a simple mathematics.

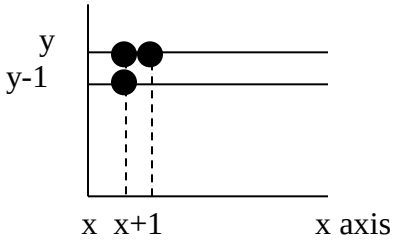


fig. ii.

On each scan-line , we start by calculating depth on a left edge of the polygon that intersect that scan-line (fig ii) 'z' at each successive position across the scan line are the calculated by equation(ii)
first determine y coordinate extents of each polygon and process from top scan to bottom scan starting at a top vertices we can recursively calculate 'x' position down a left edge of the polygon as
x' = x - 1/ m where m is the slope of edge fig(iii)

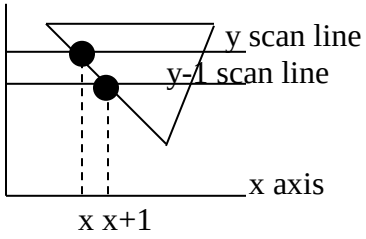


fig iii.

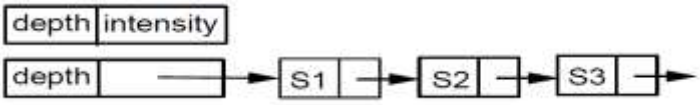
'z' values down the edge are then obtained recursively as
$z' = z + (a/m + b)/c$ or $z' = z + b/c$ as m → for vertical edge

**Drawback:**
A drawback of the depth-buffer method is that it can only find one visible surface at each pixel position. In other words, it deals only with opaque surfaces and cannot accumulate intensity values for more than one surface, as is necessary if transparent surfaces are to be displayed**.**

### 5.1.3 A- buffer method

An extension of the ideas in the depth-buffer method is the A-buffer method (at the other end of the alphabet from "z-buffer", where ***z*** represents depth). The **A** buffer method represents an antialiased, ***area-averaged, accumulation-buffer*** method developed by Lucasfilm for implementation in the surface-rendering system called REYES (an acronym for "Renders Everything You Ever Saw").
The A-buffer method expands the depth buffer so that each position in the buffer can reference a linked list of surfaces. Thus, more than one surface intensity can be taken into consideration at each pixel position, and object edges can be antialiased.
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 the depth field is positive, the number stored at that position is the depth of a single surface overlapping the corresponding pixel area. The intensity field then stores the RCB components of the surface color at that point and the percent of pixel coverage.
If the depth field is negative, this indicates multiple-surface contributions to the pixel intensity. The intensity field then stores a pointer to a linked list of surface data




Data for each surface in the linked list includes:
- RGB intensity components
- opacity parameter (percent of transparency)
- depth

- percent of area coverage
- surface identifier
- other surface-rendering parameters
- pointer to next surface

The A-buffer can be constructed using methods similar to those in the depth-buffer algorithm. Scan lines are processed to determine surface overlaps of pixels across the individual scanlines. Surfaces are subdivided into a polygon mesh and clipped against the pixel boundaries. Using the opacity factors and percent of surface overlaps, we can calculate the intensity of each pixel as an average of the contributions from the overlapping surfaces.

### 5.1.4 Scan Line Method

*ISM* for removing hidden surfaces is an extension of the scan-line algorithm for filling polygon interiors.

Instead of filling just one surface we consider multiple surfaces.

As each scan-line is processed all polygon surfaces intersecting that line are examined to determine which are visible

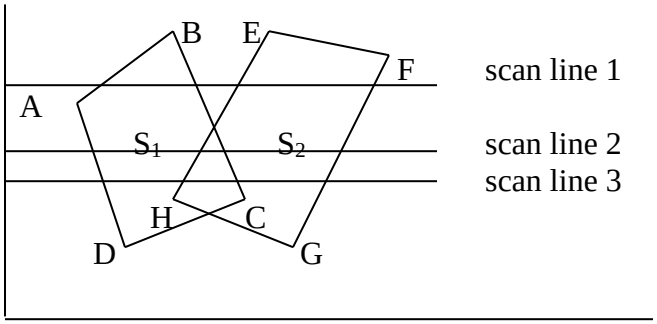
Set up Polygon tables for various surfaces (*edge, polygon etc*)

Polygon table contains *coefficients of the plane equation* for each surface intensity information for the surfaces and *possible pointers into the edge table*

An **Active edge list** consists of the edges of the polygon that a particular scan line intersects

*Active edge list* for scan-line1 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 flags for surfaces $S_1$ is on.




No depth calculation is necessary and intensity information for surface $S_1$ is entered from the polygon table into the refresh buffer

Between edges EH and FG only the flag for surface $S_2$ is on.

No other position along scan-line1 intersect surfaces so intensity values in the other areas are set to background intensity

For scan-line 2 and 3 the active edge list contains edges AD, EH, BC and FG

Along scan-line2 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 and depth calculations must be made using plane coefficients for the two surfaces,

e.g if 'z' of surface $S_1$ is less than that of $S_2$ 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 $S_2$ are stored until edge FG is passed
Any number of overlapping polygon surfaces can be processed with this scan-line method.

Steps:

1. Set up Polygon tables for polygon surfaces

2. Set up Active edge list for scan-lines

3. For each scan line, raise flag for a surface if the scan line passes through the pair of edges of that surface

4. Based on that perform the following tests:
   If flag for only one surface is on
   No depth calculation is needed
   Intensity information of surface is loaded in frame buffer

   Else if flags are on for more than one surface
   Depth calculations are needed for the two overlapping surfaces
   Intensity information of visible surface pixels are loaded in frame buffer

## List Priority Algorithms

Determines a visibility ordering for objects ensures that a correct picture results if objects are rendered in that order.
e.g. if no object overlaps in 'z' then we need only to sort objects by increasing 'z' and render them in that order.
Farther objects are obscured by closer ones as pixels from the closer polygons overwrite those of more distant ones.
If objects overlap in 'z', we may still be able to determine a correct order.
If objects cyclically overlap, or penetrate each other, then there is no correct order.
Hybrids that combine both object and image precision operations.

- Depth comparison and object splitting are done with object precision.
- Scan conversion (which relies on ability of graphics device to overwrite pixels of previously drawn objects) is done with image precision.

### 5.1.5 Depth Sorting Method

Makes use of both image and object space operations to:

i. Sort surfaces in decreasing depth order(using both image and object space)
ii. Scan converted in order starting with surface with greatest depth(using image space)

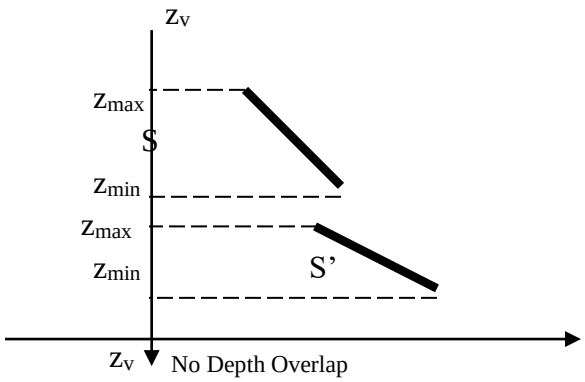
Known as painters algorithm because an artist first paints background color then most distant object then nearer object and so on. In the end the foreground objects are painted on canvas over the background. Each layer of paint covers up previous layer

Similarly, we first sort surfaces according to their distance from view plane.

Intensity values for farthest surface are entered into refresh buffer.

Taking each surface in turn (in decreasing depth order) we paint surface intensities onto frame buffer over intensity of previously processed surface

We assume we're viewing along 'z' direction. Surfaces are ordered according to largest 'z' value on each surface

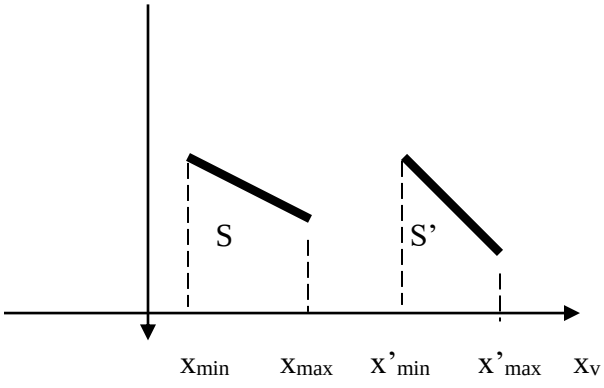



Surface with greatest depth is compared with other surface in list to see if there are any overlaps in depth

If no depth overlap then surface S is scan converted

Additional test are required for each surface in case of depth overlap with surface S:

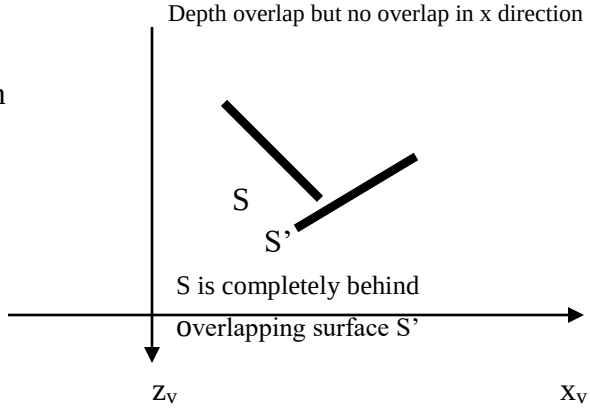
i. Bounding rectangle in xy plane for two surfaces don't overlap
ii. Surface S is completely behind overlapping surface relative to viewing position
iii. Overlapping surface S' is completely in-front of S relative to viewing position
iv. Projection of 2 surfaces onto view plane don't overlap.




Depth overlap but no overlap in x direction

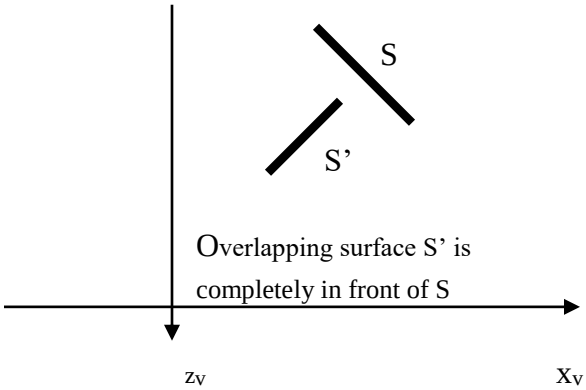
If any of these conditions is true then no reordering of surfaces is necessary. i.e. if all surfaces pass at least one of the tests then none of those surfaces are behind S.

For test ( i) we perform two steps to check overlap first in 'x' then in 'y' direction. If either of these directions show no overlap then the two planes cannot obscure one another.




For tests (ii) and (iii), we can perform inside outside polygon test. Substitute coordinates for all vertices into plane equation for overlapping surfaces and check the sign of the result obtained.

Based on plane equation if all vertices of S are inside S' then S is completely behind S'. Similarly, S' is completely in-front of S if all vertices of S are outside of S'.




For test (iv) we can use line equation in xy plane for checking intersections between bounding edges of two surfaces.