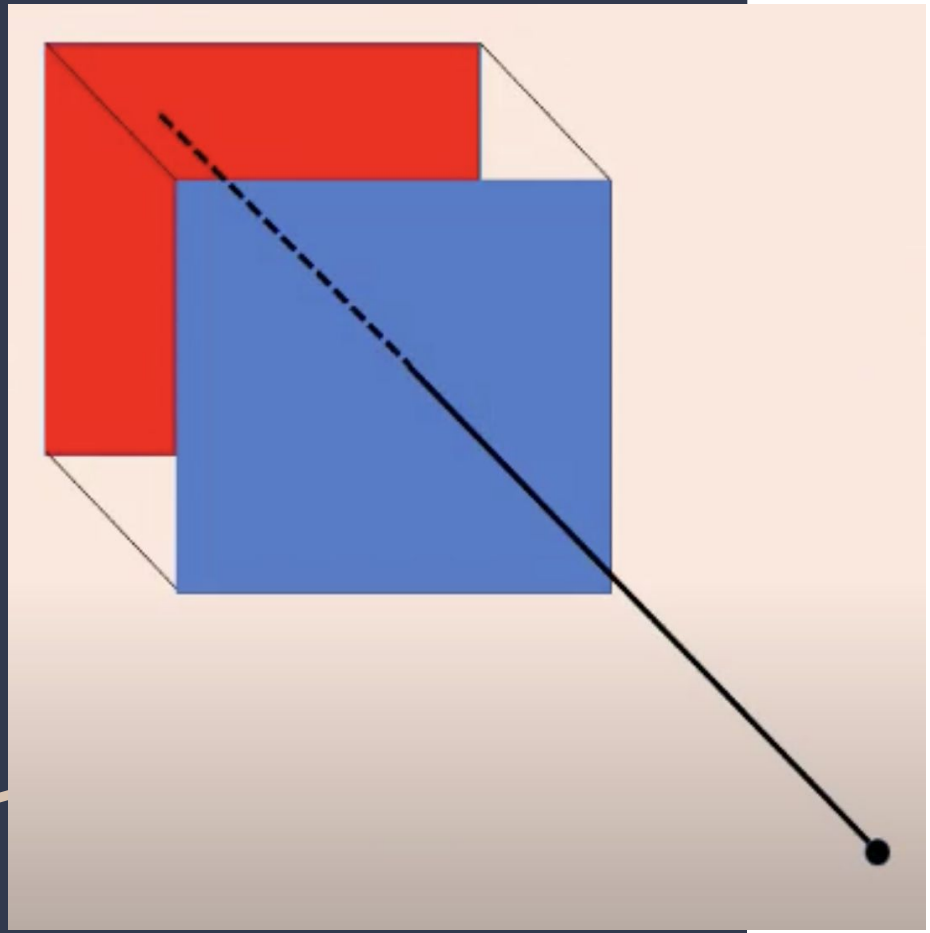
Chapter -7
Visible Surface Determination and computer graphics algorithm

Prepared by Sumesh Gajmer

Visible Surface Determination

- It is the process of identifying those parts of a scene that are visible from a chosen viewing position.There are numerous algorithms for efficient identification of visible objects for different types of applications
- These various algorithms are referred to as **visible-surface detection methods**. Sometimes these methods are also referred to as **hidden-surface elimination methods**.

To identify those parts of a scene that are visible from a chosen viewing position (visible-surface detection methods).
Surfaces which are obscured by other opaque surfaces along the line of sight are invisible to the viewer so can be eliminated (hidden-surface elimination methods).



Back Face

Front Face

Two Approaches

1. Object-Space methods:

- Efficient For Small number of objects but difficult to implement
- Deal with object definition
- Compares objects and parts of objects to each other within the scene definition to determine which surface as a whole we should label as visible.
- For N objects, may require N*N comparison operations.
- **E.g. Back-face detection method.**
- The resolution of the display device is irrelevant here as the calculation is done at the mathematical level of the object.

2. Image-Space Method:

- Visibility is decided point by point at each pixel position on the projection plane.
- Deal with projected image
- E.g. **Depth-buffer method, Scan-line method, Area-subdivision method**
- The resolution of the display device is important here as the calculation is done at the mathematical level of the object.

Note : Most visible surface detection algorithm use image-space-method but in some cases object space methods are also used for it.

Difference between Object-space and Image-space

Home Work

Back-Face Detection Method

- Back-face detection is a technique used in computer graphics to determine whether a polygon of a 3D object is visible from a particular viewpoint.
- It helps optimize rendering by eliminating polygons that are facing away from the camera, which are not visible to the viewer.

- A fast and simple **object-space method** for identifying the back faces of a polyhedron.
- It is based on the performing inside-outside test.

TWO METHODS:

First Method:

- A point *(x,* y, z) is "inside" a polygon surface with plane parameters A, B, C, and D if $Ax+By+Cz+D < 0$ (from plane equation).
- When an inside point is along the line of sight to the surface, the polygon must be a back face.
- In eq. $Ax+By+Cz+D=0$

if A,B,C remain constant , then varying value of D result in a whole family of parallel plane

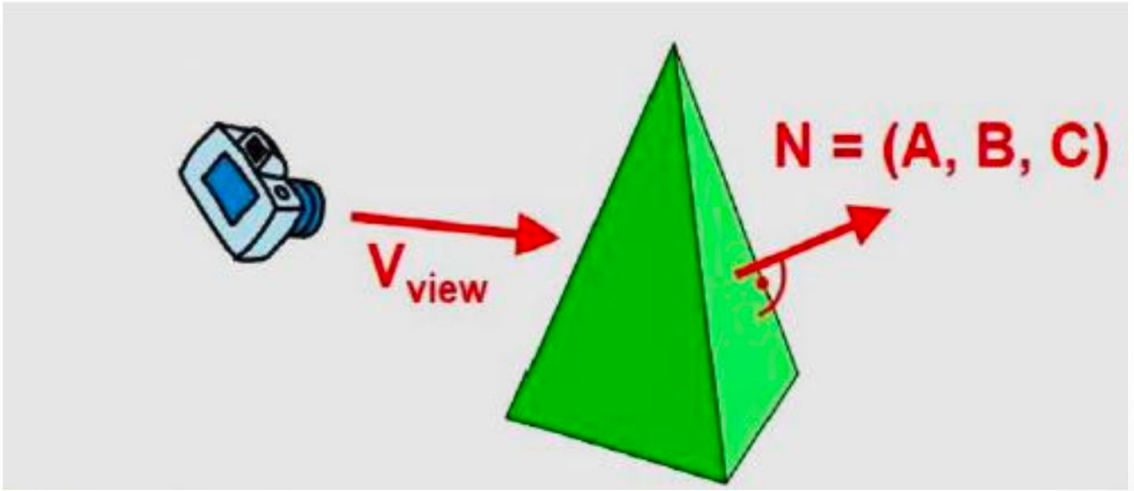
if $D>0$, plane is behind the origin (Away from observer)

if $D<0$, plane is in front of origin (toward the observer)

Second Way

- Let N be normal vector 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.**



Depth Buffer Method (Z-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.
- Also called ***z-buffer*** method since depth usually measured along z-axis. This approach compares surface depths at each pixel position on the projection plane.
- Each surface of a scene is processed separately, one point at a time across the surface. And each (x, y, z) position on a polygon surface corresponds to the projection point (x, y) on the view plane.

This method requires two buffers:

- A **z-buffer** or **depth buffer**: Stores depth values for each pixel position (x, y).
- **Frame buffer (Refresh buffer):** Stores the surface-intensity values or color values for each pixel position.
- As surfaces are processed, the image buffer is used to store the color values of each pixel position and the z-buffer is used to store the depth values for each (x, y) position.

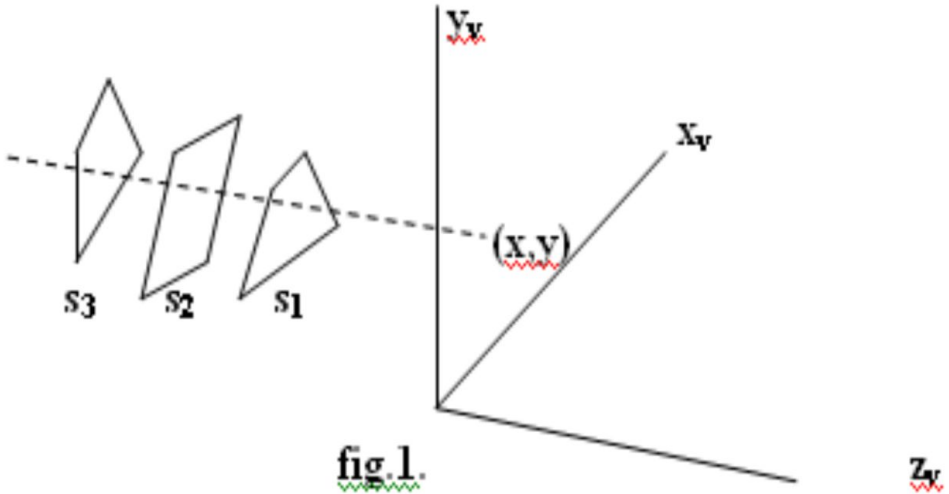
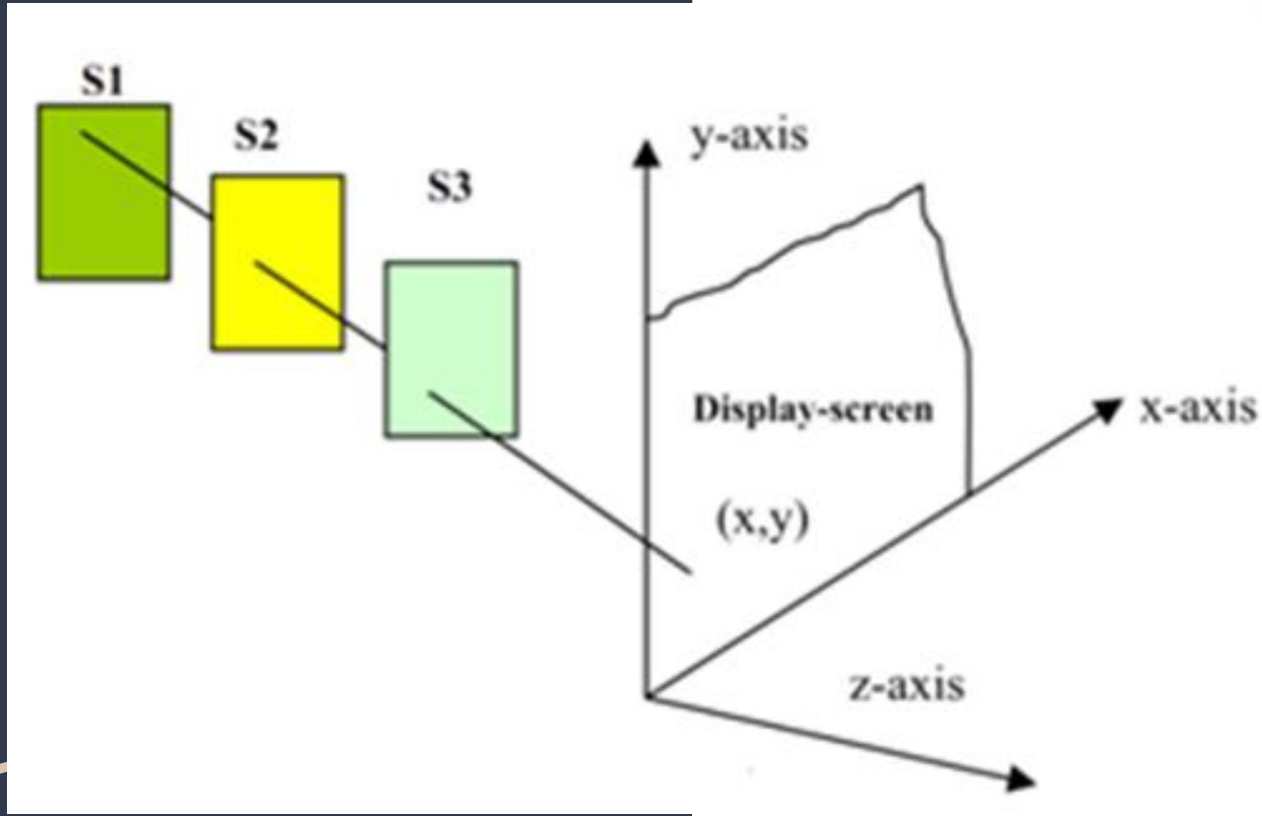


fig.1.


S1
S2
S3
y-axis
Display-screen
x-axis
(x,y)
z-axis

Algorithm:

1. Initialize both, depth buffer and refresh buffer for all buffer positions (x, y),

   depth(x, y) = 0

   refresh(x, y) = $I_{background}$,

   (where $I_{background}$ is the value for the background intensity.)

2. Process each polygon surface in a scene one at a time,

   2.1. Calculate the depth z for each (x, y) position on the polygon.

   2.2. If Z > depth(x, y), then set

   depth(x, y)=z

   refresh(x, y)= I_{surf}(x, y),

   (where I_{surf}(x, y) is the intensity value for the surface at pixel position (x, y).)

3. After all pixels and surfaces are compared, draw object using X,Y,Z from depth and intensity refresh buffer.

- 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

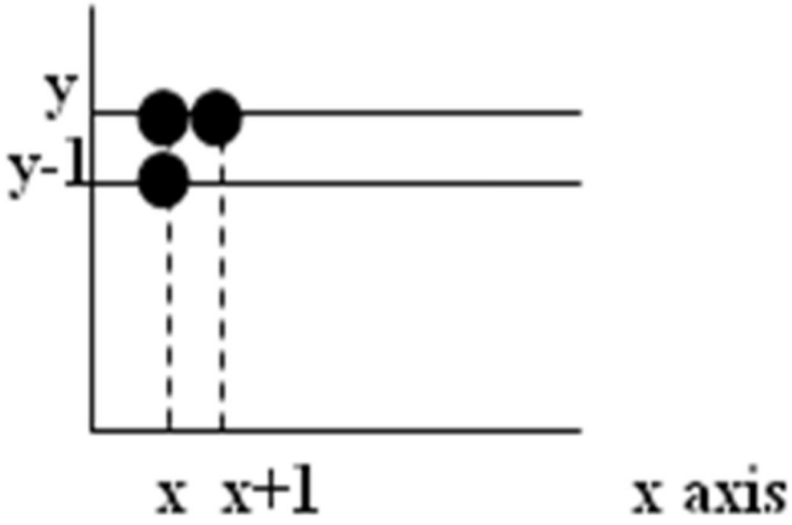
$z = (-Ax - By - D)/c$ .....(i)

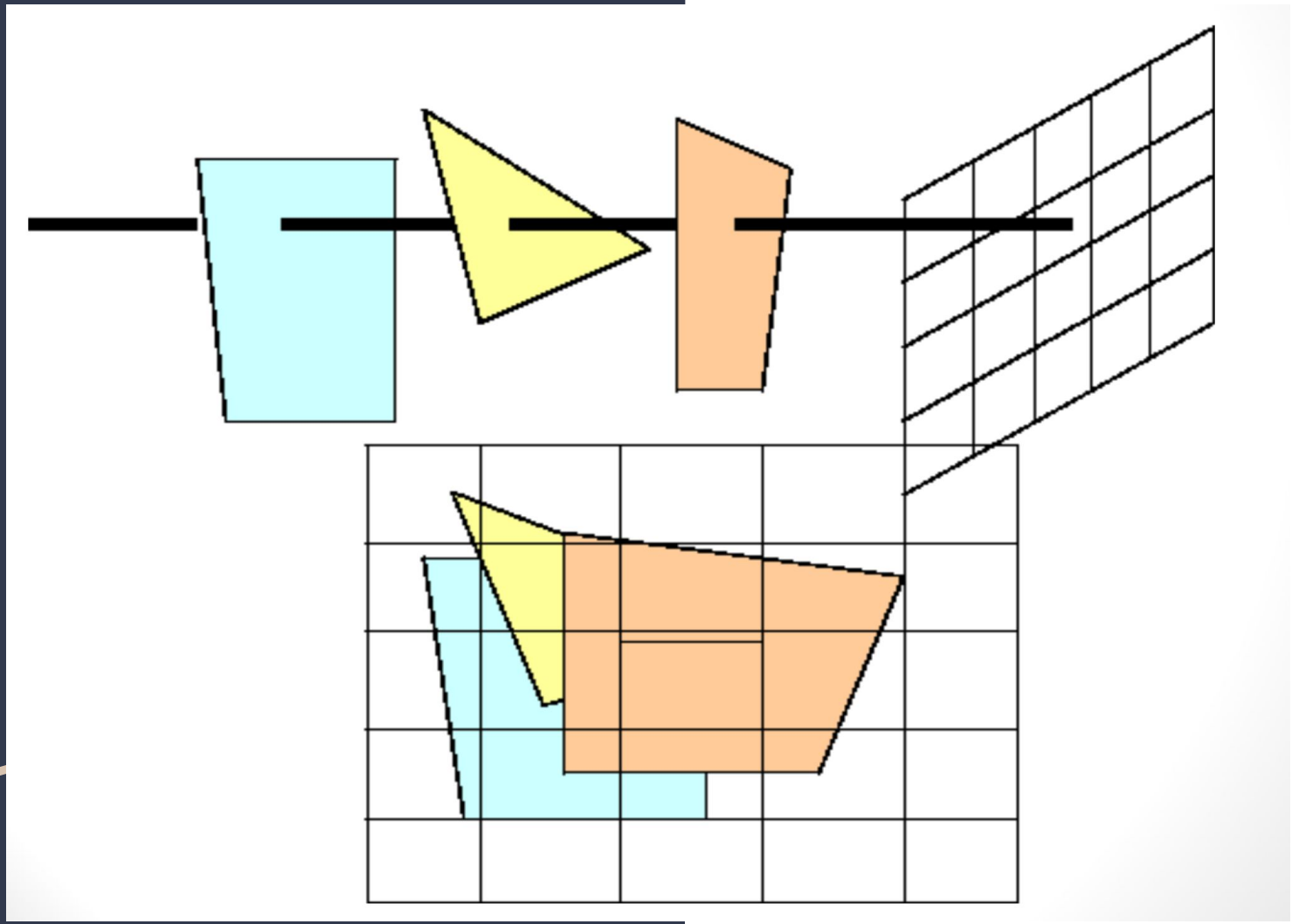
Let depth z' at (x + 1 , y)

$z' = -A(x+1) - By - D/c$

or

$z' = z - A/c$.....(ii)

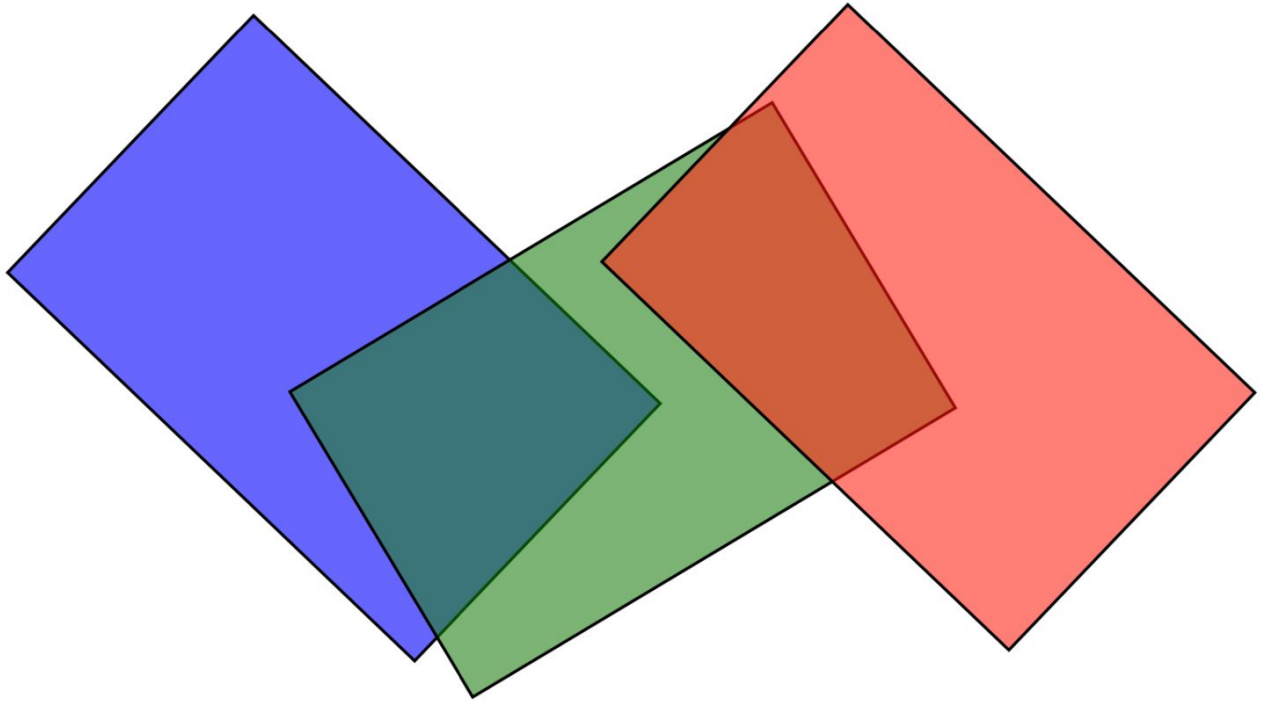




A – Buffer Method

- The **A-buffer** (**anti-aliased**, **area-averaged**, **accumulation** buffer) is an extension of the ideas in the **depth-buffer** method (other end of the alphabet from "**z-buffer**").
- A drawback of the **depth-buffer** method is that it deals only with opaque(Solid) surfaces and cannot accumulate intensity values for more than one transparent surfaces.
- The **A-buffer** method is an extension of the depth-buffer method.
- The **A-buffer** is incorporated into the REYES ("Renders Everything You Ever Saw") 3-D rendering system.
- The **A-buffer** method calculates the surface intensity for multiple surfaces at each pixel position, and object edges can be ant aliased.

- The A-buffer expands on the depth buffer method to allow transparencies. The key data structure in the A-buffer is the *accumulation buffer*



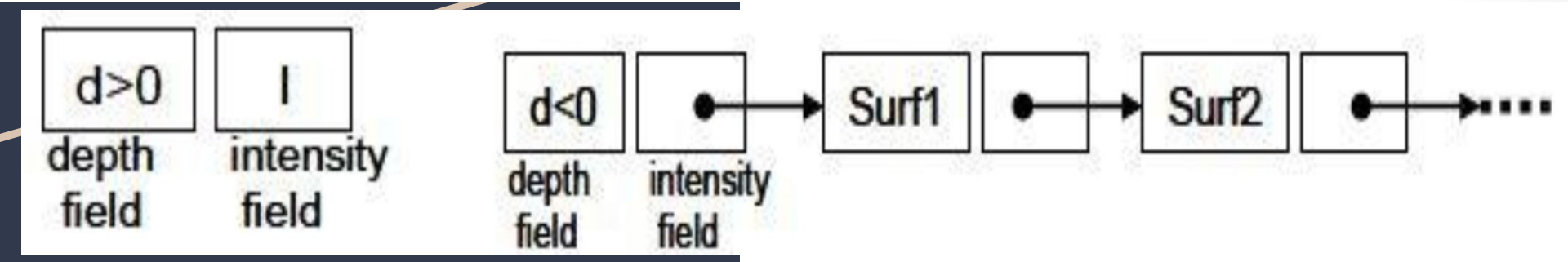
Each pixel position in the A-Buffer has two fields

Depth Field : stores a positive or negative real number

- Positive : **single** surface contributes to pixel intensity
- Negative : **multiple** surfaces contribute to pixel intensity

Intensity Field : stores surface-intensity information or a pointer value

- Surface intensity if single surface stores the RGB components of the surface color at that point
- and percent of pixel coverage Pointer value if multiple surfaces
- RGB intensity components
- Opacity parameter(per cent of transparency)
- Per cent of area coverage
- Surface identifier
- Other surface rendering parameters
- Pointer to next surface (Link List)



If depth is >= 0, then the surface data field stores the depth of that pixel position as before (**SINGLE SURFACE**)

(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, as illustrated first figure.)

If depth < 0 then the data filed stores a pointer to a linked list of surface data (**MULTIPLE SURFACE**)

(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, as in second figure. 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, and pointer to next surface)

Scan line Method

- Extension of scan line algorithm for filling polygon interiors
- Instead of filling just one surface, we deal with multiple surfaces
- As each scan line is processed, all polygon surfaces intersecting that line are examined to determine which are visible.
- Across each scan line , depth calculations are made for each overlapping surface to determine, which is nearest to view plane.
- When the visible surface has been determined , the intensity value for that position is entered into the refresh buffer.

Data Structure for scan line method

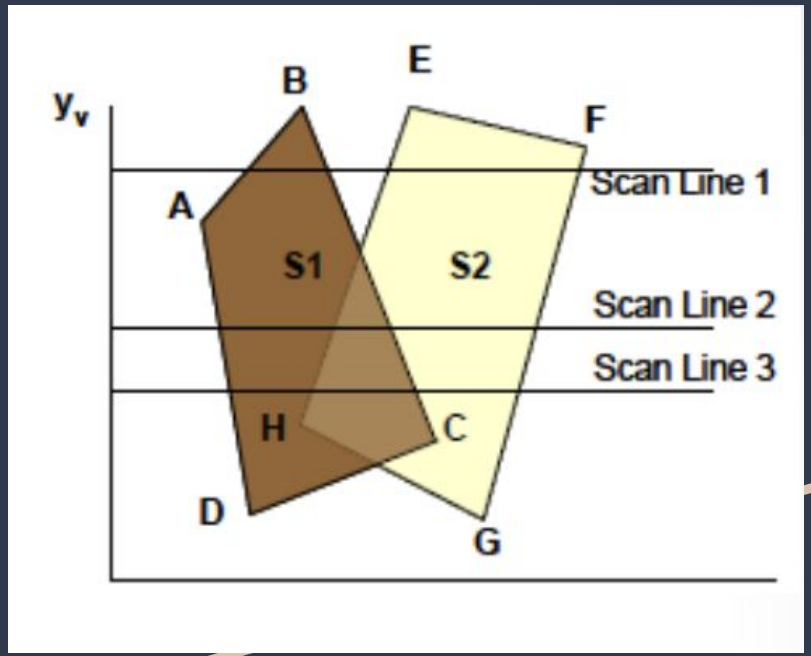
- **Edge table containing**
  - Coordinate endpoints for each line in a scene
  - Inverse slope of each line
  - Pointers into polygon table to identify the surfaces bounded by each line
- **Surface table containing**
  - Coefficients of the plane equation for each surface
  - Intensity information for each surface
  - Pointers to edge table
- **Active Edge List**
  - To keep a trace of which edges are intersected by the given scan line

Note : The edges are sorted in order of increasing x. Define flags for each surface to indicate whether a position is inside or outside the surface.

Algorithm for Scan line method

1. **Initialize the necessary data structure**
   a. Edge table containing end point coordinates, inverse slope and polygon pointer.
   b. Surface table containing plane coefficients and surface intensity
      Active Edge List
   c. Flag for each surface

2. **For each scan line repeat**
   a. update active edge list
   b. determine point of intersection and set surface on or off.
   c. If flag is on, store its value in the refresh buffer
   d. If more than one surface is on, do depth sorting and store the intensity of surface nearest to view plane in the refresh buffer



For scan line 1

- The active edge list contains edges AB,BC,EH, FG
- Between edges AB and BC, only flags for s1 == on and between edges EH and FG, only flags for s2==on
- no depth calculation needed and corresponding surface intensities are entered in refresh buffer

For scan line 2

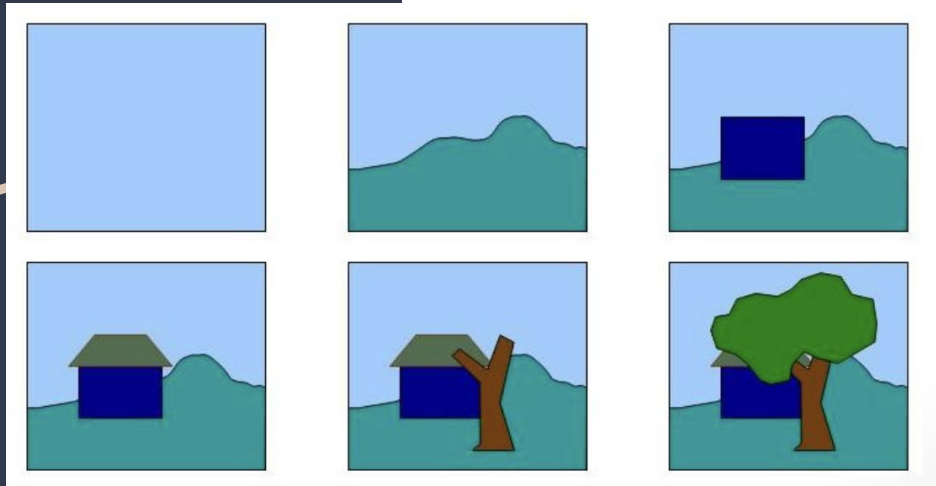
- The active edge list contains edges AD,EH,BC and FG
- Between edges AD and EH, only the flag for surface s1 == on
- Between edges EH and BC flags for both surfaces == on
- Depth calculation (using plane coefficients) is needed.
- In this example, say s2 is nearer to the view plane than s1, so intensities for surface s2 are loaded into the refresh buffer until boundary BC is encountered
- Between edges BC and FG flag for s1==off and flag for s2 == on
- Intensities for s2 are loaded on refresh buffer

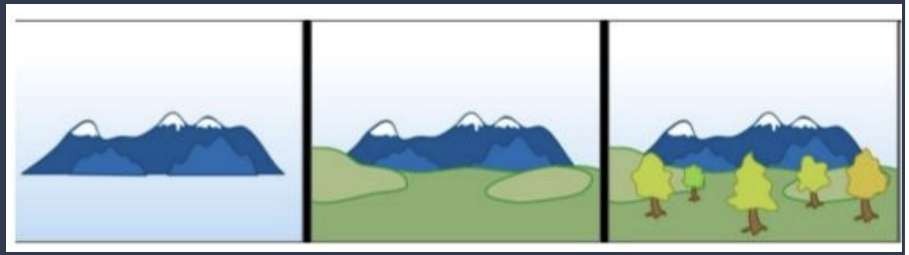
For scan line 3

- Same **coherent** property as scan line 2 as noticed from active list, so no depth

Painter Algorithm (Depth Sorting Algorithm)

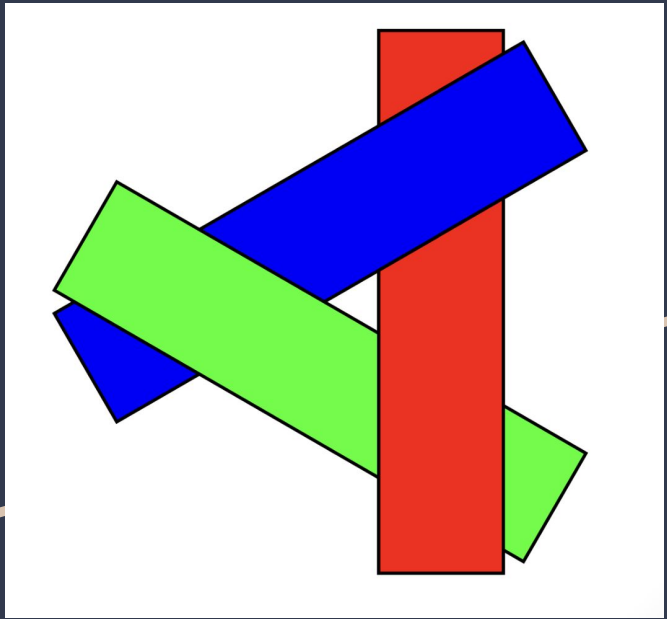
- This method uses both object space and image space method.
- In this method the surface representation of 3D object are sorted in of decreasing depth from viewer .
- Then sorted surface are scan converted in order starting with surface of greatest depth for the viewer .





- This algorithm is also called "Painter's Algorithm" as it simulates how a painter typically produces his painting by starting with the background and then progressively adding new (nearer) objects to the canvas.
- Thus, each layer of paint covers up the previous layer .
- Similarly, we 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.

Problem and its solution



One of the major problem in this algorithm is intersecting polygon surfaces. As shown in fig. below.

Problem

- Different polygons may have same depth.
- The nearest polygon could also be farthest.
- We cannot use simple depth-sorting to remove the hidden-surfaces in the images.

Solution

- For intersecting polygons, we can split one polygon into two or more polygons which can then be painted from back to front. This needs more time to compute intersection between polygons. So it becomes complex algorithm for such surface existence

- **General Case**: Given two polygons P and Q, an order may be determined between them, if at least one of the following holds:

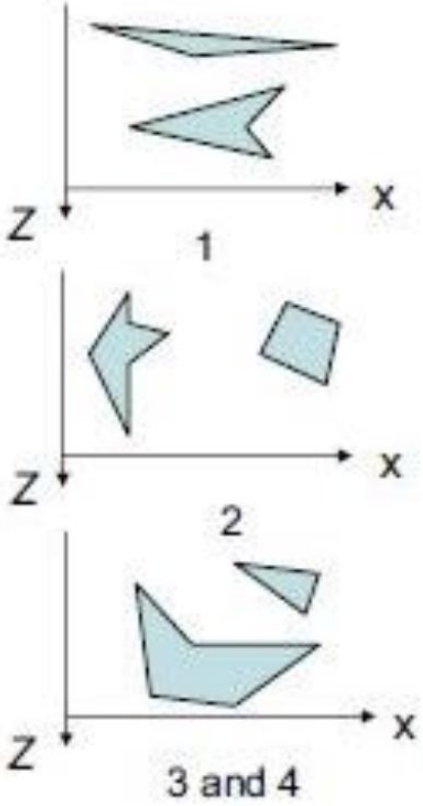
  1) z values of P and Q do not overlap

  2) The bounding rectangle in the x, y plane for P and Q do not overlap

  3) P is totally on one side of Q's plane

  4) Q is totally on one side of P's plane

  5) The bounding rectangles of Q and P do not intersect in the projection plane



BSP tree (read in chapter 6)

Octree Method : Read by yourself

Thank You