Visible Surface Detection

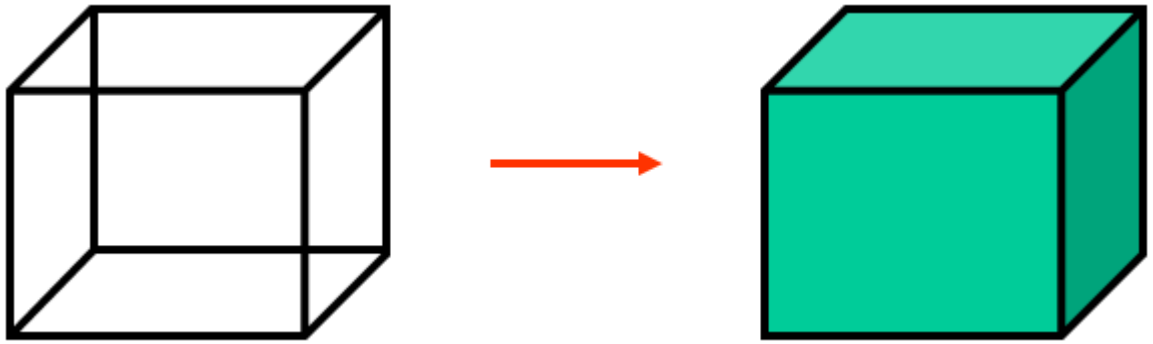
Dr. Mousumi Dutt

Visible-Surface Detection

Problem:

Given a scene and a projection,

what can we see?



Visible-Surface Detection

Terminology:

Visible-surface detection vs. *hidden-surface removal*

Hidden-line removal vs. *hidden-surface removal*

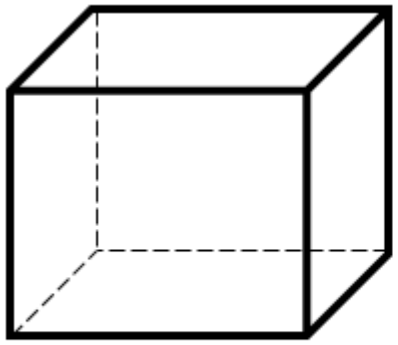
Many algorithms:

- Complexity scene
- Type of objects
- Hardware

Visible-Surface Detection

Two main types of algorithms:

- *Object space: Determine which part of the object are visible*
- *Image space: Determine per pixel which point of an object is visible*



Object space

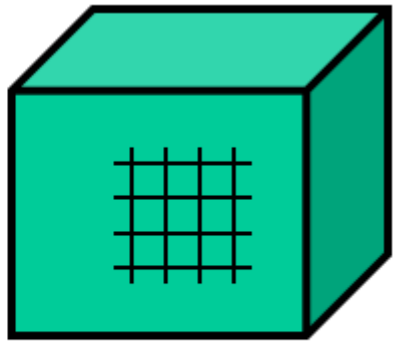


Image space

Introduction

- 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

Introduction

- 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

Introduction

- 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

CLASSIFICATION OF VISIBLE-SURFACE DETECTION ALGORITHMS

- broadly classified according to whether they deal with object definitions directly or with their projected images
- two approaches are called object-space methods and image-space methods
- 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.
- Line display algorithms, 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.

CLASSIFICATION OF VISIBLE-SURFACE DETECTION ALGORITHMS

- 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

CLASSIFICATION OF VISIBLE-SURFACE DETECTION ALGORITHMS

- Image-Space Method
  - Depth-Buffer Method
  - A-Buffer Method
  - Scan-Line Method
  - *Area-Subdivision Method*

- Object-Space Method
  - Back-Face Detection
  - BSP-Tree Method
  - *Area-Subdivision Method*
  - Octree Methods
  - Ray-Casting Method

Visible-Surface Detection

Visible-surface detection = *sort* for depth

- what and in what order varies

Performance: use *coherence*

- Objects
- Position in world space
- Position in image space
- Time

BACK FACE DETECTION ALGORITHM

A fast and simple object-space method for identifying the **back faces** of a polyhedron is based on the "inside-outside" tests discussed 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 $\mathbf{N}$ to a polygon surface, which has Cartesian components (A, B, C). In general, if $\mathbf{V}$ is a vector in the viewing direction from the eye (or "camera") position, as shown in Fig. 13-1, then this polygon is a back face if

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

BACK FACE DETECTION ALGORITHM

Furthermore, if object descriptions have been converted to projection coordinates and our viewing direction is parallel to the viewing z_v axis, then $\mathbf{V} = (0, 0, V_z)$ and

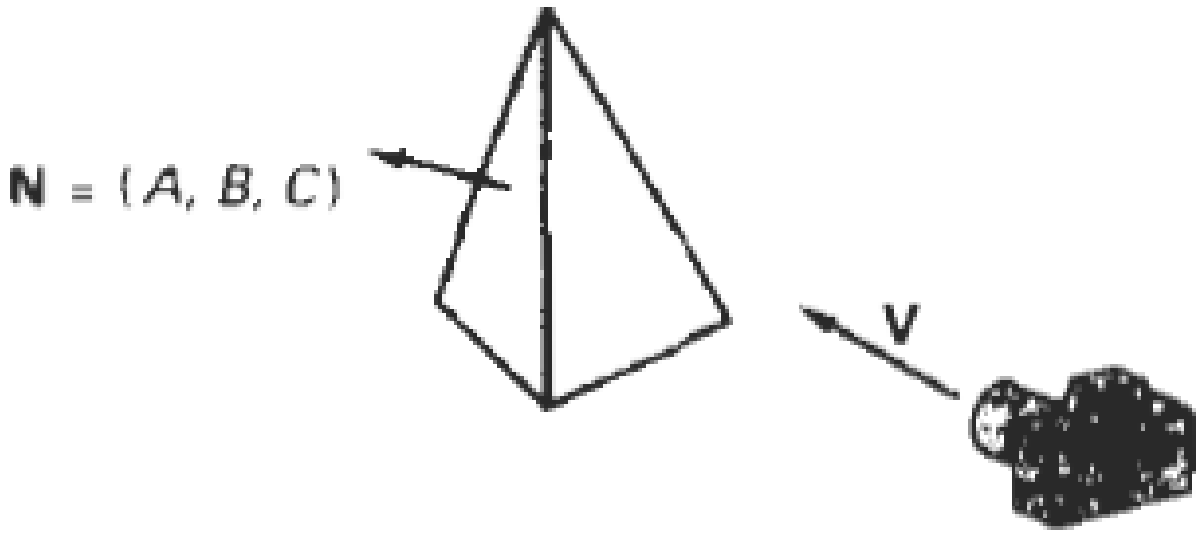
$$\mathbf{V} \cdot \mathbf{N} = V_z C$$

so that we only need to consider the sign of C, the z component of the normal vector $\mathbf{N}$

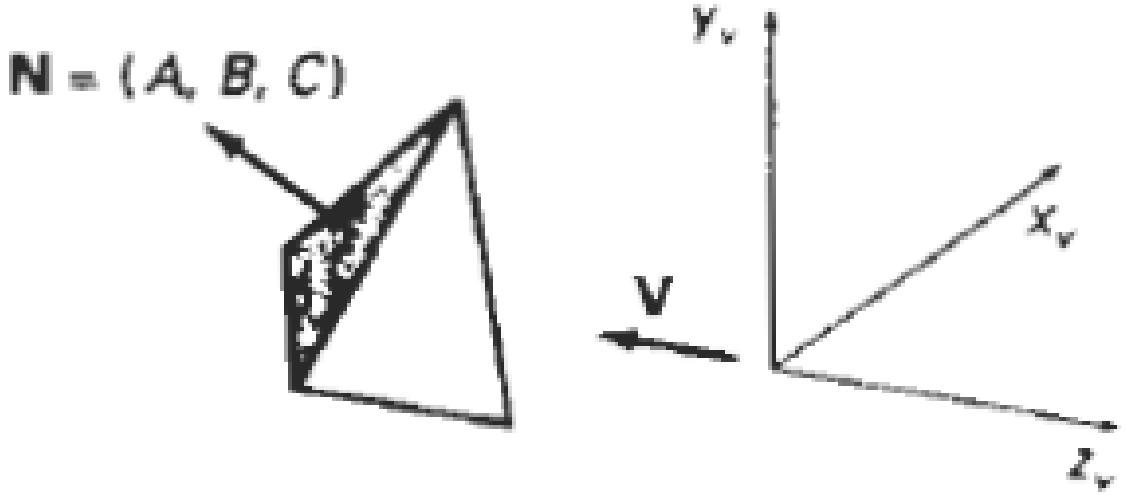
In a right-handed viewing system with viewing direction along the negative z_v axis (Fig. 13-2), the polygon is a back face if $C < 0$. Also, we cannot see any face whose normal has z component $C = 0$, since our viewing direction is grazing that polygon. Thus, in general, we can label any polygon as a back face if its normal vector has a z-component value:

$$C \leq 0$$

BACK FACE DETECTION ALGORITHM

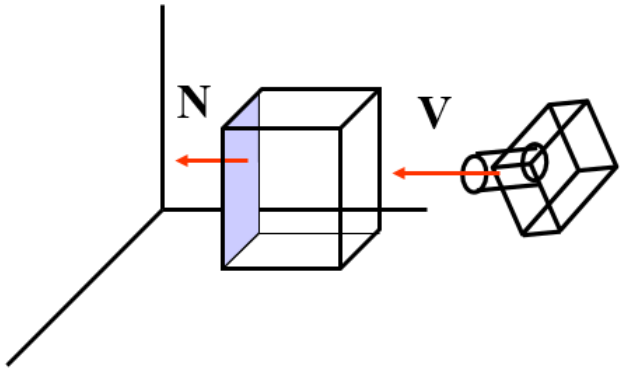


Vector **V** in the viewing direction and a back-face normal vector **N** of a polyhedron

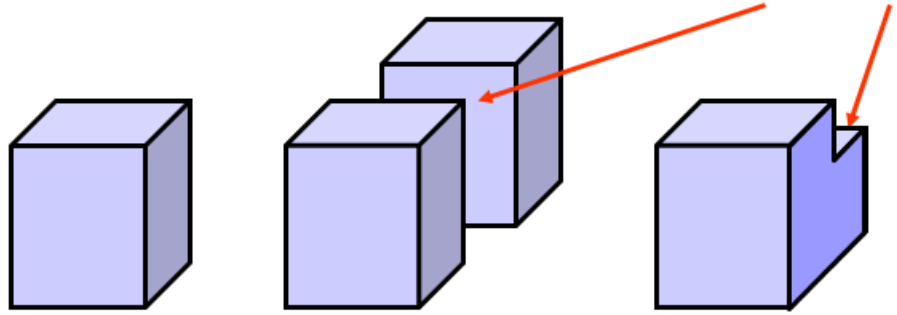


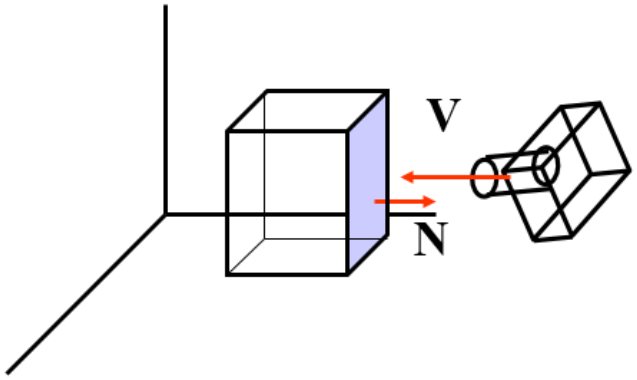
A polygon surface with plane parameter $C < 0$ in a right-handed viewing coordinate system is identified as a back face when the viewing direction is along the negative z_v axis.

BACK FACE DETECTION ALGORITHM



$\mathbf{V} \cdot \mathbf{N} > 0$: back face



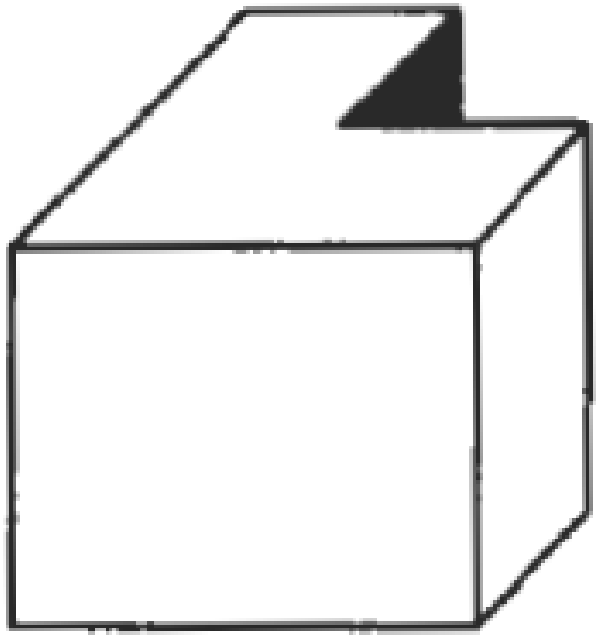


$\mathbf{V} \cdot \mathbf{N} < 0$: front face

BACK FACE DETECTION ALGORITHM

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 $C \geq 0$ when the viewing direction is along the positive z_v axis.



View of a concave polyhedron with one face partially hidden by other faces.

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, z) 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 13-4 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.

Depth Buffer Method

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 placed in the same xy location in the refresh buffer.

Depth Buffer Method

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

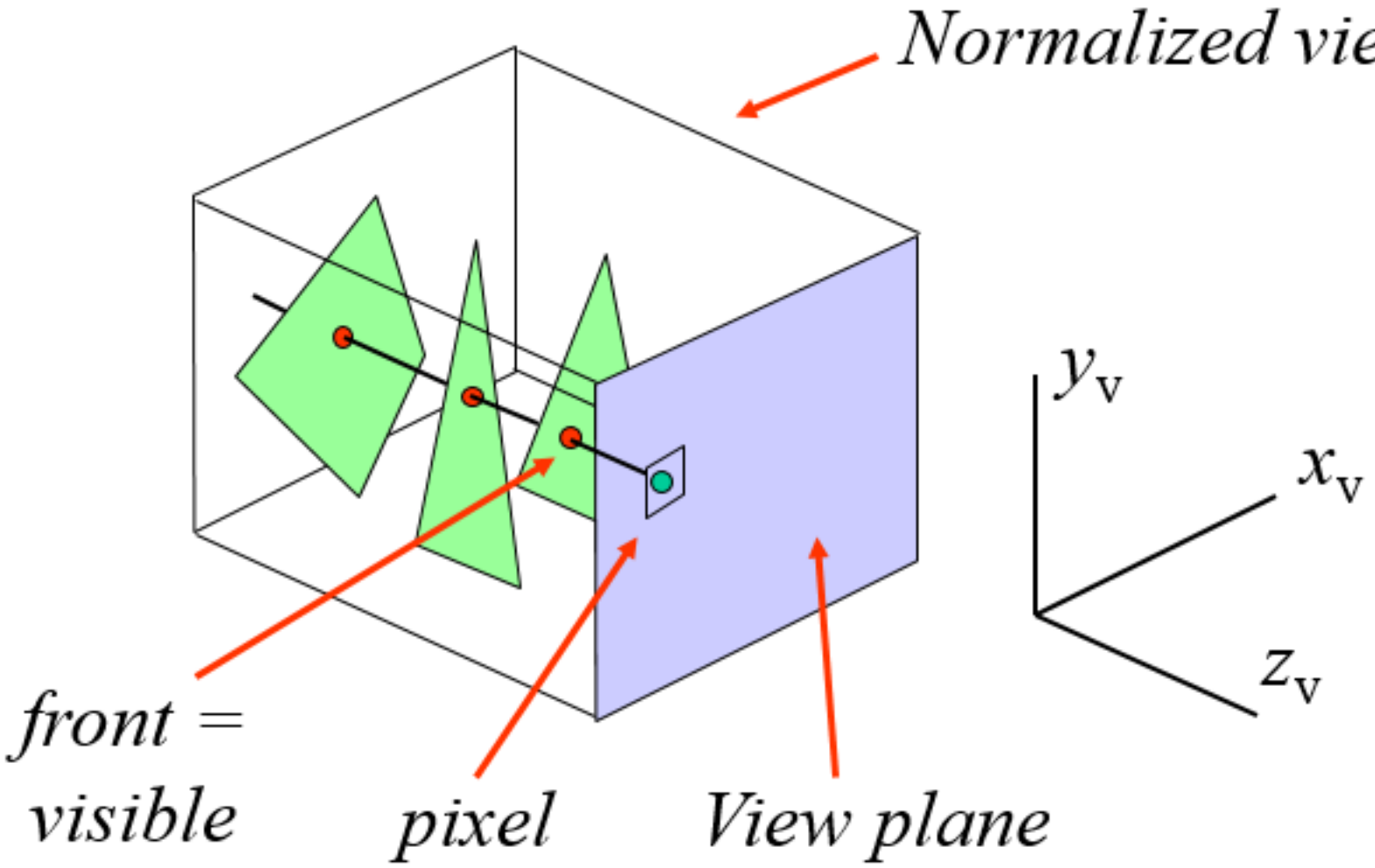
$$\text{depth}(x, y) = z, \qquad \text{refresh}(x, y) = I_{\text{surf}}(x,y)$$

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 Buffer Method

- Two buffer areas are required
  - Depth buffer
    - Store depth values for each (x, y) position
    - All positions are initialized to minimum depth
      - Usually 0 – most distant depth from the view plane
  - Refresh buffer
    - Stores the intensity values for each position
    - All positions are initialized to the background intensity

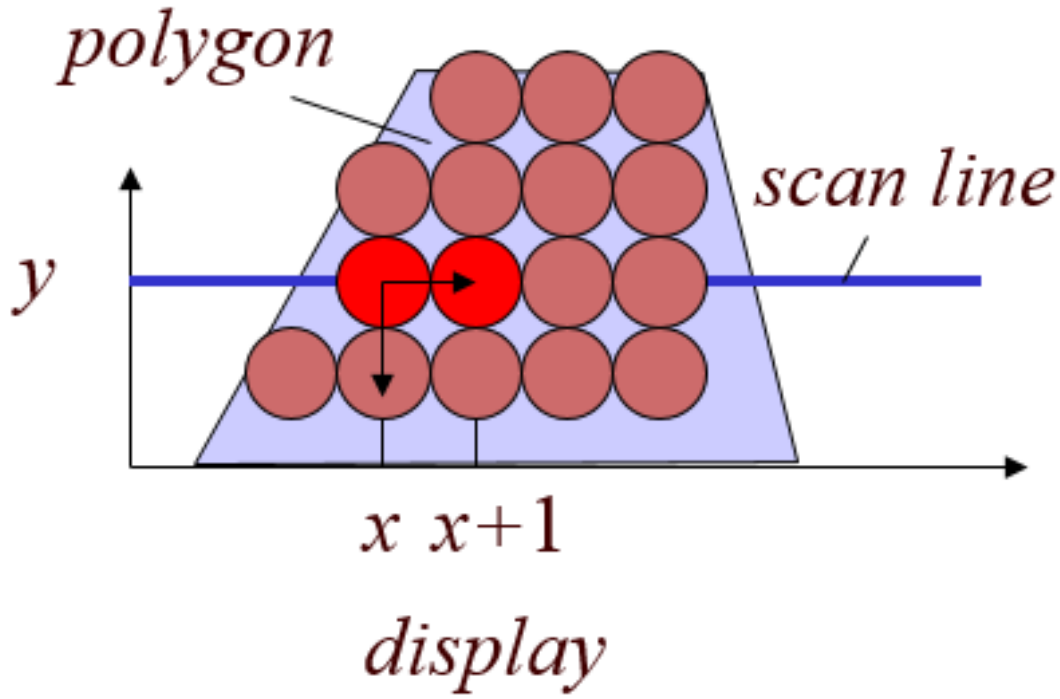
Depth Buffer Method



Algorithm:
Draw polygons,
Remember the color most in front.

Depth Buffer Method

Fast calculation z: use coherence.



$$\text{Plane}: Ax + By + Cz + D = 0$$

$$\text{Hence}: \quad z(x,y) = \frac{-Ax - By - D}{C}$$

$$\text{Also}: \quad z(x+1,y) = \frac{-A(x+1) - By - D}{C}$$

$$\text{Thus}: z(x+1,y) = z(x,y) - \frac{A}{C}$$

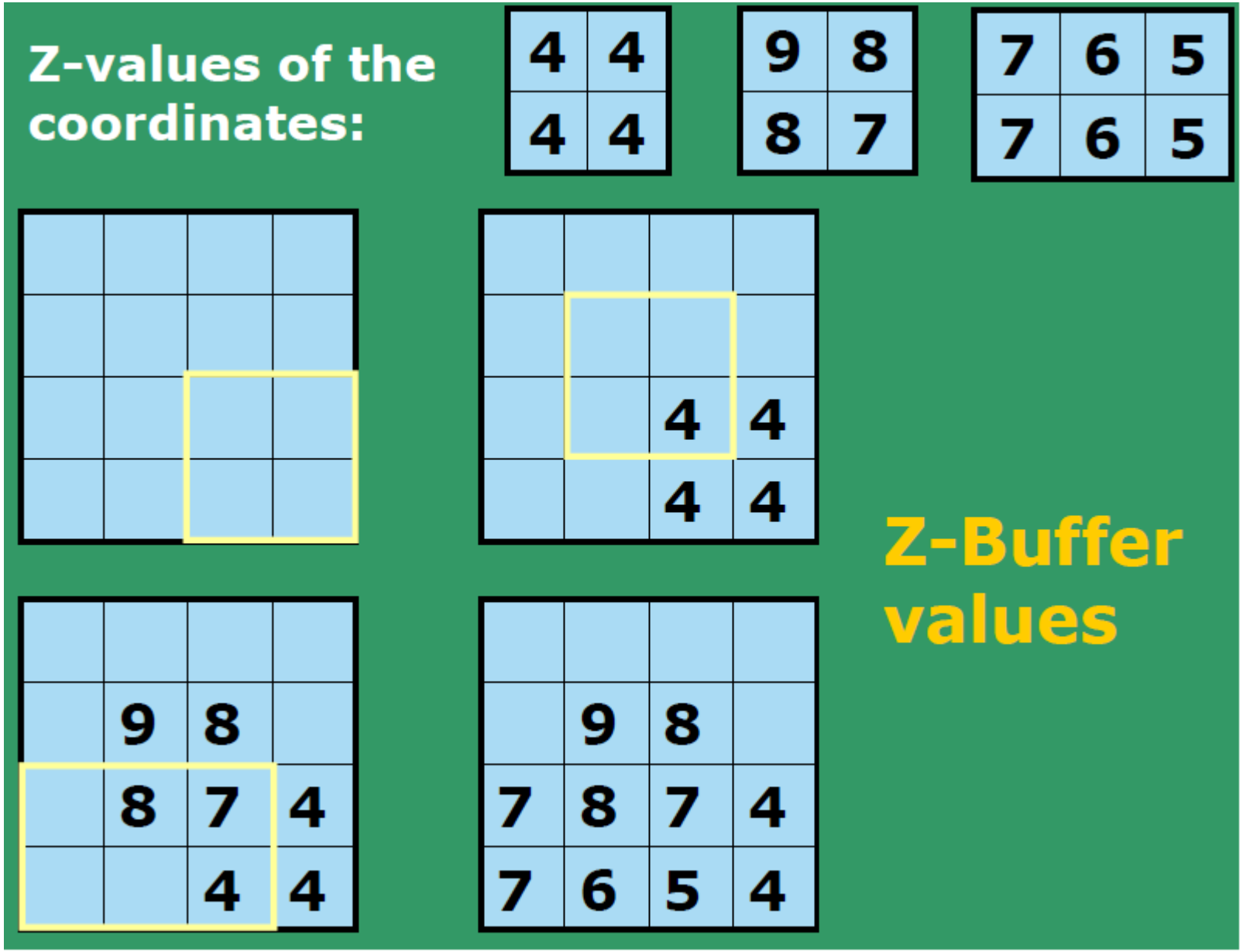
$$\text{Also}: z(x,y) = z(x,y-1) + \frac{B}{C}$$

Depth Buffer Method

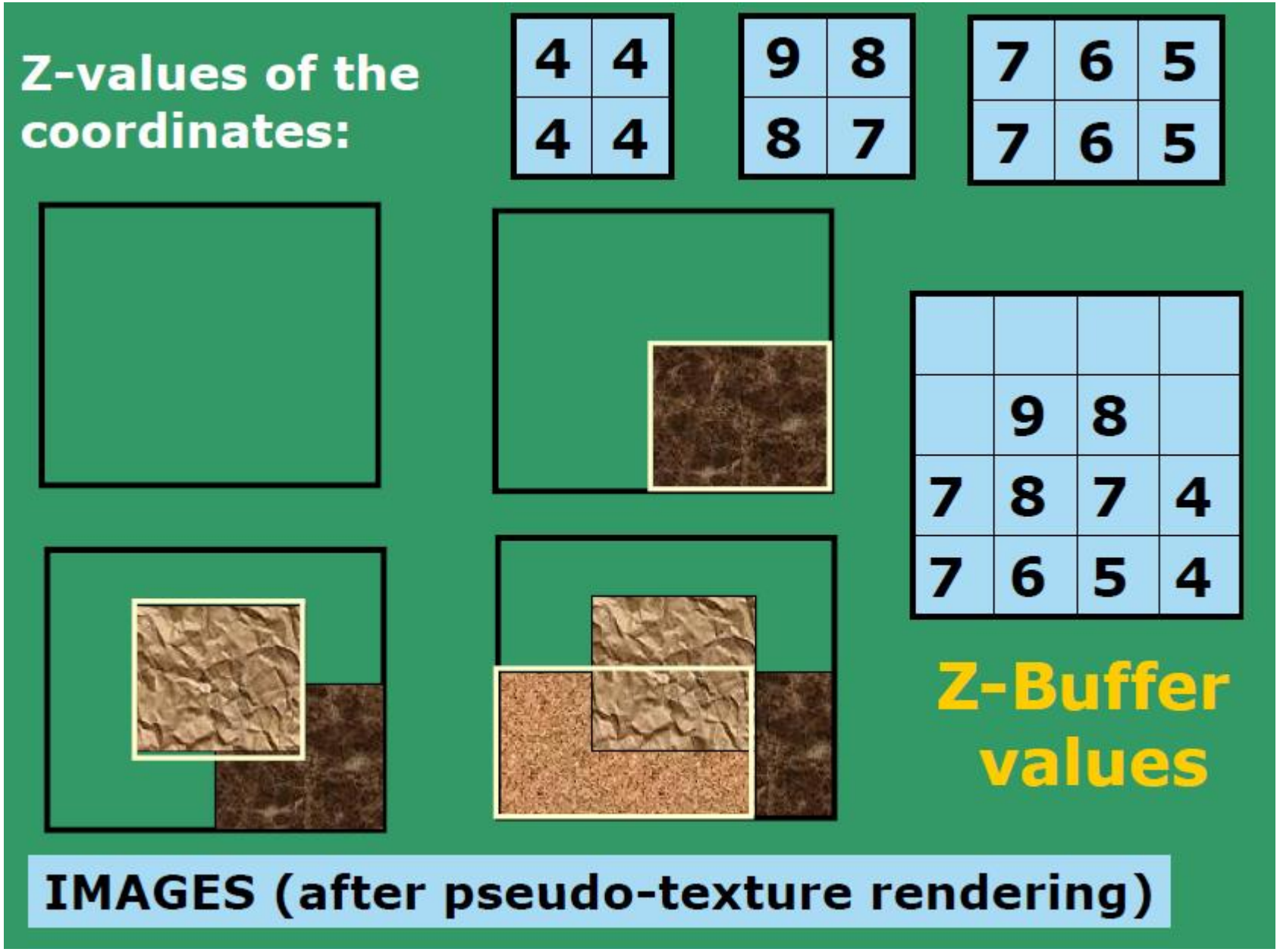
+ Easy to implement
+ Hardware supported
+ Polygons can be processed in arbitrary order
+ Fast: ~ #polygons, #covered pixels

- Costs memory
- Color calculation sometimes done multiple times
- Transparency is tricky

Depth Buffer Method



Depth Buffer Method



Scan Line Method

Extension of the scan-line algorithm for filling polygon interiors

For all polygons intersecting each scan line

Processed from left to right

Depth calculations for each overlapping surface

The intensity of the nearest position is entered into the refresh buffer

Scan Line Method

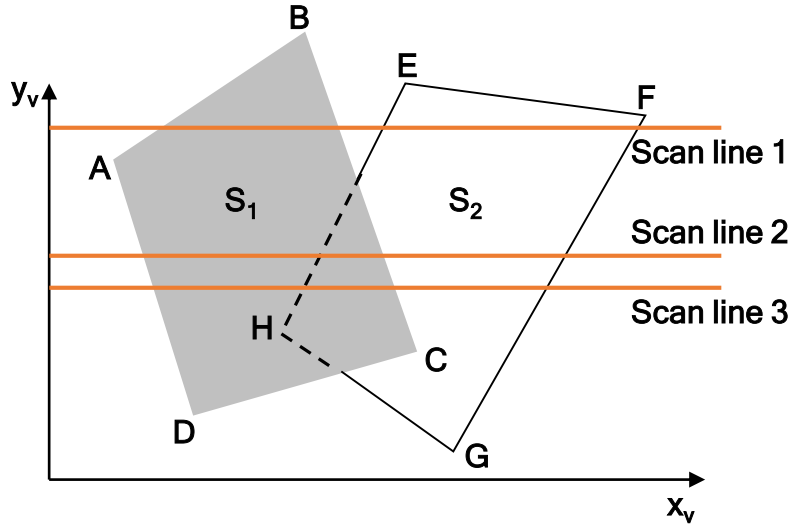
- Edge table
  - Coordinate endpoints for each line
  - Slope of each line
  - Pointers into the polygon table
    - Identify the surfaces bounded by each line
- Polygon table
  - Coefficients of the plane equation for each surface
  - Intensity information for the surfaces
  - Pointers into the edge table

Scan Line Method

- Active list
  - Contain only edges across the current scan line
  - Sorted in order of increasing x
- Flag for each surface
  - Indicate whether inside or outside of the surface
  - At the leftmost boundary of a surface
    - The surface flag is turned on
  - At the rightmost boundary of a surface
    - The surface flag is turned off

Scan Line Method

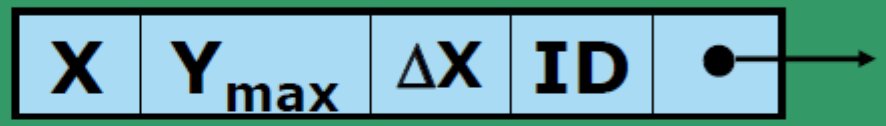
- Active list for scan line 1
  - Edge table
    - AB, BC, EH, and FG
    - Between AB and BC, only the flag for surface S_1 is on
      - No depth calculations are necessary
      - Intensity for surface S_1 is entered into the refresh buffer
    - Similarly, between EH and FG, only the flag for S_2 is on



Scan Line Method

Structure of each entry in ET:

- X-Coordn. of the end point with the smaller Y-Coordn.
- Y-Coordn. of the edge's other end point
- $\Delta X = 1/m$
- Polygon ID

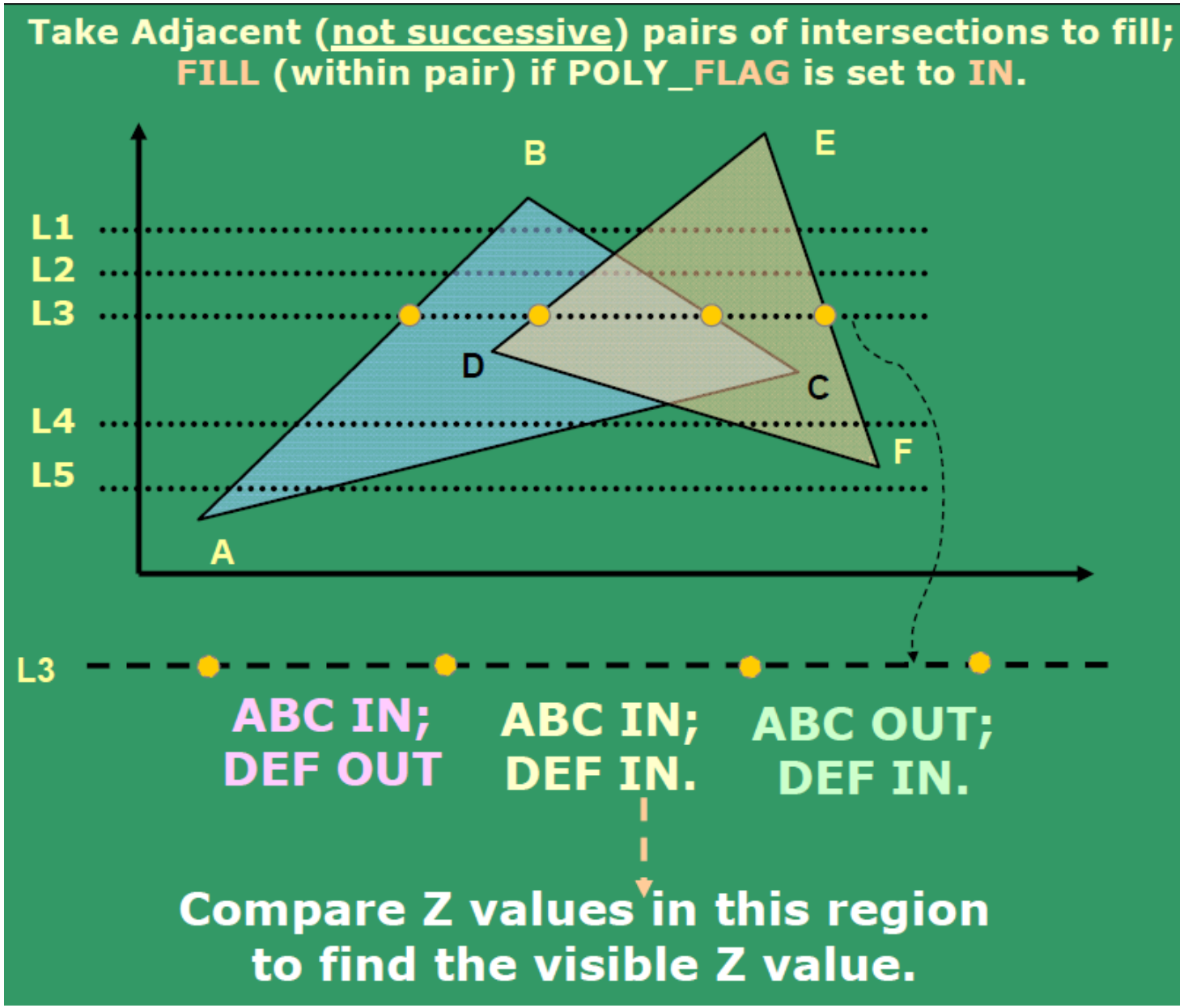


Structure of each entry in PT:

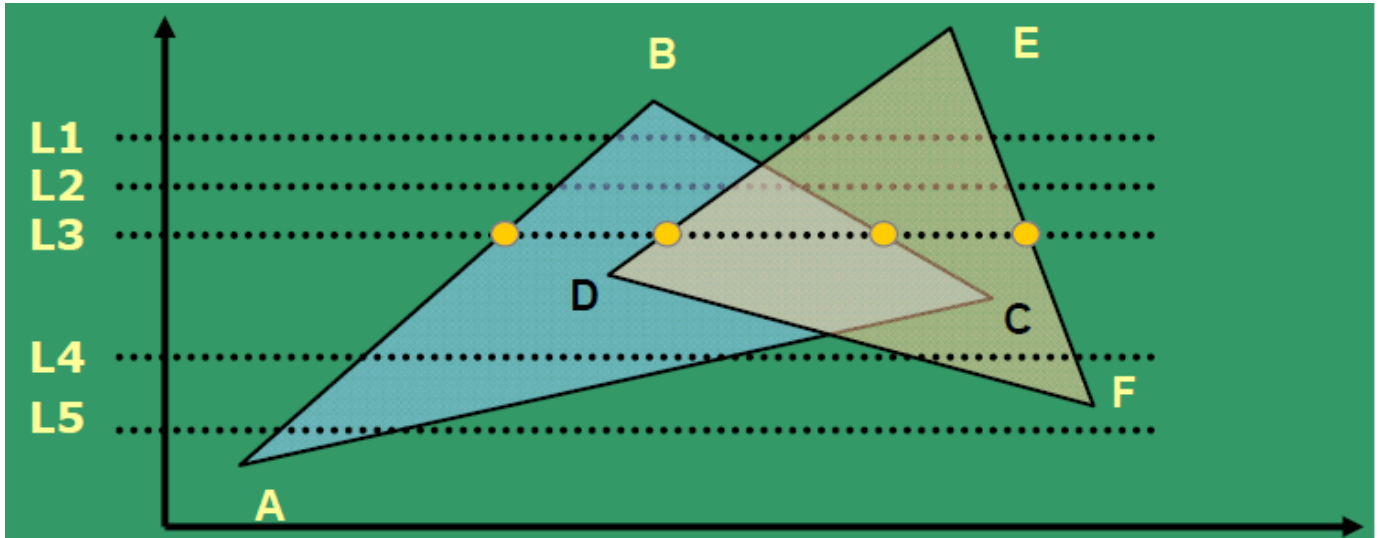
- Coefficients of the plane equations
- Shading or color information of the polygon
- Flag (IN/OUT), initialized to '*false*'

ID	Plane Coeffs.	Shading Info.	IN/OUT

Scan Line Method



Scan Line Method



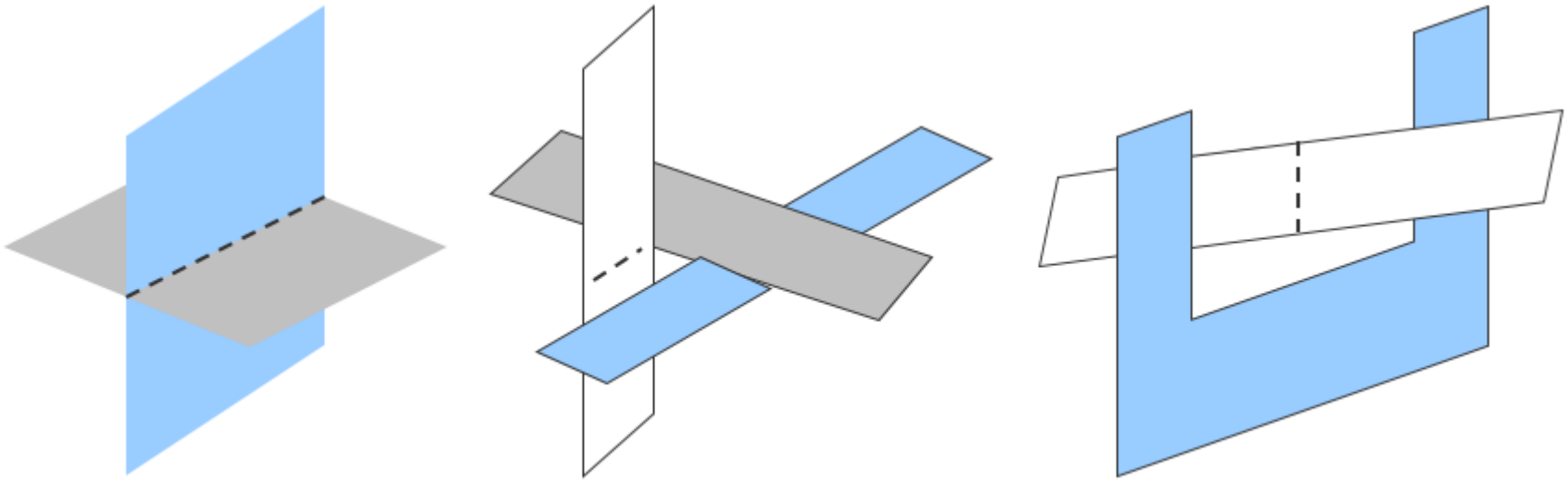
AET Contents				
Scan Line	Entries			
L5	AB	CA		
L4	AB	CA	FD	EF
L3, L2	AB	DE	BC	EF
L1	AB	BC	DE	EF

Scan Line Method

For scan line 2, 3
- AD, EH, BC, and FG
  - Between AD and EH, only the flag for S_1 is on
  - Between EH and BC, the flags for both surfaces are on
    - Depth calculation is needed
    - Intensities for S_1 are loaded into the refresh buffer until BC
- Take advantage of coherence
  - Pass from one scan line to next
  - Scan line 3 has the same active list as scan line 2
  - Unnecessary to make depth calculations between EH and BC

Scan Line Method

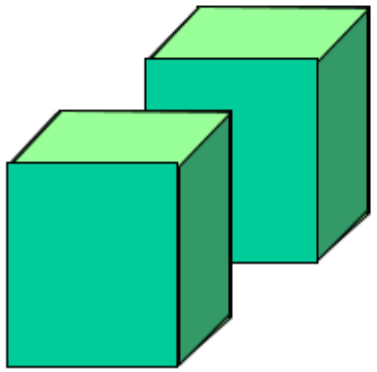
- Only if surfaces don't cut through or otherwise cyclically overlap each other
  - If any kind of cyclic overlap is present
    - Divide the surfaces



Depth Sorting Algorithm

- Image and Object space
- Aka *Painter's algorithm*

1. *Sort surfaces for depth*
2. *Draw them back to front*



The depth-sorting method performs the following basic functions:

1. Surfaces are sorted in order of decreasing depth.
2. Surfaces are scan converted in order, starting with the surface of greatest depth.

Sorting operations are carried out in both image and object space, and the scan conversion of the polygon surfaces is performed in image space.

Depth Sorting Algorithm

In creating an oil painting, an artist first paints the background colors
Next, the most distant objects are added, then the nearer objects, and so forth
At the final step, the foreground objects are painted on the canvas over the background and other objects that have been painted on the canvas
Each layer of paint covers up the previous layer. Using a similar technique, 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

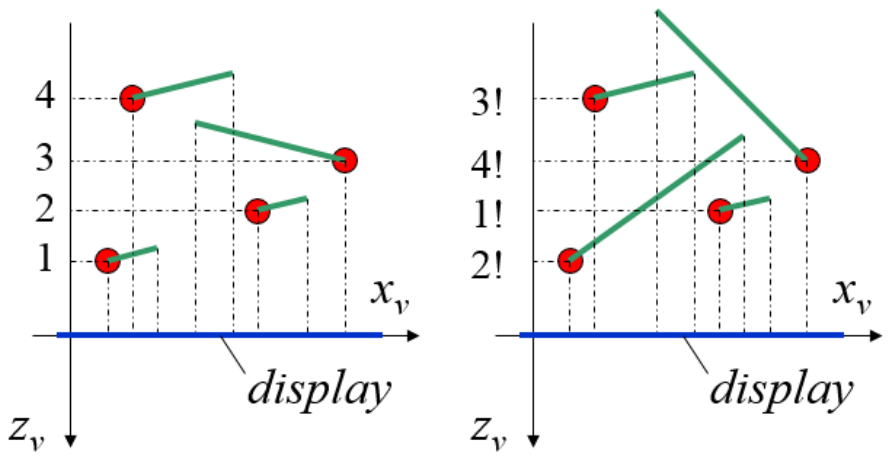
Depth Sorting Algorithm

- Painting polygon surfaces onto the frame buffer according to depth is carried out in several steps
- Assuming we are viewing along the-z direction, surfaces are ordered on the first pass according to the smallest ***z*** value on each surface
- Surface S with the greatest depth is then compared to the other surfaces in the list to determine whether there are any overlaps in depth
- If no depth overlaps occur, S is scan converted.

Depth Sorting Algorithm

Simplistic version sorting:

- Sort polygons for (average/frontal) *z*-value

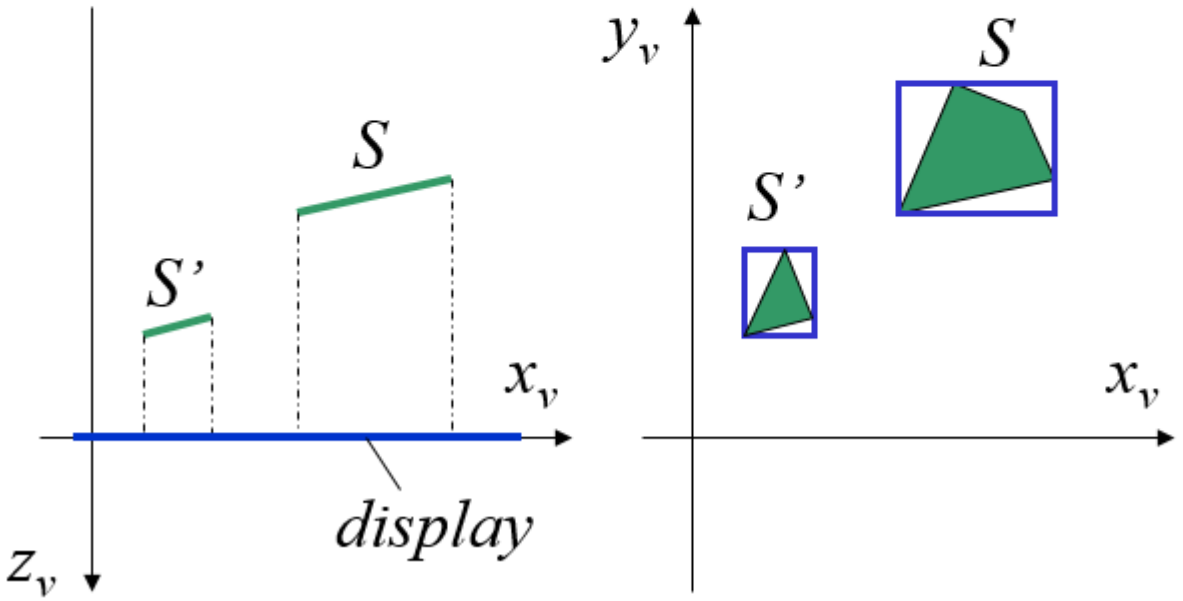


A polygon *S* can be drawn if all remaining polygons *S'* satisfy one of the following tests:

1. No overlap of *bounding rectangles* of *S* and *S'*
2. *S* is completely behind plane of *S'*
3. *S'* is completely in front of plane of *S*
4. Projections *S* and *S'* do not overlap

Depth Sorting Algorithm

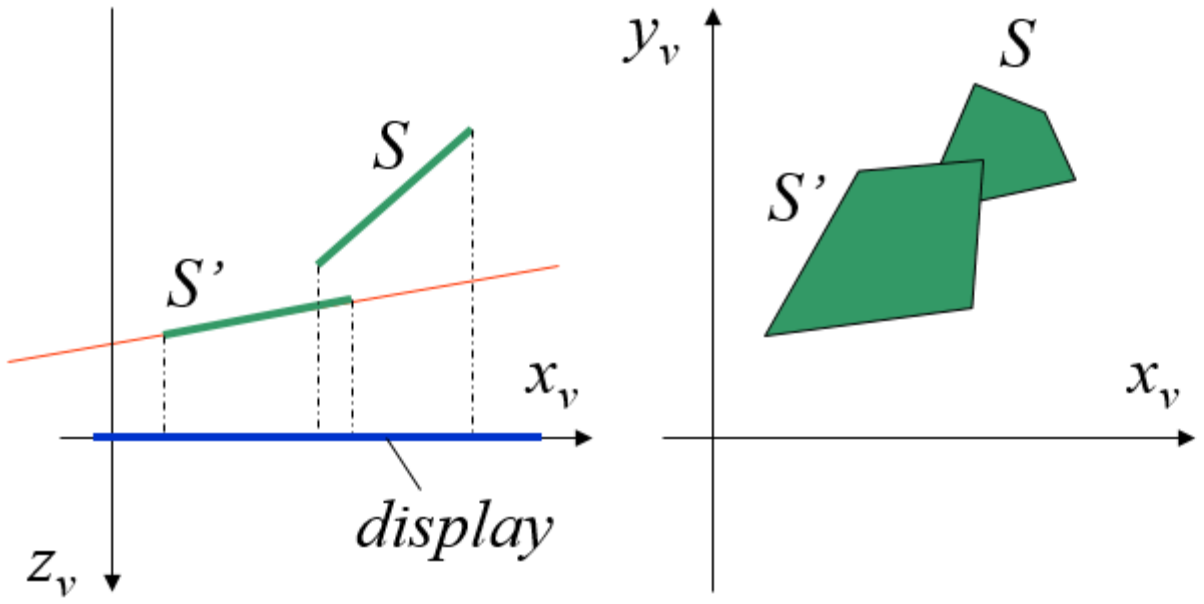
1. No overlap of *bounding rectangles* of S and S'



Depth Sorting Algorithm

2. S is completely behind plane of S'

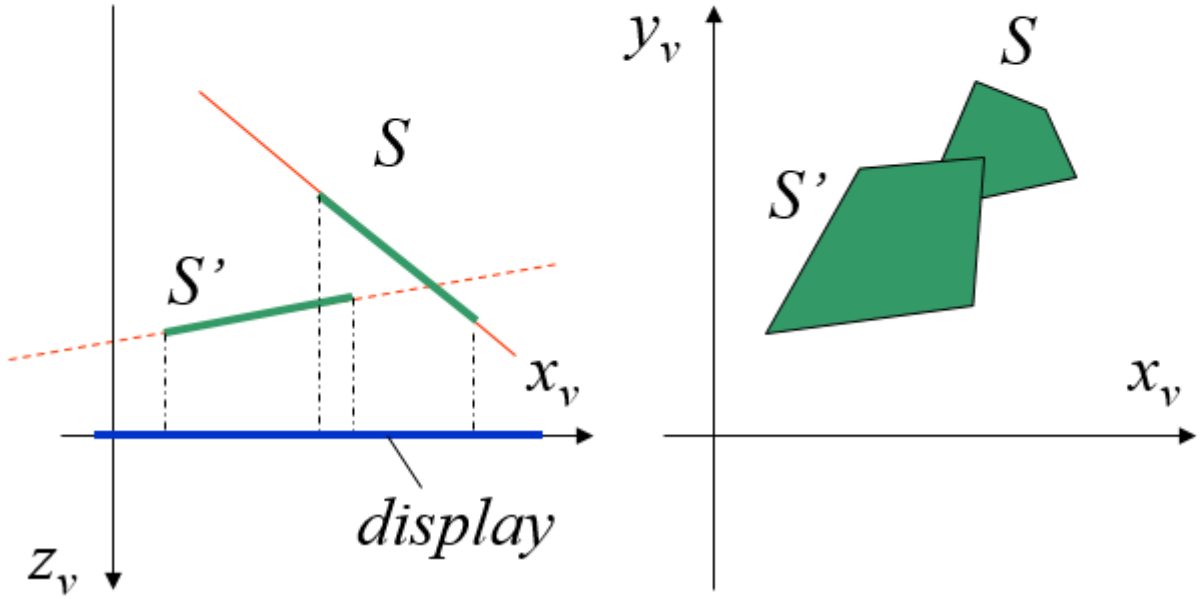
Substitute all vertices of S in plane equation S', and test if the result is always negative.



Depth Sorting Algorithm

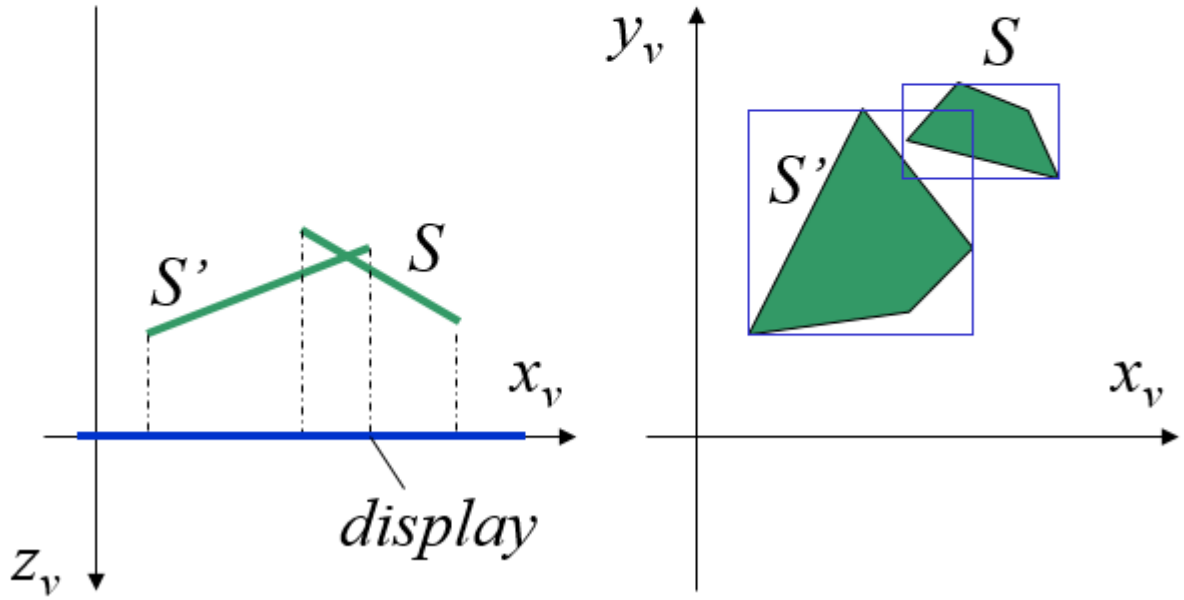
3. S' is completely in front of plane of S

Substitute all vertices of S' in plane equation of S, and test if the result is always positive



Depth Sorting Algorithm

4. Projections S and S' do not overlap

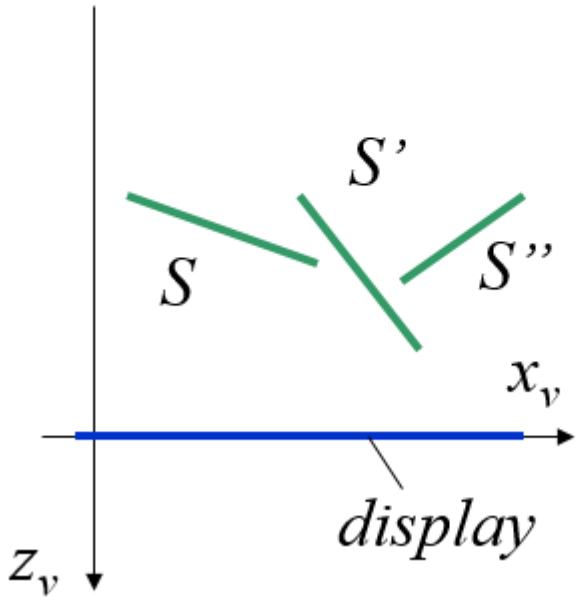


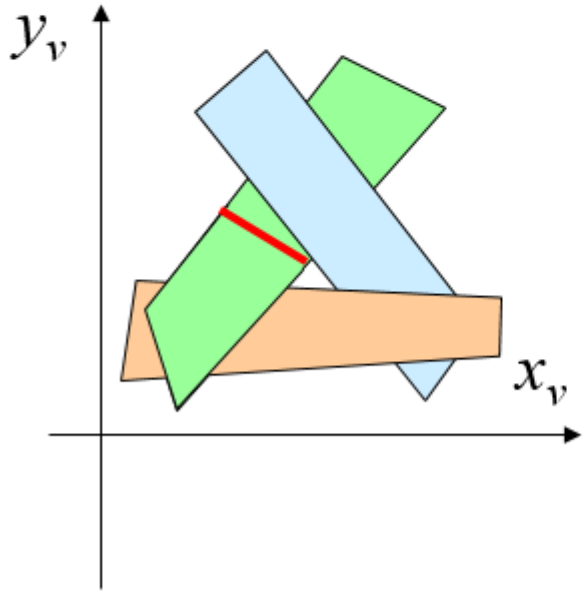
Depth Sorting Algorithm

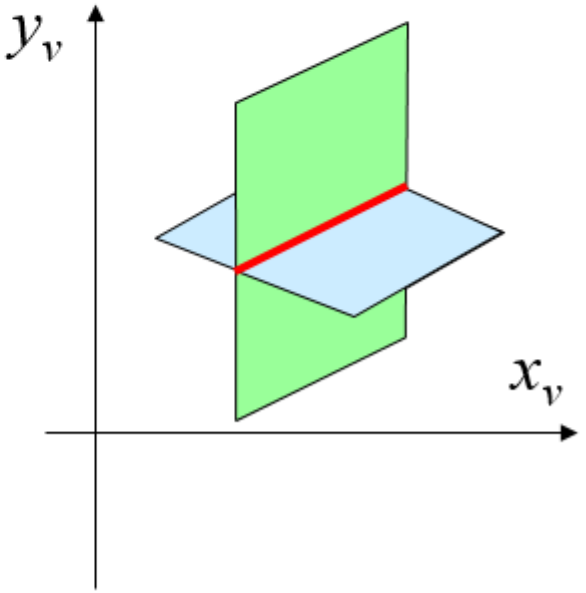
If all tests fail: Swap S and S', and restart with S'.

Problems: circularity and intersections

Solution: Cut up polygons.

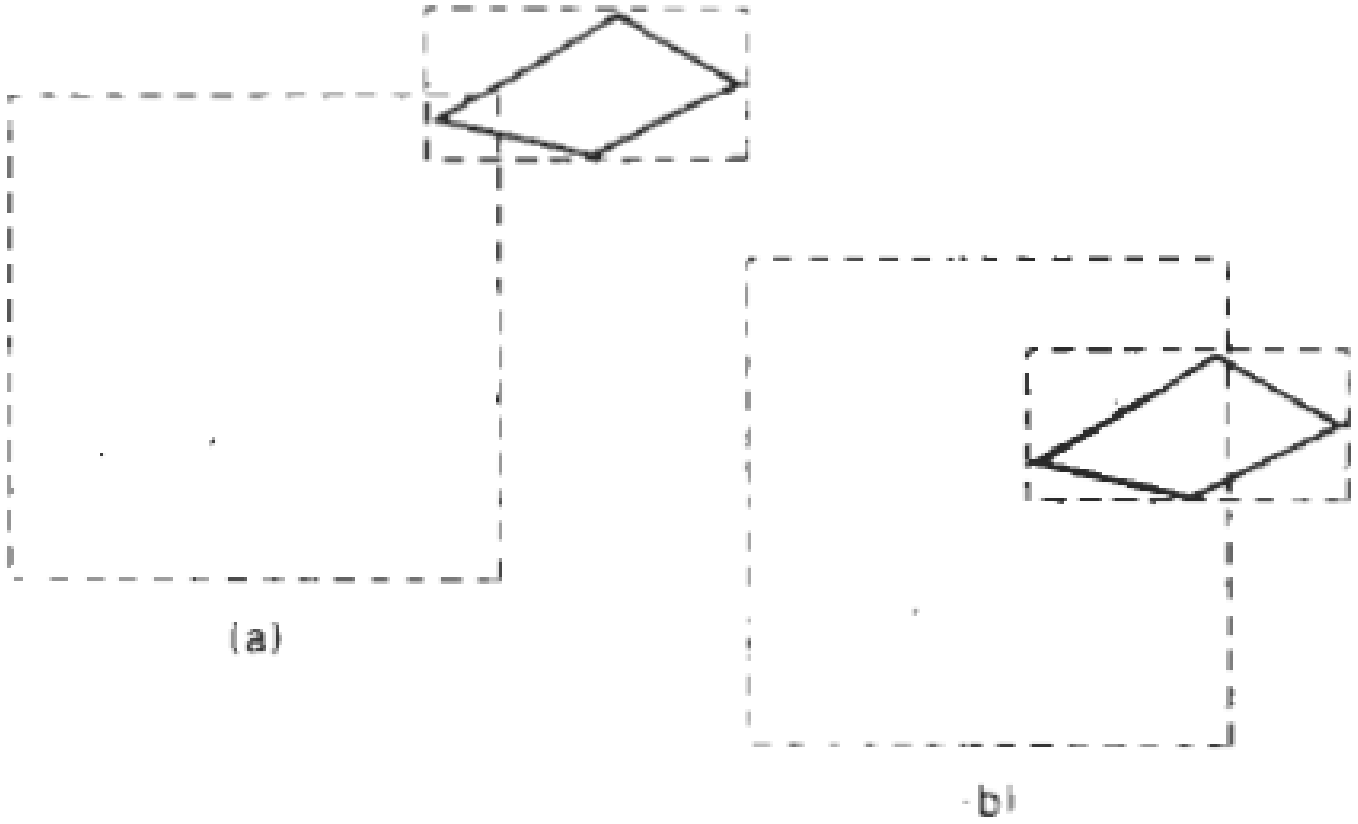






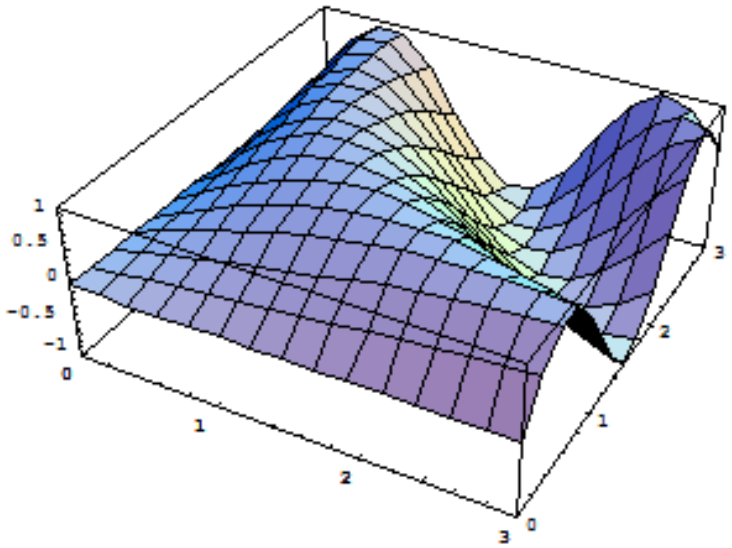
Depth Sorting Algorithm

two surfaces may or may not intersect even though their coordinate extents overlap in the *x*, y, and z directions.



Depth Sorting Algorithm

- Tricky to implement
- Polygons have to be known from the start
- Slow: ~ #polygons^2



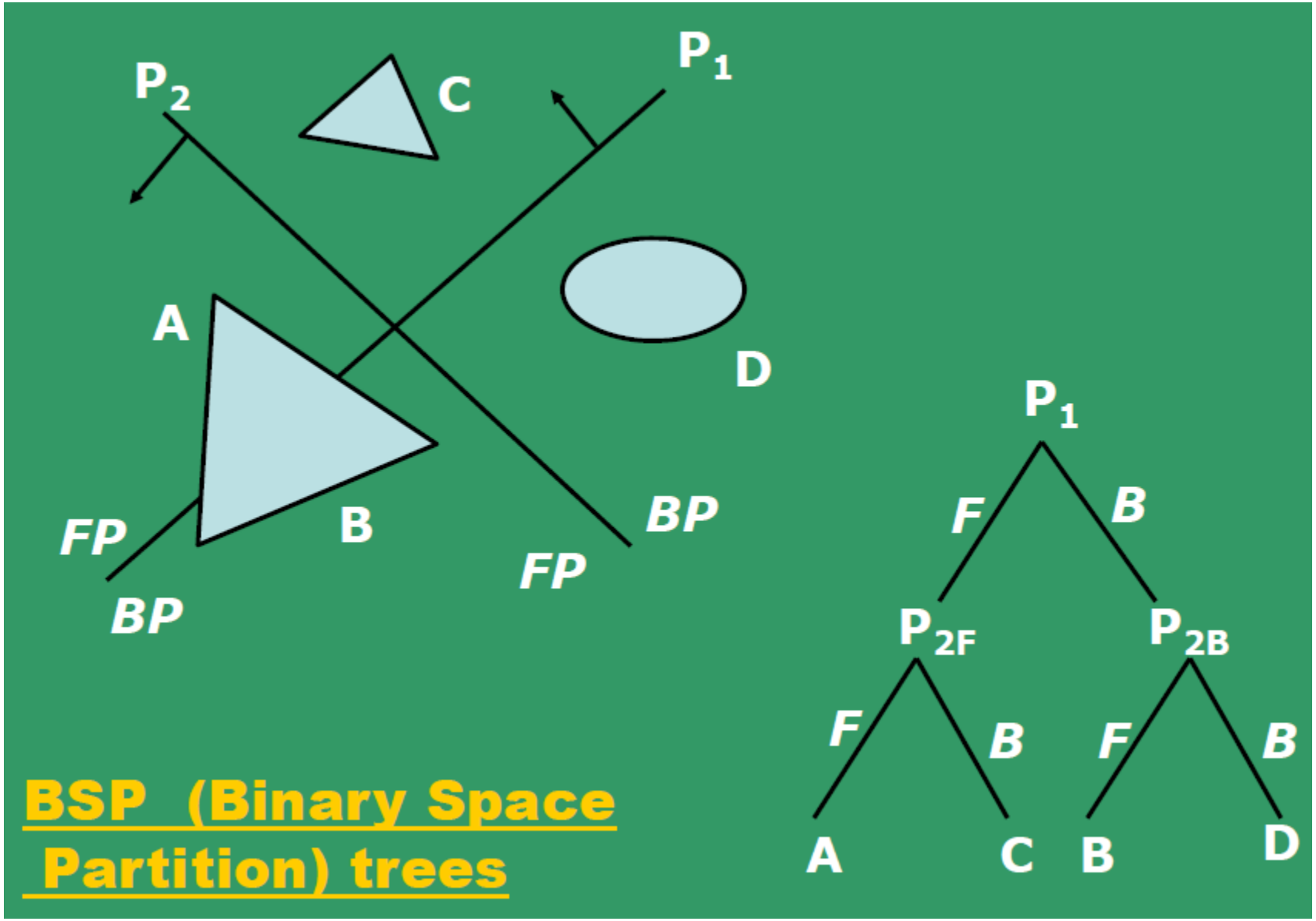
+ Fine for certain types of objects,
  such as plots of $z=f(x, y)$ or
  non-intersecting spheres
+ Produces exact boundaries polygons

BSP Method

A **binary space-partitioning (BSP)** tree is an efficient method for determining object visibility by painting surfaces onto the screen from back to front, as in the painter's algorithm. The BSP tree is particularly useful when the view reference point changes, but the objects in a scene are at fixed positions.

Applying a BSP tree to visibility testing involves identifying surfaces that are "inside" and "outside" the partitioning plane at each step of the space subdivision, relative to the viewing direction. Figure 13-19 illustrates the basic concept in this algorithm. With plane P_1, we first partition the space into two sets of objects. One set of objects is behind, or in back of, plane P_1 relative to the viewing direction, and the other set is in front of P_1. Since one object is intersected by plane P_1, we divide that object into two separate objects, labeled A and B. Objects A and C are in front of P_1, and objects B and D are behind P_1. We next partition the space again with plane P_2 and construct the binary tree representation shown in Fig. 13-19(b). In this tree, the objects are represented as terminal nodes, with front objects as left branches and back objects as right branches.

BSP Method


P2
C
P1
A
D
FP
B
BP
FP
BP
BSP (Binary Space Partition) trees
P1
F
B
P2F
P2B
F
B
F
B
A
C
B
D

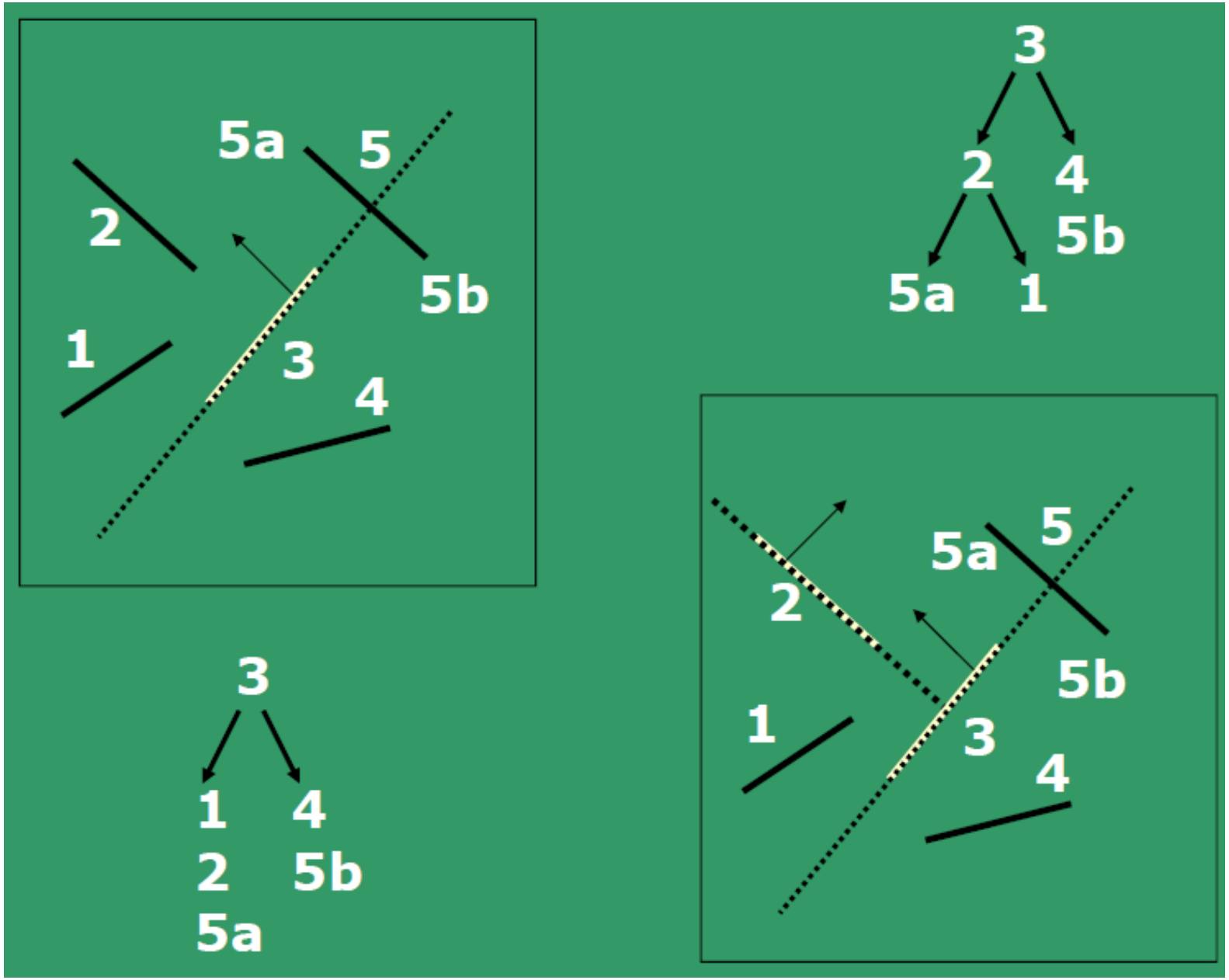
BSP Method

- Identify surfaces that are inside/front and outside/back w.r.t. the partitioning plane at each step of the space division, relative to the viewing direction.
- Start with any plane and find one set of objects behind and the rest in the front.
- In the tree, the objects are represented as terminal nodes with front objects as left branches and back objects as right branches.

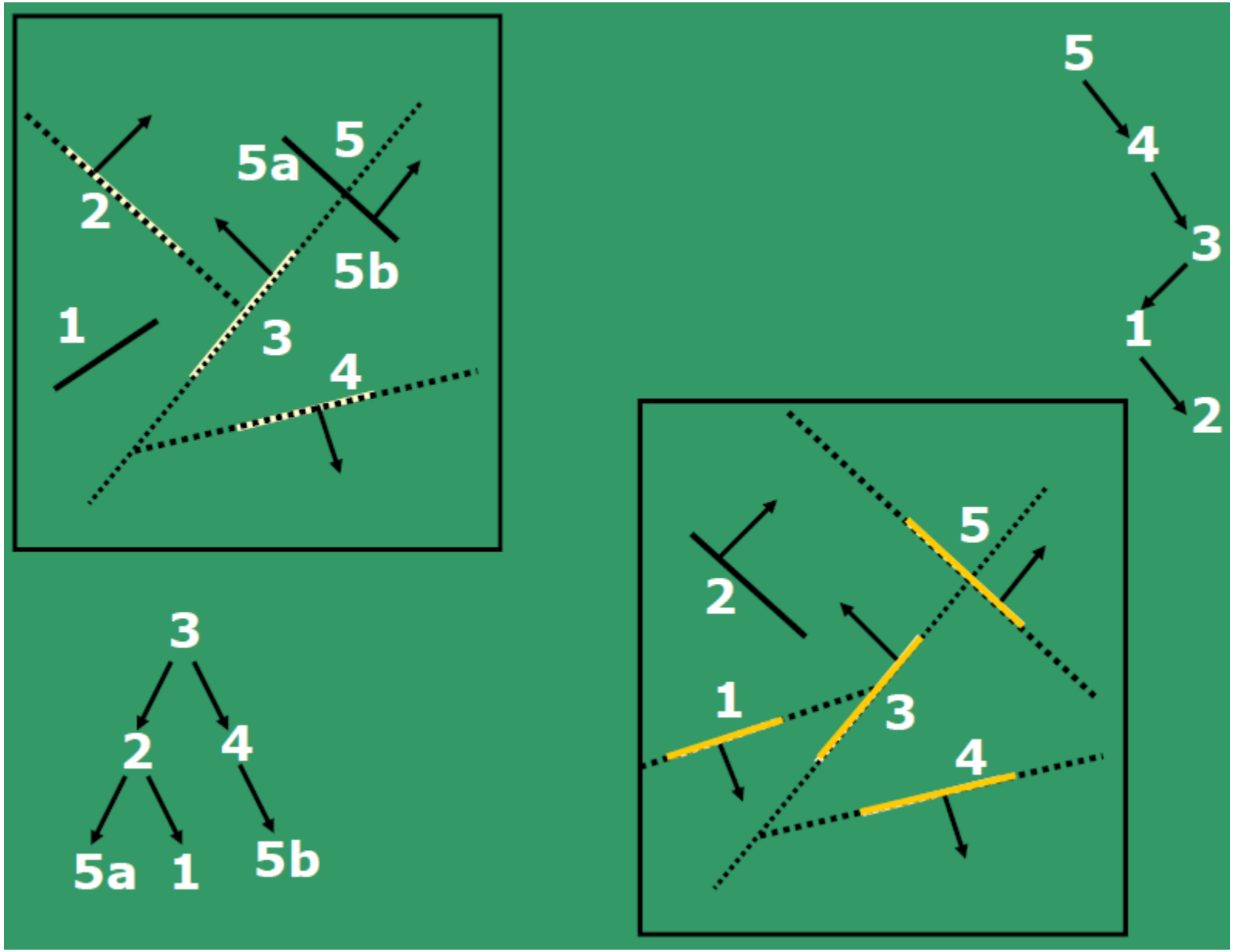
BSP Method

- BSP tree's root is a polygon selected from those to be displayed.
- One polygon each from the root polygon's front and back half plane, becomes its front and back children
- The algorithm terminates when each node contains only a single polygon.
- Intersection and sorting at object space/ precision

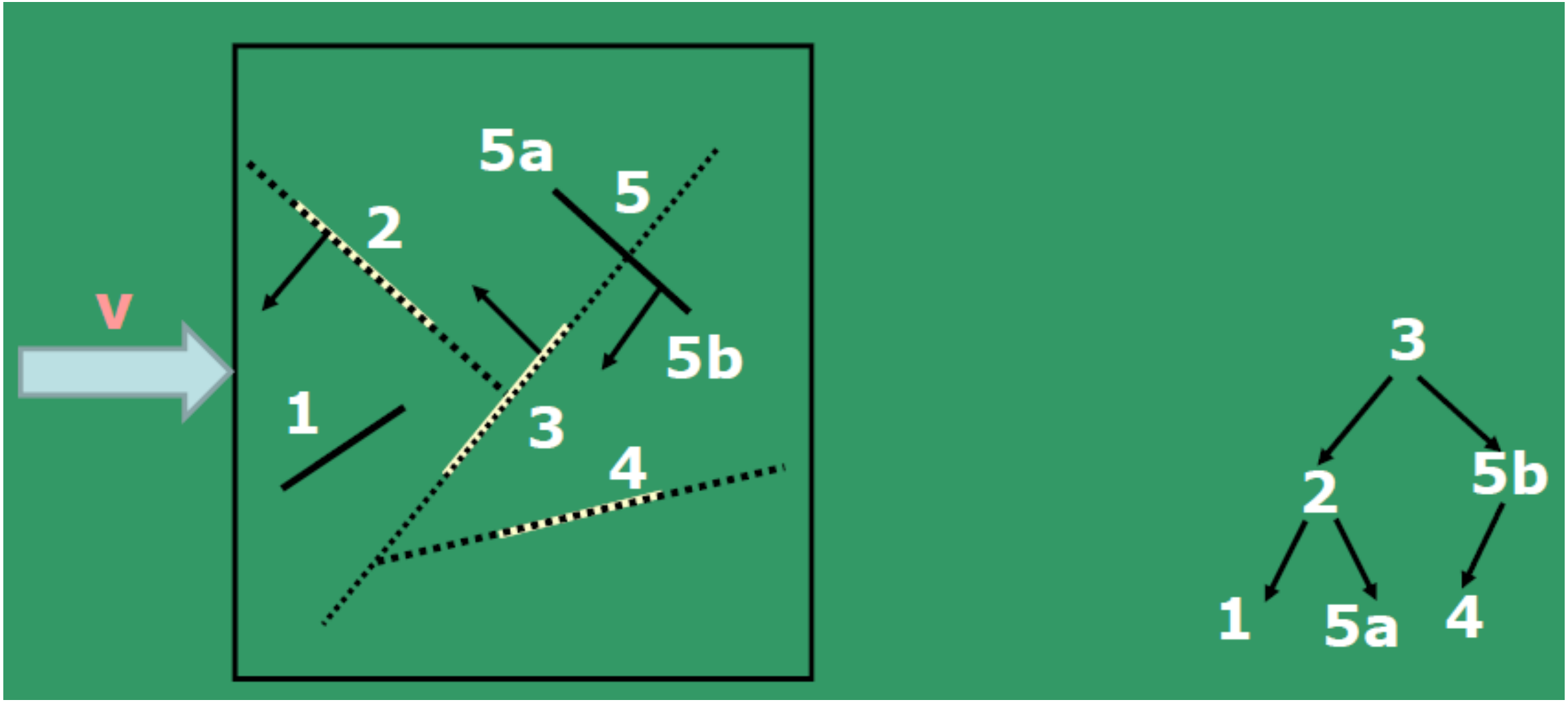
BSP Method



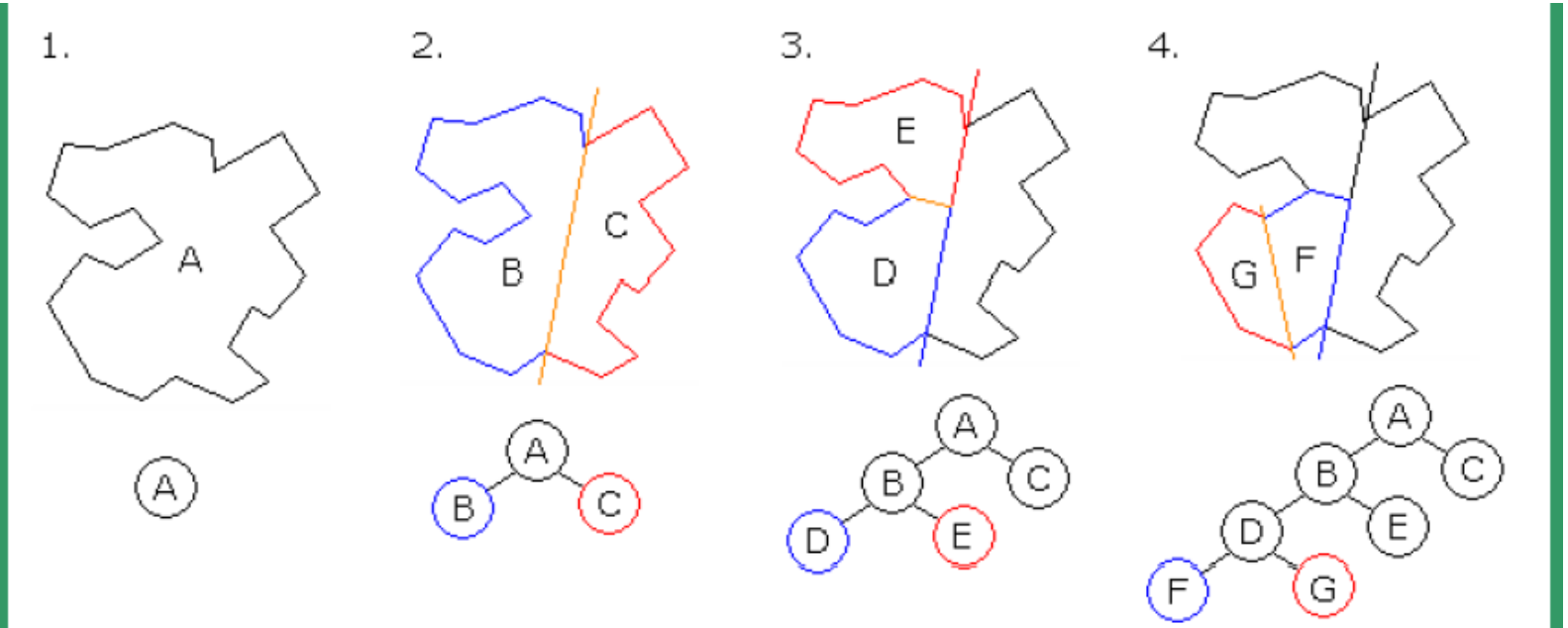
BSP Method



BSP Method



BSP Method



Also see:

potential visibility sets;

KD trees, R+-trees;

Bounding Volume Hierarchy (BVH)

and
Shadow Volume BSP Tree (SVBSP)

Other applications:

- **Robot navigation**
- **Collision Detecion**
- **GIS**
- **Image Registration**

BSP Method

- BSP tree may be traversed in a "modified in-order tree" walk to yield a correctly priority-ordered polygon list for an arbitrary viewpoint.
- If the viewer is in the root polygon's front half space, the algorithm must first display:
    1. All polygons in the root's rear half-space.
    2. Then the root.
    3. And finally all polygons in the front half-space.
- Use back face culling
- Each of the root's children is recursively processed.

BSP Method

- more polygon splitting may occur than in Painter's algorithm

- appropriate partitioning hyperplane selection is quite complicated and difficult

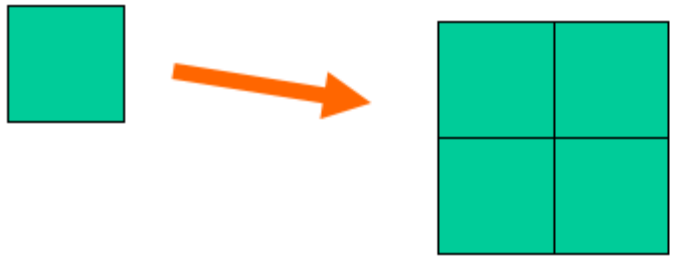
Finding Optimal BSP:

Finding an optimal root node is by testing a small number of candidates.

The node that results in the smallest number of nodes in the BSP tree should be chosen

Octree Method

- The hierarchical partitioning process is such that at any level of hierarchy each element is parted into 2^n elements of equal size by planes perpendicular to the axes.



1 partitioned to $2^2 = 4$

Octree Method

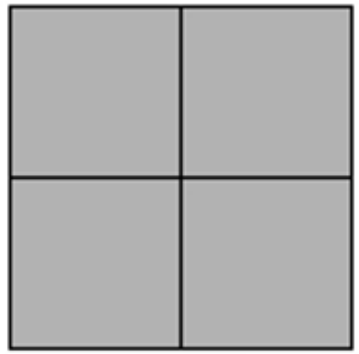
- From top to bottom
- From bottom to top

- Always recursively
- Good exercise in recursion and arrays
- Treat image as a Boolean or discrete function, ***what is the counterpart of these type of recursions?***

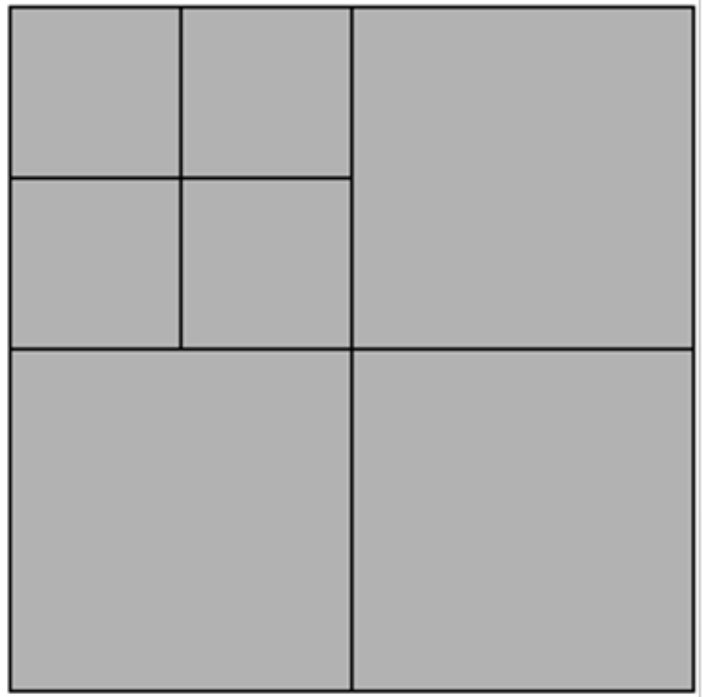
Octree Method

For simplicity, dimension = 2

Partitioning at any level i from i-1 can be done by defining a two-dimensional array **A(i)** for the i-th level



Level 2



Level 3

Octree Method

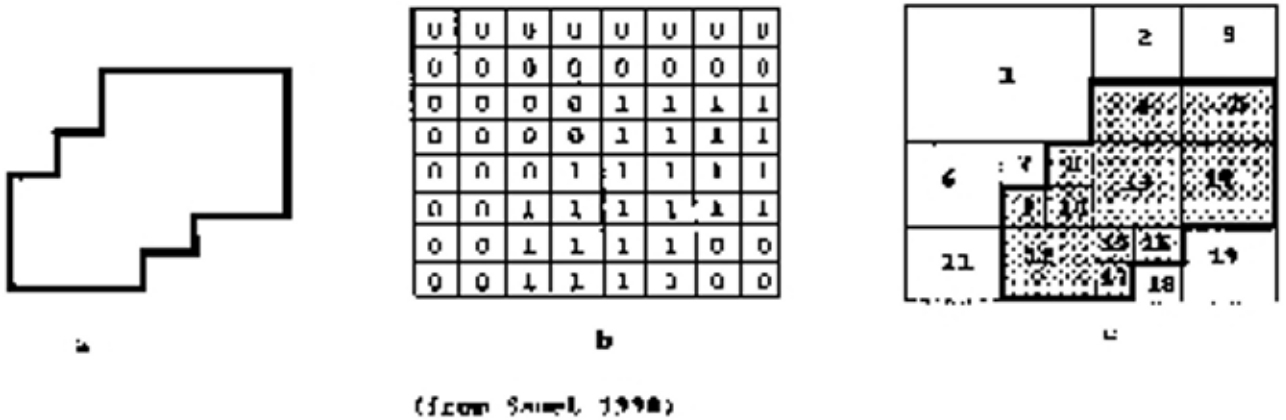
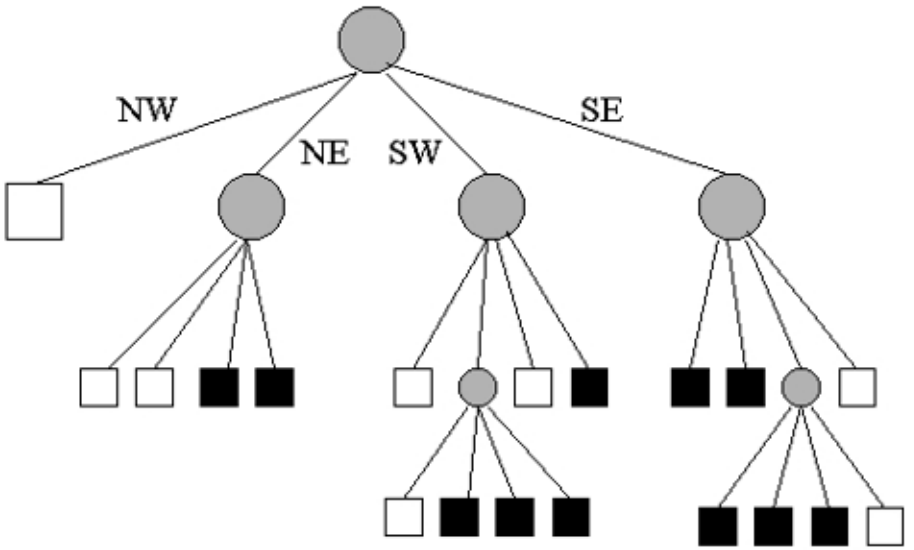


Image Binary repres

- The nodes immediately above the leaf nodes correspond to the array A(n-1) which is of size 2^n-1 by 2^n-1.
- The coarsest level, zero is a one by one matrix which can be referred to as the *root* of the tree.

We get an image represented by a quadtree



Final quadtree showing levels

Octree Method

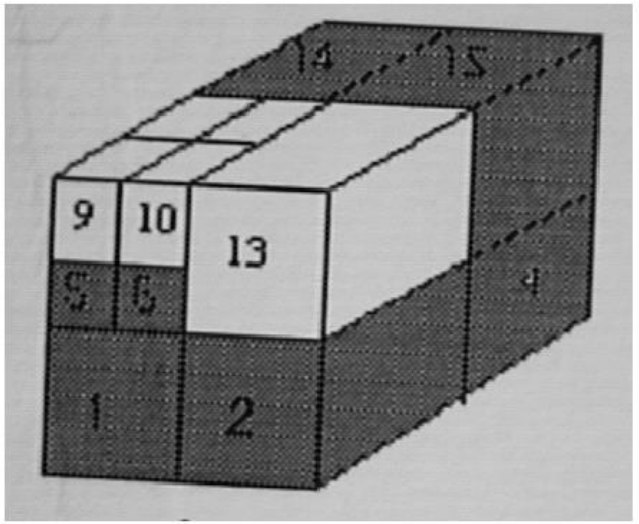
Structure of an Octree

- An Octree is a natural progression from a quadtree for the representation of 3-D space.
- Rather than an area dividing into four, octrees divide a 3-D object space into eight smaller cubes known as cells.
- If any of the cells is homogeneous, that is the cell lies entirely inside or outside the object, the sub-division stops.

Octree Method

Structure of an Octree

- If, on the other hand, the cell is heterogeneous, that is intersected by one or more of the object's bounding surfaces, the cell is sub-divided further into eight sub-cells.
- The sub-division process halts when all the leaf cells are homogeneous to some degree of accuracy.



Octree Method

- Octrees are able to provide a representation of just about any arbitrarily shaped object; whether it is convex, concave or with holes.
- Different properties such as surface area, center of mass and interface are easily evaluated at different levels.

Octree Method

- The quadtree, octree and binary tree decomposition methods are widely used in two and three dimension image processing and computer graphics

- Some of the application areas involve:
  - the image data structure,
  - region representation,
  - picture segmentation,
  - component labeling,
  - image smoothing,
  - image enhancement,
  - data compression

Octree Method

Applications

- Pattern recognition
- Shape analysis
- Image segmentation
- Region matching
- Images can be represented with pyramids and thus, both local and global feature extraction is possible

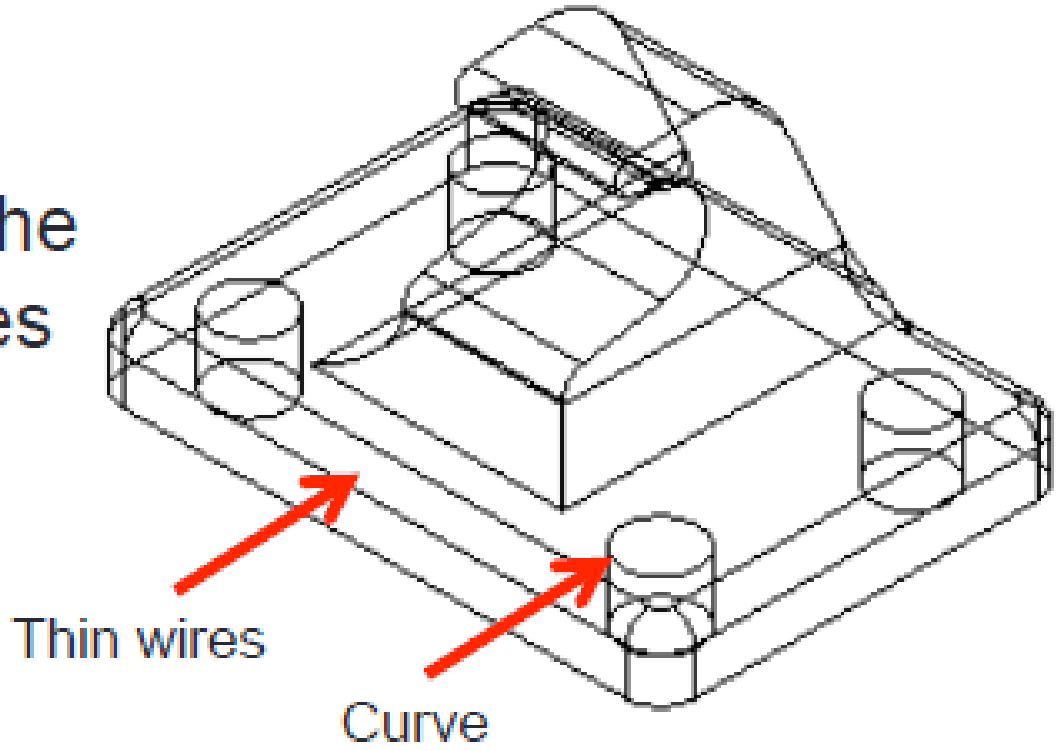
Wireframe Method

When only the outline of an object is to be displayed, visibility tests are applied to surface edges. Visible edge sections are displayed, and hidden edge sections can either be eliminated or displayed differently from the visible edges. For example, hidden edges could be drawn as dashed lines, or we could use depth cueing to decrease the intensity of the lines as a linear function of distance from the view plane. Procedures for determining visibility of object edges are referred to as **wireframe-visibility methods**. They are also called **visible-line detection methods** or **hidden-line detection methods**. Special wireframe-visibility procedures have been developed, but some of the visible-surface methods discussed in preceding sections can also be used to test for edge visibility.

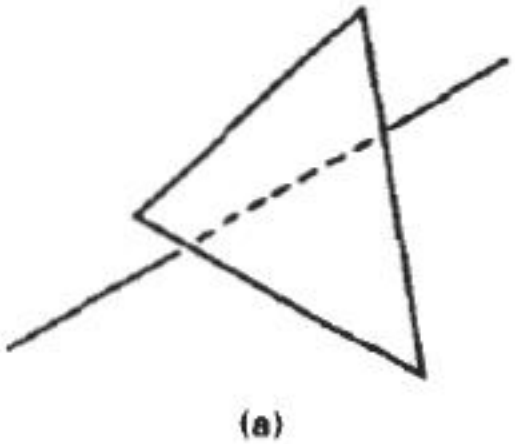
Wireframe Method

A wireframe representation is a 3-D line drawing of an object showing only the edges without any side surface in between.

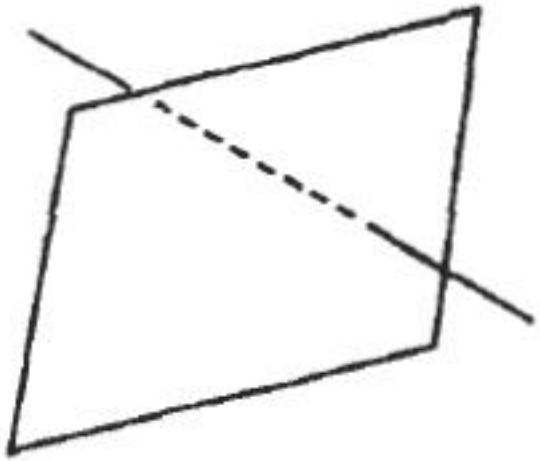
A frame constructed from thin wires representing the edges and projected lines and curves.



Wireframe Method



(a)



(b)

A direct approach to identifying the visible lines in a scene is to compare each line to each surface. The process involved here is similar to clipping lines against arbitrary window shapes, except that we now want to determine which sections of the lines are hidden by surfaces. For each line, depth values are compared to the surfaces to determine which line sections are not visible. We can use coherence methods to identify hidden line segments without actually testing each coordinate position. If both line intersections with the projection of a surface boundary have greater depth than the surface at those points, the line segment between the intersections is completely hidden, as in Fig. 13-28(a). This is the usual situation in a scene, but it is also possible to have lines and surfaces intersecting each other. When a line has greater depth at one boundary intersection and less depth than the surface at the other boundary intersection, the line must penetrate the surface interior, as in Fig. 13-28(b). In this case, we calculate the intersection point of the line with the surface using the plane equation and display only the visible sections.

Wireframe Method

There are two important aspects to the use of wire-frame models in CAD.

The first is the computer representation of an object, and this is concerned with the structure needed to encode a wire-frame model.

The second is concerned with the computational procedures needed to produce and manipulate the viewing or visualization of this representation.

Wireframe Method

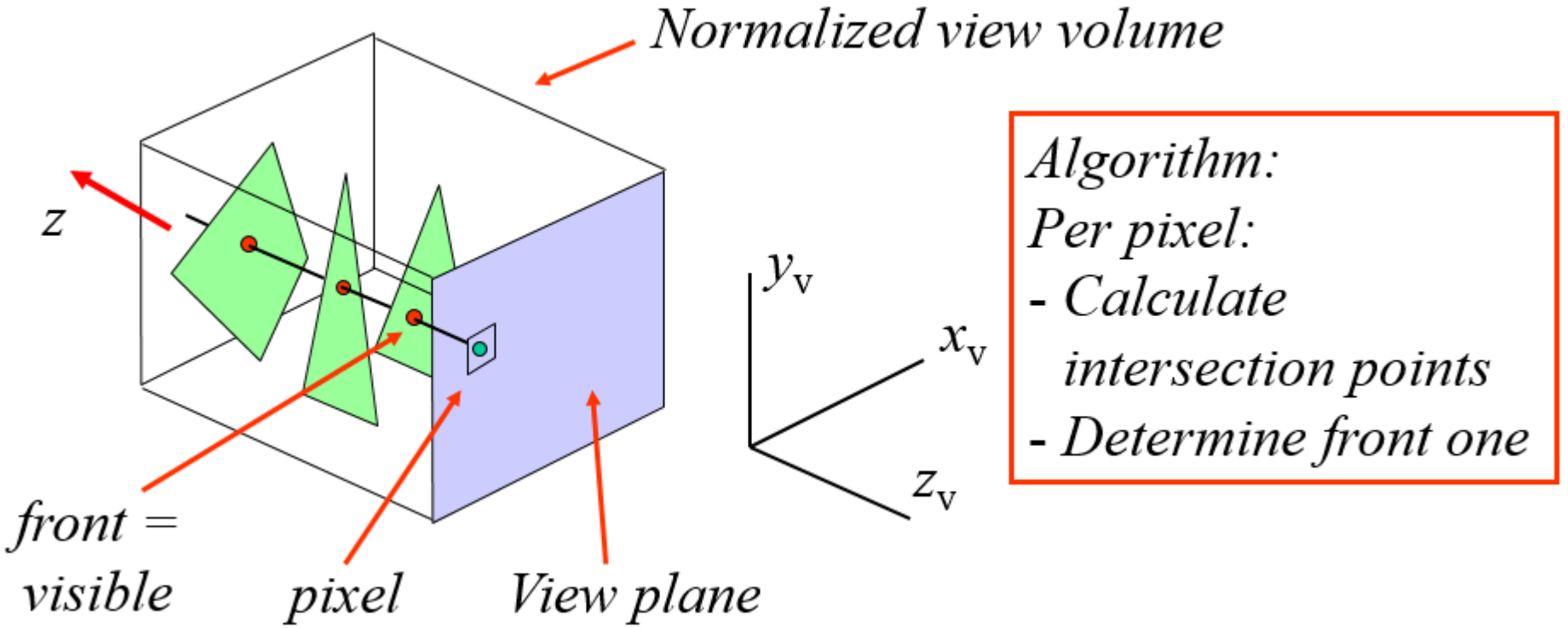
- Can quickly and efficiently convey information than multiview drawings.
- The only lines seen are the intersections of surfaces.
- Can be used for finite element analysis.
- Contain most of the information needed to create surface, solid and higher order models

Wireframe Method

- Geometric entities are lines and curves in 3D
- Volume or surfaces of object not defined
- Easy to store and display
- Hard to interpret – ambiguous
- Hidden lines are not removed
- For complex items, the result can be a jumble of lines that is impossible to determine.
- No ability to determine computationally information such as the line of intersect between two faces of intersecting models.

Ray-casting Algorithm 1

- Image-space method
- Related to depth-buffer, order is different



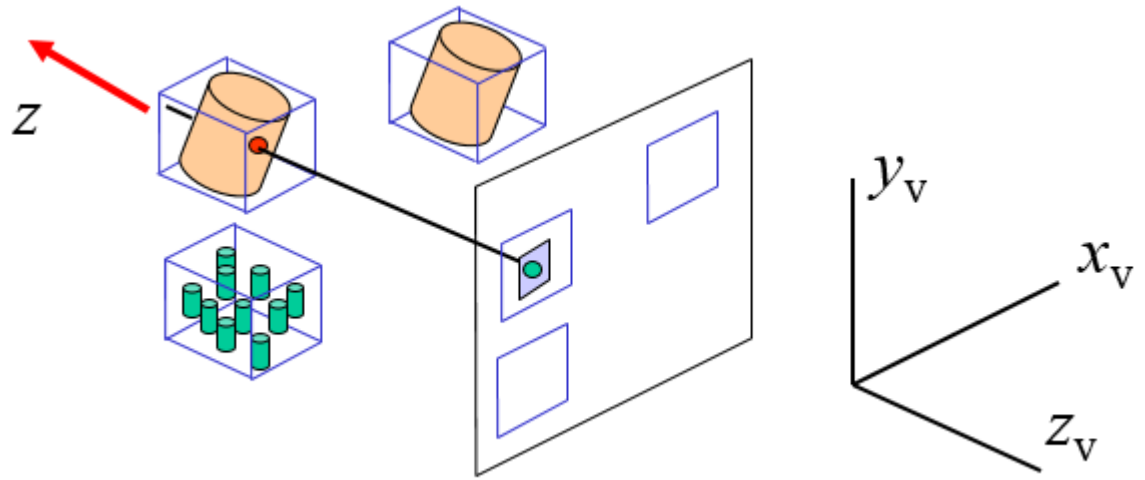
Ray-casting Algorithm 2

```
Var  fbuf: array[N,N] of colour;          { frame-buffer      }
     n : integer;                         { #intersections    }
     z : array[MaxIntsec] of real;        { intersections     }
     p : array[MaxIntsec] of object;      { corresp. objects  }
For all 1<= i, j <=N do  { for alle pixels }
   For all objects do
      Calculate intersections and add these to z and p,
      keeping z and p sorted;
    if n > 1 then fbuf[i,j] := surfacecolor(p[1], z[1]);
```

Ray-casting Algorithm 3

Acceleration intersection calculations:

Use (hierarchical) bounding boxes



Ray-casting algorithm 4

+ Relatively easy to implement
+ For some objects very suitable (for instance spheres and other quadratic surfaces)
+ Transparency can be dealt with easily

- Objects must be known in advance
- Sloooow: ~ #objects*pixels, little coherence
+ Special case of *ray-tracing*

Comparison

- Hardware available? Use depth-buffer, possibly in combination with back-face elimination or depth-sort for part of scene.
- If not, choose dependent on complexity scene and type of objects:
  - Simple scene, few objects: depth-sort
  - Quadratic surfaces: ray-casting
  - Otherwise: depth-buffer
- Many additional methods to boost performance (kD-trees, scene decomposition, etc.)