# COMPUTER GRAPHICS

# UNIT-V

# VISIBLE SURFACE DETECTION METHOD

**R.MANIMEGALAI**

DEPARTMENT OF COMPUTER SCIENCE

PERIYAR GOVT ARTS COLLEGE

CUDDALORE.

# Visible Surface Detection

## Visible Surface Detection

- Visible surface detection or hidden surface removal.
- Realistic scenes: closer objects occludes the others.
- Classification:
  - Object space methods
  - Image space methods

## Object Space Methods

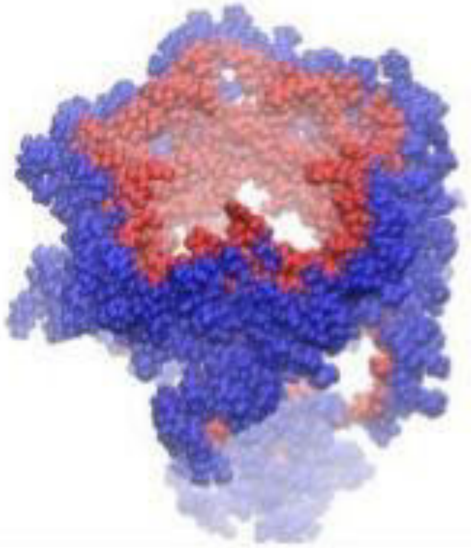
- Algorithms to determine which parts of the shapes are to be rendered in 3D coordinates.
- Methods based on comparison of objects for their 3D positions and dimensions with respect to a viewing position.
- For *N* objects, may require *N*N* comparision operations.
- Efficient for small number of objects but difficult to implement.
- Depth sorting, area subdivision methods.

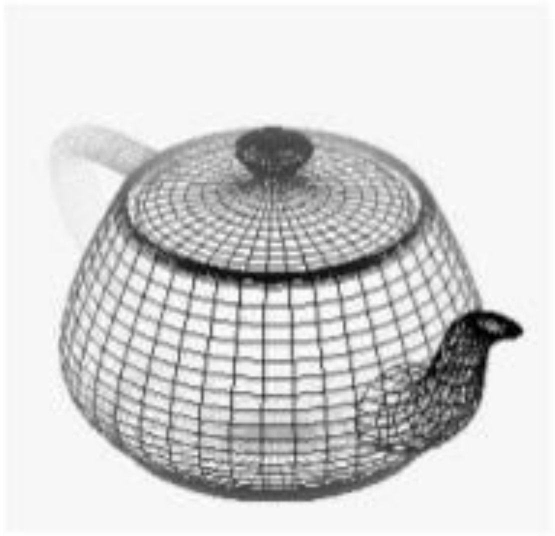
## Image Space Methods

- Based on the pixels to be drawn on 2D. Try to determine which object should contribute to that pixel.
- Running time complexity is the number of pixels times number of objects.
- Space complexity is two times the number of pixels:
  - One array of pixels for the frame buffer
  - One array of pixels for the depth buffer
- Coherence properties of surfaces can be used.
- Depth-buffer and ray casting methods.

## Depth Cueing

- Hidden surfaces are not removed but displayed with different effects such as intensity, color, or shadow for giving hint for third dimension of the object.
- Simplest solution: use different colors-intensities based on the dimensions of the shapes.





## Back-Face Detection

- Back-face detection of 3D polygon surface is easy
- Recall the polygon surface equation:

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

- We need to also consider the viewing direction when determining whether a surface is back-face or front-face.
- The normal of the surface is given by:

$$\mathbf{N} = (A, B, C)$$

## Back-Face Detection

- A polygon surface is a back face if:

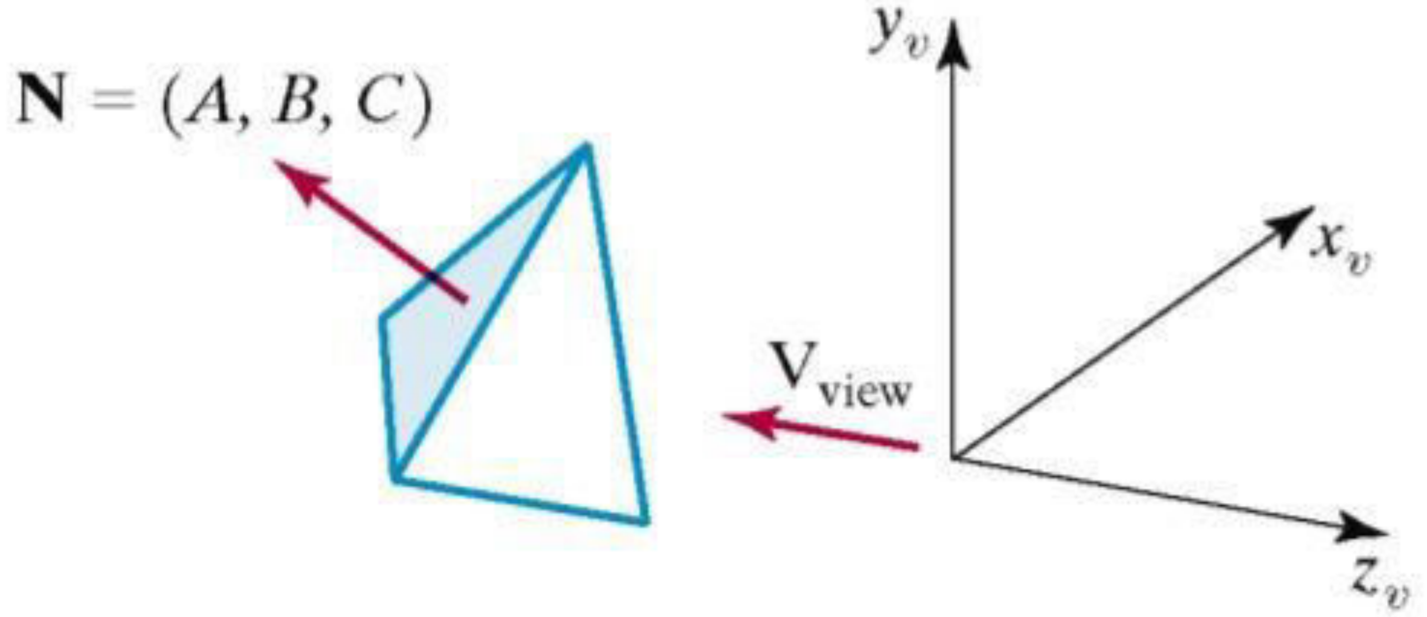
$$\mathbf{V}_{\text{view}} \cdot \mathbf{N} > 0$$

- However, remember that after application of the viewing transformation we are looking down the negative *z*-axis. Therefore a polygon is a back face if:

$$(0,0,-1) \cdot \mathbf{N} > 0$$

$$\text{or if } C < 0$$

## Back-Face Detection



- We will also be unable to see surfaces with $C$=0. Therefore, we can identify a polygon surface as a back-face if:

$$C \leq 0$$

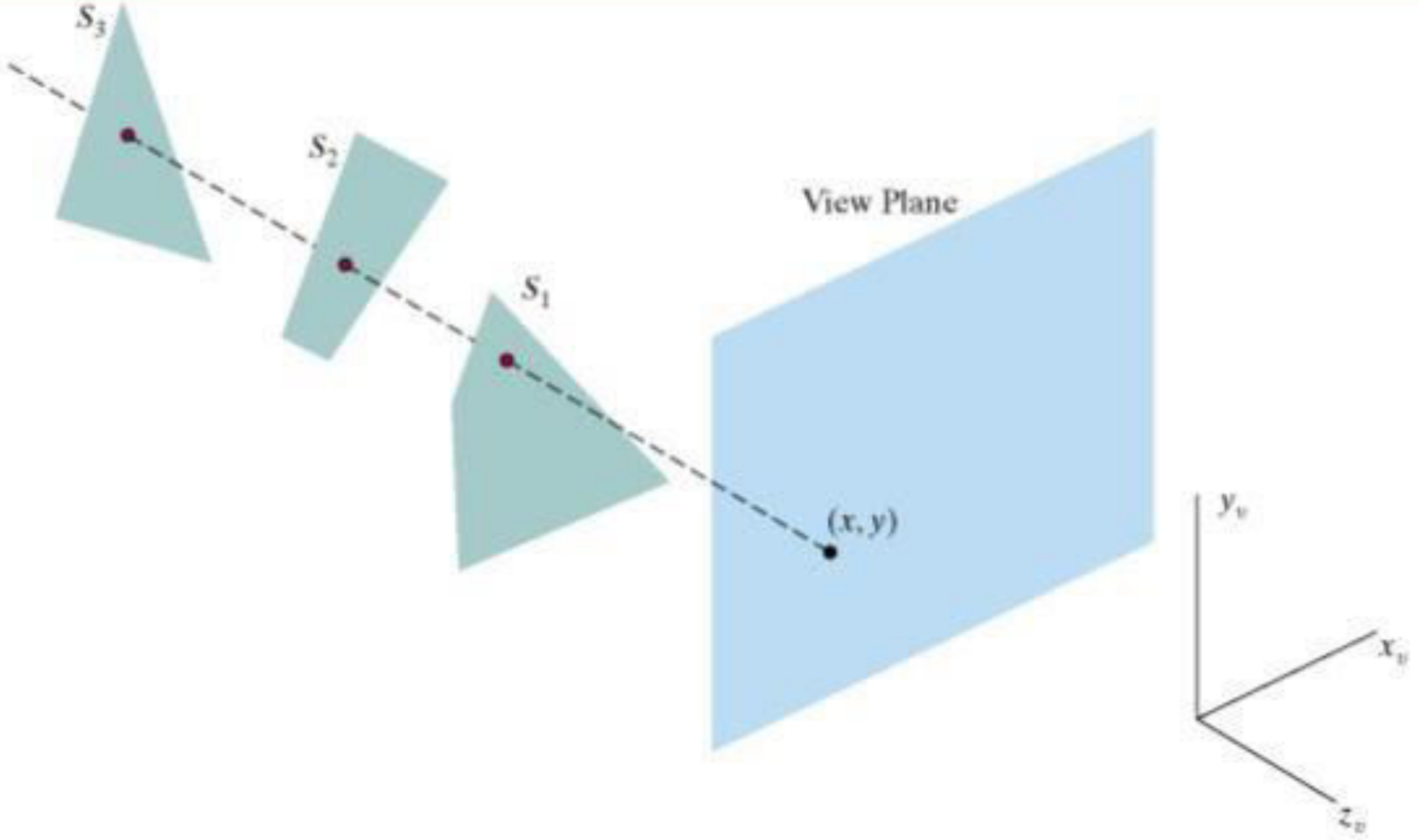
## Back-Face Detection

- Back-face detection can identify all the hidden surfaces in a scene that contain non-overlapping convex polyhedra.
- But we have to apply more tests that contain overlapping objects along the line of sight to determine which objects obscure which objects.

## Depth-Buffer Method

- Also known as *z*-buffer method.
- It is an image space approach
  - Each surface is processed separately one pixel position at a time across the surface
  - The depth values for a pixel are compared and the closest (smallest z) surface determines the color to be displayed in the frame buffer.
  - Applied very efficiently on polygon surfaces
  - Surfaces are processed in any order

# Depth-Buffer Method



## Depth-Buffer Method

- Two buffers are used
  - Frame Buffer
  - Depth Buffer
- The z-coordinates (depth values) are usually normalized to the range [0,1]

## Depth-Buffer Algorithm

- Initialize the depth buffer and frame buffer so that for all buffer positions $(x,y)$,

  depthBuff (x,y) = 1.0,  frameBuff (x,y) =bgColor

- Process each polygon in a scene, one at a time
  - For each projected $(x,y)$ pixel position of a polygon, calculate the depth $z$.
  - If $z$ < depthBuff (x,y), compute the surface color at that position and set

  depthBuff (x,y) = z,  frameBuff (x,y) = surfCol (x,y)

## Calculating depth values efficiently

- We know the depth values at the vertices. How can we calculate the depth at any other point on the surface of the polygon.
- Using the polygon surface equation:

$$z = \frac{-Ax - By - D}{C}$$

## Calculating depth values efficiently

- For any scan line adjacent horizontal *x* positions or vertical *y* positions differ by 1 unit.
- The depth value of the next position (x+1,y) on the scan line can be obtained using

$$z' = \frac{-A(x+1) - By - D}{C}$$

$$= z - \frac{A}{C}$$

## Calculating depth values efficiently

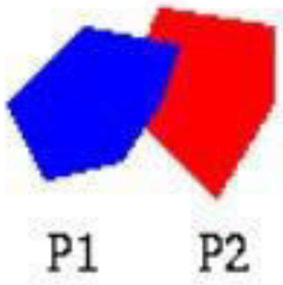
- For adjacent scan-lines we can compute the x value using the slope of the projected line and the previous x value.

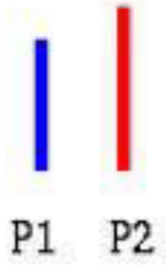
$$x' = x - \frac{1}{m}$$

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

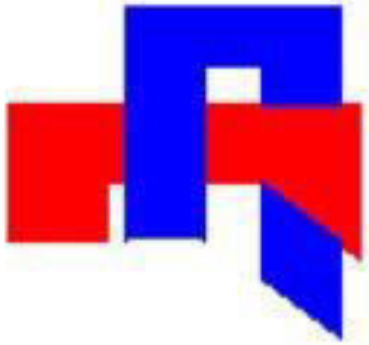
## Depth-Buffer Method

- Is able to handle cases such as





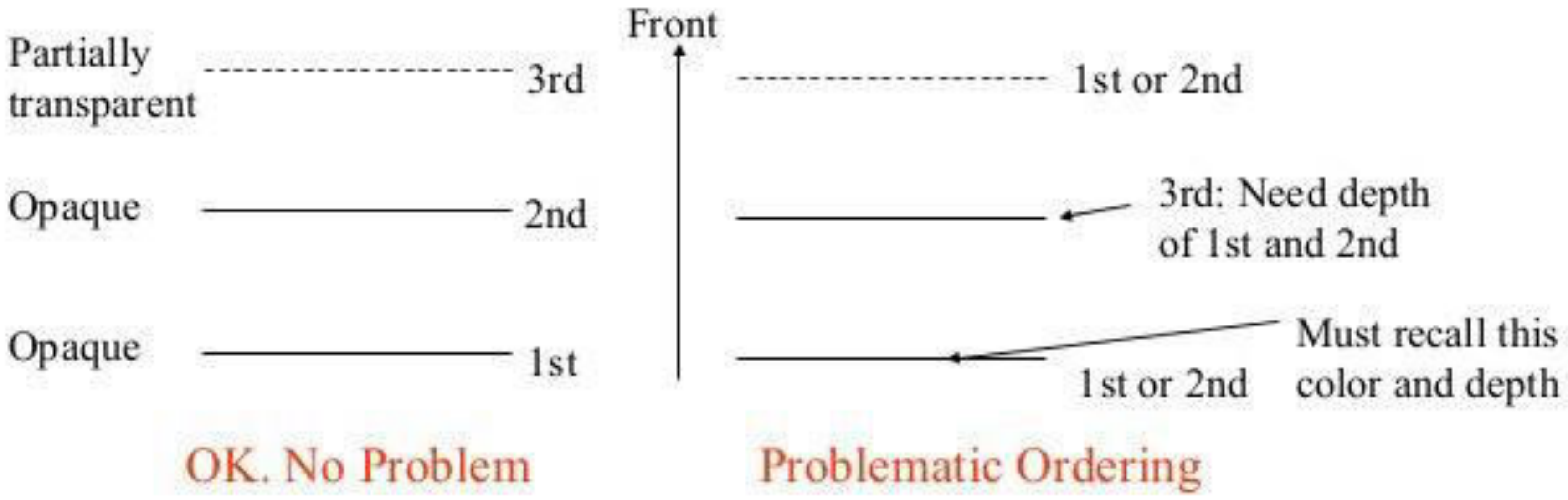
View from the Right-side



These polygons are both in front of and behind one another.

## Z-Buffer and Transparency

- We may want to render transparent surfaces (alpha $\neq 1$) with a z-buffer
- However, we must render in back to front order
- Otherwise, we would have to store at least the first opaque polygon behind transparent one

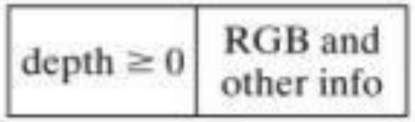


## A-Buffer Method

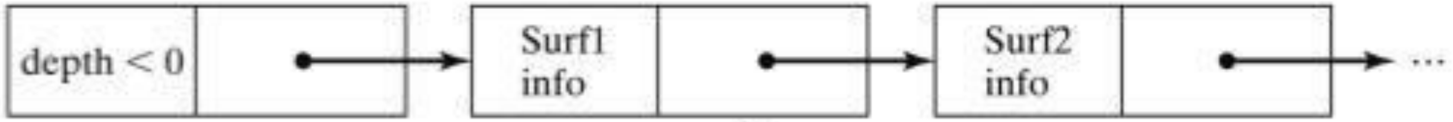
- Extends the depth-buffer algorithm so that each position in the buffer can reference a linked list of surfaces.
- More memory is required
- However, we can correctly compose different surface colors and handle transparent surfaces.

## A-Buffer Method

- Each position in the A-buffer has two fields:
  - a depth field
  - surface data field which can be either surface data or a pointer to a linked list of surfaces that contribute to that pixel position



(a)



(b)

## 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

## Tables for The Various Surfaces

- 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

## Active List & Flag

- 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

## Depth Buffer Method

- In computer graphics, **z-buffering**, also known as **depth buffering**, is the management of image depth coordinates in 3D graphics, usually done in hardware, sometimes in software. It is one solution to the visibility problem, which is the problem of deciding which elements of a rendered scene are visible, and which are hidden.

- This method is developed by Cutmull.
- It is an image-space approach. The basic idea is to test the Z-depth of each surface to determine the closest (visible) surface.
- In this method each surface is processed separately one pixel position at a time across the surface. The depth values for a pixel are compared and the closest (smallest z) surface determines the colour to be displayed in the frame buffer.
- It is applied very efficiently on surfaces of polygon. Surfaces can be processed in any order. To override the closer polygons from the far ones, two buffers named **frame buffer** and **depth buffer**, are used.

- Depth Values for a surface position(x , y) are calculated from the plane equation for each surface by:

$$z = \frac{(-Ax-By-D)}{C}$$

- Depth Values across the edge are calculated by:

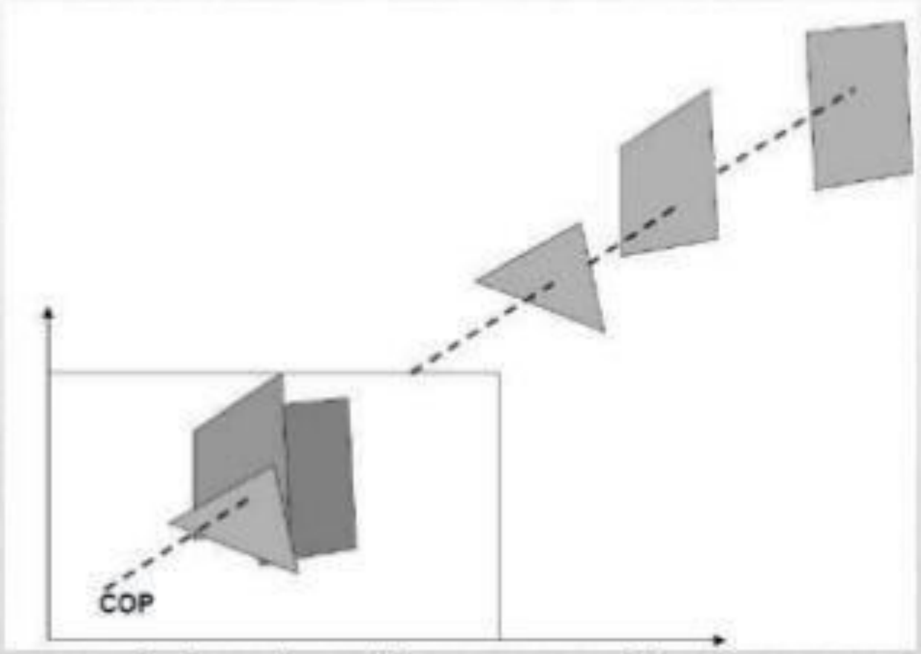
$$z` = \frac{(-A(x+1)-By-D)}{C}$$

- Depth Values down the edge are recursively calculated by:

$$z` = z + \frac{(A/m)+B}{C}$$

- **Depth buffer** is used to store depth values for (x, y) position, as surfaces are processed ($0 \leq \text{depth} \leq 1$).
- The **frame buffer** is used to store the intensity value of colour value at each position (x, y).

The z-coordinates are usually normalized to the range [0, 1]. The 0 value for z-coordinate indicates back clipping pane and 1 value for z-coordinates indicates front clipping pane.



## Algorithm:

**Step-1** – Set the buffer values –

Depthbuffer (x, y) = 0

Framebuffer (x, y) = background colour

**Step-2** – Process each polygon (One at a time)

For each projected (x, y) pixel position of a polygon, calculate depth z.

If Z > depthbuffer (x, y)

Compute surface color,

set depthbuffer (x, y) = z,

framebuffer (x, y) = surfacecolor (x, y)

## Advantages

- It is easy to implement.
- It reduces the speed problem if implemented in hardware.
- It processes one object at a time.
- Accurate performance.

## Disadvantages

- It requires large memory.
- It is time consuming process.

## Uses:

- The Z-buffer is a technology used in almost all contemporary computers, laptops and mobile phones for performing 3D graphics, for example for computer games. The Z-buffer is implemented as hardware in the silicon ICs (integrated circuits) within these computers. The Z-buffer is also used (implemented as software as opposed to hardware) for producing computer-generated special effects for films.
- Furthermore, Z-buffer data obtained from rendering a surface from a light's point-of-view permits the creation of shadows by the shadow mapping technique.

## A-Buffer Method

Prerequisite : depth-buffer (or Z Buffer) method

A-Buffer method in computer graphics is a general hidden face detection mechanism suited to medium scale virtual memory computers. This method is also known as anti-aliased or area-averaged or accumulation buffer. This method extends the algorithm of depth-buffer (or Z Buffer) method. As the depth buffer method can only be used for opaque object but not for transparent object, the A-buffer method provides advantage in this scenario. Although the A buffer method requires more memory, but different surface colors can be correctly composed using it. Being a descendent of the Z-buffer algorithm, each position in the buffer can reference a linked list of surfaces. The key data structure in the A buffer is the accumulation buffer.

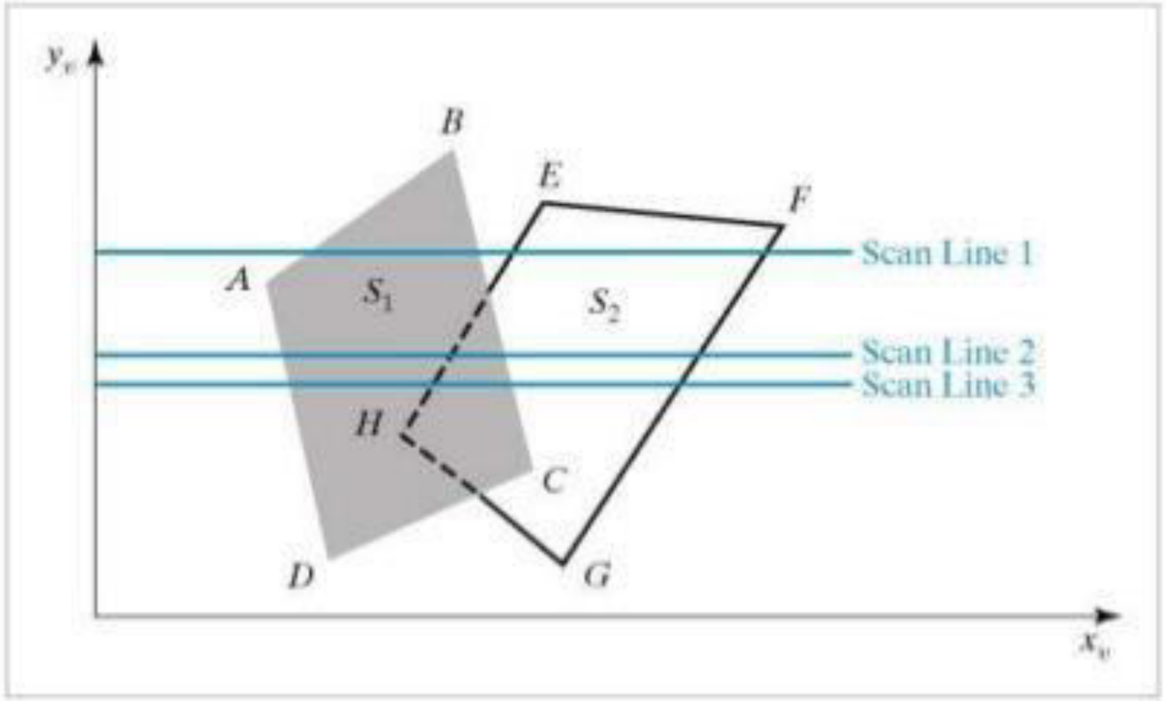
## Scan-Line Method

- It is an image-space method to identify visible surface. This method has a depth information for only single scan-line. In order to require one scan-line of depth values, we must group and process all polygons intersecting a given scan-line at the same time before processing the next scan-line. Two important tables, **edge table** and **polygon table,** are maintained for this.
- **The Edge Table** – It contains:-
  - coordinate endpoints of each line in the scene
  - the inverse slope of each line
  - and pointers into the polygon table to connect edges to surfaces.
- **The Polygon Table** – It contains:-
  - the plane coefficients
  - surface material properties
  - other surface data
  - may be pointers to the edge table.

## Scan-Line Method

- Each of the scan line is processes from left to right. The surface flag is turned on at left intersection and at right intersection it is turned off.
- In this method, as each scan line is processed, all polygon surfaces intersecting that line are examined to determine which are visible. A cross each scan line, depth calculations are made for each overlapping surface to determine which is nearest to the view plane. When the visible surface has been determined, the intensity value for that position is entered into the image buffer.
- The search for the surfaces that cross a given scan line can be facilitated by an active list of edges. Only the edges that cross the scan line is stored by the active list. For indicating whether the position along a scan line is inside or outside the surface, a flag is set.
- Each of the scan line is processes from left to right. The surface flag is turned on at left intersection and at right intersection it is turned off.

## Scan-Line Method



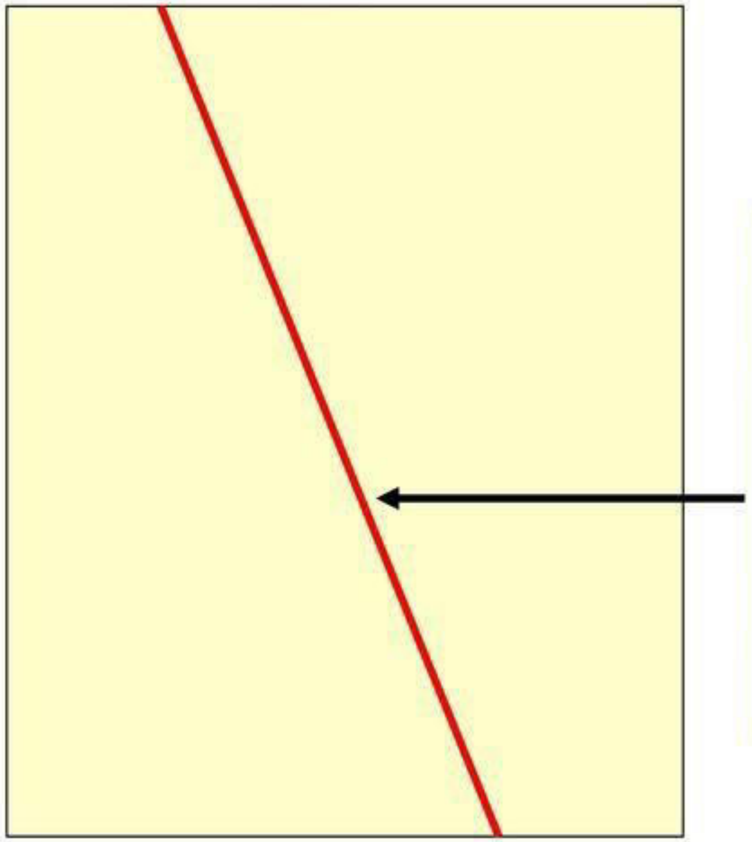
Scan lines corssing the projection of two furfaces S1, S2 in the view plane. Dashed lines indicate the boundaries of hidden surfaces.

## BSP Trees

- Binary space partitioning trees.
- Used to store a collection of objects in n-dimensional space.
- Tree recursively divides n-dimensional space using (n-1)-dimensional hyperplanes.

## Space Partitioning

n-dimensional space



splitting hyperplane

(n-1)-dimensional

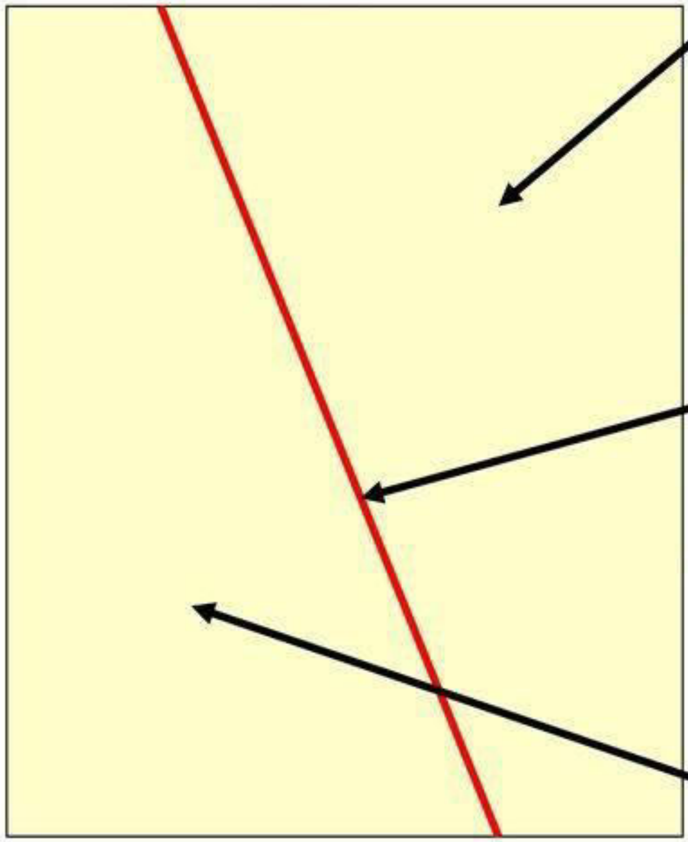
$a_1x_1 + a_2x_2 + \ldots a_nx_n + a_{n+1} = 0$

$ax + by + c = 0$ (2D)

$ax+by+cz+d = 0$ (3D)

# Space Partitioning

n-dimensional space



+ve half space

$ax + by + c > 0$

$ax+by+cz+d > 0$

coincident

$ax + by + c = 0$

$ax+by+cz+d = 0$

-ve half space

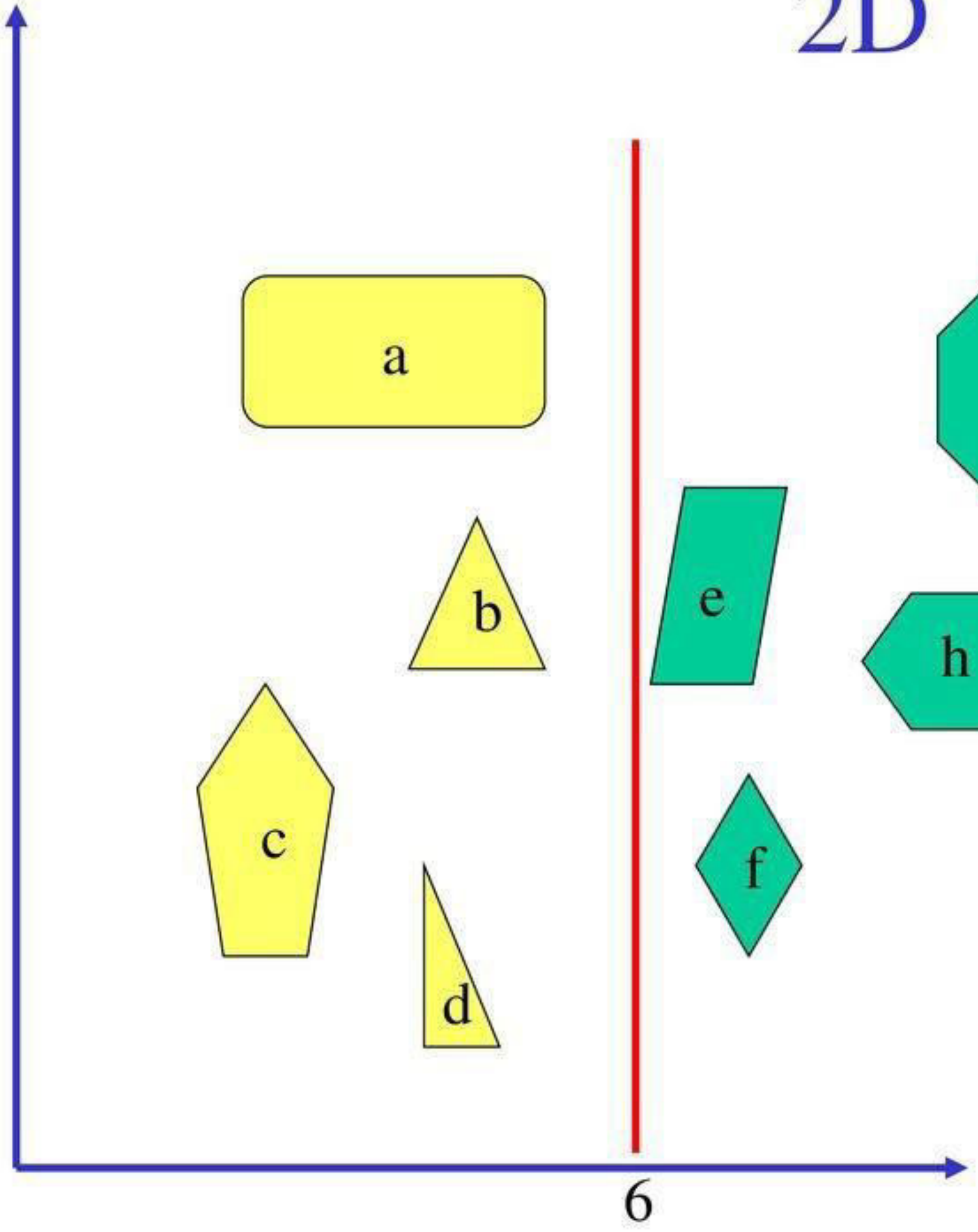
$ax + by + c < 0$

$ax+by+cz+d < 0$

## Classifying Object *z*

- In 2D, *ph* is the line $ax + by + c = 0$.
  - Compute $ax + by + c$ for all vertices of *z*.
  - If all values are = 0; *z* is coincident to *ph*.
  - If all values are <= 0; *z* is left of *ph*.
  - If all values are >= 0; *z* is right of *ph*.
  - Otherwise, *z* spans *ph* and is to be split by finding intersection points with *ph*.

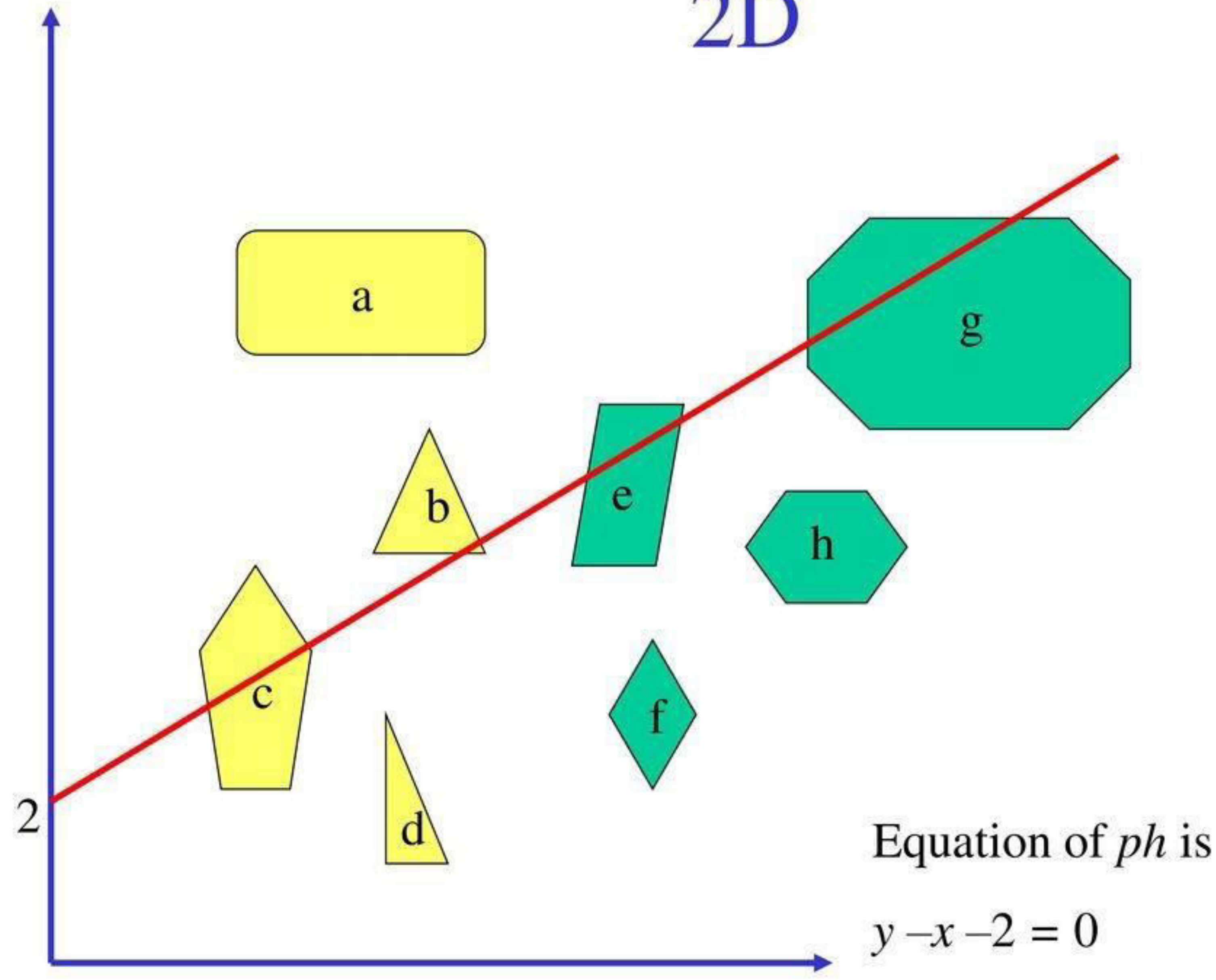
# 2D



Equation of *ph* is

$x - 6 = 0$

## 2D



Equation of *ph* is

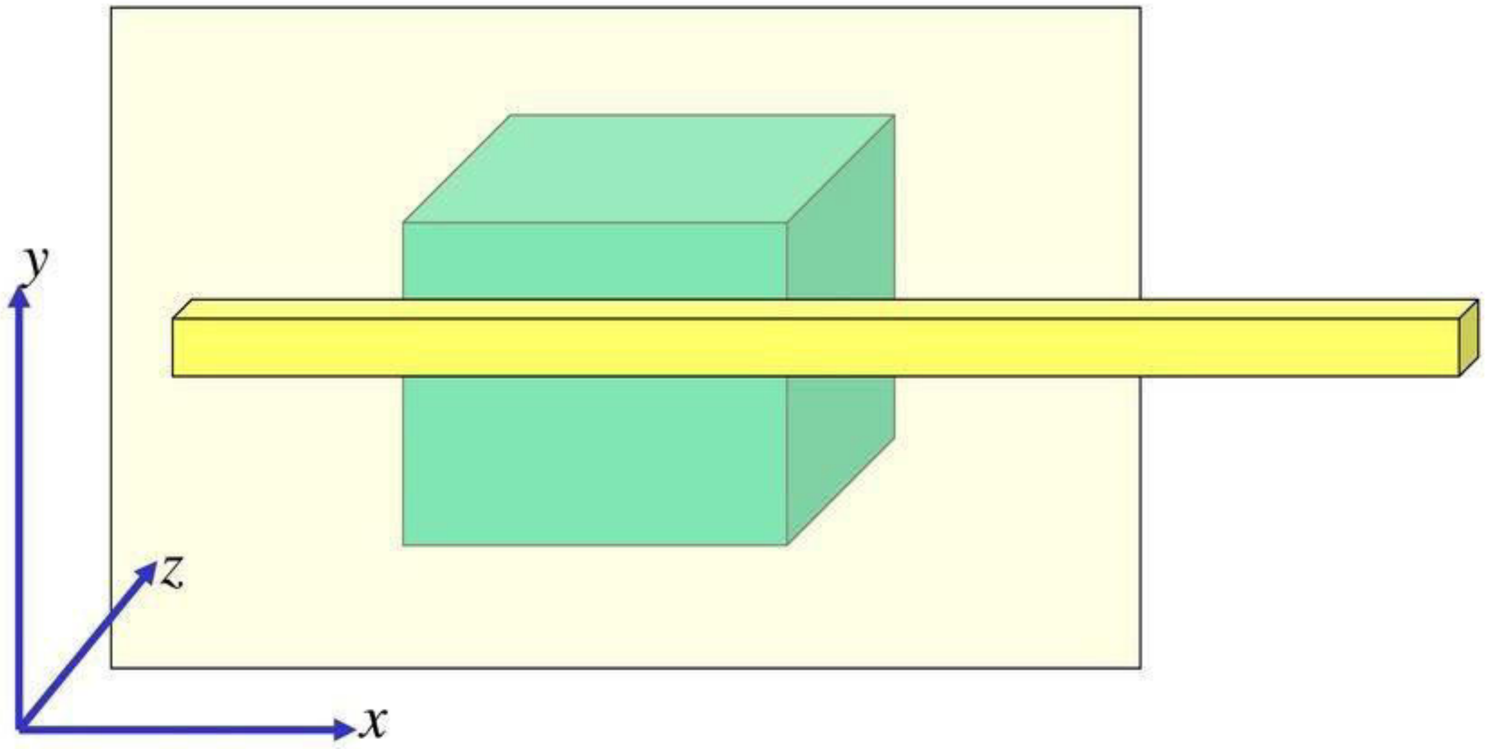
$y - x - 2 = 0$

## 3D

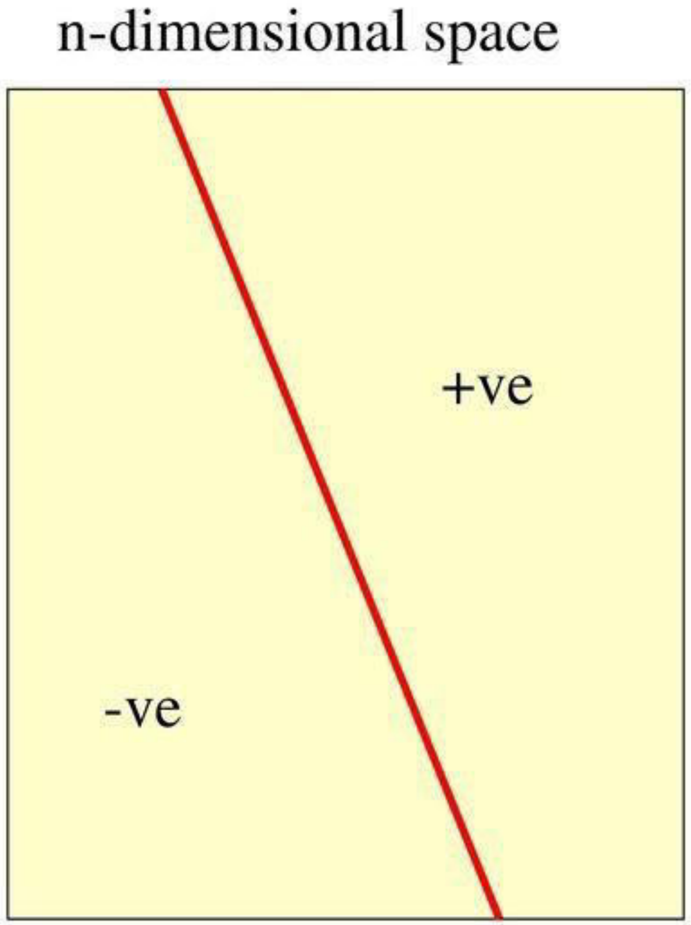
Equation of *ph* is

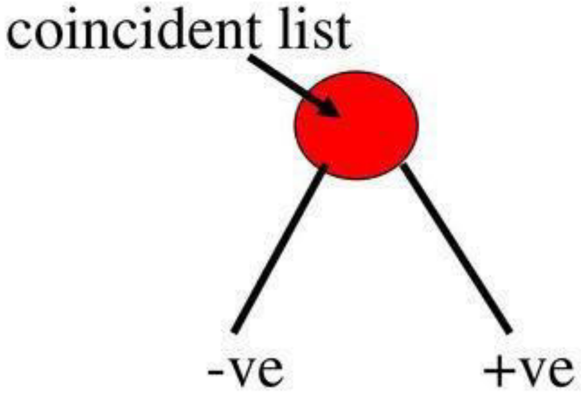
$z - 2 = 0$

General: $ax + by + cz + d = 0$

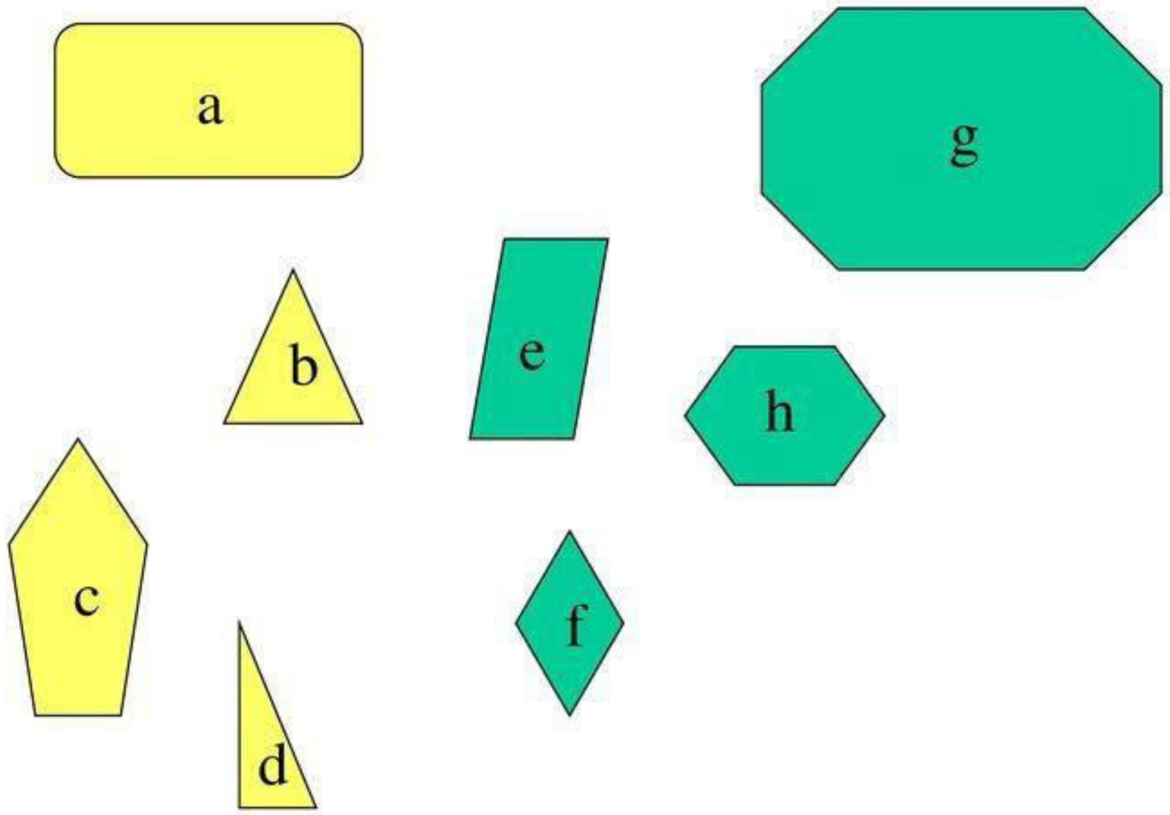


# Space Partitioning

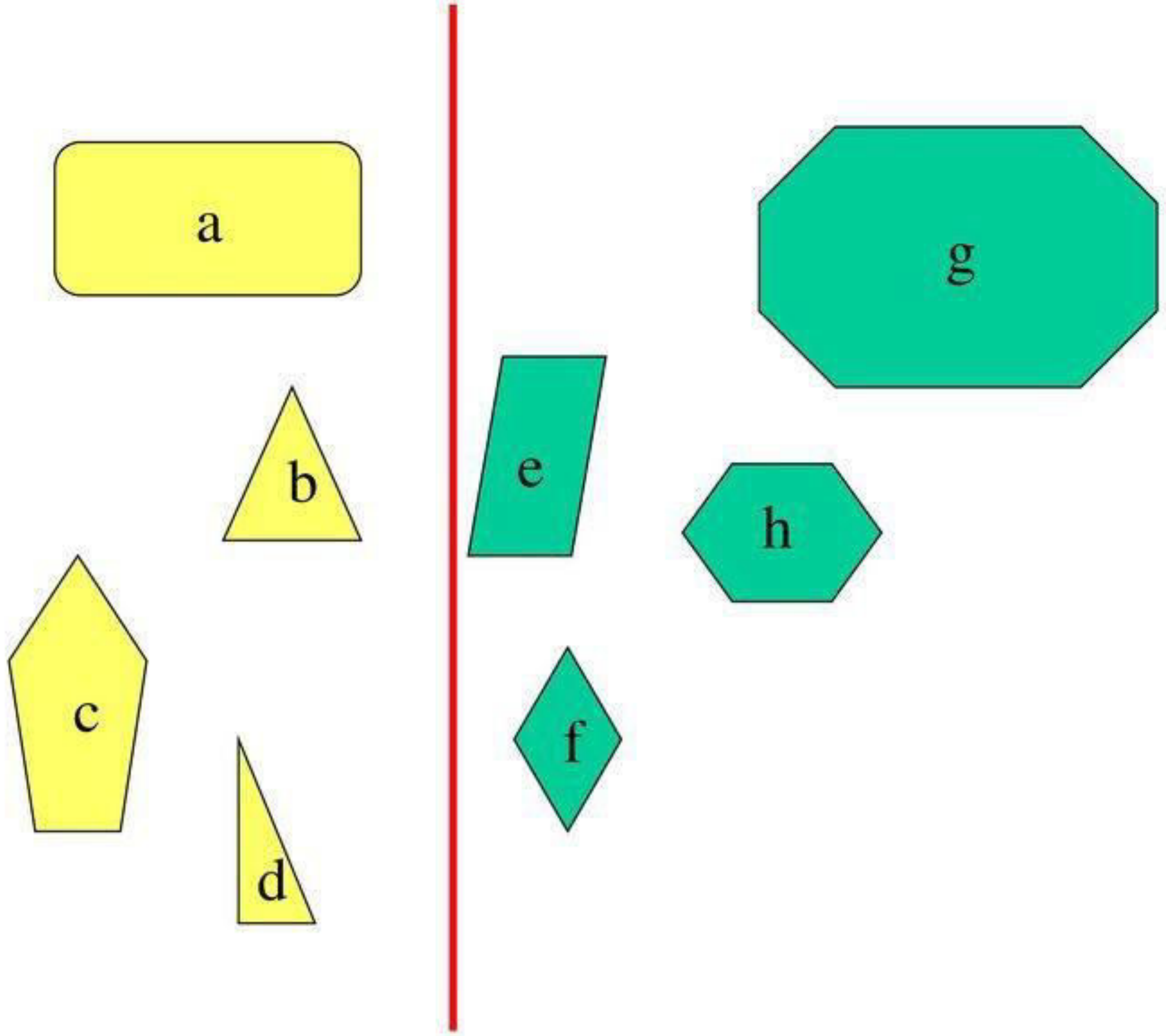




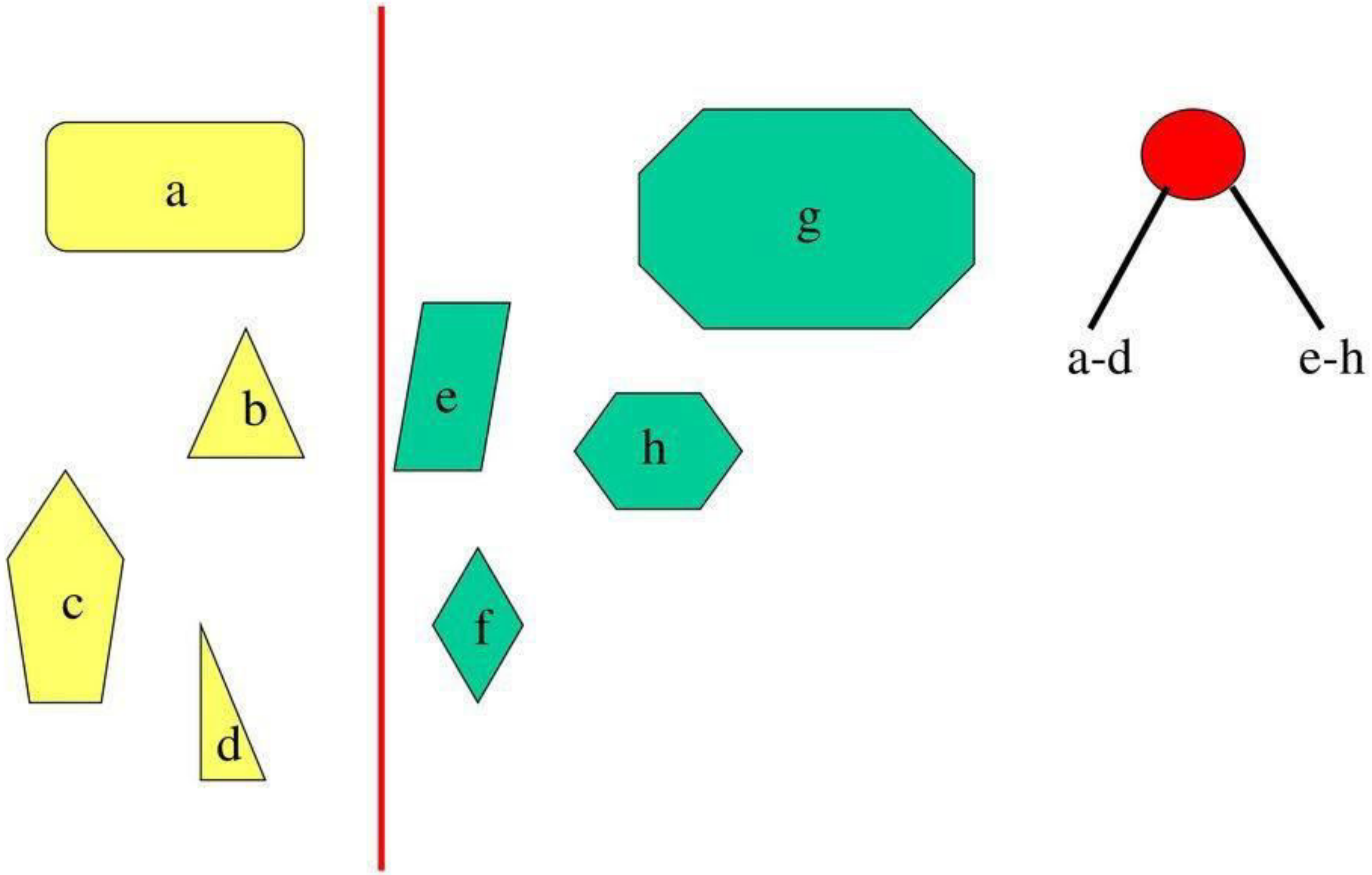
# Objects in 2D



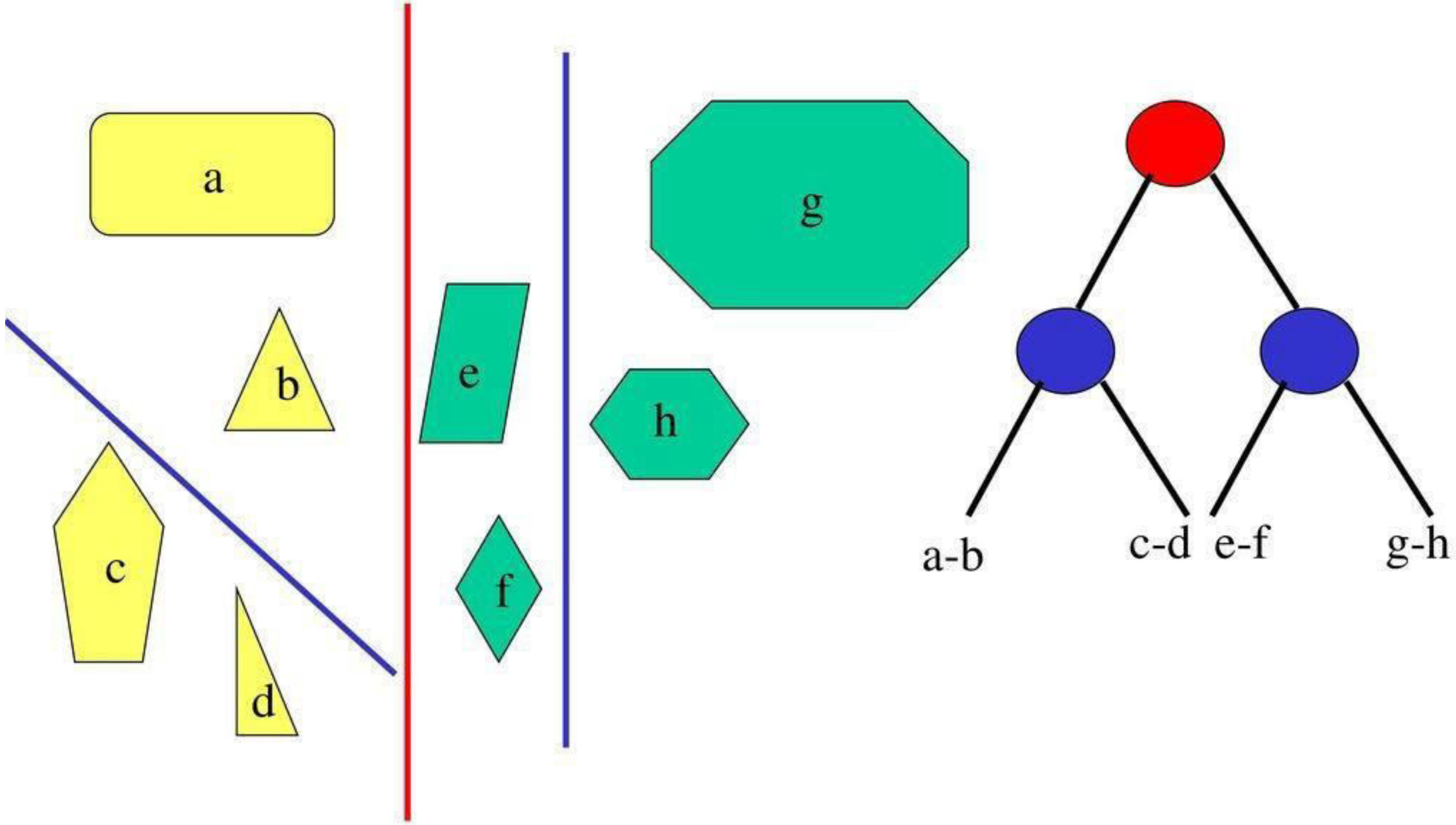
# Objects in 2D



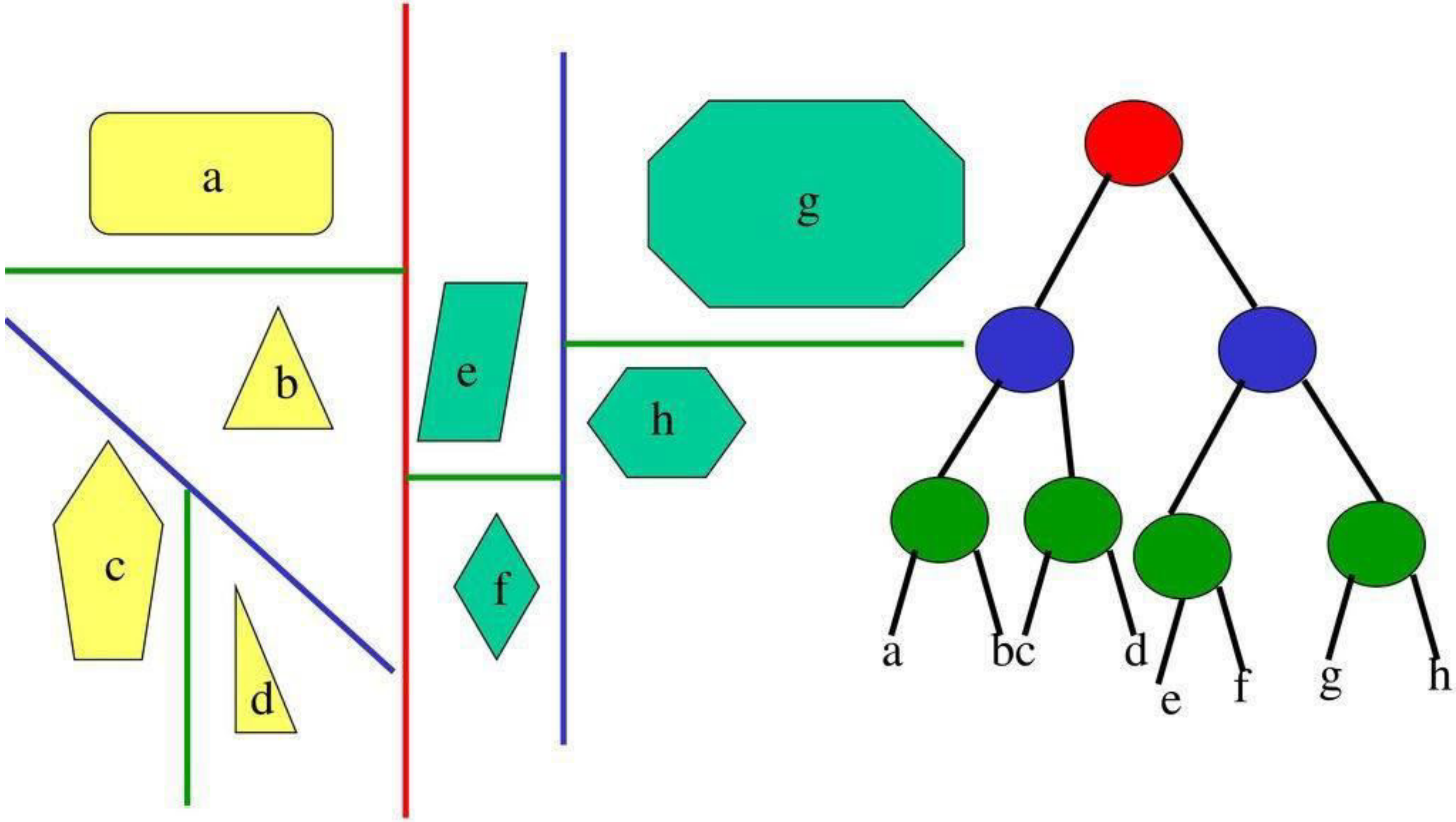
# Objects in 2D



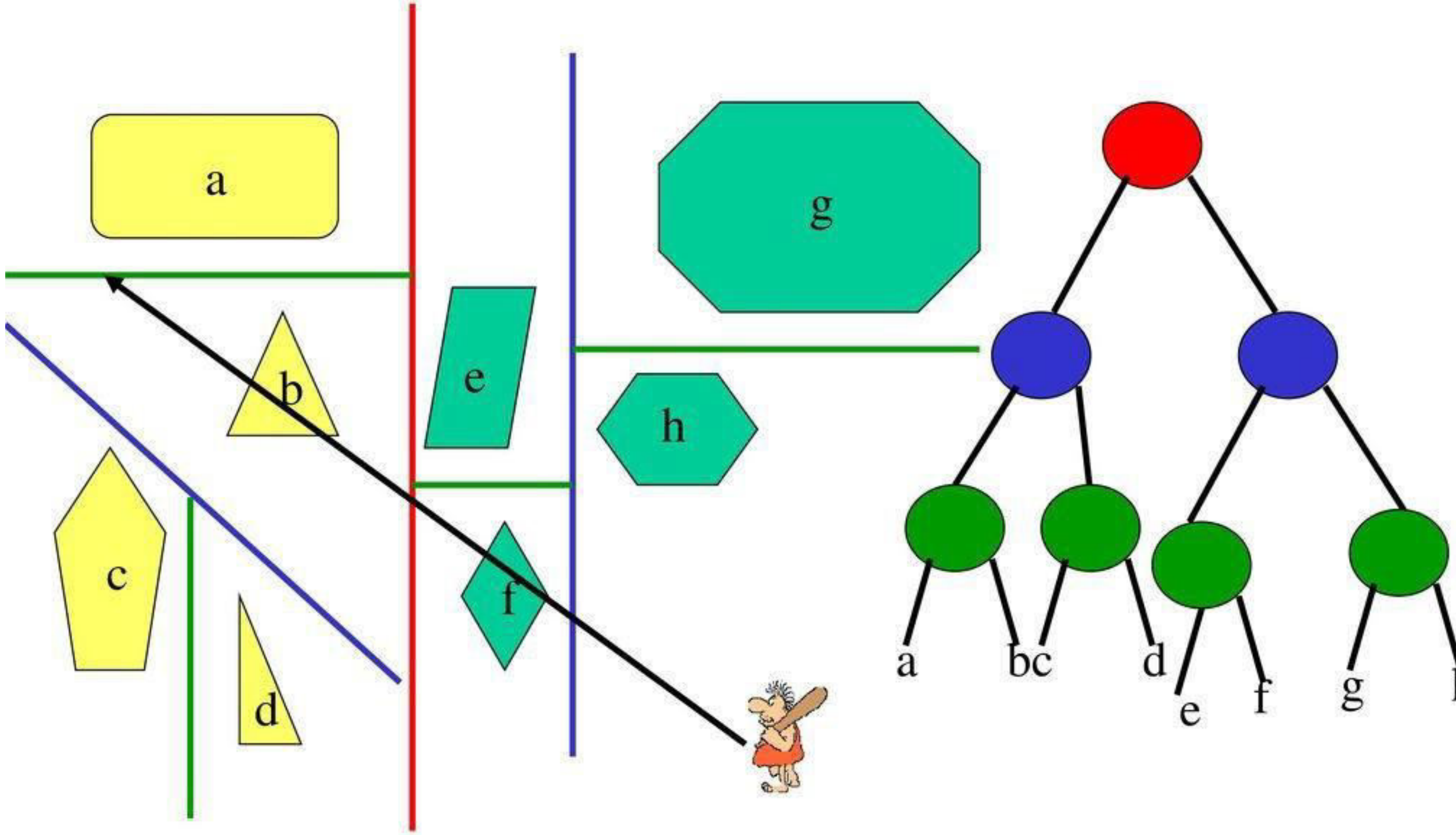
# Objects in 2D



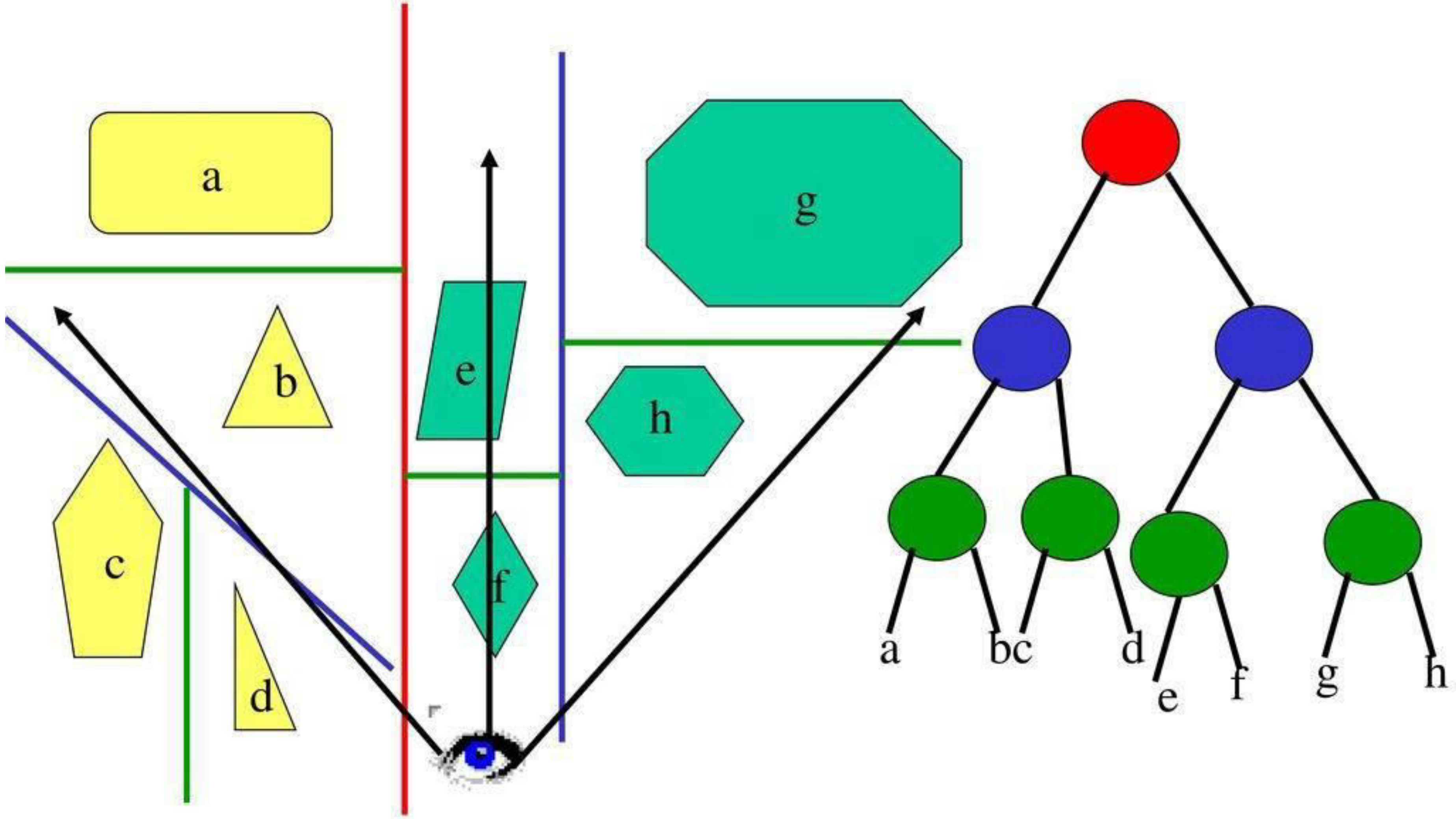
# Objects in 2D



# Collision Detection



# Visibility Ordering

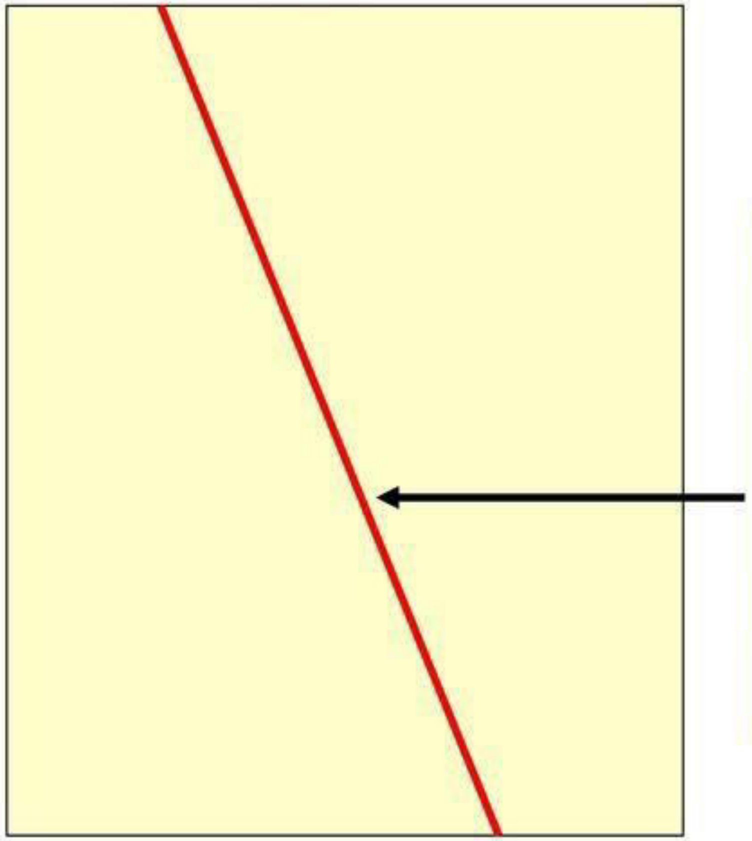


## BSP Trees

- Binary space partitioning trees.
- Used to store a collection of objects in n-dimensional space.
- Tree recursively divides n-dimensional space using (n-1)-dimensional hyperplanes.

# Space Partitioning

n-dimensional space



splitting hyperplane

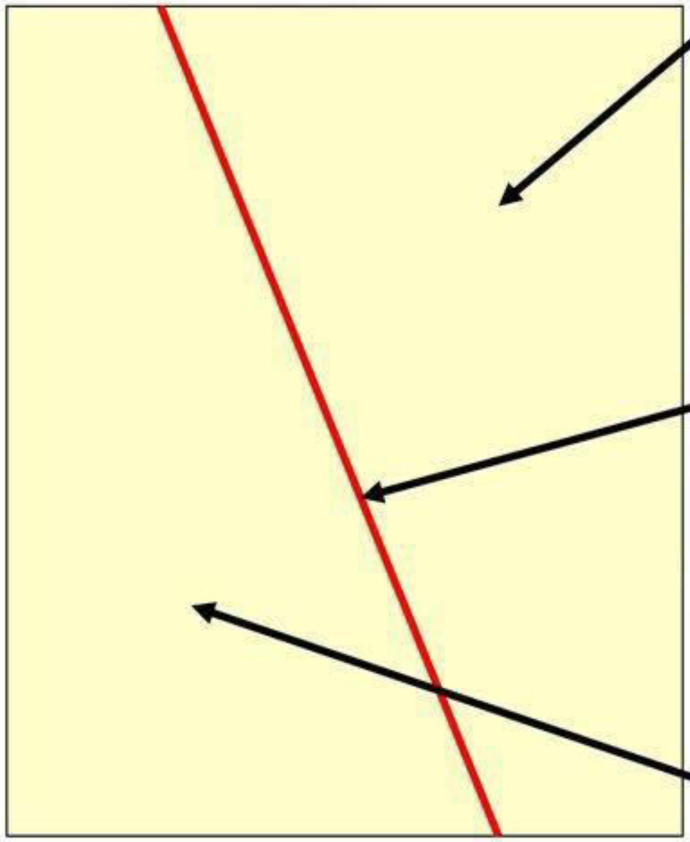
(n-1)-dimensional

$a_1x_1 + a_2x_2 + \ldots a_nx_n + a_{n+1} = 0$

$ax + by + c = 0$ (2D)

$ax+by+cz+d = 0$ (3D)

# Space Partitioning



+ve half space

$ax + by + c > 0$

$ax+by+cz+d > 0$

coincident

$ax + by + c = 0$

$ax+by+cz+d = 0$

-ve half space

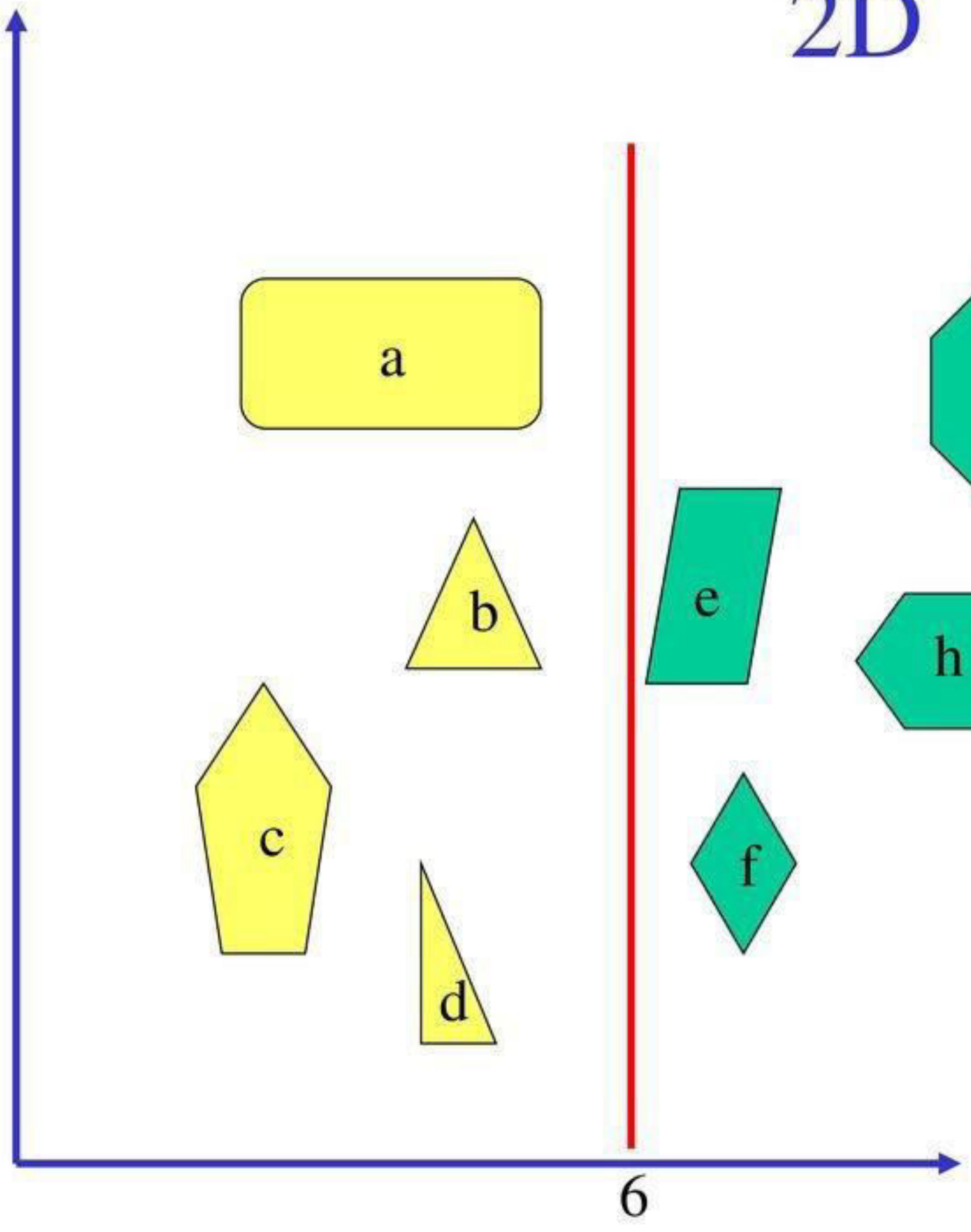
$ax + by + c < 0$

$ax+by+cz+d < 0$

## Classifying Object *z*

- In 2D, *ph* is the line $ax + by + c = 0$.
  - Compute $ax + by + c$ for all vertices of *z*.
  - If all values are $= 0$; *z* is coincident to *ph*.
  - If all values are $<= 0$; *z* is left of *ph*.
  - If all values are $>= 0$; *z* is right of *ph*.
  - Otherwise, *z* spans *ph* and is to be split by finding intersection points with *ph*.

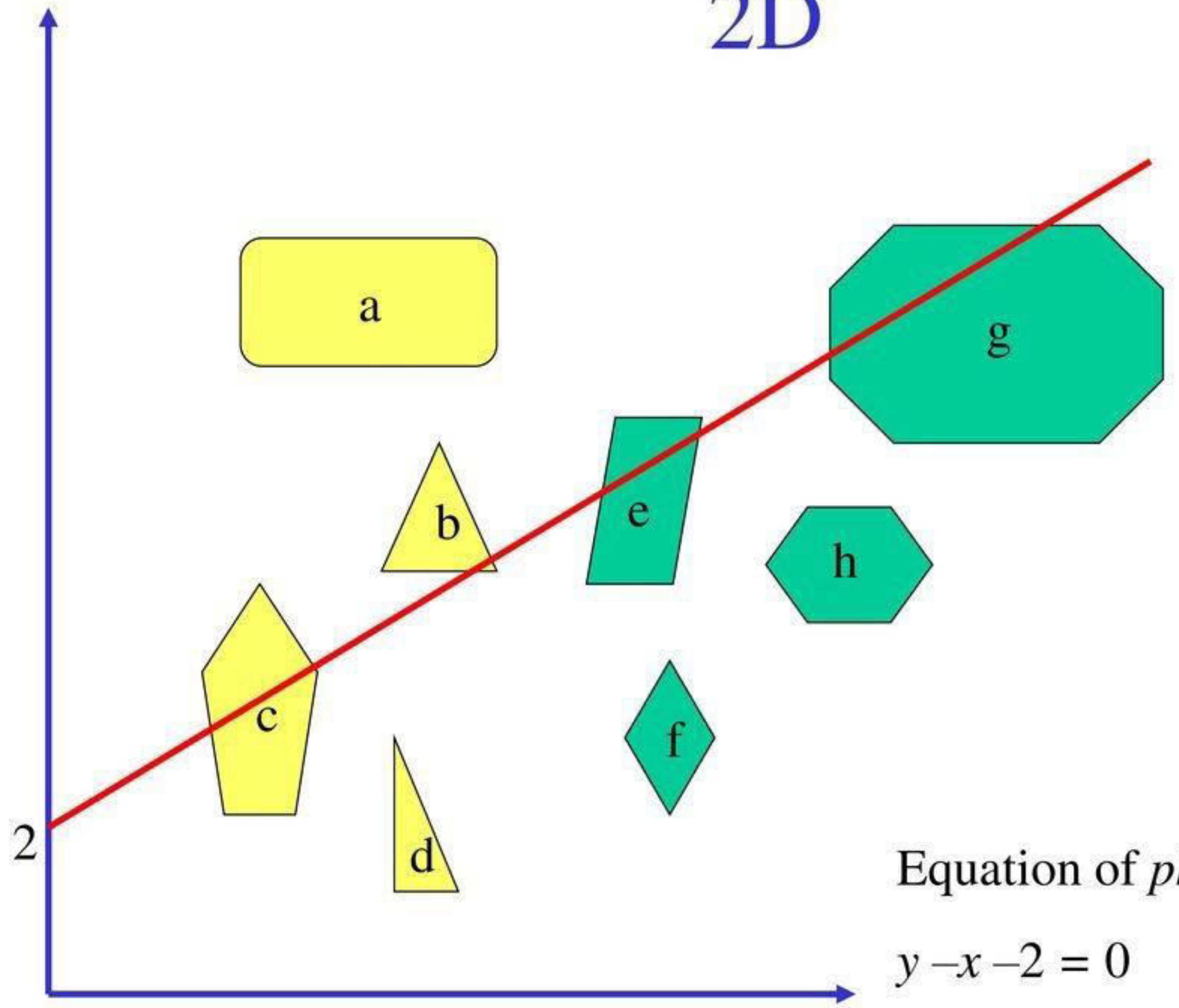
# 2D



Equation of *ph* is

$x - 6 = 0$

# 2D



Equation of *ph* is

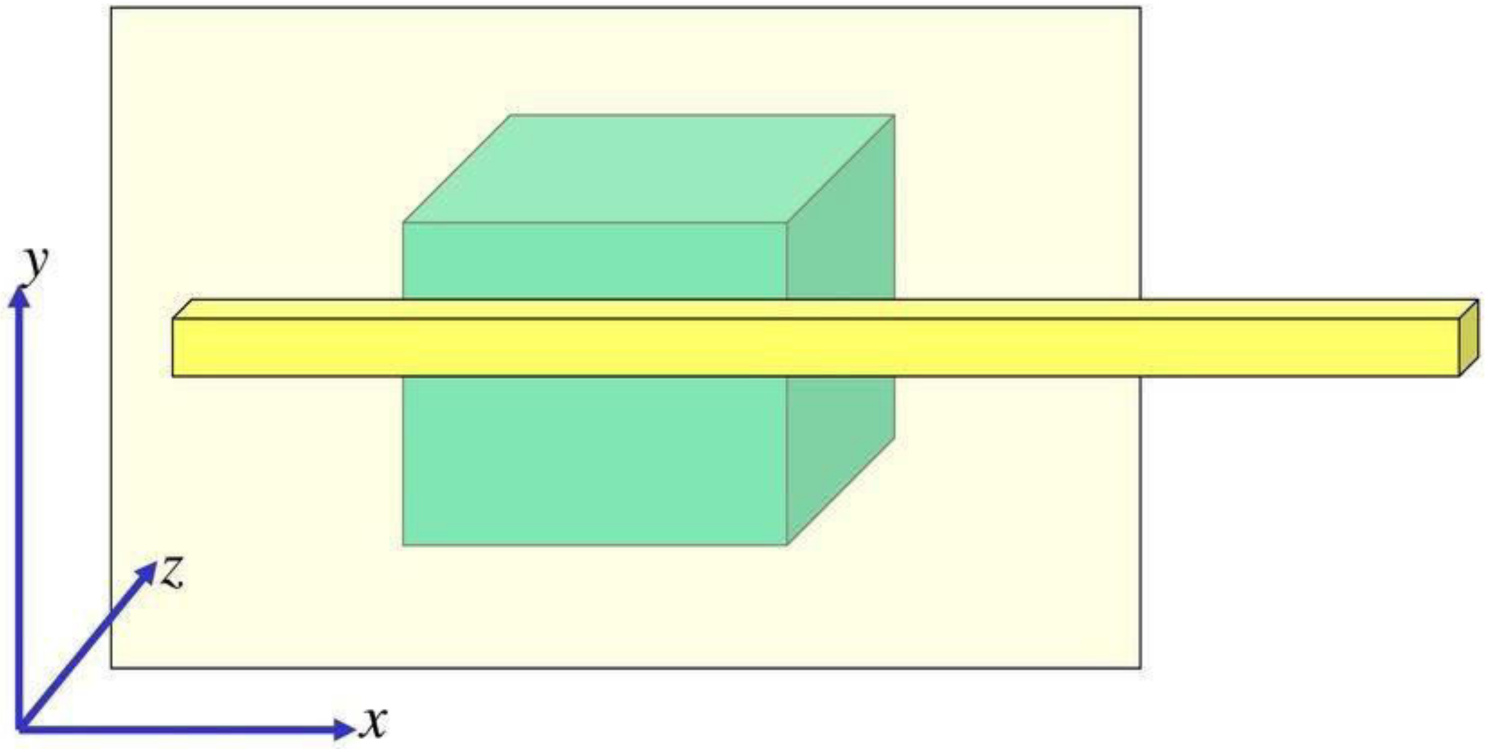
$y - x - 2 = 0$

## 3D

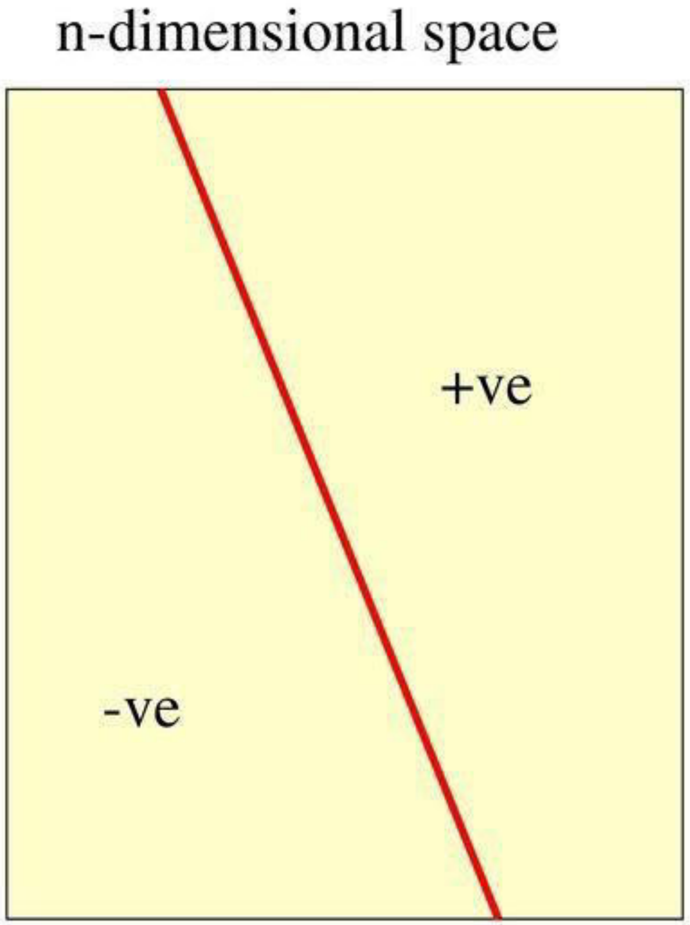
Equation of *ph* is

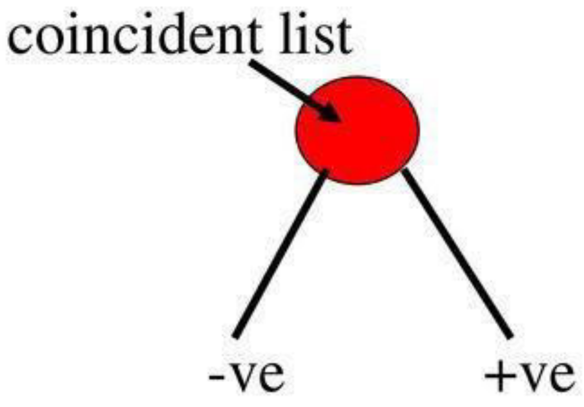
$z - 2 = 0$

General: $ax + by + cz + d = 0$

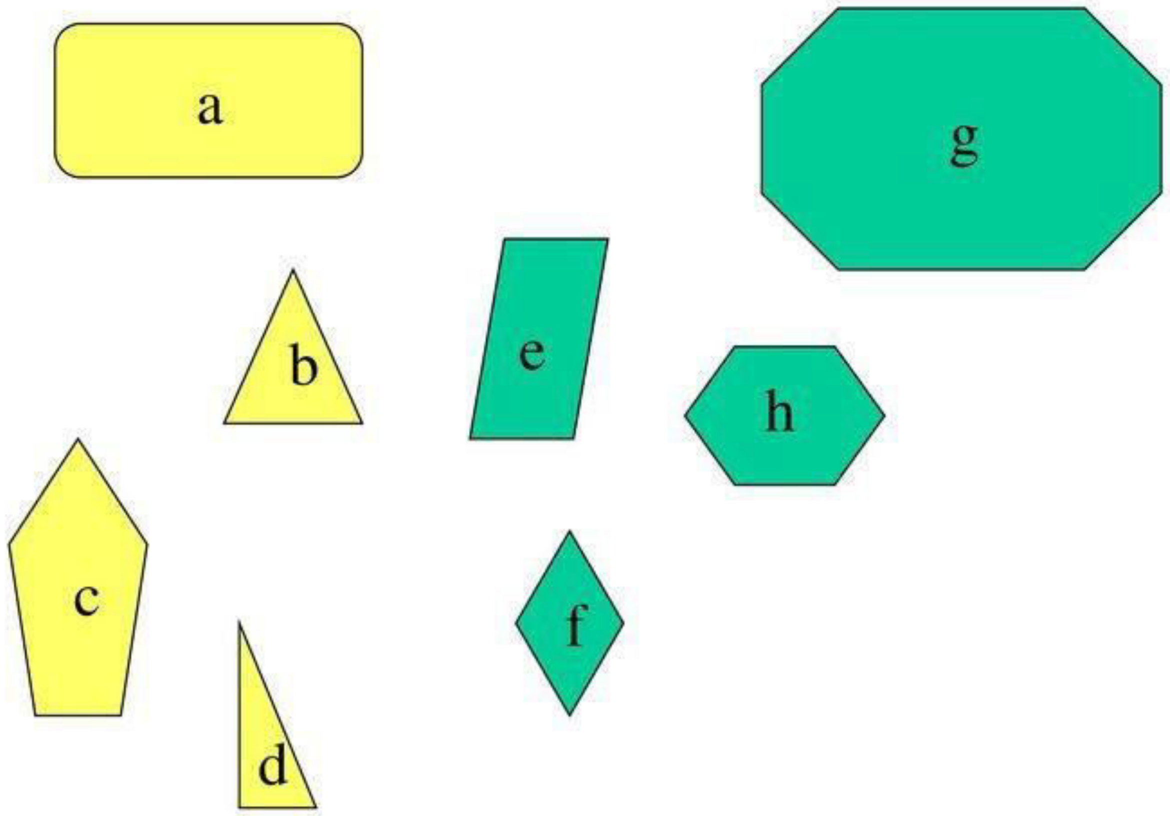


# Space Partitioning

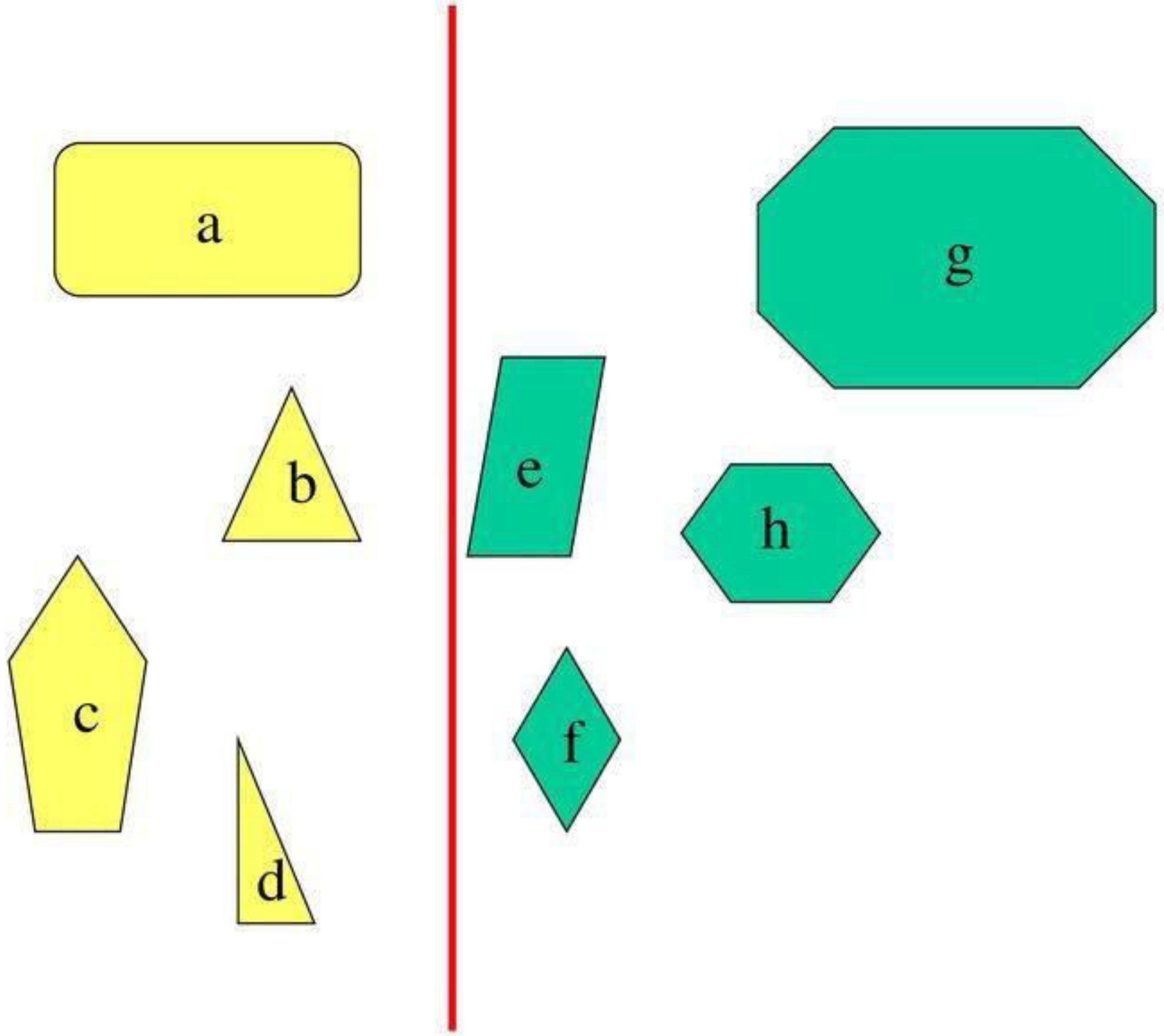




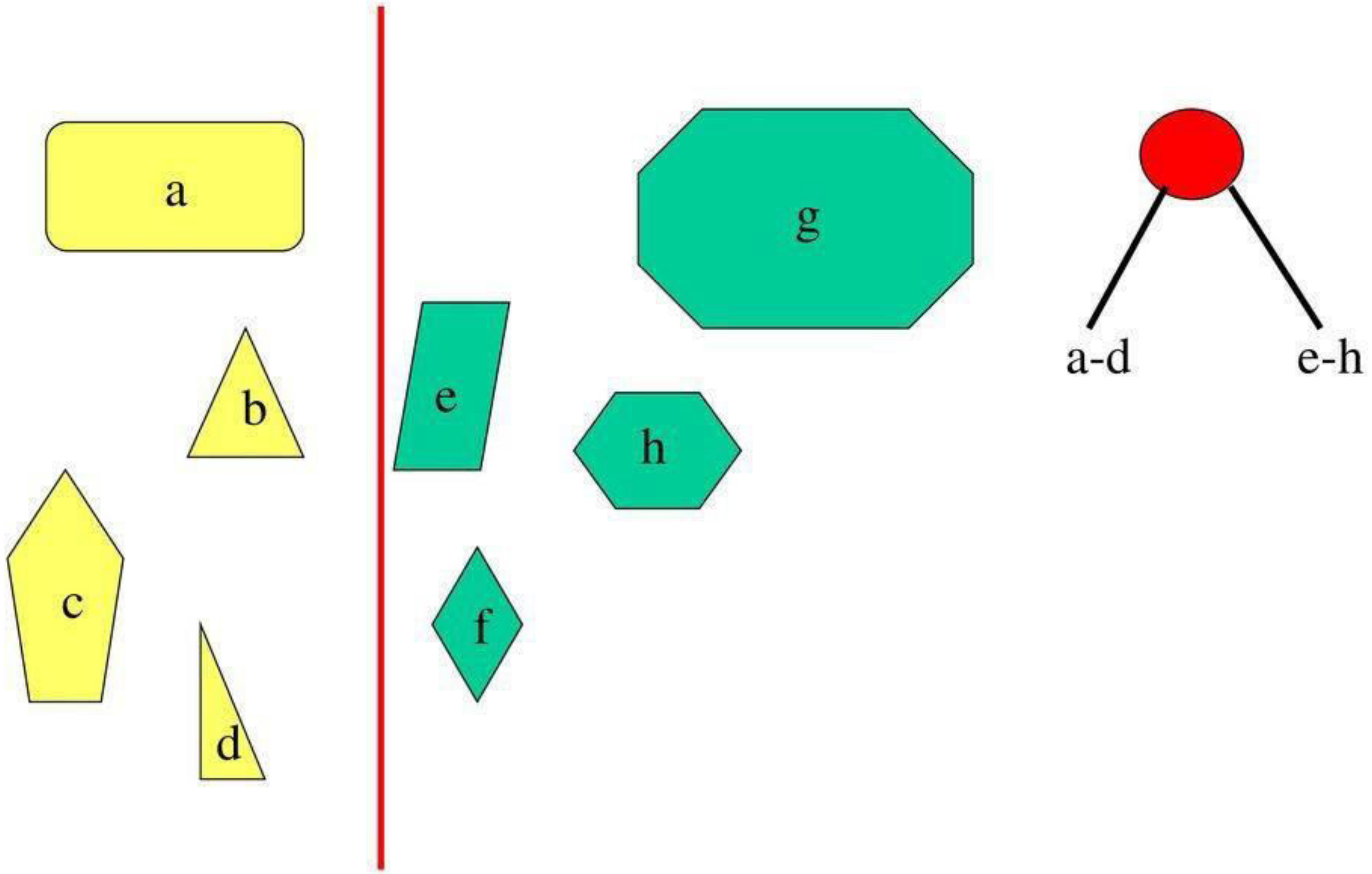
# Objects in 2D



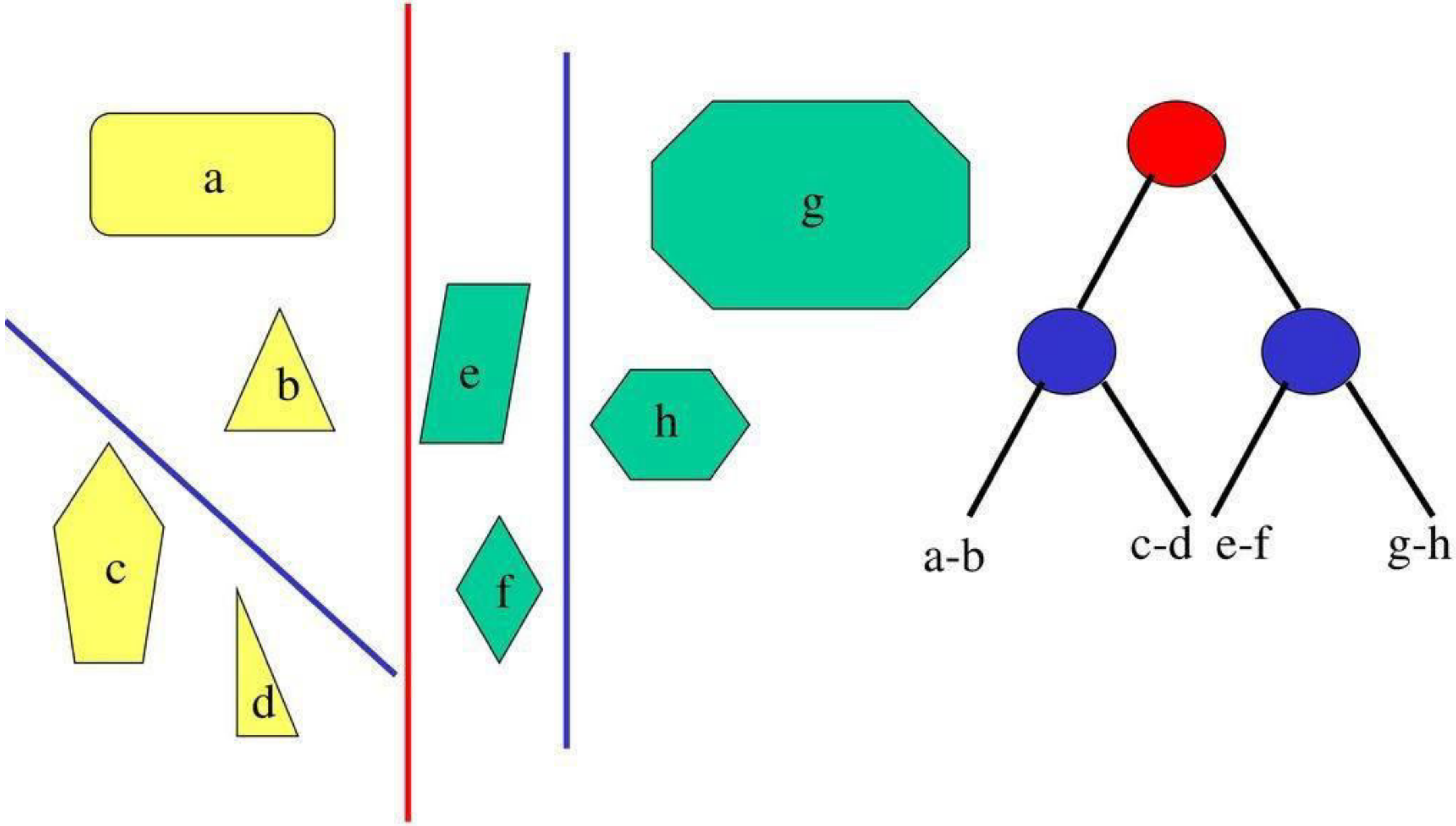
# Objects in 2D



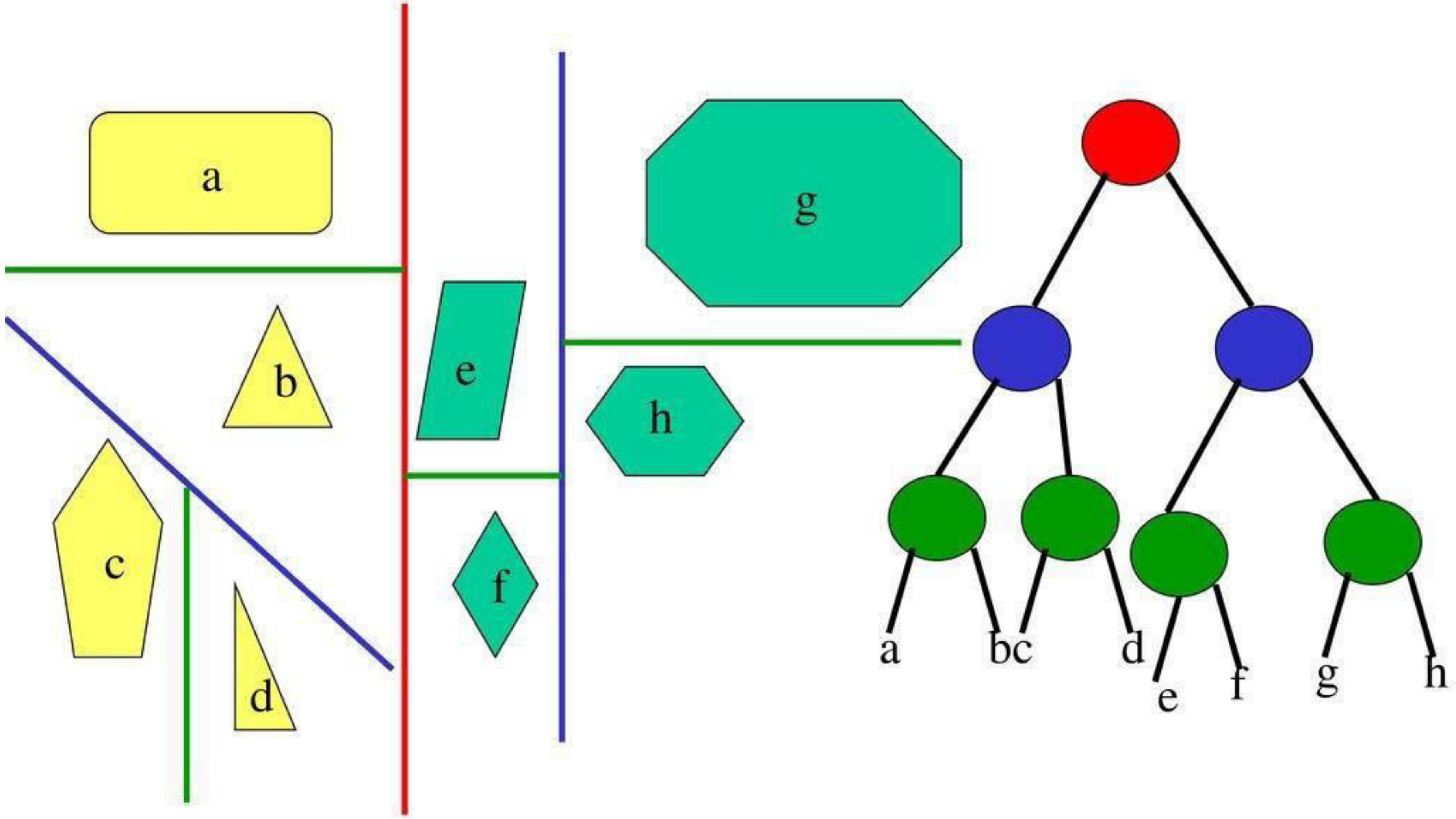
# Objects in 2D



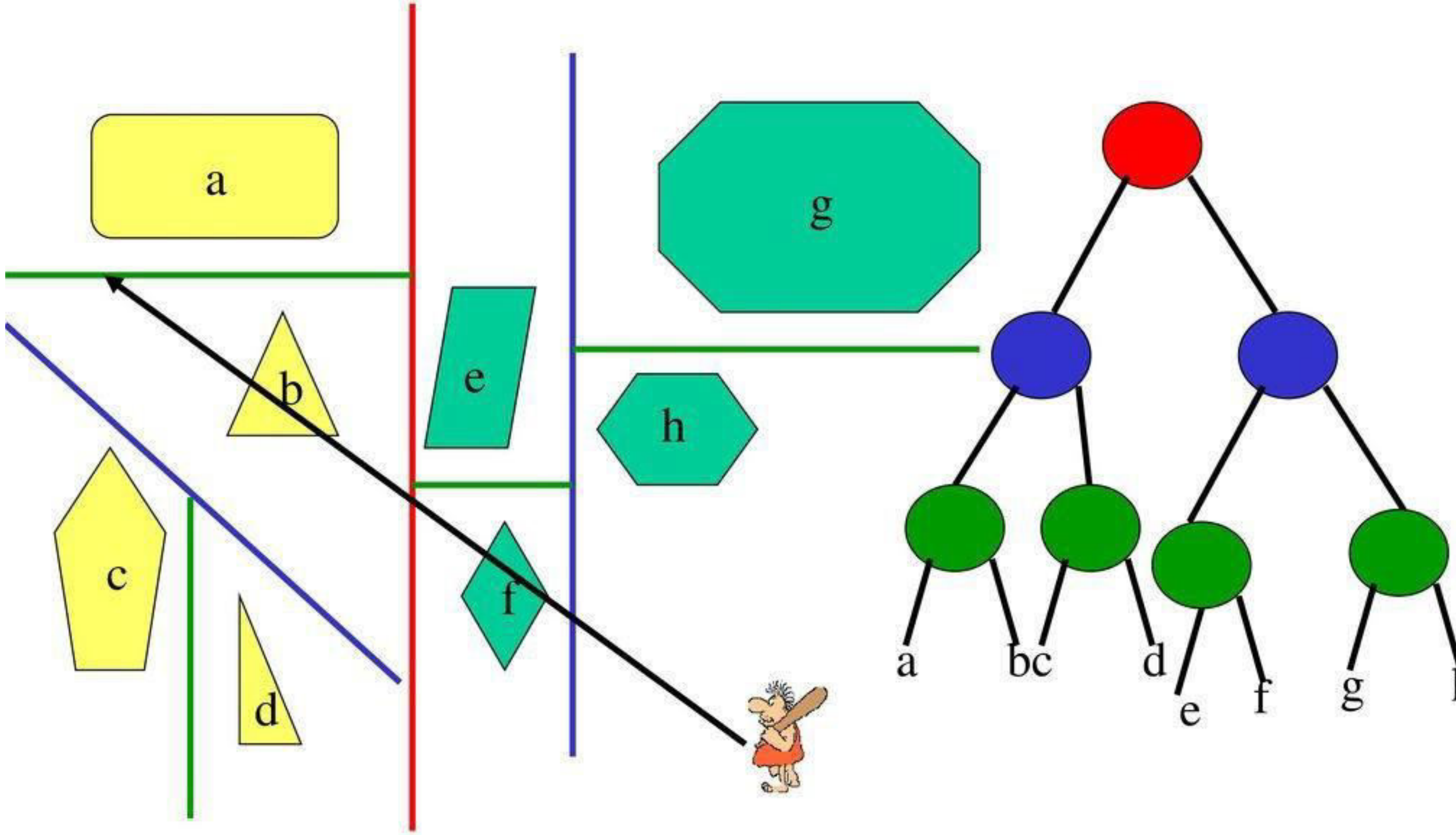
# Objects in 2D



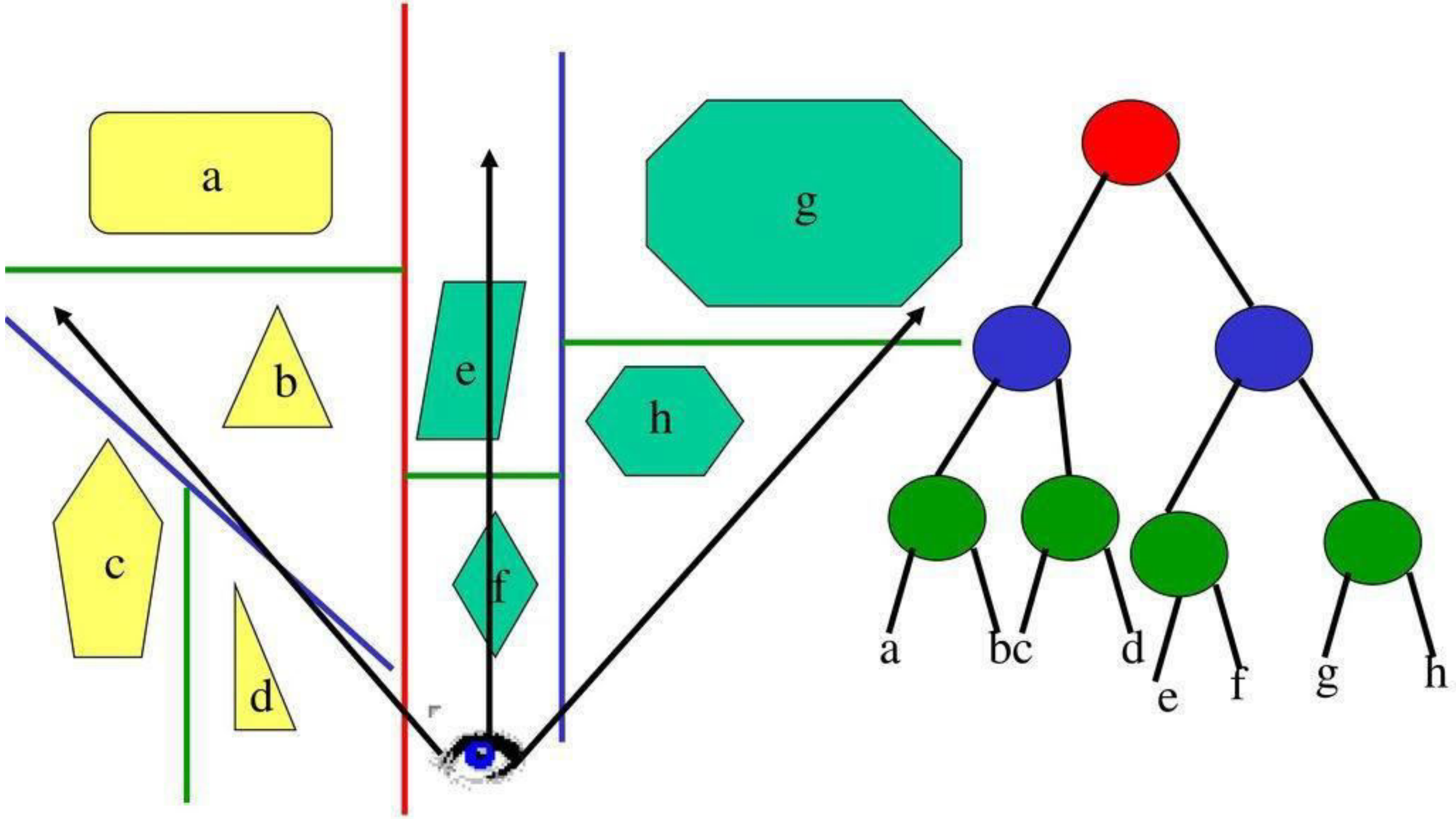
# Objects in 2D



# Collision Detection



# Visibility Ordering



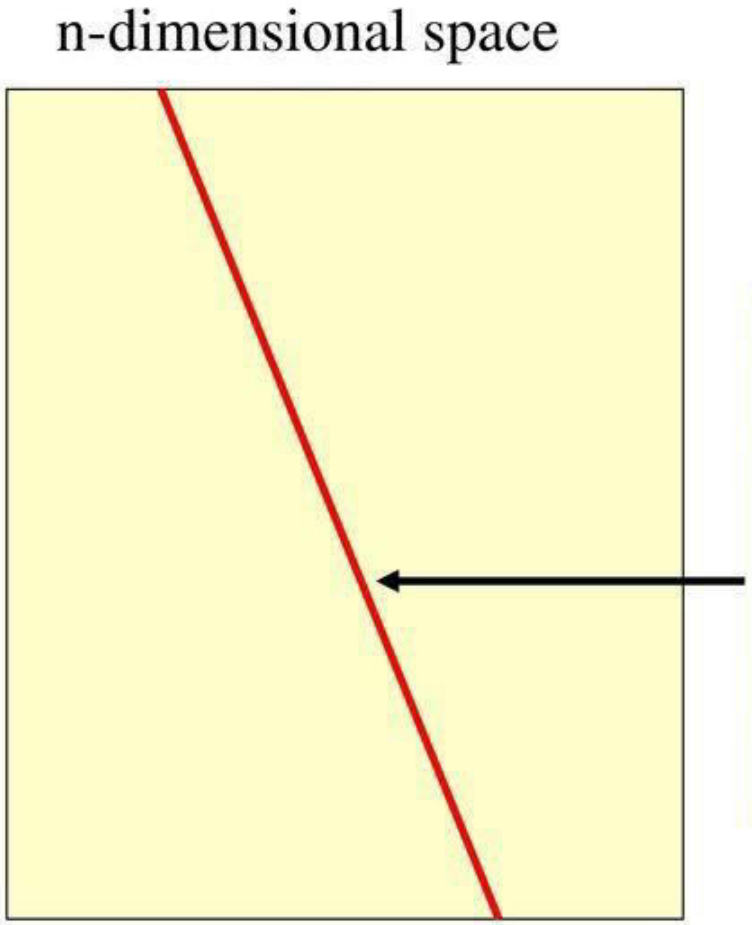
## Scan-Line Method

- Figure illustrates the scan-line method for locating visible portions of surfaces for pixel positions along the line. The active list for line 1 contains information from the edge table for edges AB, BC, EH, and FG. For positions along this scan line between edges AB and BC, only the flag for surface S1 is on. Therefore no depth calculations are necessary, and intensity information for surface S1, is entered from the polygon table into the refresh buffer. Similarly, between edges EH and FG, only the flag for surface S2 is on. NO other positions along scan line 1 intersect surfaces, so the intensity values in the other areas are set to the background intensity. The background intensity can be loaded throughout the buffer in an initialization routine.
- For scan lines 2 and 3 in Fig. , the active edge list contains edges AD, EH, BC, and FG. Along scan line 2 from edge AD to edge EH, only the flag for surface S1 is on. But between edges EH and BC, the flags for both surfaces are on. In this interval, depth calculations must be made using the plane coefficients for the two surfaces. For this example, the depth of surface S1 is assumed to be less than that of S2, so intensities for surface S1 are loaded into the refresh buffer until boundary BC is encountered. Then the flag for surface S1 goes off, and intensities for surface S2 are stored until edge FG is passed.
- We can take advantage of-coherence along the scan lines as we pass from one scan line to the next. In Fig. 4 , scan line 3 has the same active list of edges as scan line 2. Since no changes have occurred in line intersections, it is unnecessary again to make depth calculations between edges EH and BC. The two surfaces must be in the same orientation as determined on scan line 2, so the intensities for surface S1 can be entered without further calculations.

## BSP Trees

- Binary space partitioning trees.
- Used to store a collection of objects in n-dimensional space.
- Tree recursively divides n-dimensional space using (n-1)-dimensional hyperplanes.

# Space Partitioning



splitting hyperplane

(n-1)-dimensional

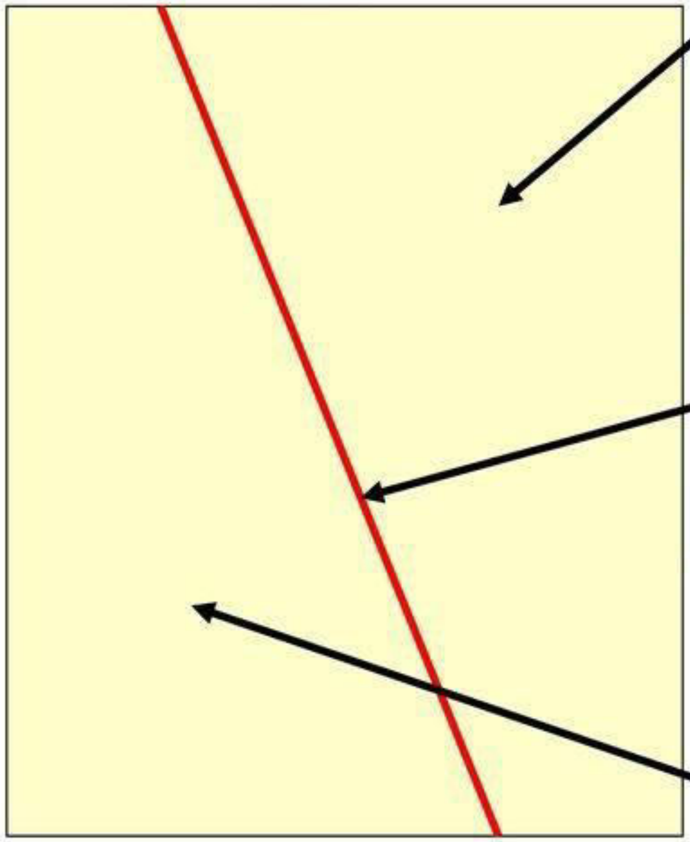
$a_1x_1 + a_2x_2 + \ldots\ a_nx_n + a_{n+1} = 0$

$ax + by + c = 0$ (2D)

$ax+by+cz+d = 0$ (3D)

# Space Partitioning

n-dimensional space



+ve half space

$ax + by + c > 0$

$ax+by+cz+d > 0$

coincident

$ax + by + c = 0$

$ax+by+cz+d = 0$

-ve half space

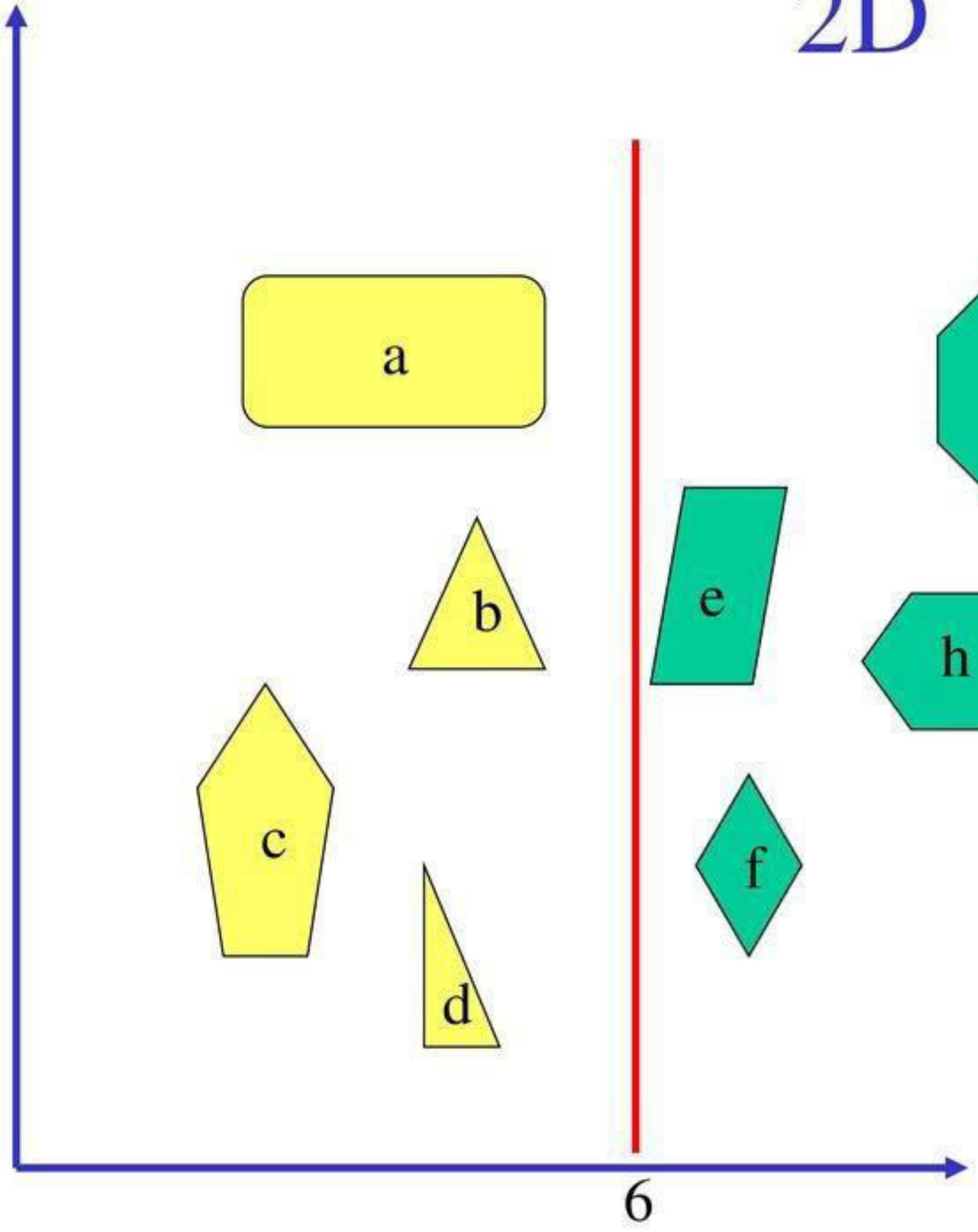
$ax + by + c < 0$

$ax+by+cz+d < 0$

## Classifying Object *z*

- In 2D, *ph* is the line $ax + by + c = 0$.
  - Compute $ax + by + c$ for all vertices of *z*.
  - If all values are = 0; *z* is coincident to *ph*.
  - If all values are <= 0; *z* is left of *ph*.
  - If all values are >= 0; *z* is right of *ph*.
  - Otherwise, *z* spans *ph* and is to be split by finding intersection points with *ph*.

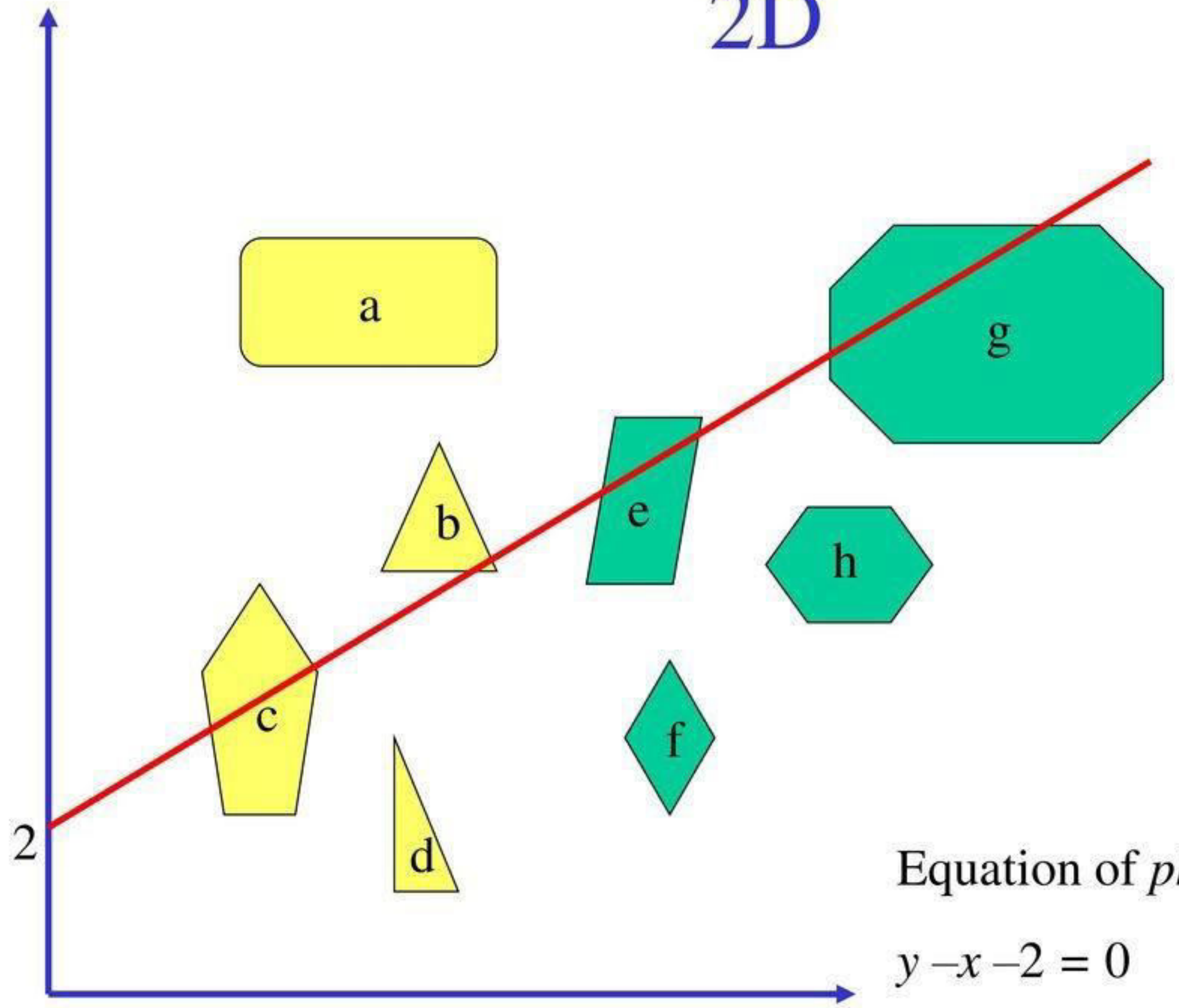
## 2D



Equation of *ph* is

$x - 6 = 0$

# 2D



Equation of *ph* is

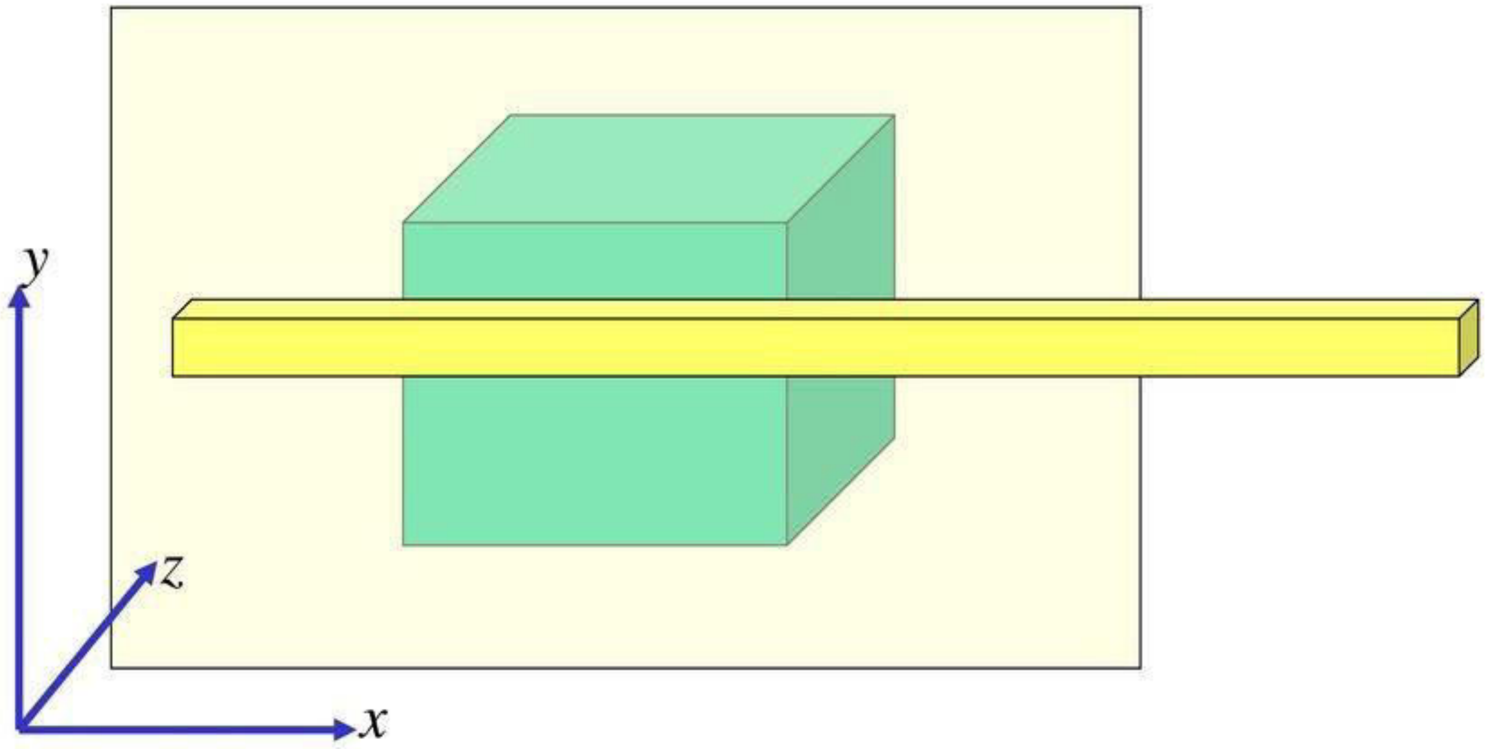
$y - x - 2 = 0$

## 3D

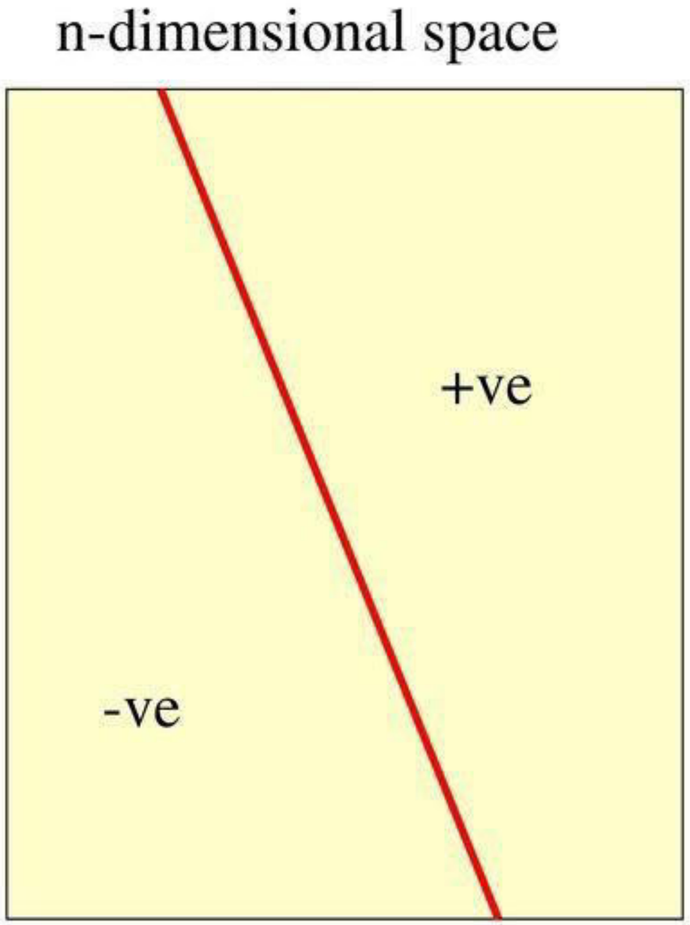
Equation of *ph* is

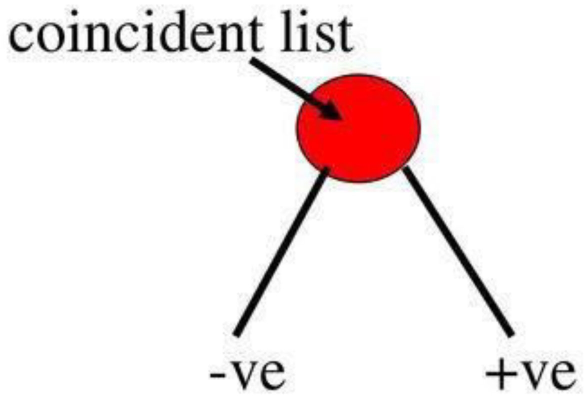
$z - 2 = 0$

General: $ax + by + cz + d = 0$

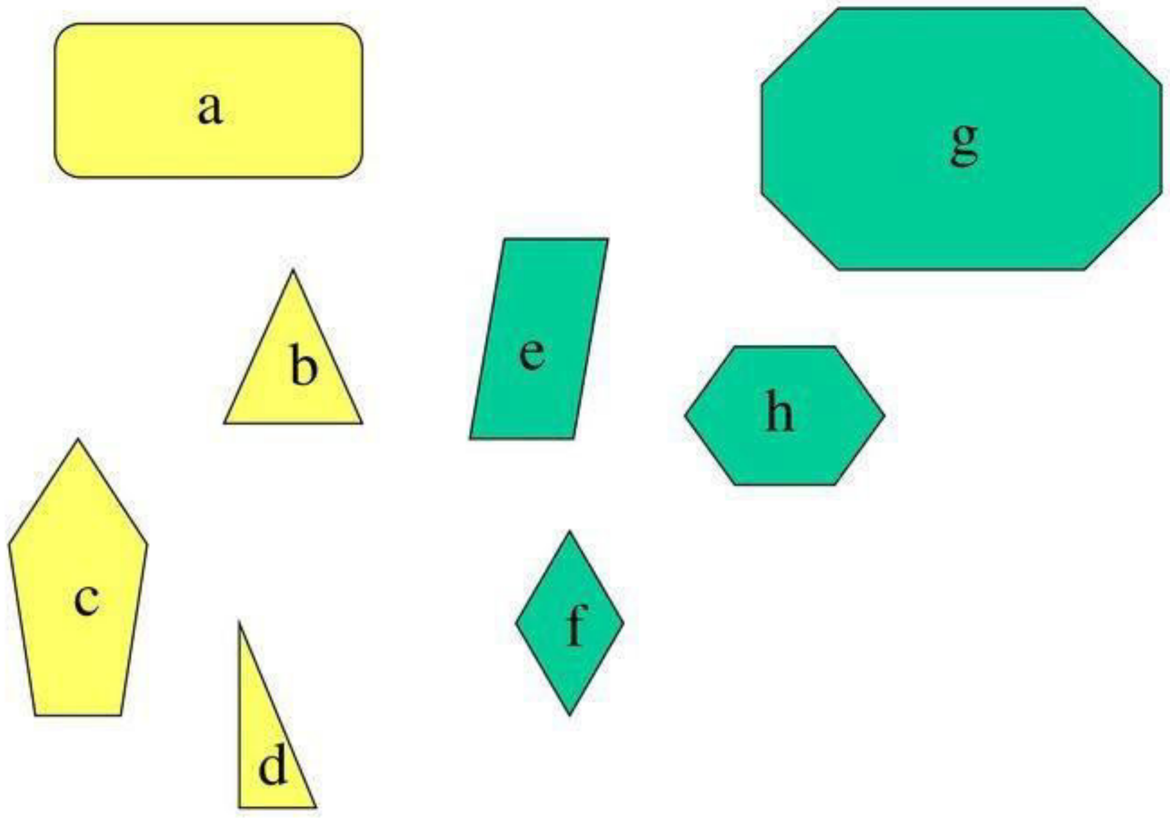


# Space Partitioning

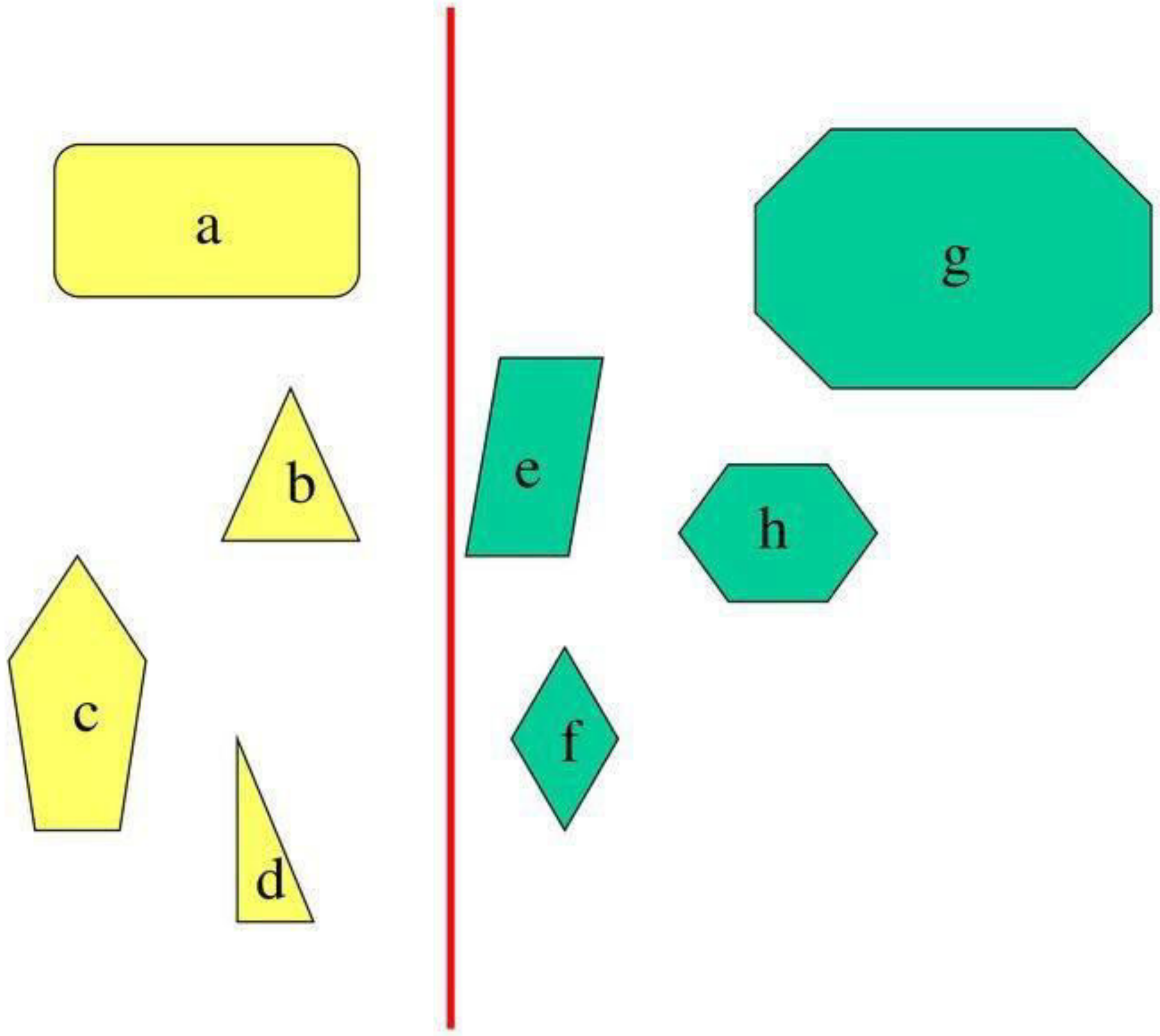




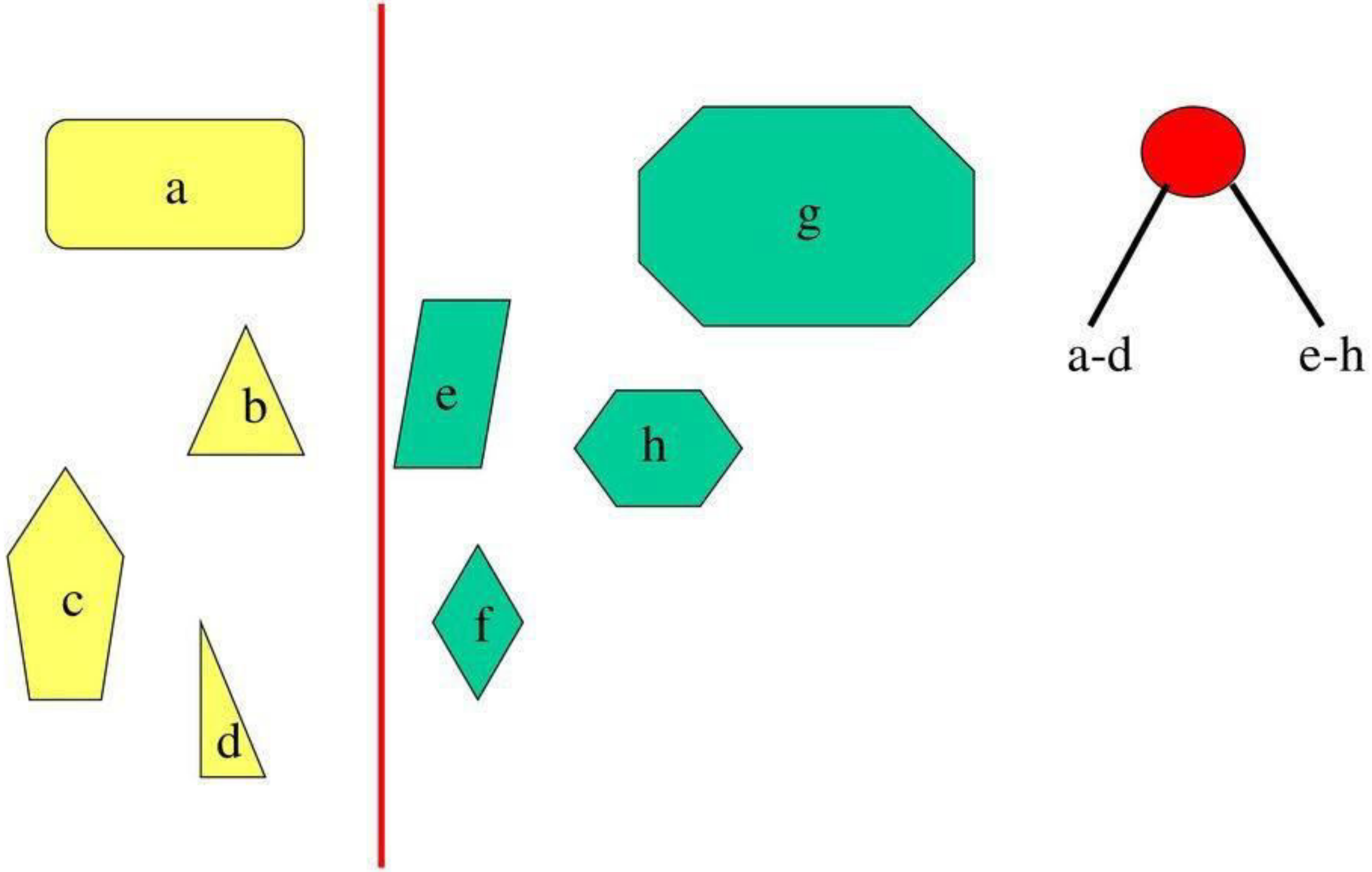
# Objects in 2D



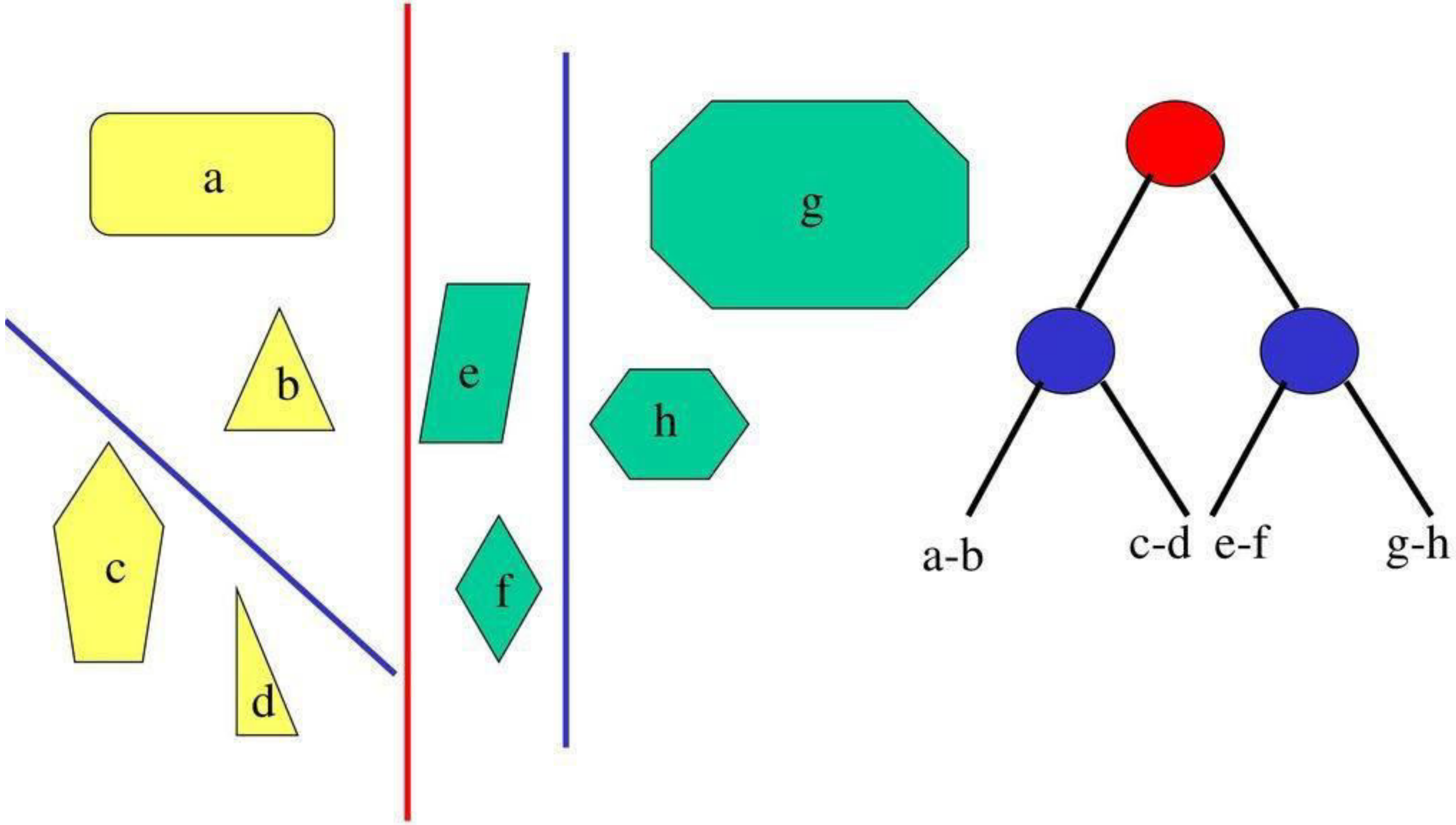
# Objects in 2D



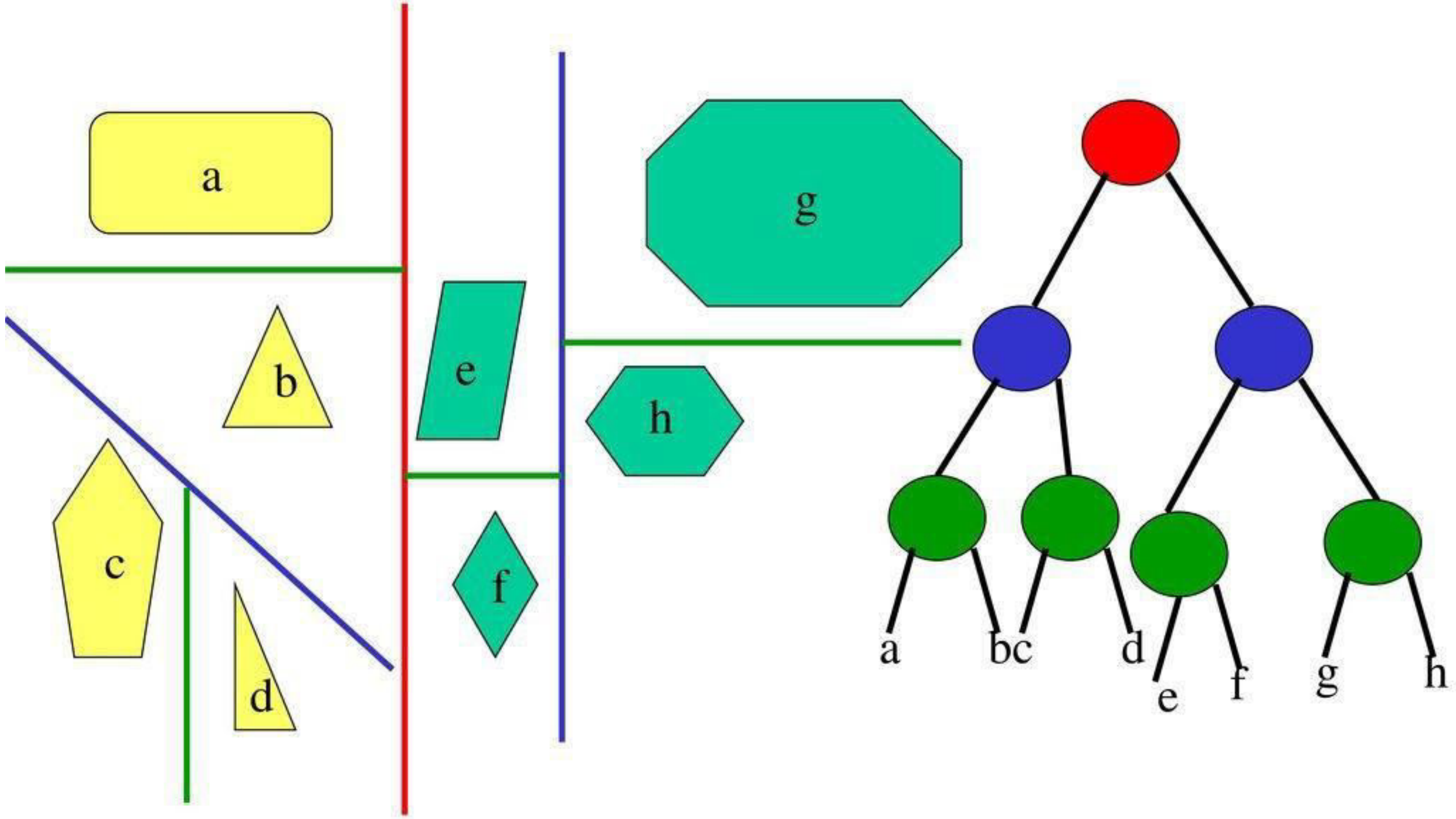
# Objects in 2D



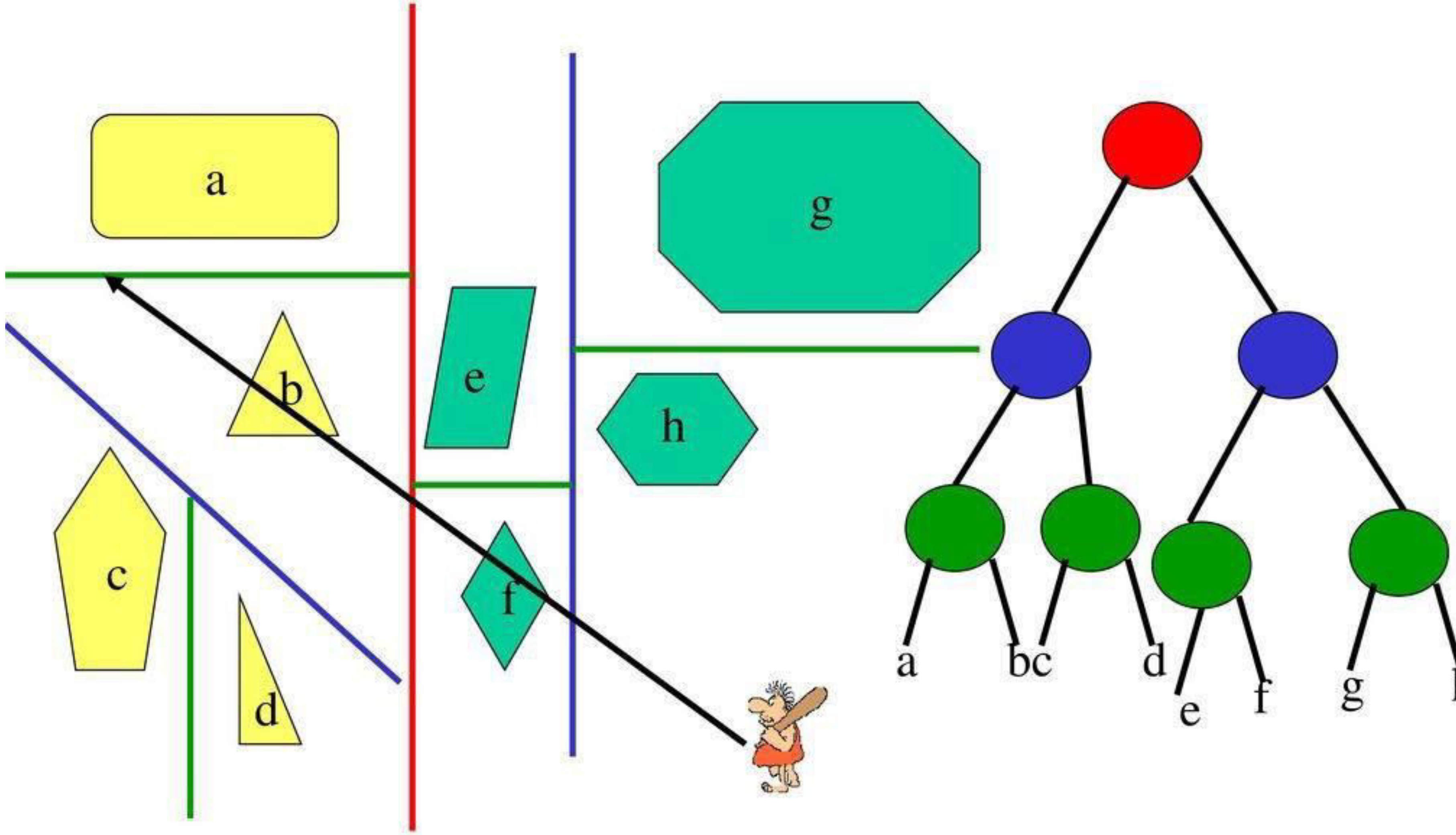
# Objects in 2D



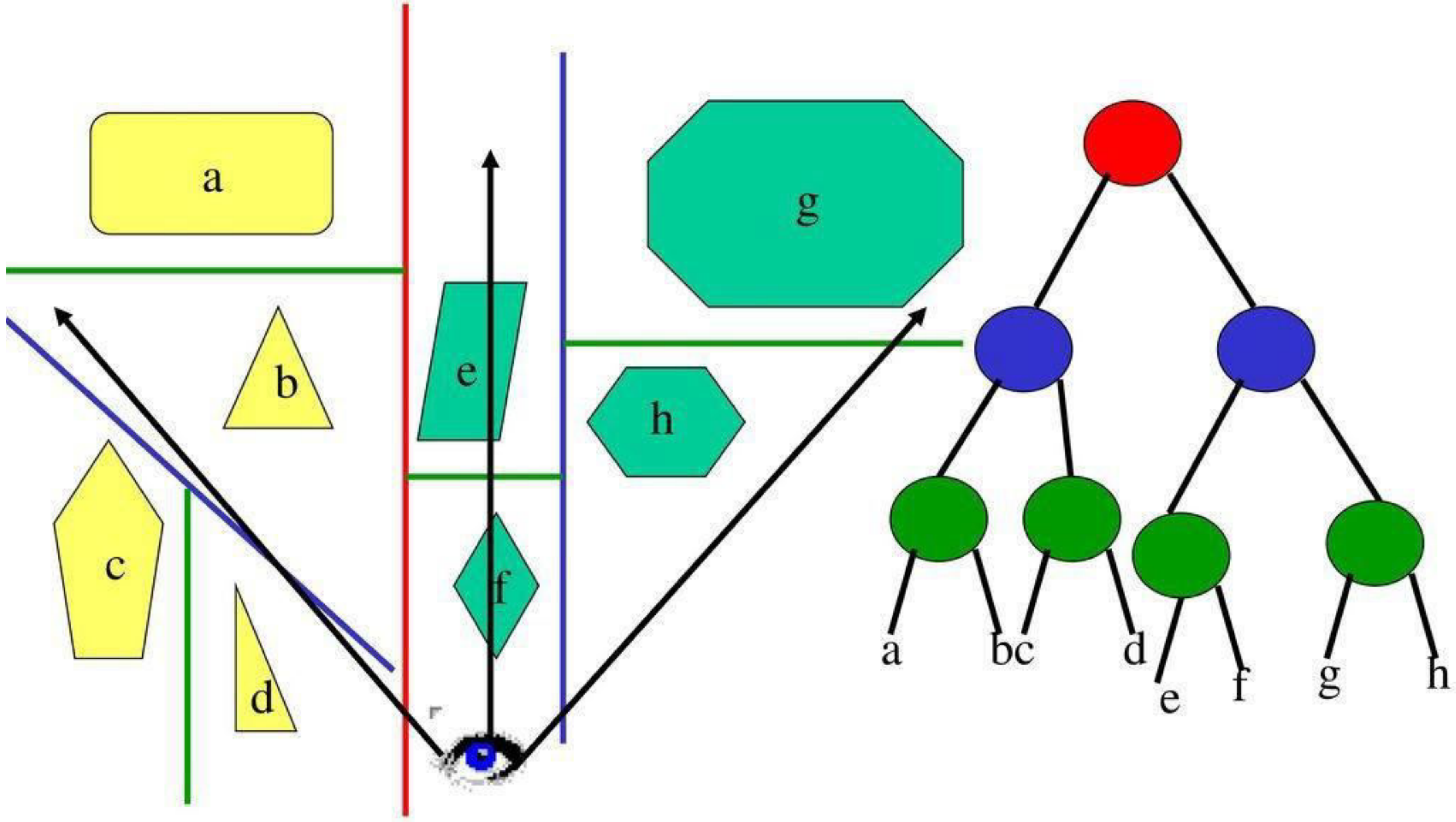
# Objects in 2D



# Collision Detection



# Visibility Ordering



**Area subdivision Method**

This hidden surface removal technique is essentially an image space method

But object space operations can be used to accomplish depth ordering of surfaces

Takes advantage of area coherence

Small areas of an image are likely to be covered by only a single polygon

Applied by successively dividing the total view plane into smaller and smaller

rectangles until each rectangle area contains either

The projection of part of a single visible surface

No surface projections

Or the area has been reduced to the size of a pixel

Starting with a total view, apply tests to determine whether area should be subdivided

The basic principle

If the area is sufficiently complex, then subdivide An easy approach is to

Successively divide the area into four equal parts each step

(same as A 1024 x 1024 viewing area could be subdivided ten times before a subarea is reduced to the size of a single pixel

Four surface classifications

Surrounding surface – Surface completely encloses area

Overlapping surface – Surface partly inside and partly outside

Inside surfac – Surface completely inside

Outside surface – Surface completely outside

No further subdivision required it for specified area one of the following conditions is true.

Condition 1

Area has no inside, overlapping, or surrounding surfaces (i.e. all surfaces are outside the area).

Condition 2

Area has only one inside, overlapping, or surrounding surface.

Condition 3

Area has one surrounding surface that obscures all other surfaces within the area boundaries.

## OCTREE METHOD

An octree is a tree data structure in which each internal node has exactly eight children. Octrees are most often used to partition a three-dimensional space by recursively subdividing it into eight octants. Octrees are the three-dimensional analog of quadtrees. The name is formed from oct + tree, but note that it is normally written "octree" with only one "t". Cortes are often used in 3D graphics and 3D game engines.

Ray casting is the methodological basis for 3-D CAD/CAM solid modeling and image rendering. It is essentially the same as ray tracing for computer graphics where virtual light rays are "cast" or "traced" on their path from the focal point of a camera through each pixel in the camera sensor to determine what is visible along the ray in the 3-D scene. The term "Ray Casting" was introduced by Scott Roth while at the General Motors Research Labs from 1978-1980. His paper, "Ray Casting for Modeling Solids"[1], describes modeled solid objects by combining primitive solids, such as blocks and cylinders, using the set operators union (+), intersection (&), and difference. The general idea of using these binary operators for solid modeling is largely due to Voelcker and Requicha's geometric modelling group at the University of Rochester[2][3]. See Solid modeling for a broad overview of solid modeling methods. This figure on the right shows a U-Joint modeled from cylinders and blocks in a binary tree using Roth's ray casting system, circa 1979.

## RAY CASTING METHOD

Ray-casted image of idealized universal joint with shadow

Before ray casting (and ray tracing), computer graphics algorithms projected surfaces or edges (e.g., lines) from the 3-D world to the image plane where visibility logic had to be applied. The world-to-image plane projection is a 3-D homogeneous coordinate system transformation (aka: 3D projection, affine transformation, or projective transform (Homograph). Rendering an image in that way is difficult to achieve with hidden surface/edge removal. Plus, silhouettes of curved surfaces have to be explicitly solved for whereas it is an implicit by-product of ray casting, so there is no need to explicitly solve for it whenever the view changes.

Ray casting greatly simplified image rendering of 3-D objects and scenes because a line transforms to a line. So, instead of projecting curved edges and surfaces in the 3-D scene to the 2-D image plane, transformed lines (rays) are intersected with the objects in the scene. A homogeneous coordinate transformation is represented by 4x4 matrix. The mathematical technique is common to computer graphics and geometric modeling.[4] A transform includes rotations around the three axes, independent scaling along the axes, translations in 3-D, and even skewing. Transforms are easily concatenated via matrix arithmetic. For use with a 4x4 matrix, a point is represented by [X, Y, Z, 1] and a direction vector is represented by [Dx, Dy, Dz, 0]. (The fourth term is for translation and that does not apply to direction vectors.)

While simplifying the mathematics, the ray casting algorithm is very computer-processing intensive. Pixar has large render farms, buildings with 1000's of CPUs, to make their animations using ray tracing [aka "ray casting"] as a core technique.