ICT 4203
Computer Graphics and Animation

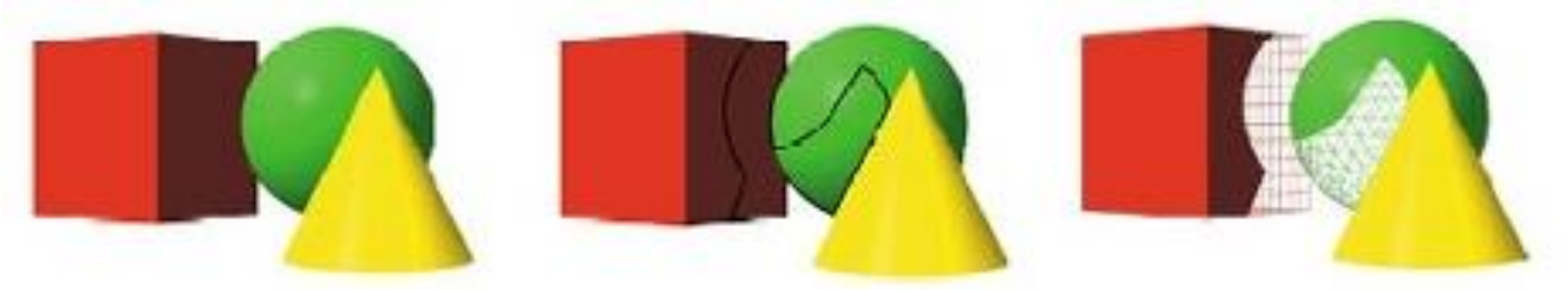
Lecture 16

Hidden surface removal

Md. Mahmudur Rahman
Lecturer
Institute of Information Technology

Introduction

- **Need for Hidden Surface Removal**
- We must determine what is visible within a scene from a chosen viewing position
- For 3D worlds this is known as **visible surface detection** or **hidden surface elimination**



Hidden surface removal

Drawing polygonal faces on screen consumes CPU cycles

- ✓ Illumination

We cannot see every surface in scene

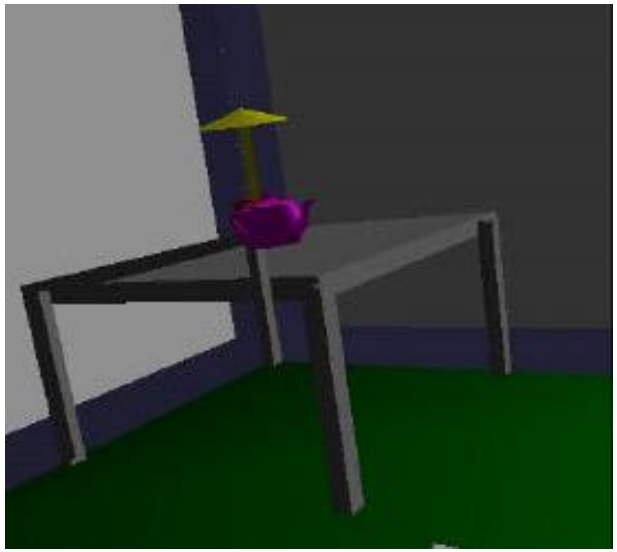
- ✓ We don’t want to waste time rendering primitives which don’t contribute to the final image.

Visibility (hidden surface removal)

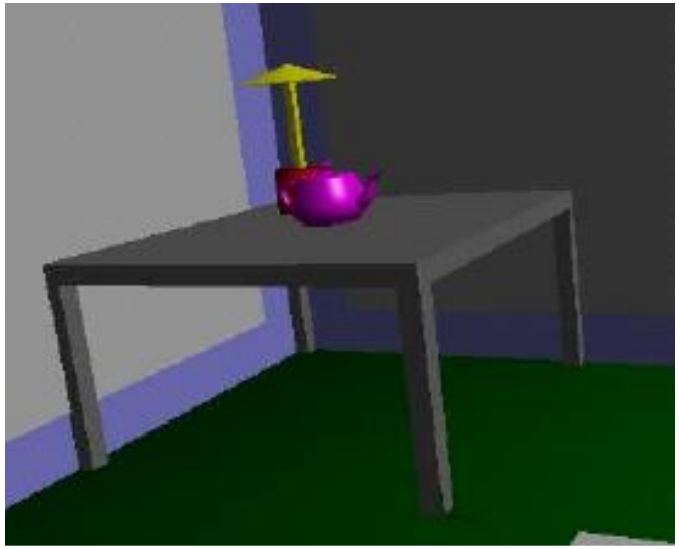
A correct rendering requires correct visibility calculations
Correct visibility

when multiple opaque polygons cover the same screen space, only the closest one is visible (remove the other hidden surfaces)

wrong visibility



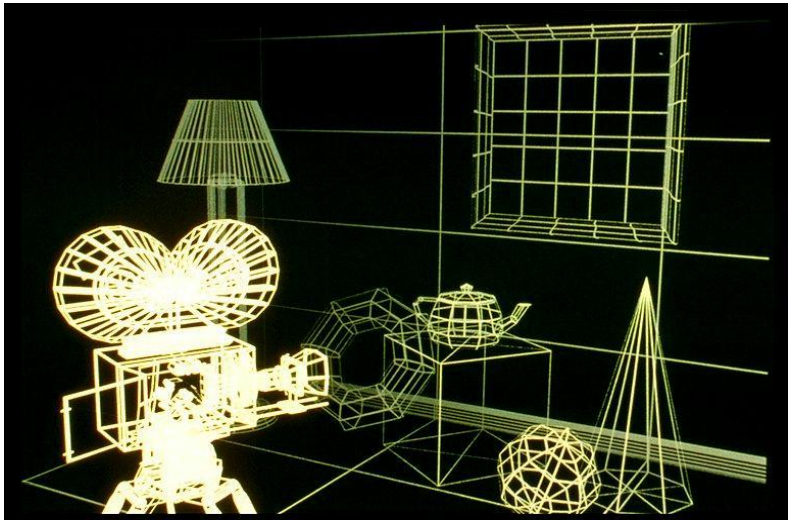
correct visibility

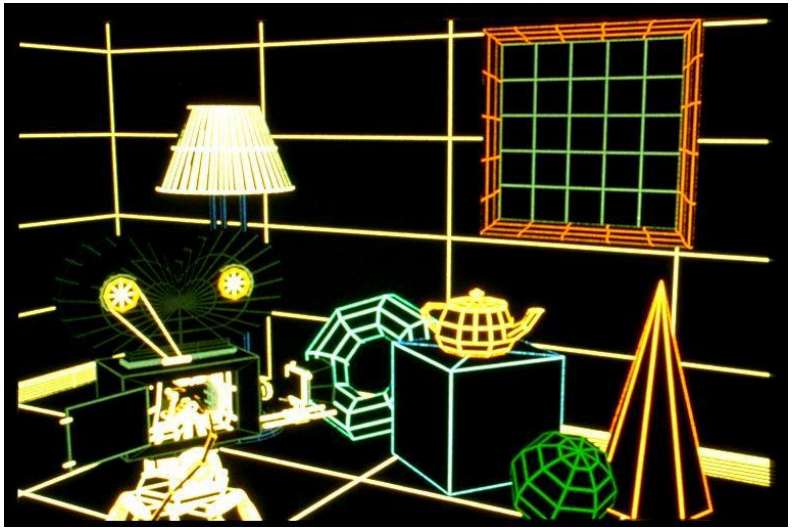


Visibility of primitives

A scene primitive can be invisible for 3 reasons:

- Primitive lies outside field of view
- Primitive is *back-facing*
- Primitive is occluded by one or more objects nearer the viewer





Visible surface detection algorithm

Visible surface detection algorithms are broadly classified as:

Object Space Methods: Compares objects and parts of objects to each other within the scene definition to determine which surfaces are visible

- *Back face culling, Painter's algorithm, BSP trees*

Image Space Methods: Visibility is decided point-by-point at each pixel position on the projection plane

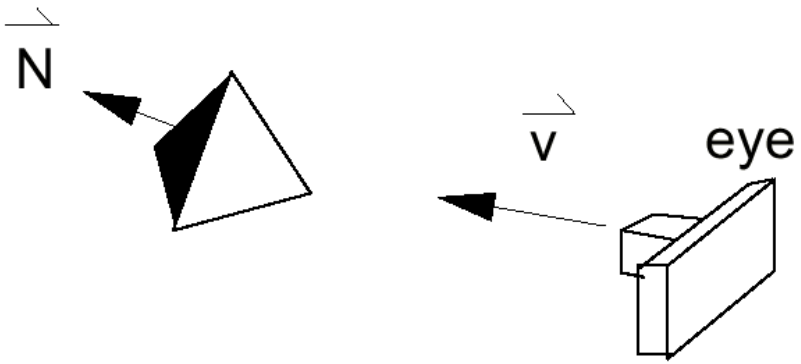
- *Z-buffering*

Image space methods are by far the more common

Back face culling

Backface culling is a technique used in 3D computer graphics to improve rendering efficiency. It helps by not rendering the faces of an object that are not visible to the viewer, essentially those that are facing away from the camera.

The simplest thing we can do is find the faces on the backs of polyhedra and discard them



How Backface Culling Works:

1. **Determine the Face Orientation**:
    1. For each face (polygon) of a 3D model, calculate the normal vector, which is perpendicular to the surface of the face.
    2. The normal vector helps determine which way the face is oriented.
2. **Check Visibility**:
    1. Compare the normal vector of the face to the direction of the camera's view.
    2. If the angle between the camera's viewing direction and the normal vector indicates that the face is facing away from the camera, it is considered a backface.
3. **Cull Backfaces**:
    1. Backfaces are not rendered, as they are not visible to the viewer. This reduces the number of polygons that need to be processed and rendered, leading to improved performance.

Back-Face Detection

- We know from before that a point (*x*, *y*, *z*) is behind a polygon surface if:

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

where *A*, *B*, *C* & *D* are the plane parameters for the surface

- When an inside point is along the line of sight to the surface, the polygon must be a back face.
- It is also called culling.
- This is an object space method.

Back Face Removed Algorithm

Repeat for all polygons in the scene.

Do numbering of all polygons in clockwise direction i.e.
v1 v2 v3.....vz
Calculate normal vector i.e. N1
N1=(v2-v1)*(v3-v2)
Consider projector P, it is projection from any vertex
Calculate dot product
Dot=N.P
Test and plot whether the surface is visible or not.
If Dot ≥ 0 then
surface is visible
else
Not visible

Benefits of Backface Culling

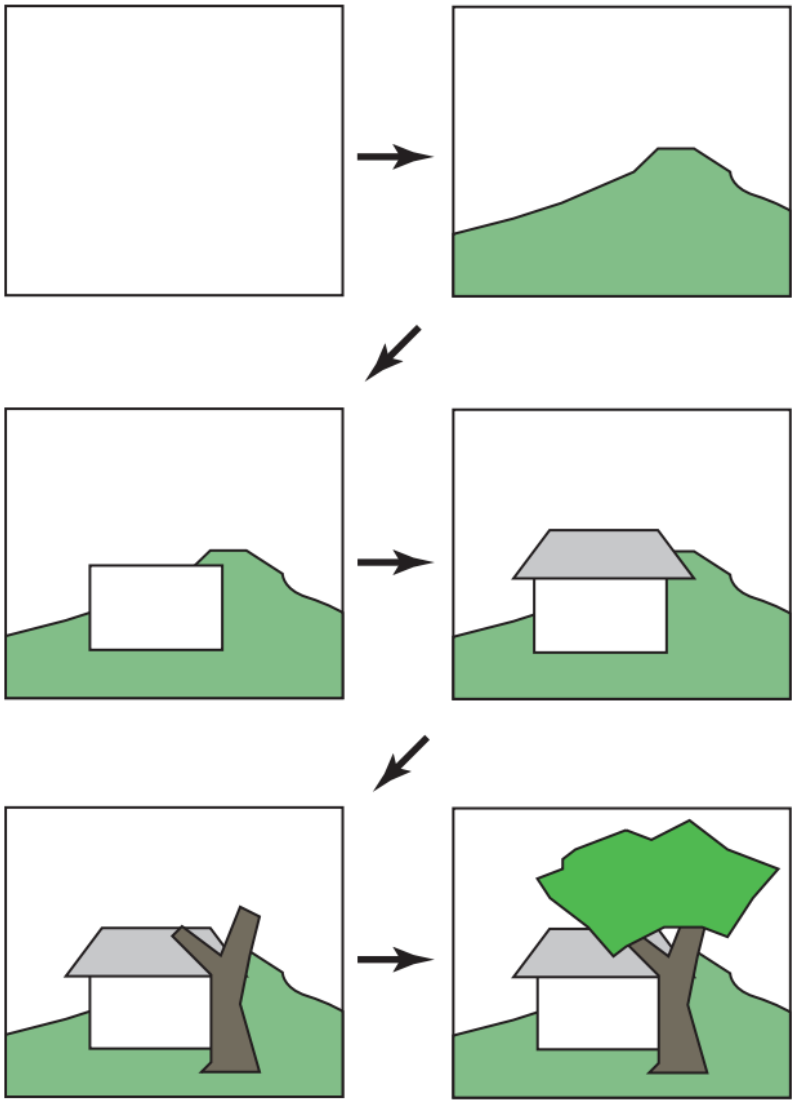
1. **Performance Improvement**: Reduces the number of polygons to be processed, leading to faster rendering times.
2. **Memory Efficiency**: Less memory is needed as fewer polygons are stored for rendering.
3. **Simplicity in Shading**: Simplifies shading calculations by ignoring faces that are not visible.

Painters' algorithm

It came under the category of list priority algorithm. It is also called a **depth-sort** algorithm. In this algorithm ordering of visibility of an object is done. If objects are reversed in a particular order, then correct picture results.

Objects are arranged in increasing order to z coordinate. Rendering is done in order of z coordinate. Further objects will obscure near one. Pixels of rear one will overwrite pixels of farther objects.

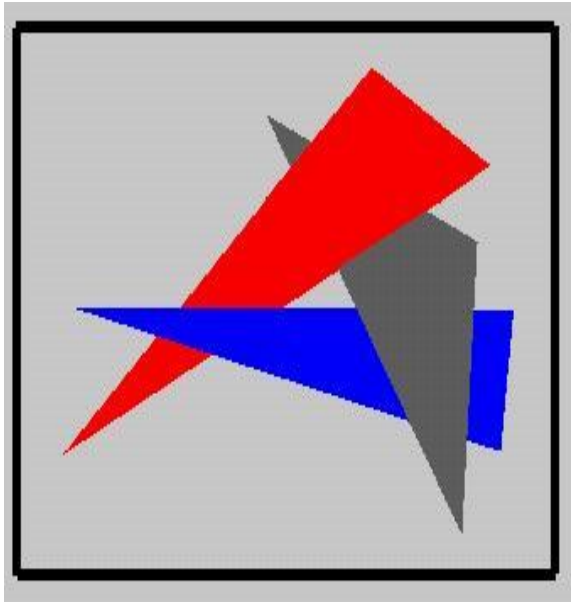
Painters' algorithm



A painter's algorithm starts with a blank image and then draws the scene one object at a time from back-to-front, overdrawing whatever is already there. This automatically eliminates hidden surfaces.

Painters' algorithm (object space).

- Draw surfaces in back to front order – nearer polygons “paint” over farther ones.
- Supports transparency.
- Key issue is order determination.
- Doesn’t always work – see image at right.



Painter Algorithm

Steps performed in-depth sort

1.Sort all polygons according to z coordinate.

2.Find ambiguities of any, find whether z coordinate overlap, split polygon if necessary.

3.Scan convert each polygon in increasing order of z coordinate.

Painter Algorithm

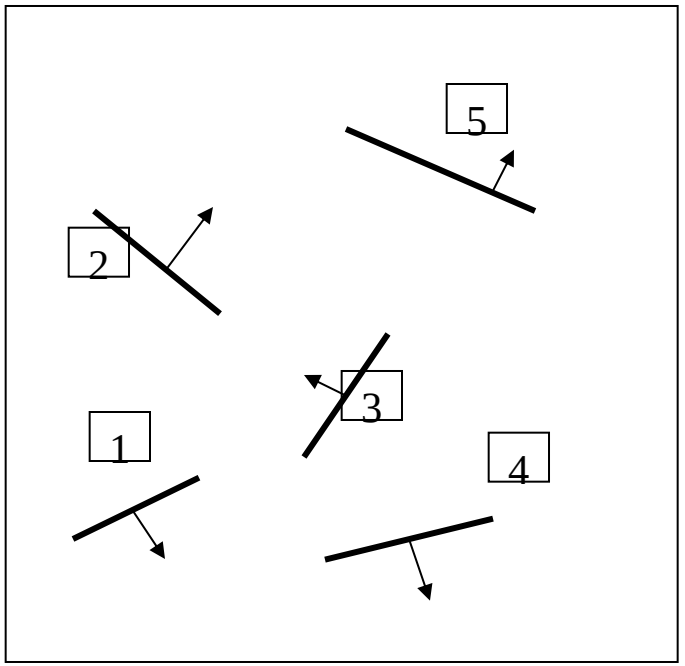
Step1: Start Algorithm

Step2: Sort all polygons by z value keep the largest value of z first.

Step3: Scan converts polygons in this order.

BSP (Binary Space Partitioning) Tree

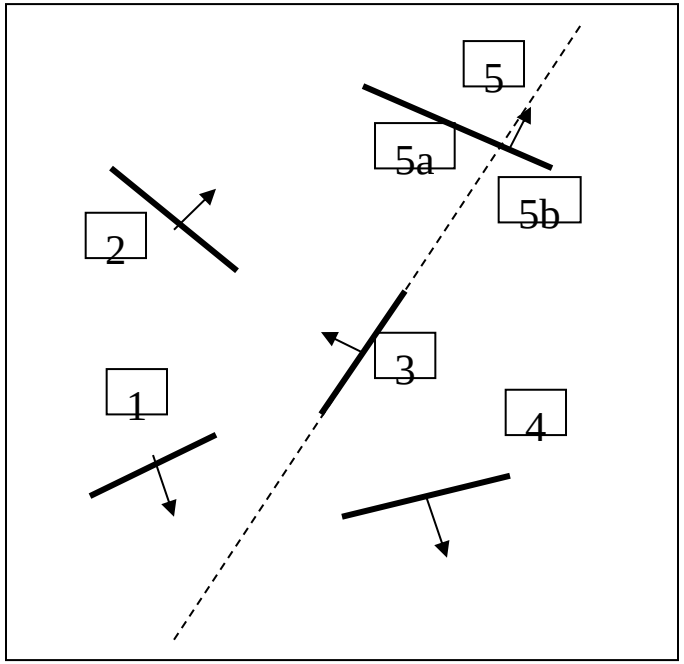
- One of class of "list-priority" algorithms – returns ordered list of polygon fragments for specified view point (static pre-processing stage).
- Choose polygon arbitrarily
- Divide scene into front (relative to normal) and back half-spaces.
- Split any polygon lying on both sides.
- Choose a polygon from each side – split scene again.
- Recursively divide each side until each node contains only 1 polygon.

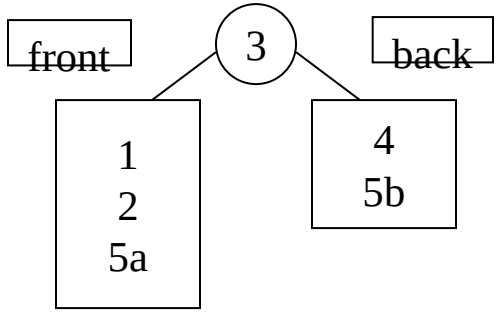


View of scene from above

BSP Tree

- **Choose polygon arbitrarily**
- **Divide scene into front (relative to normal) and back half-spaces.**
- **Split any polygon lying on both sides**.
- Choose a polygon from each side – split scene again.
- Recursively divide each side until each node contains only 1 polygon.





BSP Tree.

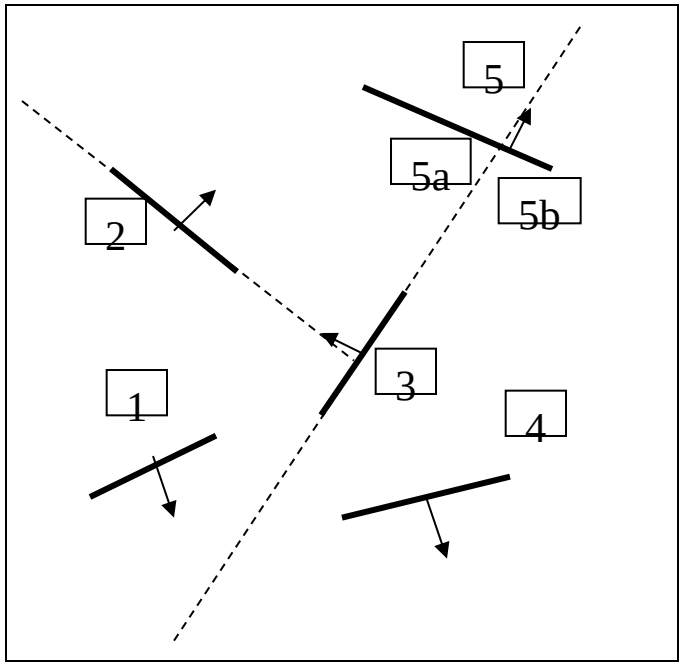
•Choose polygon arbitrarily

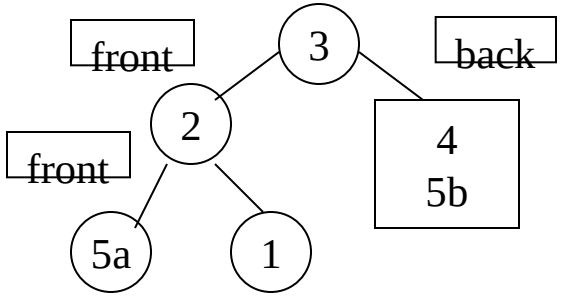
•Divide scene into front (relative to normal) and back half-spaces.

•Split any polygon lying on both sides.

•**Choose a polygon from each side – split scene again.**

•Recursively divide each side until each node contains only 1 polygon.





BSP Tree.

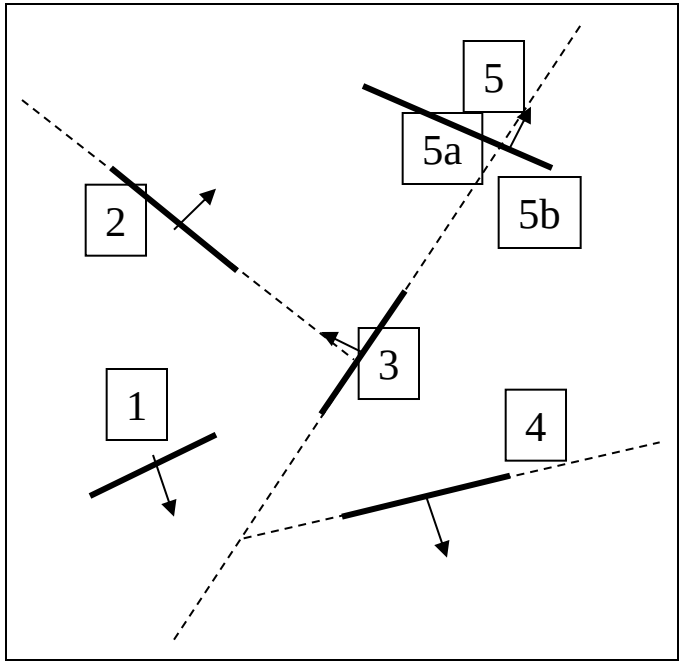
•Choose polygon arbitrarily

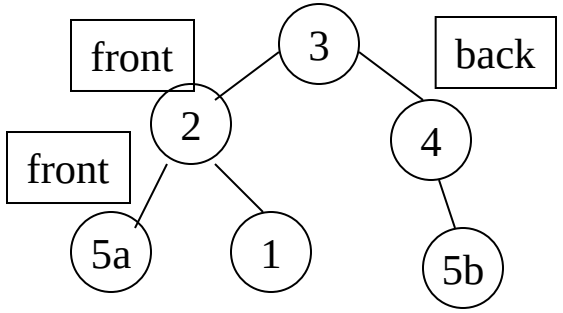
•Divide scene into front (relative to normal) and back half-spaces.

•Split any polygon lying on both sides.

•Choose a polygon from each side – split scene again.

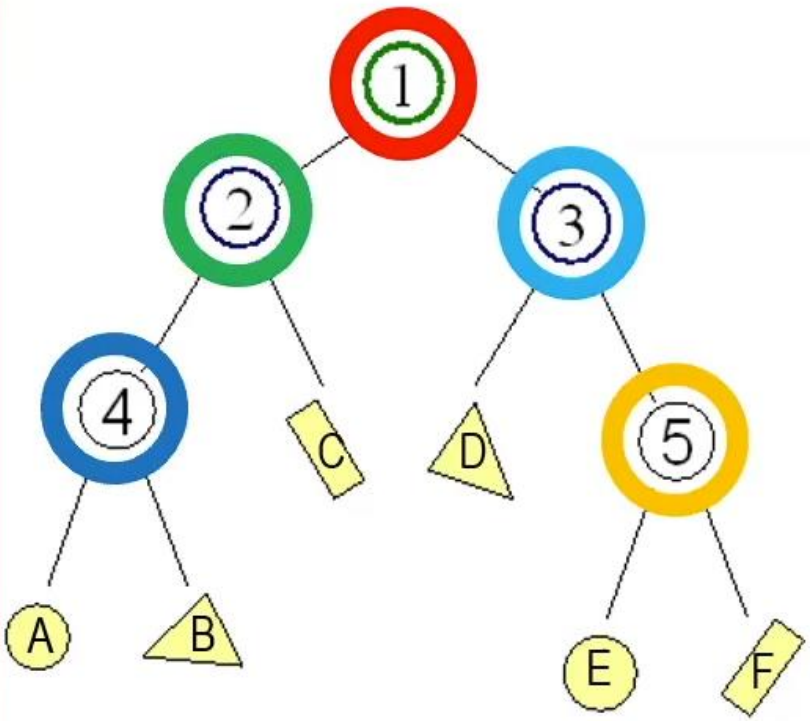
•**Recursively divide each side until each node contains only 1 polygon.**

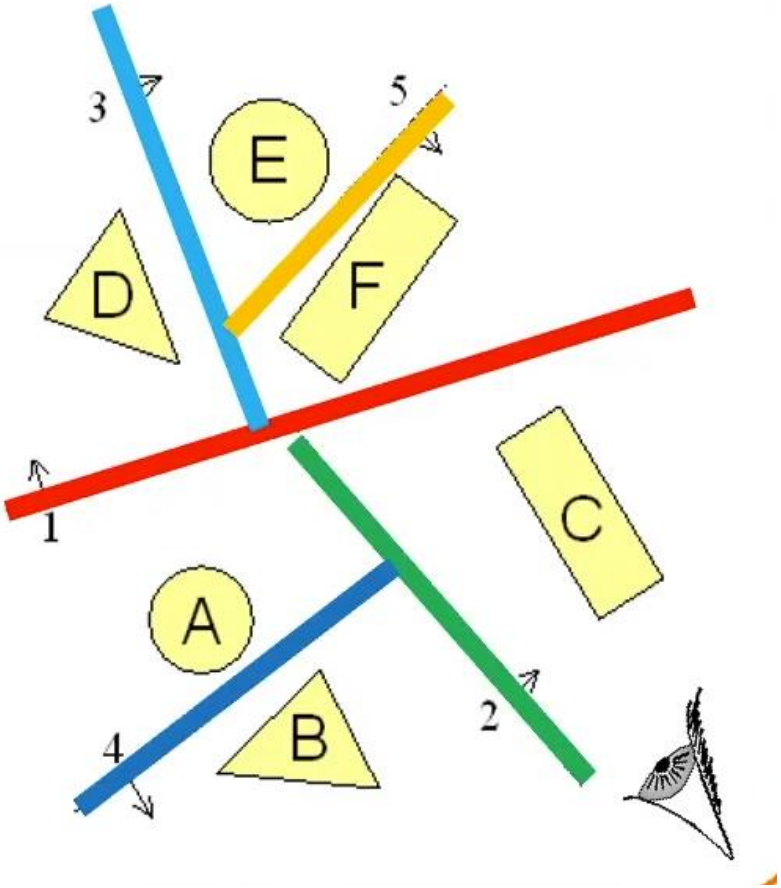




BSP Tree

- Recursively partition space by planes
  - Every cell is a convex polyhedron





Drawing order: D, E, F, A, B, C

Displaying a BSP tree.

Once we have the regions – need priority list
BSP tree can be traversed to yield a correct priority list for an arbitrary viewpoint.
Start at root polygon.

- If viewer is in front half-space, draw polygons behind root first, then the root polygon, then polygons in front.
- If polygon is on edge – either can be used.
- Recursively descend the tree.

If eye is in rear half-space for a polygon – then can back face cull.

Why BSP Trees are Needed:

1. Efficient Rendering:

•**Visibility Determination**: In 3D rendering, determining which objects are visible from a certain viewpoint is crucial. BSP trees help in quickly determining visibility by providing a structured way to test whether objects or parts of objects are in view.

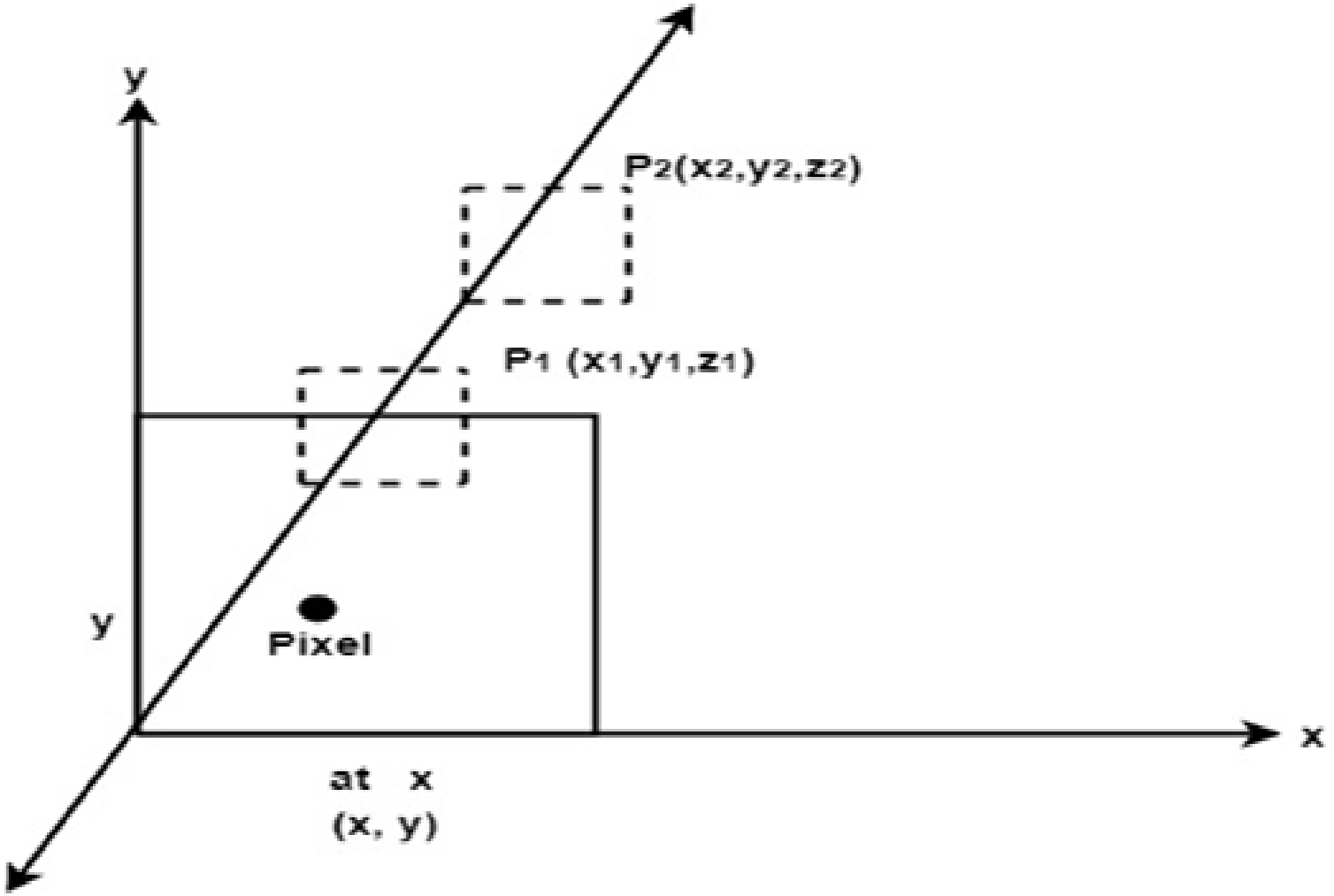
•**Painter's Algorithm**: BSP trees can be used to implement the Painter's Algorithm, which involves rendering objects from back to front to handle occlusion (i.e., closer objects obscure further objects).

Z-buffering : image space approach

We say that a point in display space is “seen” from pixel (x, y) if the projection of the point is scan- converted to this pixel . The Z-buffer algorithm essentially keeps track of the smallest z coordinate (also called the depth value) of those points which are seen from pixel (x, y). These Z values are stored in what is called the Z buffer.

Basic Z-buffer idea:

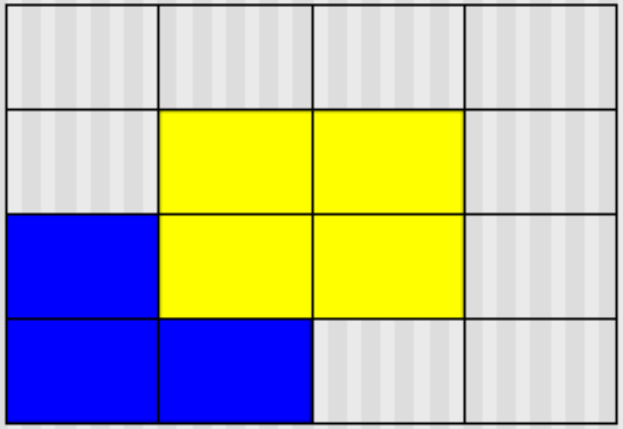
- Rasterize every input polygon
- For every pixel in the polygon interior, calculate its corresponding z value (by interpolation)
- Track depth values of closest polygon (smallest z) so far
- Paint the pixel with the color of the polygon whose z value is the closest to the eye.


y
P2(x2,y2,z2)
P1 (x1,y1,z1)
y
Pixel
x
at x
(x, y)

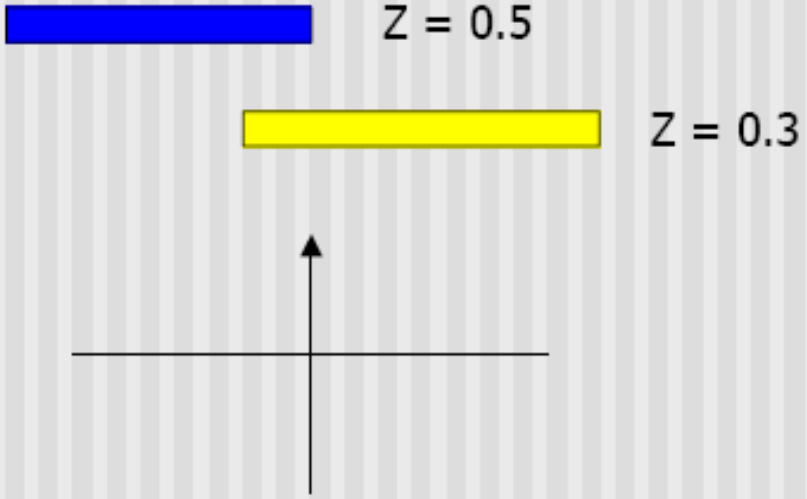
Z-buffer

- Z-buffer, which is also known as the **Depth-buffer** method is one of the commonly used method for hidden surface detection.
- It is an ***Image space*** method. Image space methods are based on the pixel to be drawn on 2D.
- For these methods, the running time complexity is the number of pixels times number of objects.
- And the space complexity is two times the number of pixels because two arrays of pixels are required, one for frame buffer and the other for the depth buffer.

Z buffer example



Correct Final image



Top View

Z buffer example

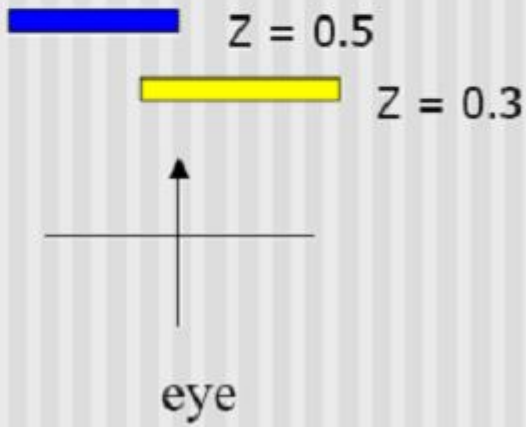
Step 1: Initialize the depth buffer

1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0

Z buffer example

Step 2: Draw the blue polygon (assuming the program draws blue polyon first – the order does not affect the final result any way).

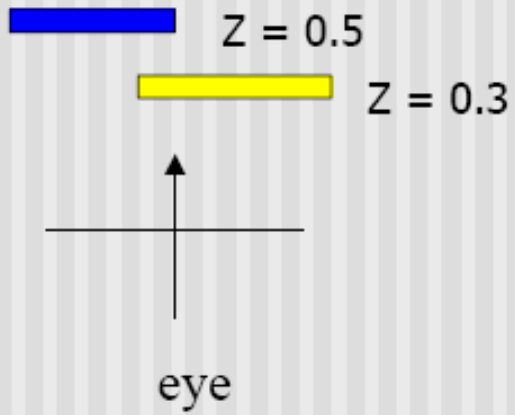
1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0
0.5	0.5	1.0	1.0
0.5	0.5	1.0	1.0



Z buffer example

Step 3: Draw the yellow polygon

1.0	1.0	1.0	1.0
1.0	0.3	0.3	1.0
0.5	0.3	0.3	1.0
0.5	0.5	1.0	1.0



z-buffer drawback: wastes resources by rendering a face and then drawing over it

Implementation.

- **Initialization**:
  - Create a Z-buffer with the same dimensions as the frame buffer (i.e., the display screen resolution).
  - Initialize the Z-buffer to the maximum possible depth value (e.g., infinity or a very large value) for each pixel.
  - Initialize the frame buffer to the background color.
- **Process Each Polygon**:
  For each polygon in the scene, perform the following steps:
- **For Each Pixel**:
  For each pixel covered by the polygon, calculate the depth (Z-value) of the polygon at that pixel.
- **Compare Depths**:
  1. Compare the calculated Z-value with the current value in the Z-buffer at that pixel.
  2. If the calculated Z-value is less (i.e., the polygon is closer to the viewer) than the value in the Z-buffer, update the Z-buffer and the frame buffer:

     Update Z-buffer: Set the Z-buffer value at that pixel to the calculated Z-value.

     Update Frame Buffer: Set the color of that pixel in the frame buffer to the color of the polygon at that pixel.

Why is z-buffering so popular ?

Advantage

- Simple to implement in hardware.
  - Memory for z-buffer is now not expensive
- Diversity of primitives – not just polygons.
- Unlimited scene complexity
- Don’t need to calculate object-object intersections.

Disadvantage

- Extra memory and bandwidth
- Waste time drawing hidden objects
- Z-precision errors
- May have to use point sampling

Given points P1(1,2,0), P2(3,6,20), and P3(2,4,6) and a viewpoint C(0, 0,-10), determine which points obscure the others when viewed from C.

We can determine whether viewpoint obscures another by examining the lines formed from the viewpoint (C) to each point in question. Specifically, we need to find whether the given points (P1, P2, P3) are on the same line
(i.e., they are collinear) formed with viewpoint C.

If the points are collinear, the point that is closer to the viewpoint will obscure the farther one when seen from the viewpoint.

If P1, P2, P3 and C are collinear, then vector CP1 = a1 * CP2,
vector CP1 = a2 * CP3 and vector CP2 = a3 * CP3.

Let's calculate these vectors and check:

Vector CP1: (1 - 0, 2 - 0, 0 – (-10)) = (1, 2, 10)

Vector CP2: (3 - 0, 6 - 0, 20 – (-10)) = (3, 6, 30)

Vector CP3: (2 - 0, 4 - 0, 6 – (-10)) = (2, 4, 16)

We can say that one vector is a scalar multiple of another vector if all components are multiplied by the same scalar value.

Therefore,

- CP2 = 3CP1 since (3, 6, 30) = 3(1, 2, 10)
- CP3 ≠ any multiple of CP1, or of CP2 as

(2, 4, 16) ≠ any multiple of (1, 2, 10) or (3, 6, 30)

So, from point C,

- (A) P1 obscures P2, as P2 lies on the line segment from C to P1.
- (C) P3 does not obscure P1, as they do not lie on the same line segment.
- (D) P2 does not obscure P3, as they do not lie on the same line segment.

Thank You