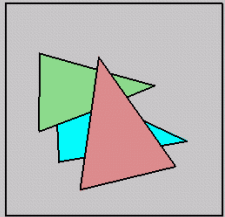


Hidden Surface Removal

Foley : Upto 15.1, 15.2.4, 15.4, 15.5.1, 15.5.2, 15.6

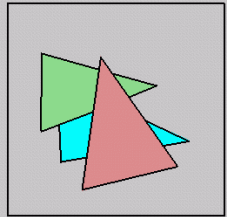


Recap

- *Fundamentals*
 - Introduction
 - Graphics pipeline
 - Vectors

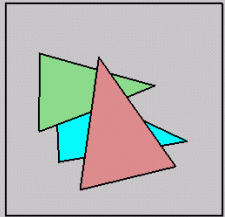
- *Modeling Transformation*
 - Transformation to position objects e.g. translation, rotation, scaling etc.

- *Projection and View Transformation*
 - Project and view a 3D model on a 2D screen

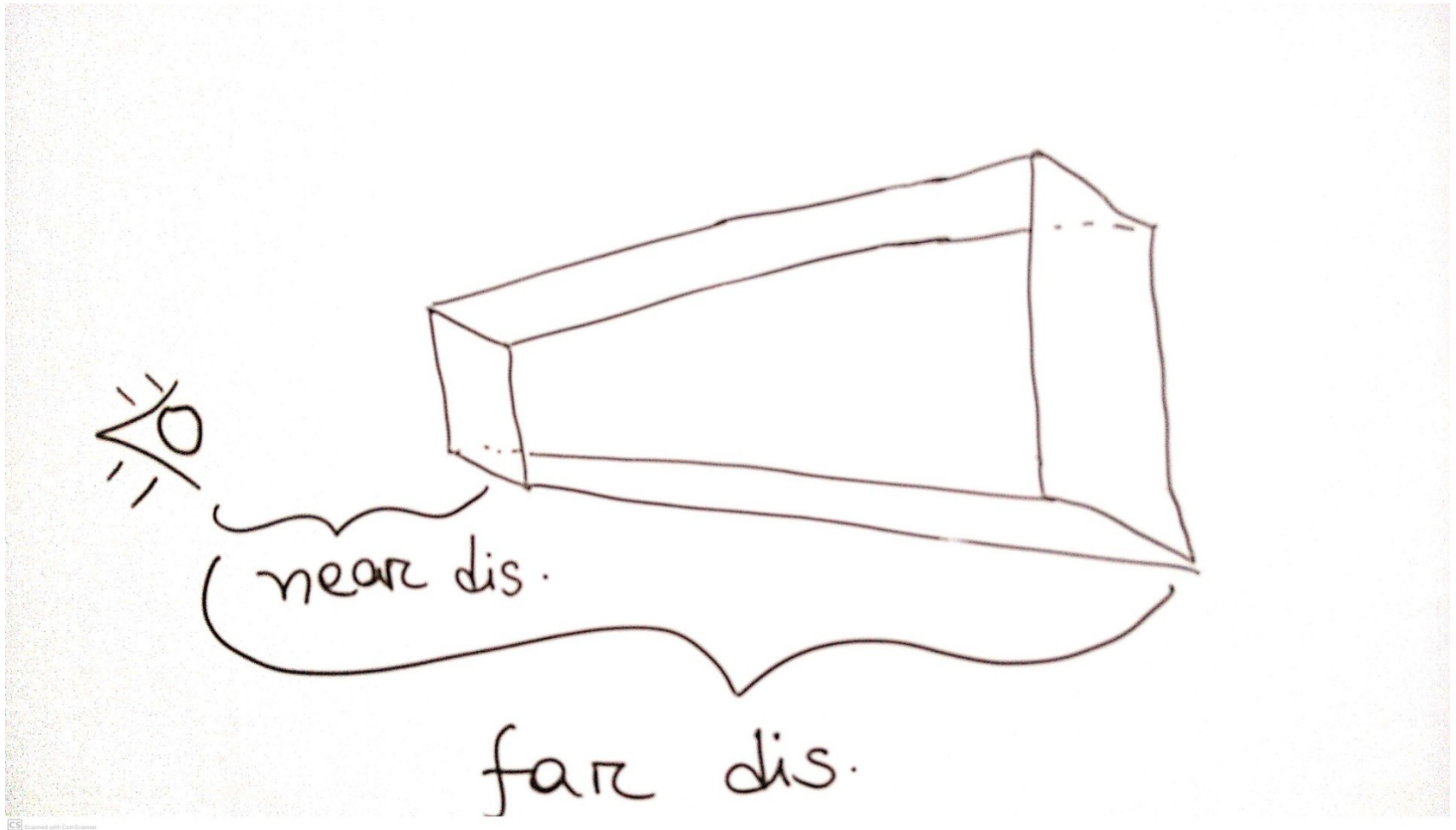


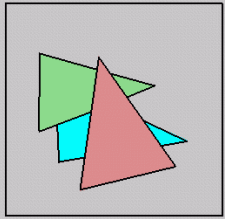
Visible Surface

- Problem: Given a set of 3D objects and a viewing specification, determine which lines or surfaces are visible
- This is called *visible surface(line) determination*, or *hidden surface(line) removal*
- Two general approaches:
 - Image precision
 - Object precision



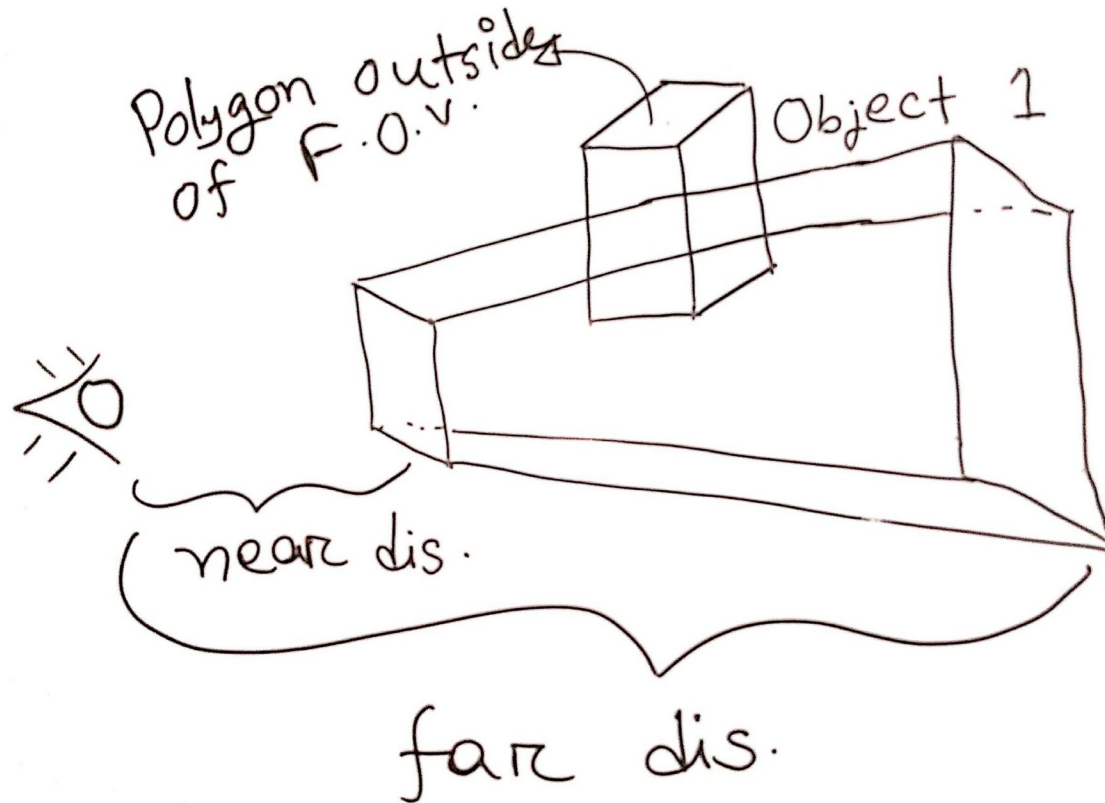
Invisible Primitives

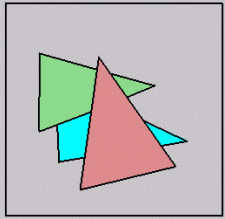




Invisible Primitives

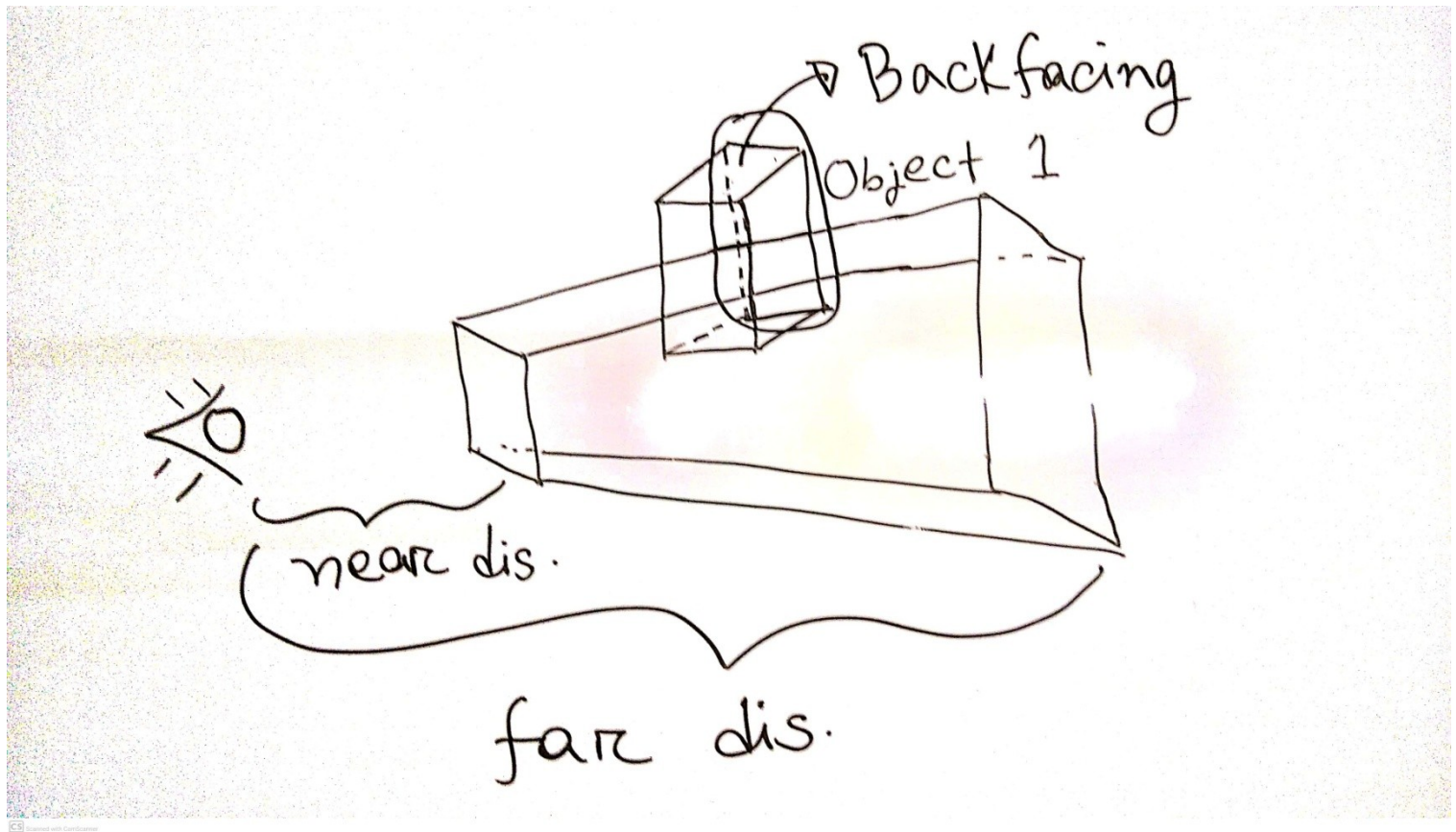
- Why might a polygon be invisible?
 - Polygon outside the *field of view*

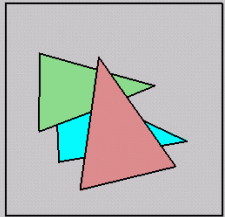




Invisible Primitives

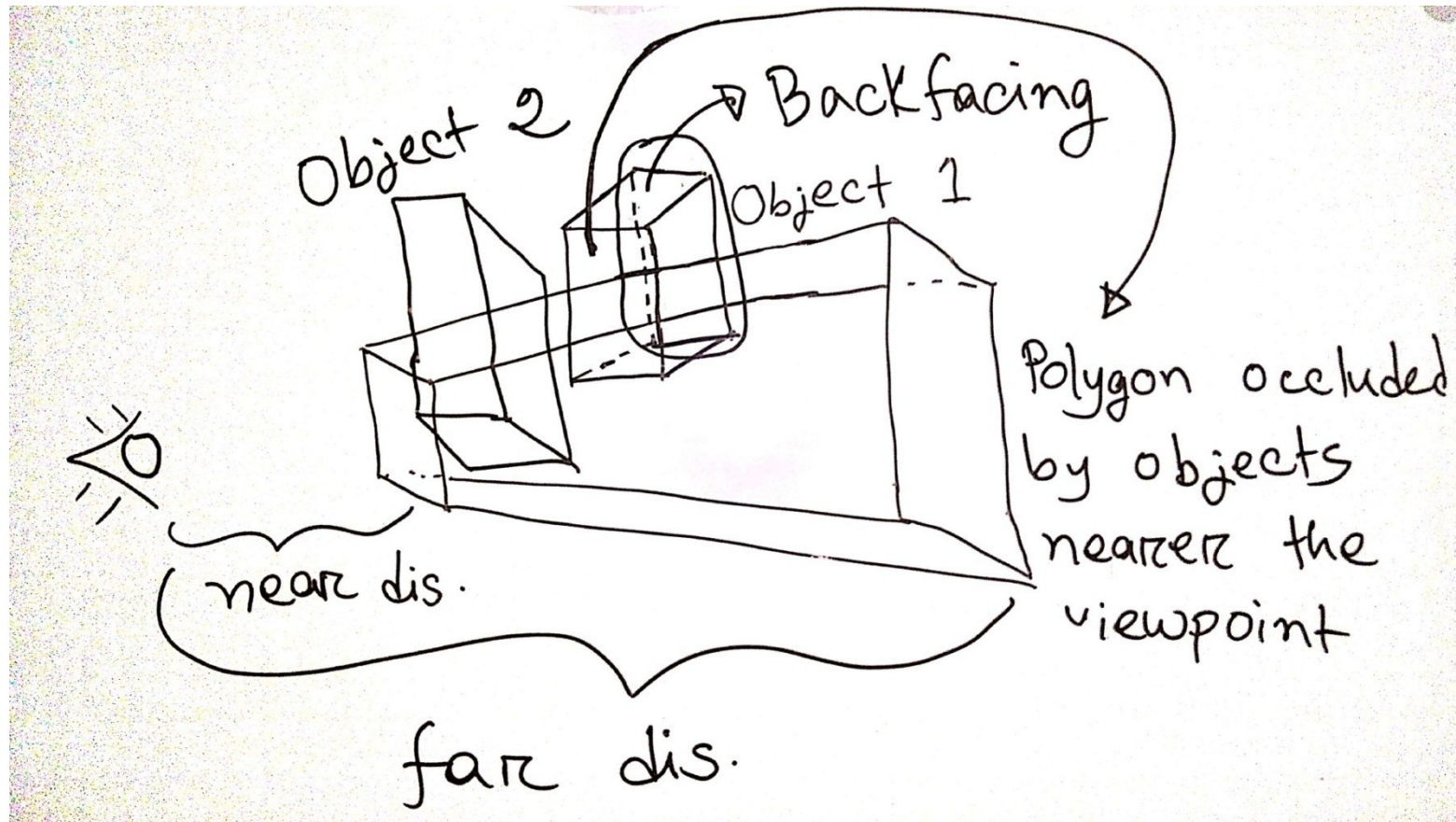
- Why might a polygon be invisible?
 - Polygon is *backfacing*

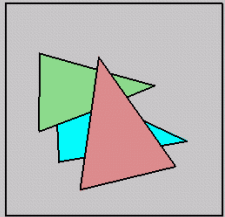




Invisible Primitives

- Why might a polygon be invisible?
 - Polygon is *occluded* by object(s) nearer to the viewpoint





Invisible Primitives

- Why might a polygon be invisible?
 - Polygon outside the *field of view*
 - Polygon is *backfacing*
 - Polygon is *occluded* by object(s) nearer to the viewpoint
[Figures HSR-IP-1,2,3,4]
- For efficiency reasons, we want to avoid spending work on polygons outside field of view or backfacing
- For efficiency and correctness reasons, we need to know when polygons are occluded

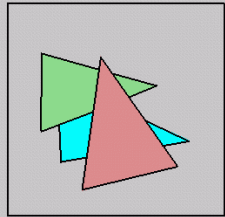
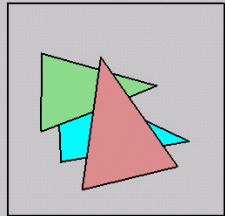


Image Precision

```
for each pixel in the image
    determine the closest object pierced by
        the ray through the pixel
    draw the pixel in the appropriate color
```

- Brute force approach:
 - For each pixel, examine all n objects, find which is closest
- $O(np)$ where $p = \text{\#pixels}$
- Needs to be redone if the image is resized



Object Precision

for each object in the world
 determine the parts of the object that
 are unobscured by other parts
 of it or other objects
 draw those parts in the appropriate color

Brute force approach:

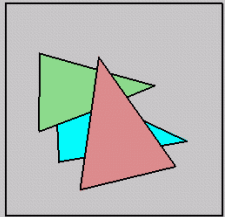
 For each object, examine all n objects

$O(n^2)$

$n \ll p$, so this seems good...

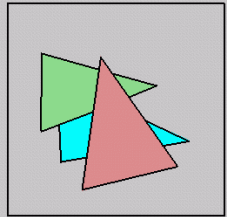
But each computation much more expensive

Tends to be slower in practice



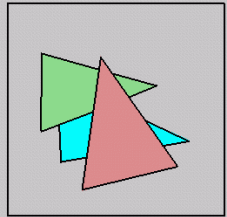
HSR Algo Classes

- Three classes of algorithms
 - “Conservative” visibility testing: only trivial reject – does not give final answer!
 - e.g., back-face culling, canonical view volume clipping, spatial subdivision
 - have to feed results to algorithms mentioned below
 - Image precision – resolve visibility at discrete points in image
 - sample model, then resolve visibility – i.e., figure out which objects it makes sense to compare with
 - e.g., ray tracing, or Z-buffer and scan-line algo.
 - Object precision – resolve for all possible view directions from a given eye point
 - irrespective of view direction or sampling density
 - resolve visibility exactly, then sample the results
 - e.g., 3-D depth sort, BSP trees



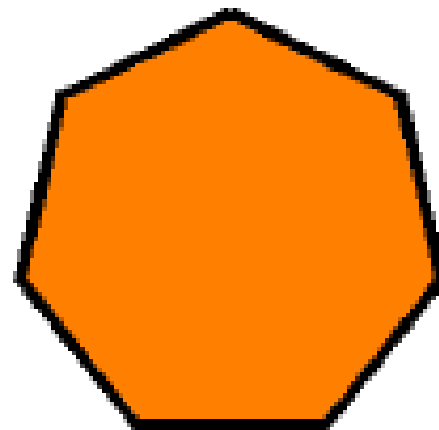
Algorithms to be Studied

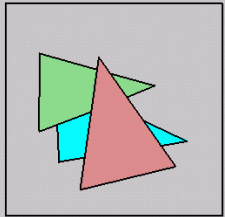
1. Back-Face Culling
2. Depth-Sort
3. Z-Buffer (Depth- Buffer)
4. BSP-Tree
5. Scanline Algorithm



Back-Face Culling

Back-face culling addressing a special case of occlusion called *convex self-occlusion*. Basically, if an object is closed (having a well defined inside and outside) then *some parts of the outer surface must be blocked by other parts of the same surface*. We'll be more precise with our definitions in a minute. On such surfaces we need only consider the normals of surface elements to determine if they are invisible.





Removing Back Faces

Idea: Compare the normal of each face with the viewing direction

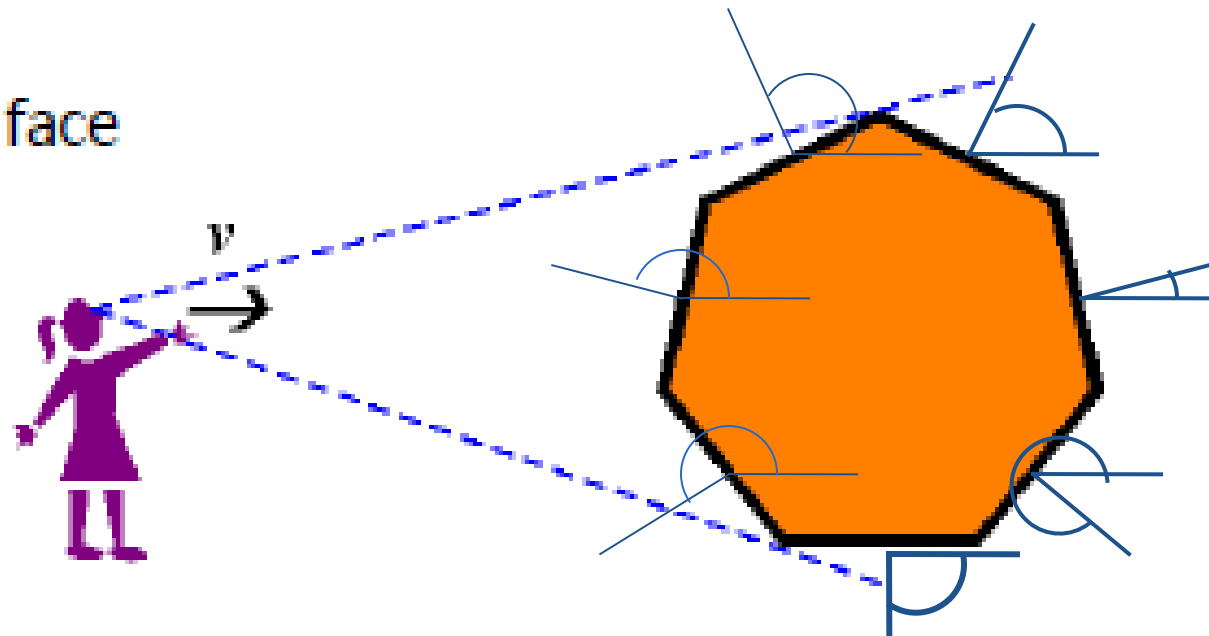
Given \mathbf{n} , the outward-pointing normal of F

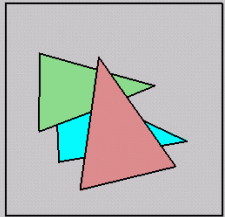
foreach face F of object

if $(\mathbf{n} \cdot \mathbf{v} > 0)$

throw away the face

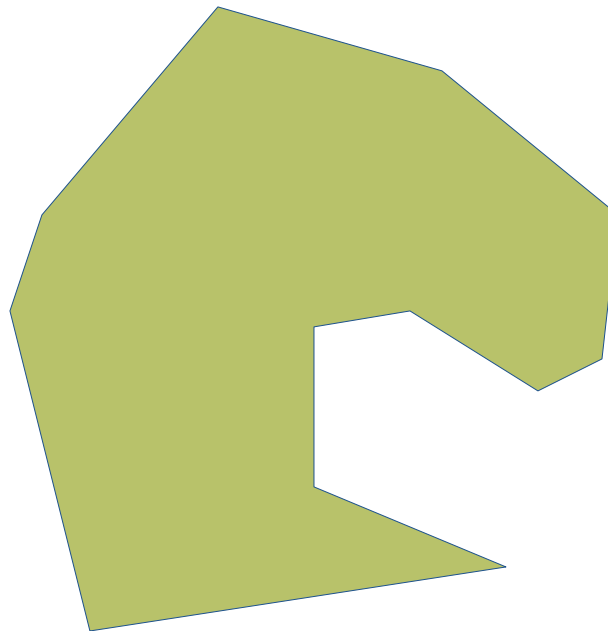
Does it work?

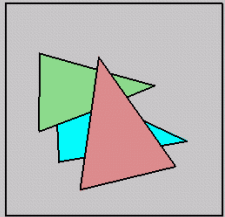




Back-Face Culling

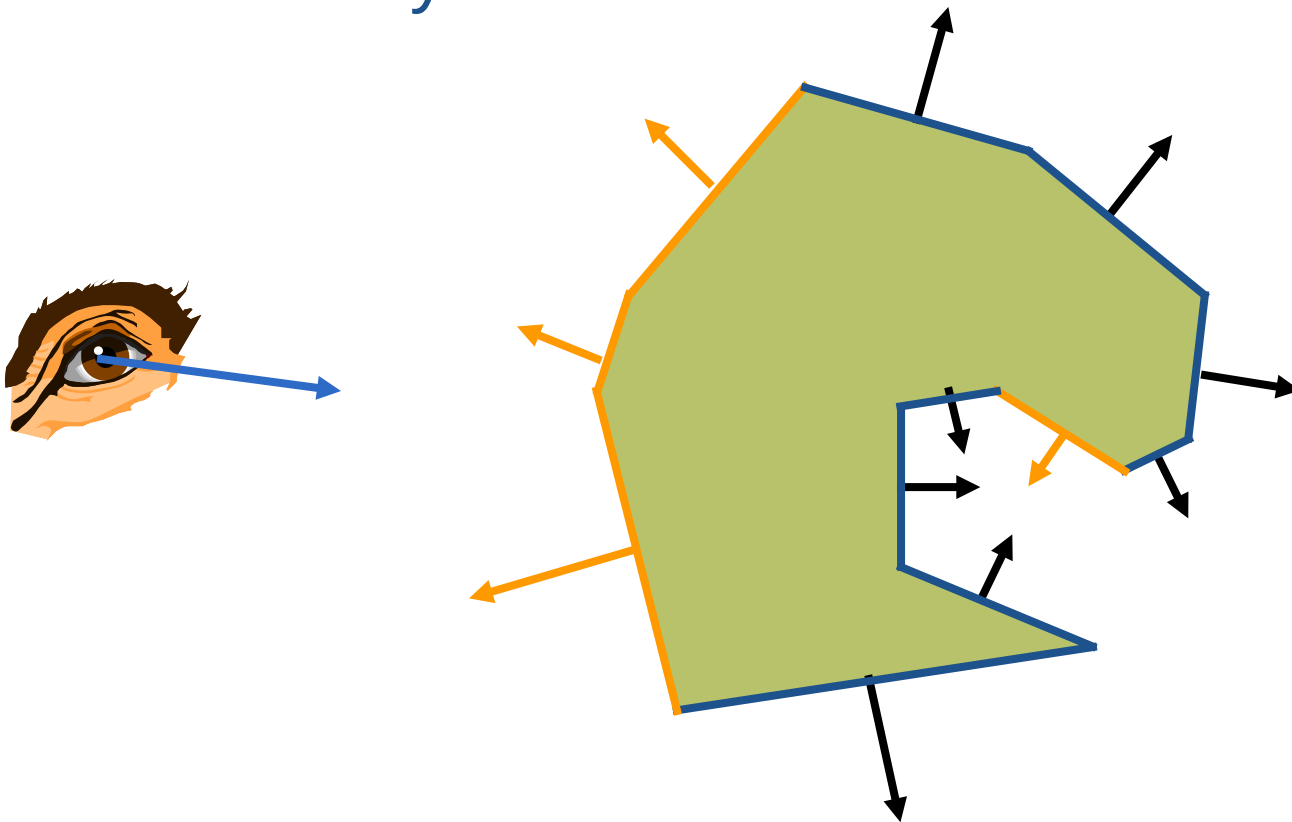
- On the surface of a closed manifold, polygons whose normals **point away from the camera** are always occluded:
Plane Normal \cdot (Point on plane - COP) $> 0 \Rightarrow$ Back face

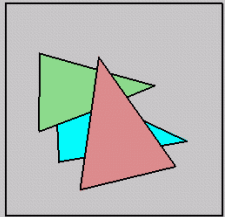




Back-Face Culling

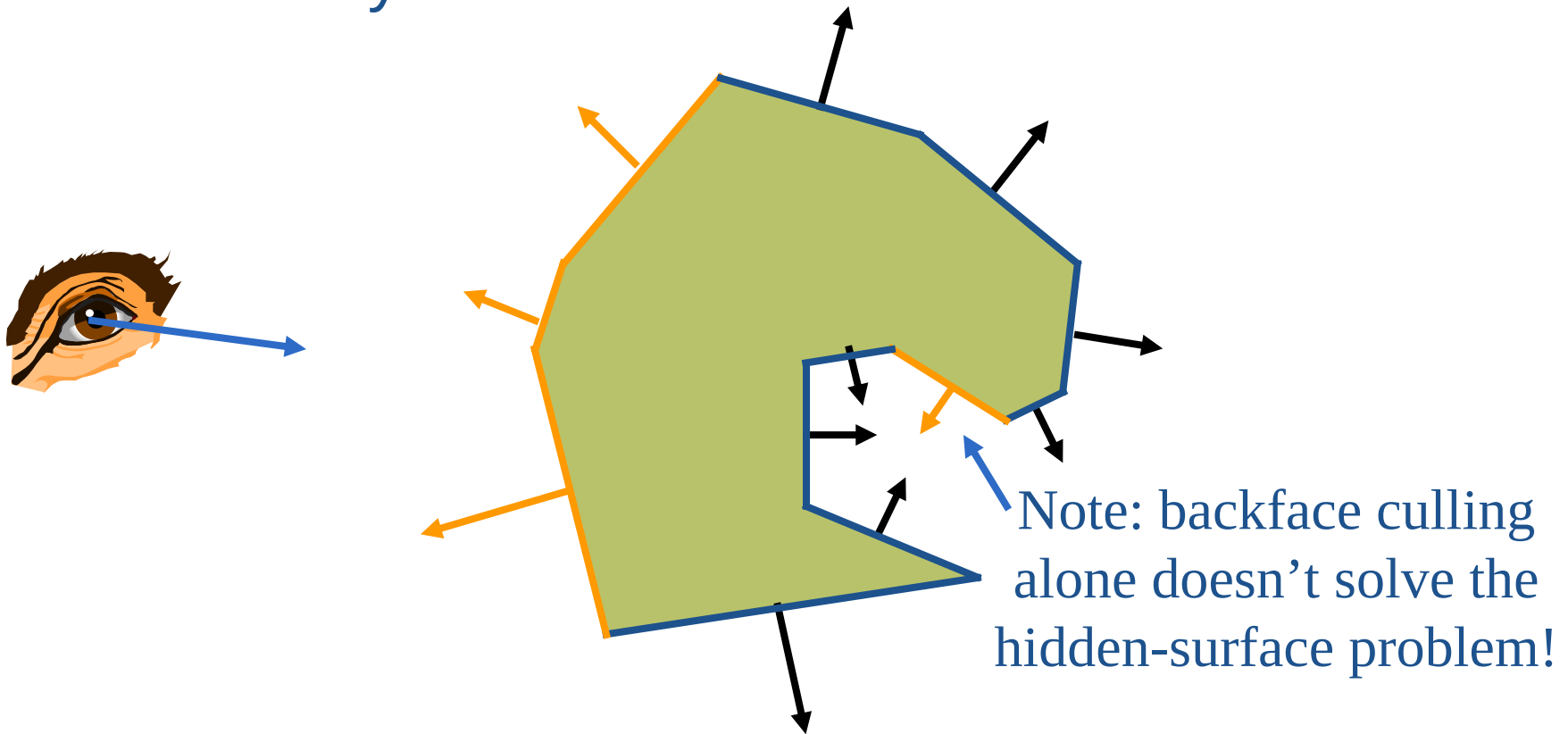
- On the surface of a closed manifold, polygons whose normals point away from the camera are always occluded:

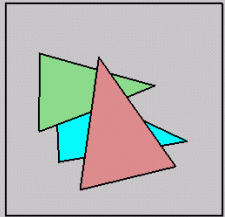




Back-Face Culling - Problems

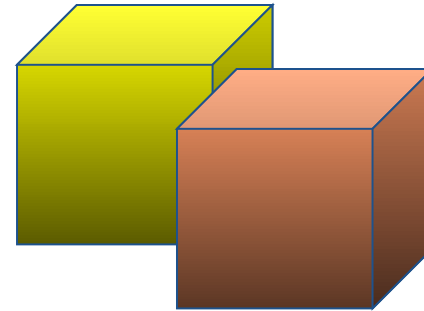
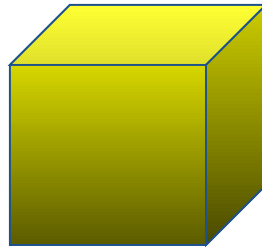
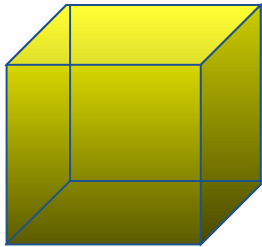
- On the surface of a closed manifold, polygons whose normals point away from the camera are always occluded:

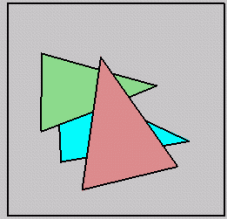




Back-Face Culling - Problems

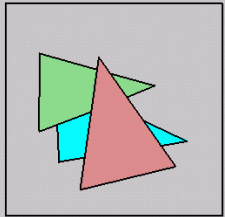
- One polyhedron may obscure another





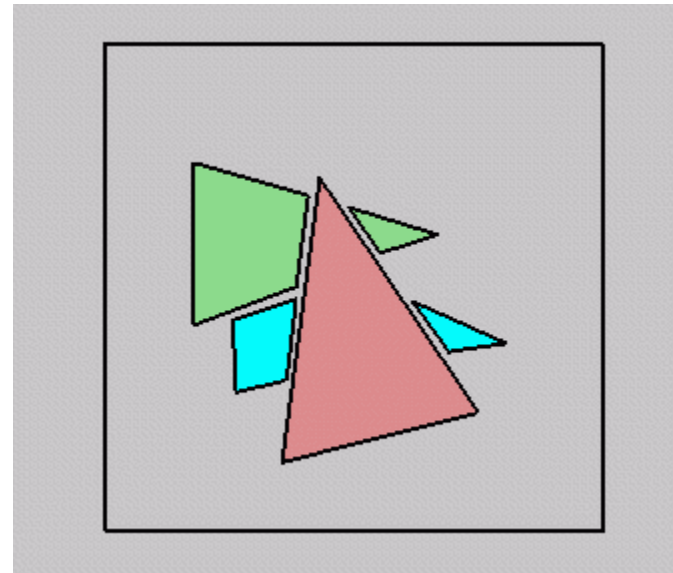
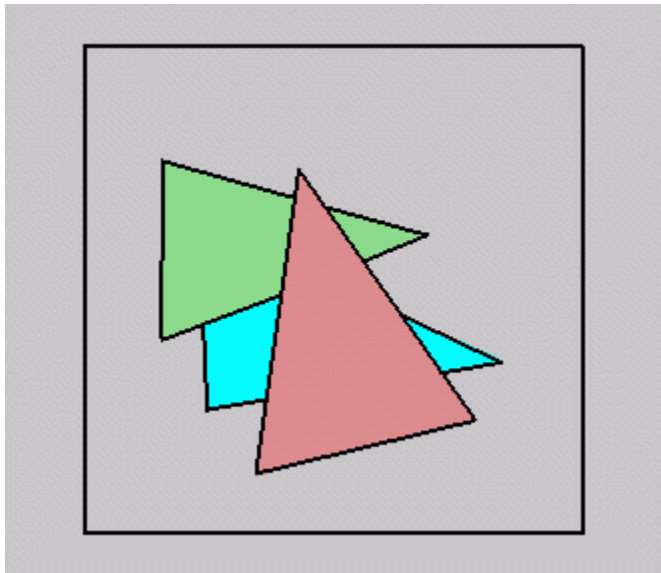
Back-Face Culling - Advantages

- On average, approximately one-half of a polyhedron's polygons are back-facing.
- Back-Face culling halves the number of polygons to be considered for each pixel in an image precision algorithm.

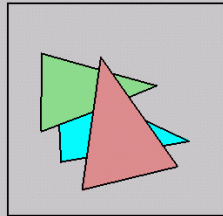


Occlusion

- For most interesting scenes, some polygons will overlap:



- To render the correct image, we need to determine which polygons *occlude* which



Painter's Algorithm

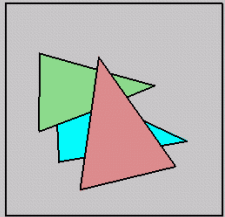
A Painter's Algorithm

The painter's algorithm, sometimes called depth-sorting, gets its name from the process which an artist renders a scene using oil paints. First, the artist will paint the background colors of the sky and ground. Next, the most distant objects are painted, then the nearer objects, and so forth. Note that oil paints are basically opaque, thus each sequential layer completely obscures the layer that it covers.

A very similar technique can be used for rendering objects in a three-dimensional scene. First, the list of surfaces are sorted according to their distance from the viewpoint. The objects are then painted from back-to-front.

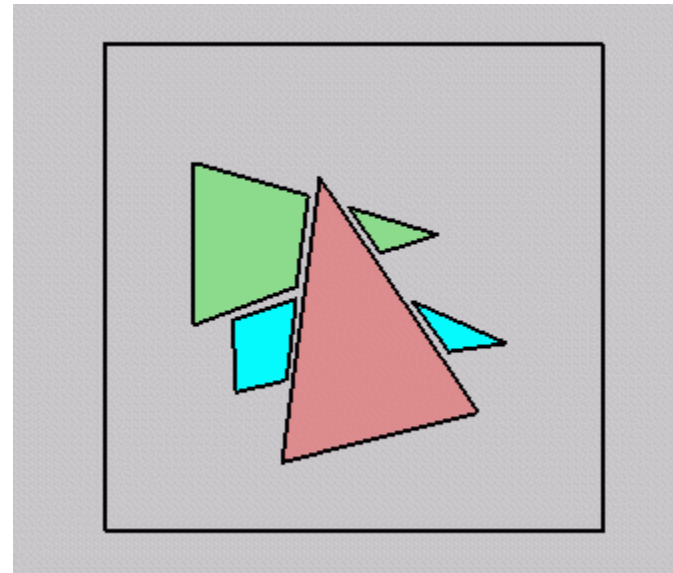
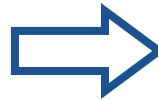
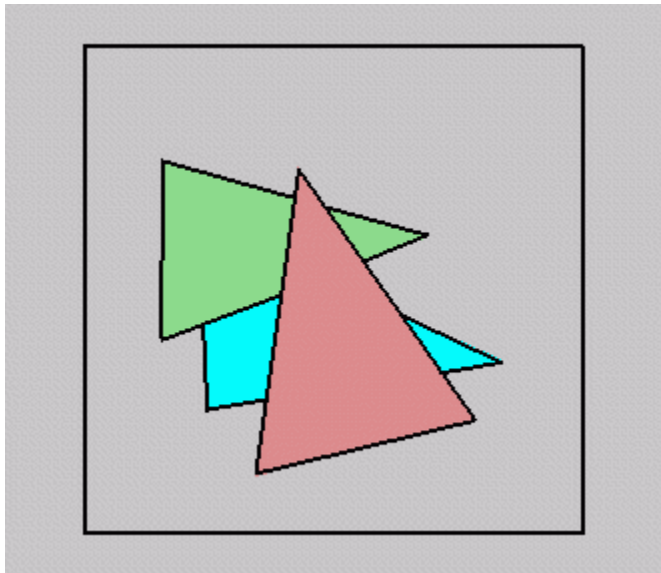
While this algorithm seems simple there are many subtleties. The first issue is which depth-value do you sort by? In general a primitive is not entirely at a single depth. Therefore, we must choose some point on the primitive to sort by.



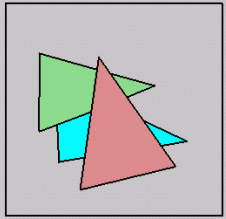


Painter's Algorithm

- Simple approach: render the polygons from back to front, “painting over” previous polygons:



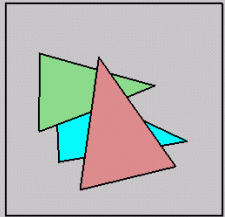
- Draw blue, then green, then pink
- *Will this work in general?*



Painter's Algorithm: Problems

- *Intersecting polygons* present a problem
- Even non-intersecting polygons can form a cycle with no valid visibility order:

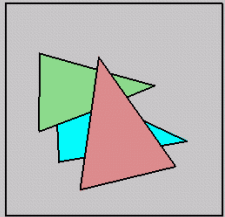




Painter's Algorithm: Problems

- *Intersecting polygons* present a problem
- Even non-intersecting polygons can form a cycle with no valid visibility order:

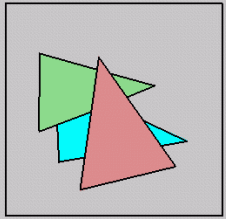




Painter's Algorithm: Problems

- *Intersecting polygons* present a problem
- Even non-intersecting polygons can form a cycle with no valid visibility order:

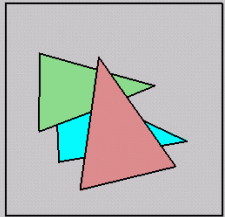




Painter's Algorithm: Problems

- *Intersecting polygons* present a problem
- Even non-intersecting polygons can form a cycle with no valid visibility order:





Z-Buffer Algorithm

○ Requires two “buffers”

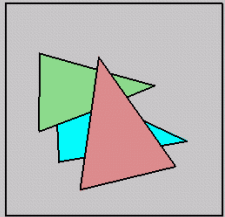
Intensity Buffer

- our familiar RGB pixel buffer
- initialized to background color

Depth (“Z”) Buffer

- depth of scene at each pixel
- initialized to far depth

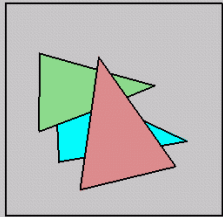
- ## ○ Polygons are scan-converted in arbitrary order.
- When pixels overlap, use Z-buffer to decide which polygon “gets” that pixel



Z-Buffer Algorithm

Algorithm:

- Initialize:
 - *Each z- buffer cell \leftarrow Max z value*
 - *Each frame buffer cell \leftarrow background color*
- *For each polygon:*
 - *Compute $z(x, y)$, polygon depth at the pixel (x, y)*
 - ***If** $z(x, y) < z$ buffer value at pixel (x, y) **then***
 - *$z\ buffer(x, y) \leftarrow z(x, y)$*
 - *$pixel(x, y) \leftarrow$ color of polygon at (x, y)*



Z-Buffer

255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255

+

127	127	127	127	127	127	127	
127	127	127	127	127	127		
127	127	127	127	127			
127	127	127	127				
127	127	127					
127	127						
127							
127							

=

127	127	127	127	127	127	127	255
127	127	127	127	127	127	255	255
127	127	127	127	127	255	255	255
127	127	127	127	255	255	255	255
127	127	127	255	255	255	255	255
127	127	255	255	255	255	255	255
127	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255

127	127	127	127	127	127	127	255
127	127	127	127	127	127	255	255
127	127	127	127	127	255	255	255
127	127	127	127	255	255	255	255
127	127	127	255	255	255	255	255
127	127	255	255	255	255	255	255
127	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255

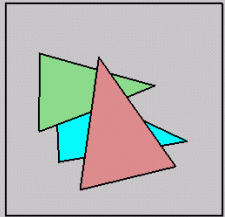
+

128							
127	127						
120	120	120					
115	115	115	115				
115	115	115	115	115			

=

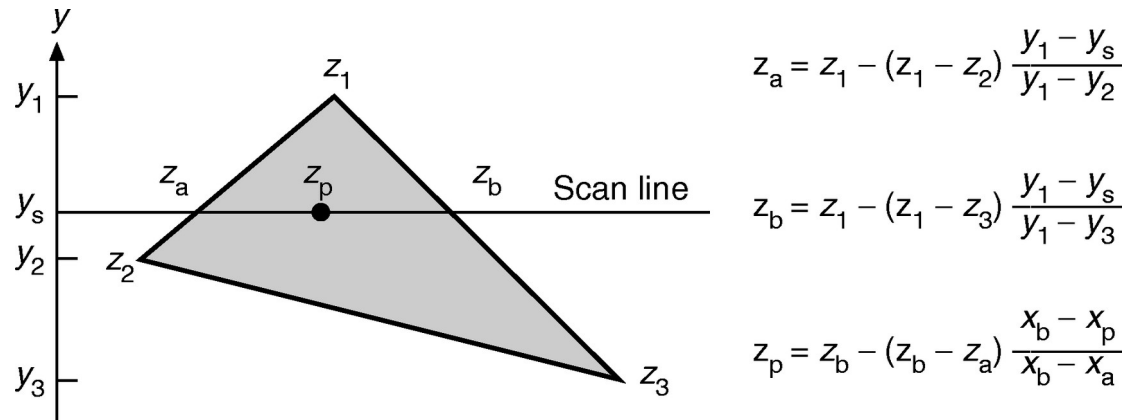
127	127	127	127	127	127	127	255
127	127	127	127	127	127	255	255
127	127	127	127	127	255	255	255
127	127	127	127	255	255	255	255
127	127	127	255	255	255	255	255
120	120	120	255	255	255	255	255
115	115	115	115	255	255	255	255
115	115	115	115	115	255	255	255

Above: example using integer Z-buffer with near = 0, far = 255



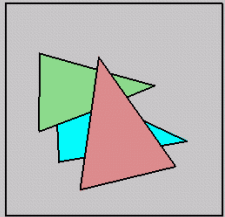
Computing Z values

- Interpolation along scan line
- Interpolation along edge
- How do we compute it efficiently?
 - Answer is simple: do it **incrementally**!
 - As we move along Y-axis, track x position where each edge intersects scan-line
 - do same thing for z coordinate using with y-z slope



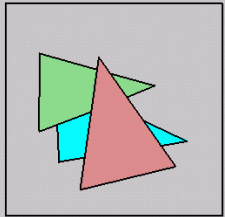
- Once we have z_a and z_b for each edge, can **incrementally calculate z_p as we scan**

* Something similar is done for calculating color per pixel in Gouraud shading



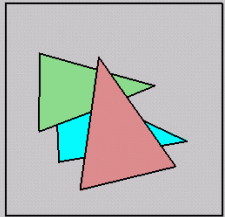
Z-Buffer: Advantages

- Simplicity lends itself well to hardware implementations:
FAST
 - used by graphics cards
- Polygons do not have to be compared in any particular order: no presorting in z necessary, big gain!
- Only consider one polygon at a time
- Z-buffer can be stored with an image; allows you to correctly composite multiple images (easy!) without having to merge models (hard!)
- Can be used for non-polygonal surfaces



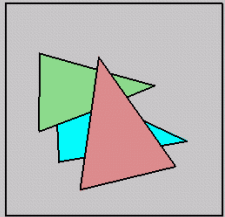
Z-Buffer: Disadvantages

- A pixel may be drawn many times
 - A rough front to back depth sort can be performed
- High amount of memory required due to Z buffer
 - Image can be scan converted in strips, at the expense of multiple passes over the objects
- Lower precision for higher depth
 - May lack enough precision for objects with millimeter detailed, positioned at kilometer apart
 - Shared edges of polyhedron where some visible pixels come from one polygon, and other come from neighbor



Scan-Line Algorithm

- Image precision algorithm
- Renders scene scan line by scan line
- Maintain various lists
 - Edge Table
 - Active Edge Table
 - Polygon Table
 - Active Polygon Table



Scan-Line Algorithm

▪ Edge Table (ET)

- Horizontal edges are ignored
- Sorted into buckets based on each edge's smaller y-coordinate
- Within buckets, edges are ordered by increasing x-coordinate of their lower end point

$x_{at\ y_{min}}$	y_{max}	Δx	ID	
-------------------	-----------	------------	------	--

$\Delta x = \Delta y / m \rightarrow$ used in stepping from one scan line to next
 $\Delta y = 1$

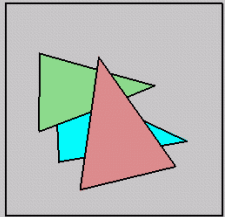
▪ Polygon Table (PT)

ID	Plane eqn.	Shading Info	in-out
------	------------	--------------	--------

Boolean Flag

Used during scan line processing

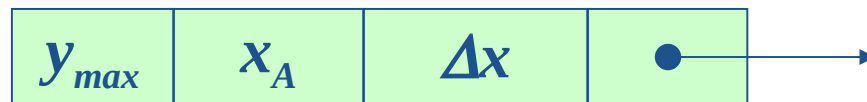
Initial value = 0



Scan-Line Algorithm

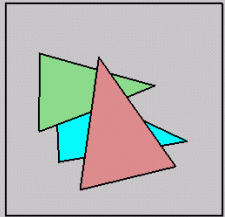
- **Active Edge Table (AET)**

- Stores the list of edges intersecting current scanline in increasing order of current x-coordinate

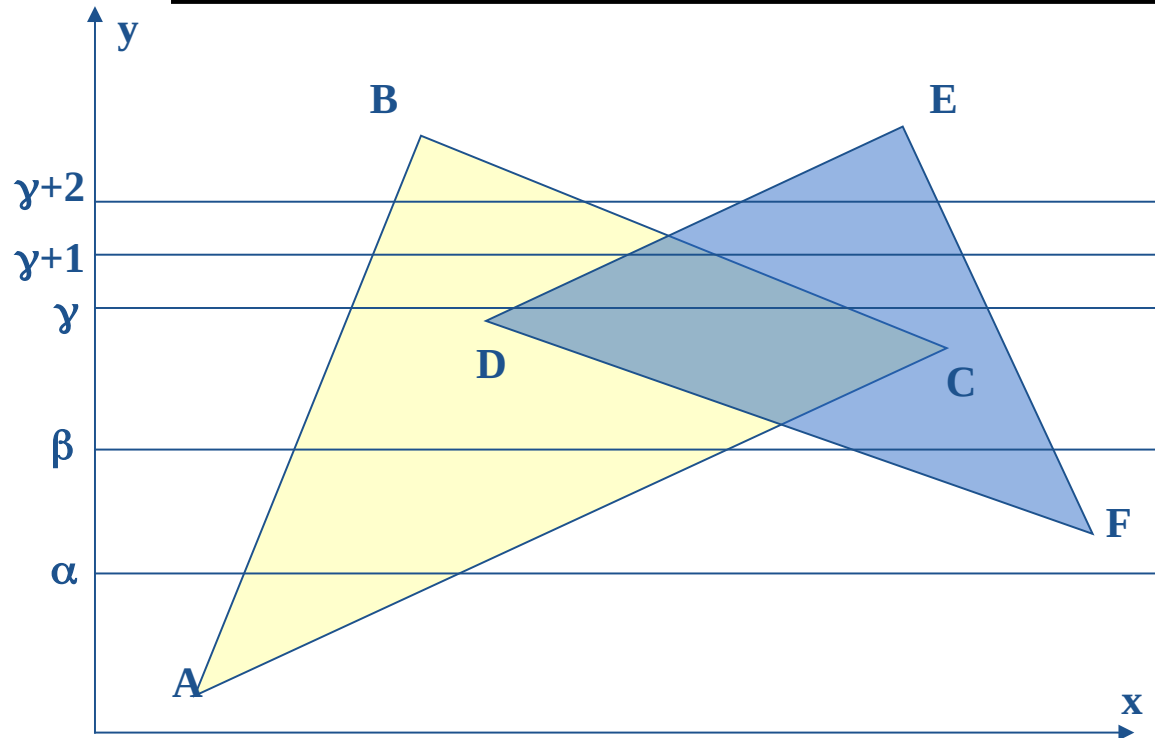


- *Active Polygon Table (APT)*

- *At each x -scan value this table contains the list of polygons whose in-out flag is set to true*



Scan-Line Algorithm



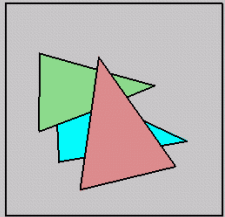
ET contents

AB AC DF EF CB DE

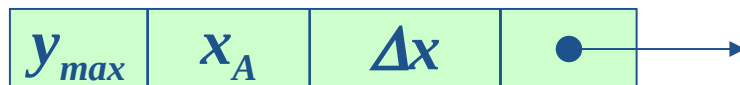
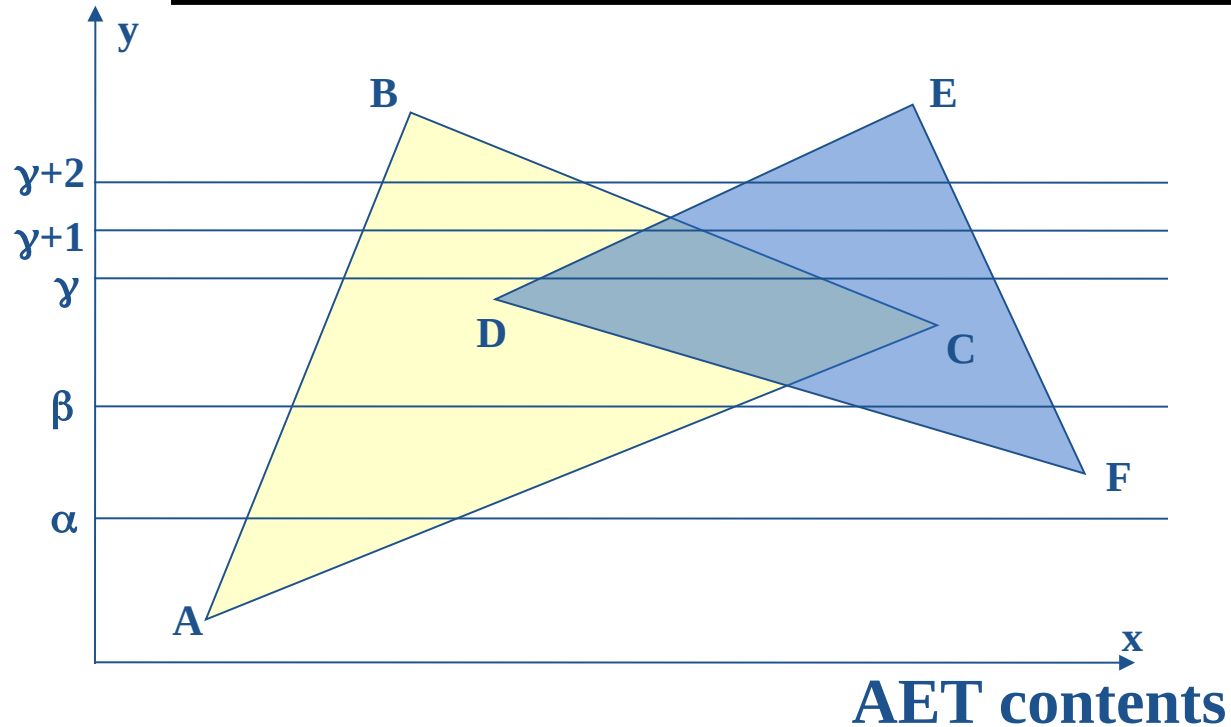
PT contents

ABC

DEF



Scan-Line Algorithm



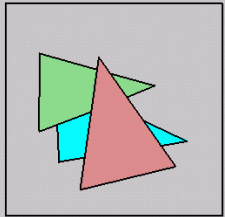
Scan line Entries

α AB AC

β AB AC FD FE

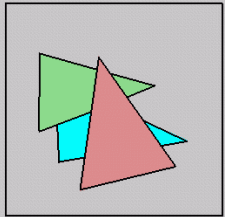
$\gamma, \gamma+1$ AB DE CB FE

$\gamma+2$ AB CB DE FE



Scan-Line Algorithm

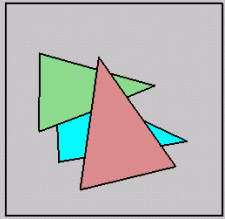
- Initialization
 - Initialize the AET to empty
 - Initialize each screen pixel to background-color
 - Set y to the first nonempty cell value in edge table



Scan-Line Algorithm

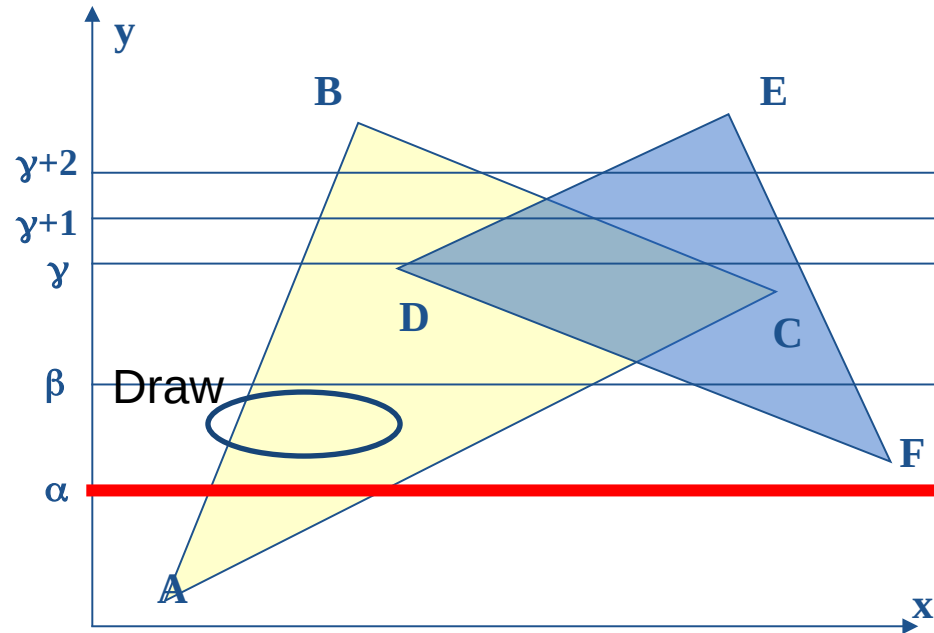
- **For each scan line y do**
 - $AET \leftarrow AET \cup$ Edges from ET that are in current scan-line
 - sort AET in order of increasing x_A
 - **For each edge e (except last one) in AET do**
 - invert in-out flag of the polygon that contains e
 - Update APT
 - Determine polygon p in APT with smallest z value at $(e.x_A, y)$
 - The pixels from e upto next edge in AET are set to the color of p
 - $AET \leftarrow AET -$ Edges from ET with
$$y_{\max} = y$$
 - **for each edge e in AET do**
$$e.x_A = e.x_A + e.\Delta x$$
 - Sort AET on x

Simulation – Scan-Line Algorithm



Toggle:
ABC.inout=0→1

Toggle:
ABC.inout=1→0



Draw up to
next edge in
AET

AET

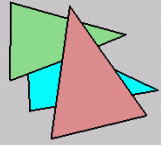
AB

AC

APT

ABC

Simulation – Scan-Line Algorithm

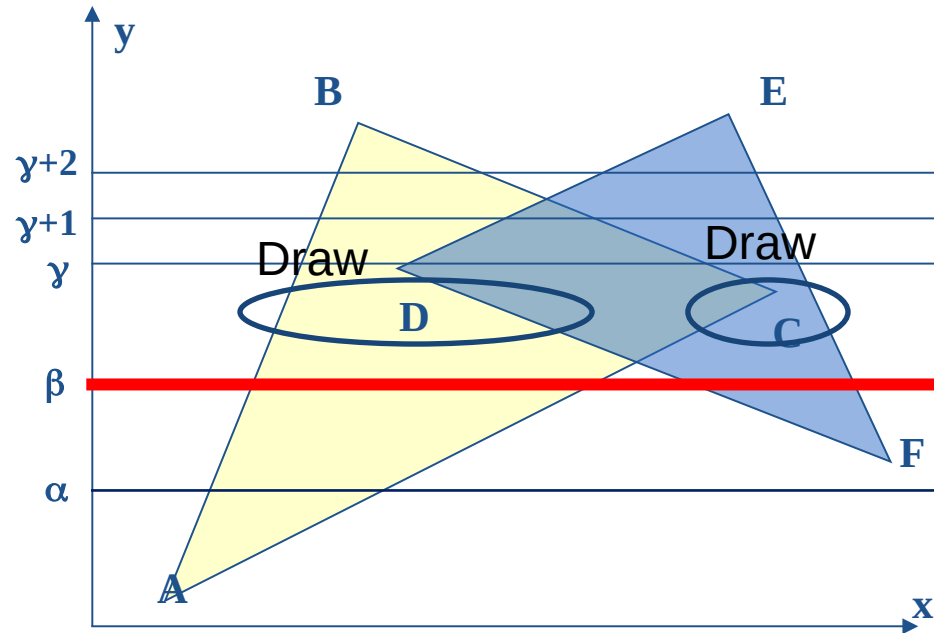


Toggle:
ABC.inout=0→1

Toggle:
ABC.inout=1→0

Toggle:
DEF.inout=0→1

Toggle:
DEF.inout=1→0



AET

AB

AC

DF

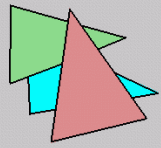
EF

APT

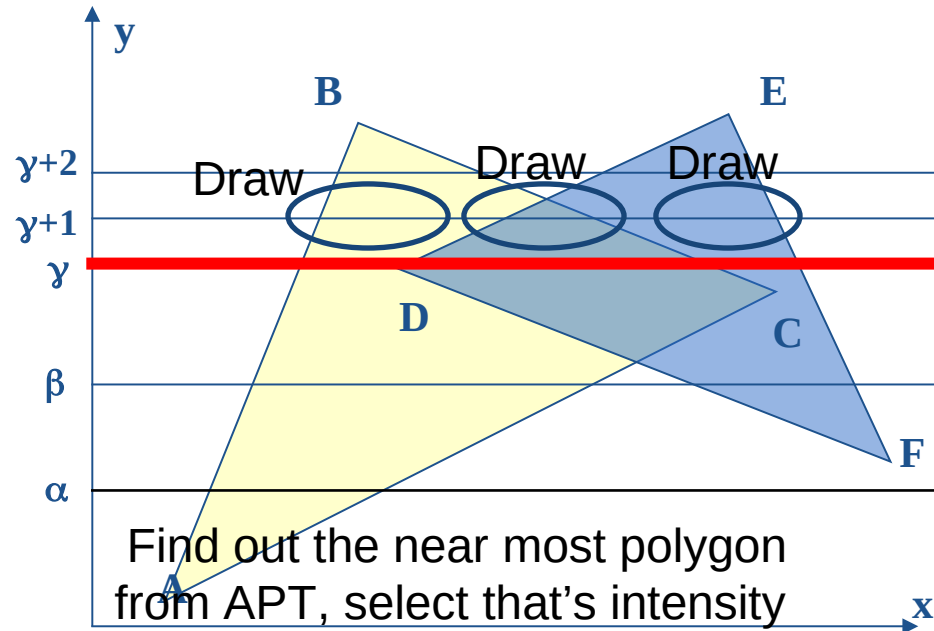
ABC

DEF

Simulation – Scan-Line Algorithm



Toggle:
ABC.inout=0→1
Toggle:
DEF.inout=0→1
Toggle:
ABC.inout=1→0
Toggle:
DEF.inout=1→0



Draw up to
next edge in
AET

Draw with
selected
intensity up to
next edge in
AET

AET

AB

DE

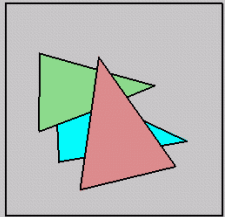
BC

EF

APT

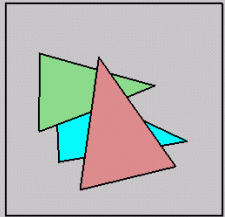
ABC

DEF

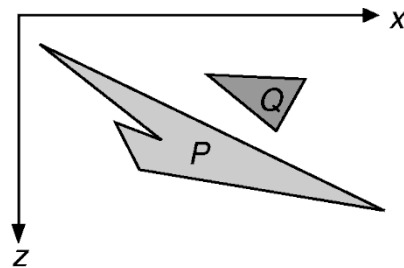


List-Priority Algorithms

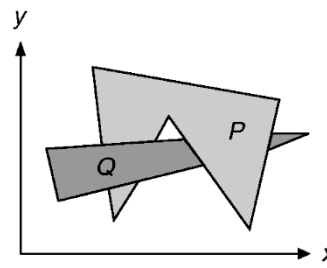
- Determine a visibility ordering for objects
 - No overlap in z ➡ Only sort by increasing z
 - Overlap in z
 - May determine a correct order with some tests
 - May not be a correct order which may require splitting of polygons to determine the order
- Depth comparison + object splitting with
Object precision, Scan conversion with
Image precision
- Differ in determining the order, splitting



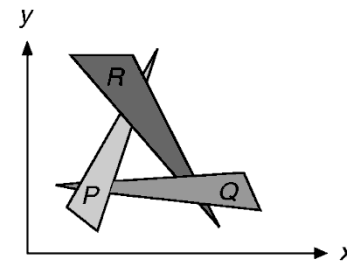
Depth-Sort Algorithm



(a)

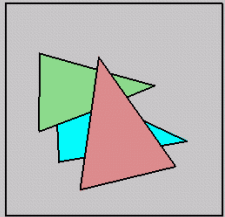


(b)



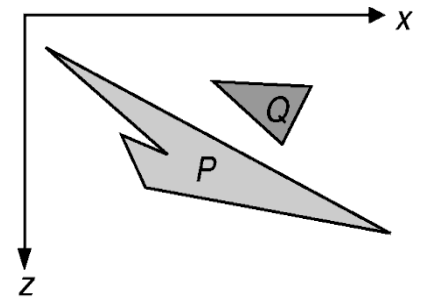
(c)

1. Sort all polygons according to the smallest (farthest) z coordinate of each
2. Resolve any ambiguities this may cause when the polygons' z extents overlap, splitting polygons if necessary
3. Scan convert each polygon in ascending order of smallest z coordinate (i.e., back to front).

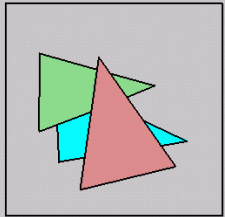


Depth-Sort Algorithm

- Select $P \rightarrow$ Polygon at the far end of the sorted list
- Before scan converting P ,
 - Test against each polygon Q with which it's z-extent overlaps
 - If P cannot obscure any such Q , then P can be scan converted
- 5 tests are performed in successive order



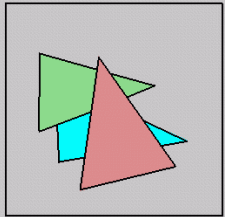
(a)



Depth-Sort Algorithm

- Checking 5 tests to see whether polygon P obscures polygon Q (overlaps in z-extent)
 1. Are the x-extents of P and Q disjoint?
 2. Are the y-extents of P and Q disjoint?
 3. Is P entirely on the opposite side of Q's plane from the eye?
 4. Is Q entirely on the same side of P's plane as the eye?
 5. Are the projections of P and Q on the screen are disjoint?

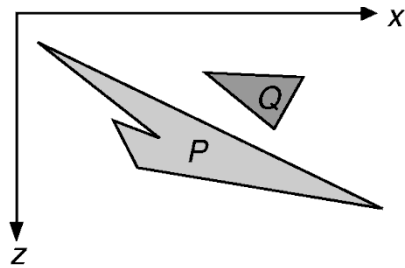
* As soon as a test succeeds, P has been shown not to obscure Q



Depth-Sort Algorithm

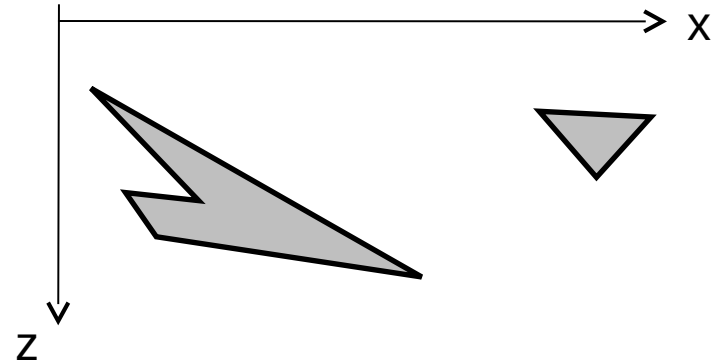
- Checking 5 tests to see whether polygon P obscures polygon Q

1. Are the x-extents of P and Q disjoint?



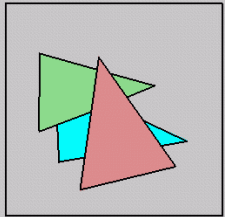
(a)

NO



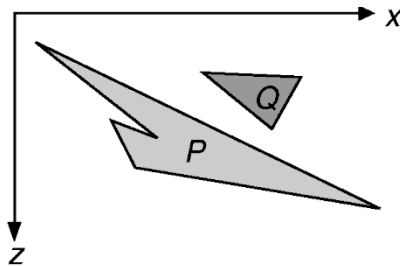
YES

2. Are the y-extents of P and Q disjoint?



Depth-Sort Algorithm

- Checking 5 tests to see whether polygon P obscures polygon Q
 - Is P entirely on the opposite side of Q's plane from the viewpoint?



(a)
NO

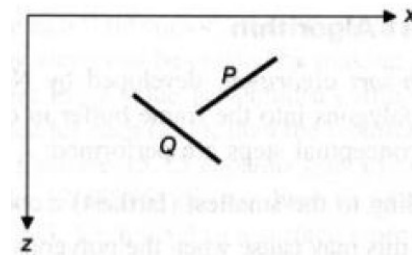


Fig. 15.25 Test 3 is true.

YES

- Is Q entirely on the same side of P's plane as the eye?

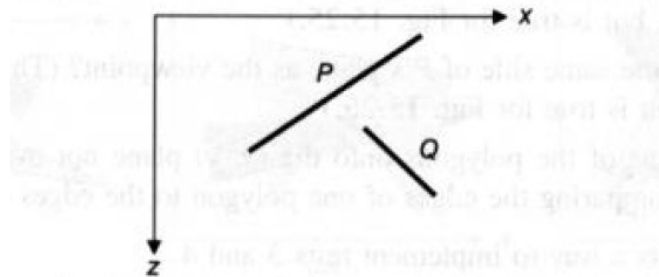
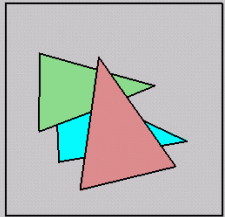


Fig. 15.26 Test 3 is false, but test 4 is true.

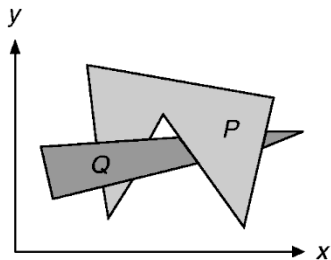
YES



Depth-Sort Algorithm

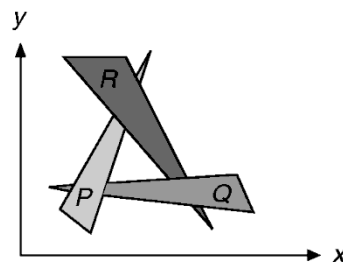
- Checking 5 tests to see whether polygon P obscures polygon Q

5. Are the projections of P and Q on the screen disjoint?

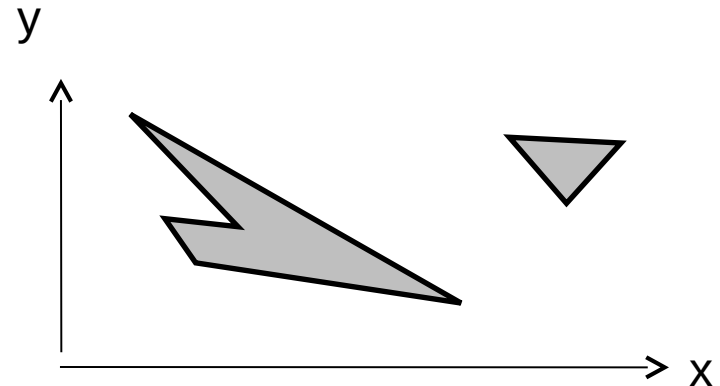


(b)

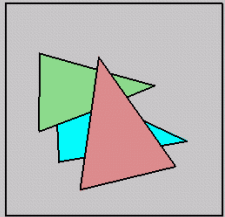
NO



(c)

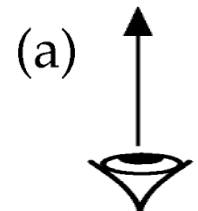
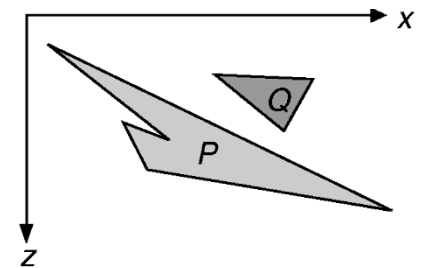


YES

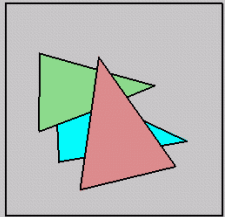


Depth-Sort Algorithm

- Checking 5 tests to see whether polygon P obscures polygon Q
- As soon as a test returns TRUE \rightarrow P does not obscure Q
 - So P can be scan converted before Q
 - If there are other polygons whose z extent overlaps P, then repeat the 5 tests for those also
- If all tests return FALSE \rightarrow P obscures Q
 - Can we scan convert Q before P?
 - 3'. Is Q entirely on the opposite side of P's plane from the eye?
 - 4'. Is P entirely on the same side of Q's plane as the eye?
 - If returns YES \rightarrow Q becomes the new P and included at the end of the list

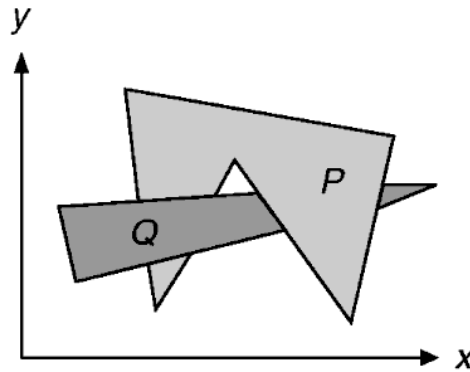


YES



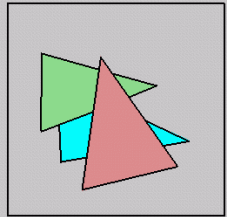
Depth-Sort Algorithm

- What if both tests 3' and 4' return NO ?
 - Q cannot totally obscure P



(b)

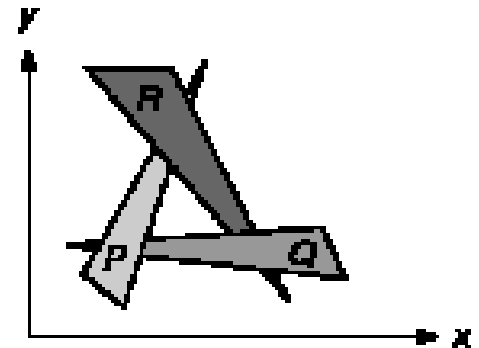
- So SPLIT (since none obscure other fully)
 - Original unsplit polygons are discarded
 - Pieces are inserted into the list in proper z order

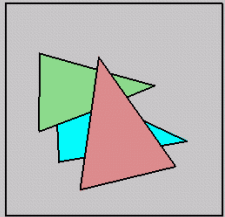


Depth-Sort Algorithm

Special Case:

- $P, Q, R \rightarrow$ any one of them can be inserted at the end of the list to place it in correct order relative to one other polygon, but not two others \rightarrow results in infinite loop
- Solution:
 - mark the polygon once it is moved to the end of the list
 - If 5 tests return NO
 - do not try steps 3' and 4'
 - Split the polygons and insert the pieces in to the list

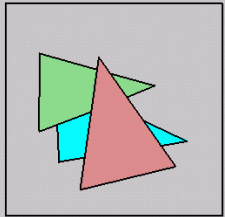




Depth-Sort Algorithm

- Advantages:
 - Fast enough for simple scenes
 - Fairly intuitive

- Disadvantages
 - Slow for even moderately complex scenes
 - Hard to implement and debug
 - Lots of special cases
 - May cause unnecessary split



Depth-Sort Algorithm

Unnecessary Split:

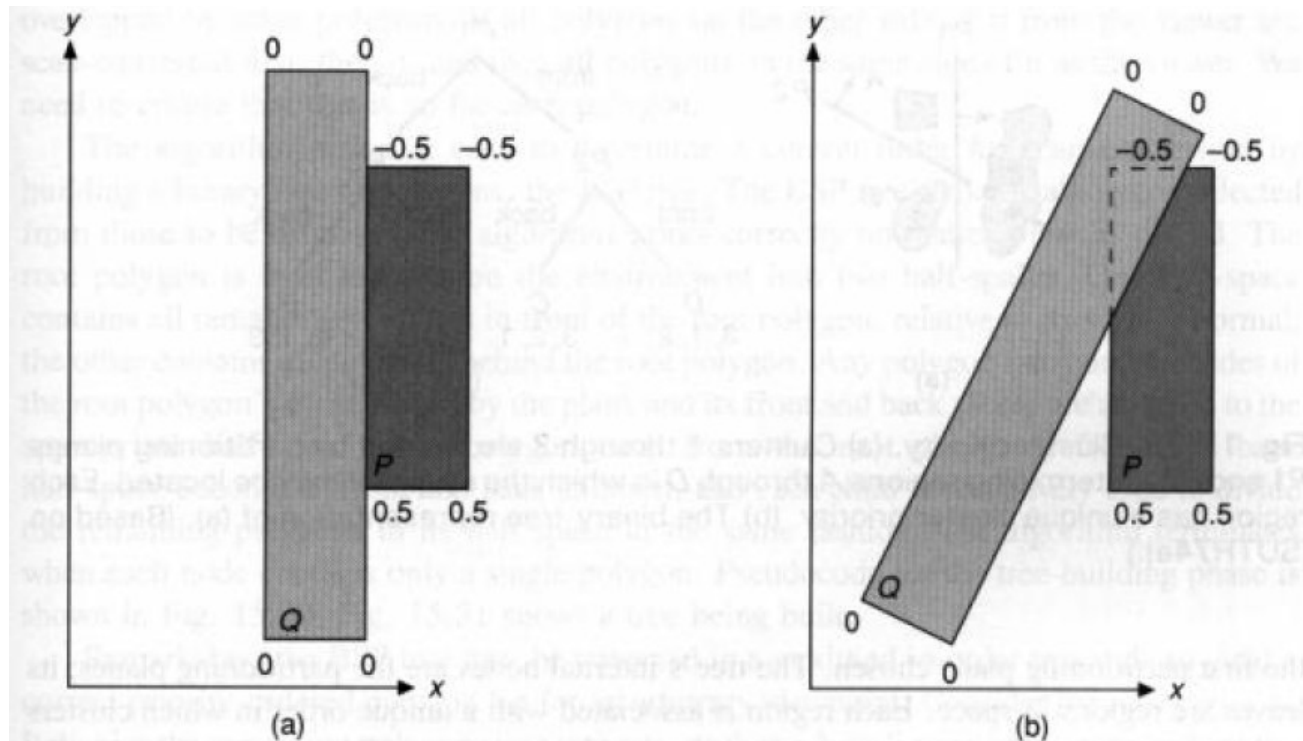
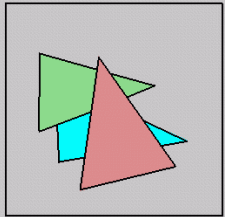
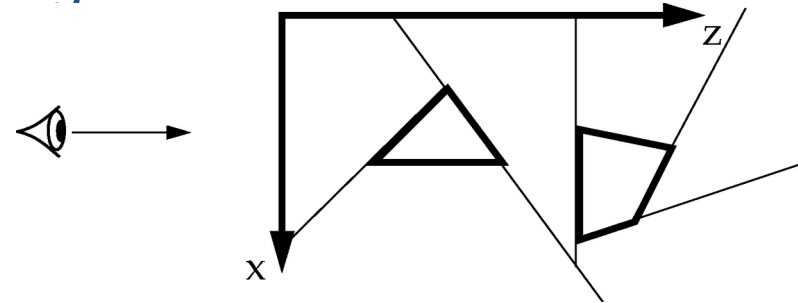


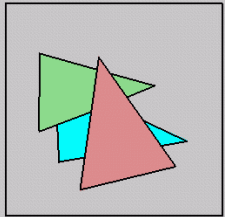
Fig. 15.27 Correctly ordered polygons may be split by the depth-sort algorithm. Polygon vertices are labeled with their z values. (a) Polygons P and Q are scan-converted without splitting. (b) Polygons P and Q fail all five tests even though they are correctly ordered.



BSP Tree

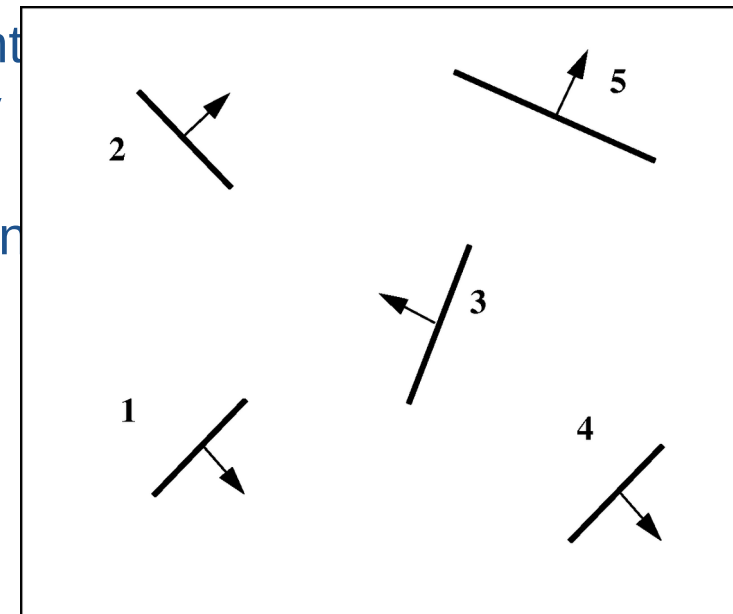
- BSP = Binary Space Partitioning
- Split space with any line (2D) or plane (3D)
- Repetitive subdivision of space to give a draw order
- Divide and conquer:
 - Select any polygon as root
 - Then display all polygons on “far” (relative to viewpoint) side of that polygon, then that polygon, then all polygons on polygon’s “near” side
- Runs an initial time and space intensive preprocessing step
 - View independent subdivision of the space
- Allows a linear time display algorithm that is executed whenever viewing direction change
- Suitable for cases where the view point changes but objects remain static



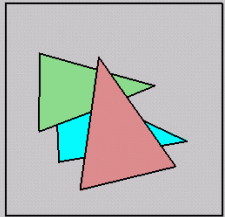


BSP Tree

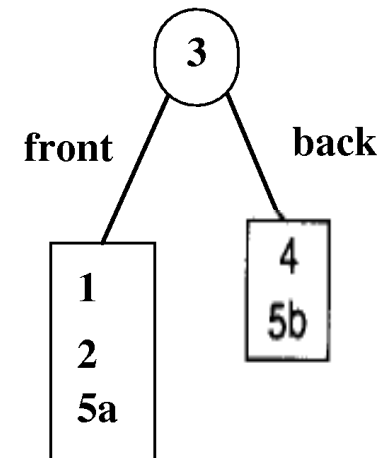
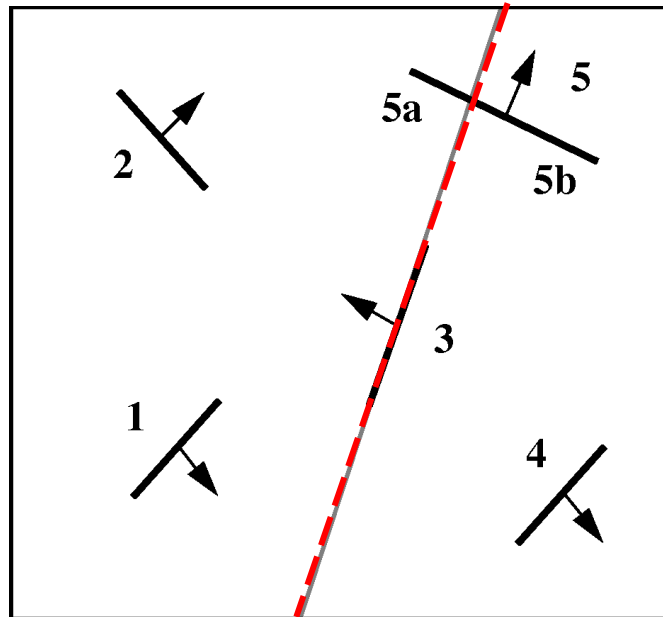
- Perform view-independent step once each time scene changes:
 - recursively subdivide environment into a hierarchy of half-spaces by dividing polygons in a half-space by the plane of a selected polygon
 - build a BSP tree representing this hierarchy
 - each selected polygon is the root of a sub-tree



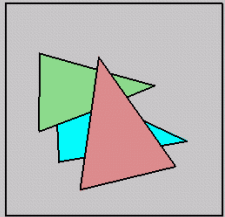
Initial Scene



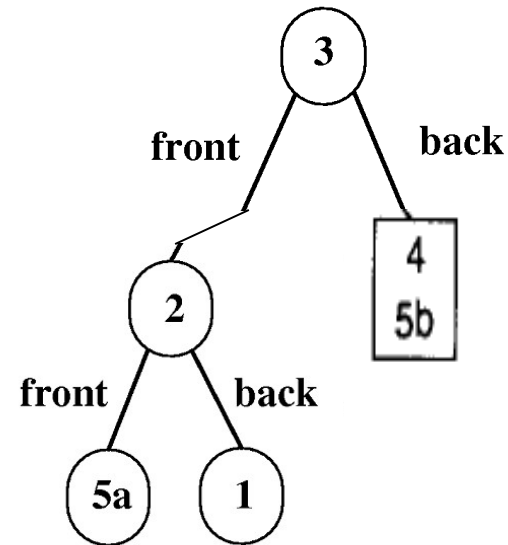
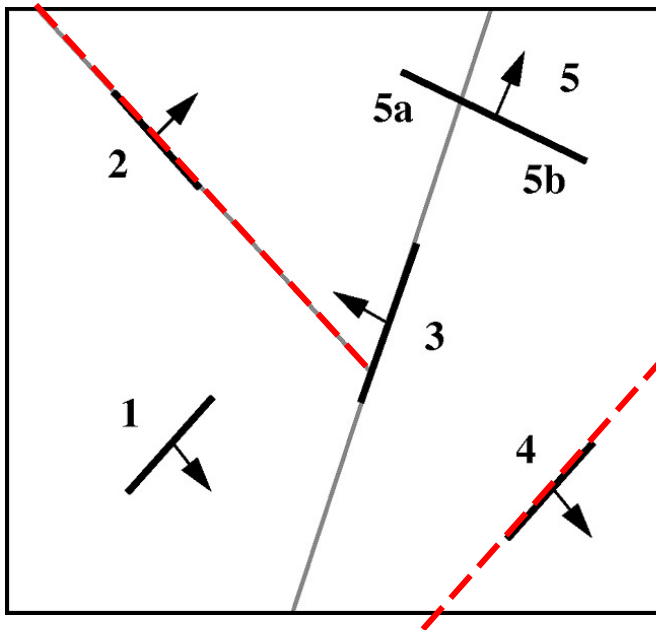
BSP Tree



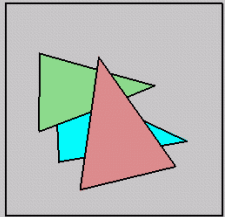
Step-1: Choose any polygon (e.g., polygon 3) and subdivide others by its plane, splitting polygons when necessary



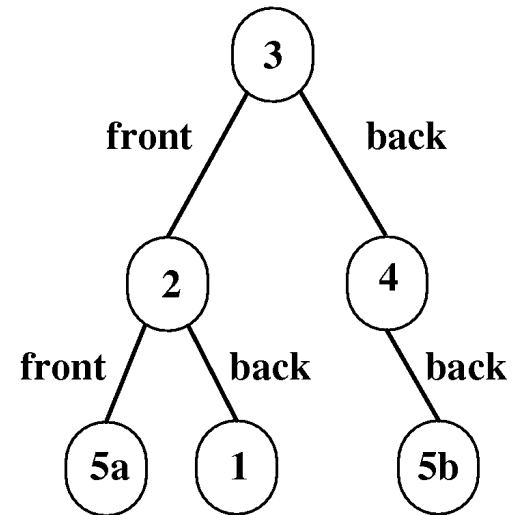
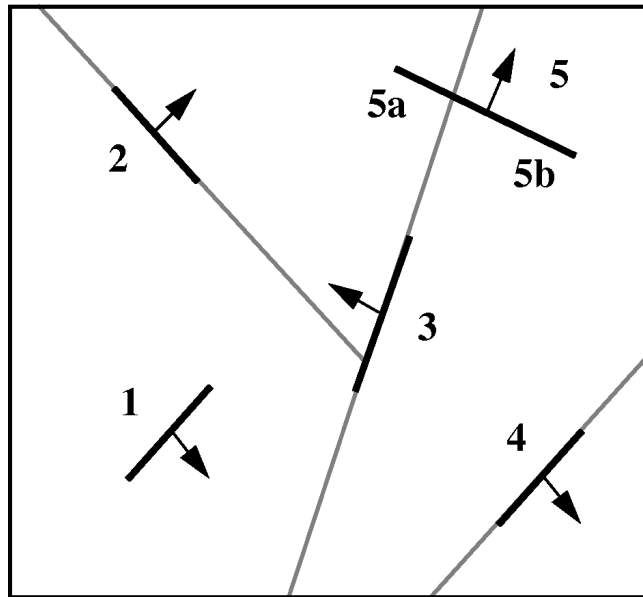
BSP Tree



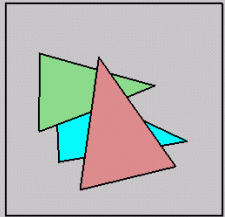
Step-2: Process front sub-tree recursively



BSP Tree

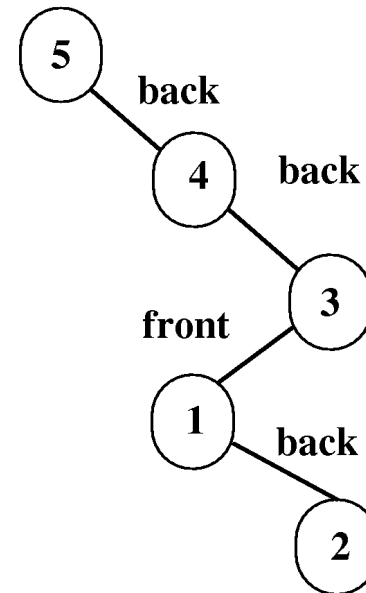
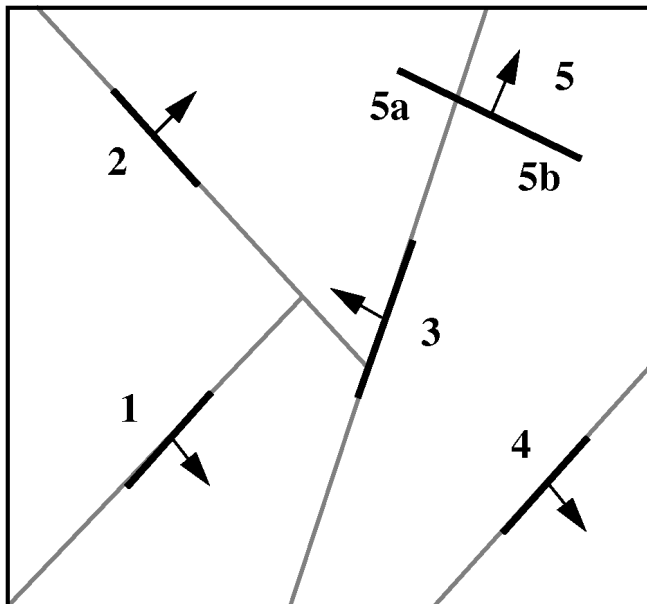


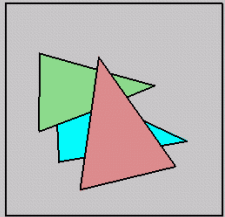
Step-3: Process back sub-tree recursively



BSP Tree

An alternative BSP tree with polygon 5 at the root





Building BSP Tree

■ Pseudocode - Textbook Fig 15.30

```
typedef struct {
```

```
    polygon root;
```

```
    BSP_tree *backChild, *frontChild;
```

```
} BSP_tree;
```

```
BSP_tree *BSP_makeTree(polygon *polylist)
```

```
    polygon root, *backList, *frontList, p, backPart, frontPart;
```

```
    if (polyList == NULL) return NULL;
```

```
    root = BSP_selectAndRemovePoly(&polyList);
```

```
    backList = frontList = NULL;
```

```
    for (each remaining polygon p in polylist)
```

```
        if (polygon p in front of root)
```

```
            BSP_addToList (p, &frontList);
```

```
        else if (polygon p in back of root)
```

```
            BSP_addToList (p, &backList);
```

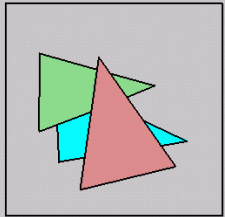
```
        else
```

```
            BSP_splitPoly(p, root, &frontPart, &backPart);
```

```
            BSP_addToList(frontPart, &frontList);
```

```
            BSP_addToList(backPart, &backList);
```

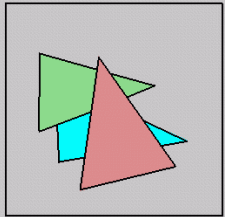
```
    return BSP_combineTree(BSP_makeTree(frontList), root, BSP_makeTree(backList))
```



Displaying BSP Tree

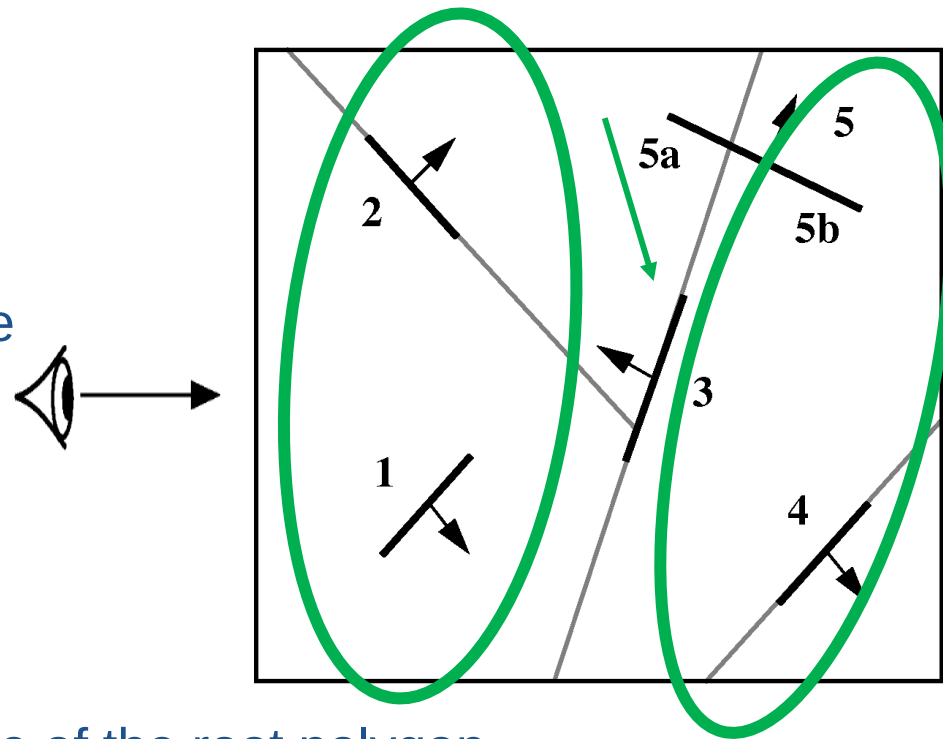
- Pseudocode - Textbook Fig 15.32

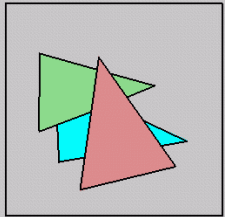
```
void BSP_displayTree(BSP_tree *tree) {  
    if (tree != NULL) {  
        if (viewer is in front of tree->root) {  
            /*Display back child, root, and front child */  
            BSP_displayTree(tree->backChild);  
            displayPolygon(tree->root);  
            BSP_displayTree(tree->frontChild);  
        } else {  
            /*Display front child, root, and back child */  
            BSP_displayTree(tree->frontChild);  
            displayPolygon(tree->root);  
            BSP_displayTree(tree->backChild);  
        }  
    }  
}
```



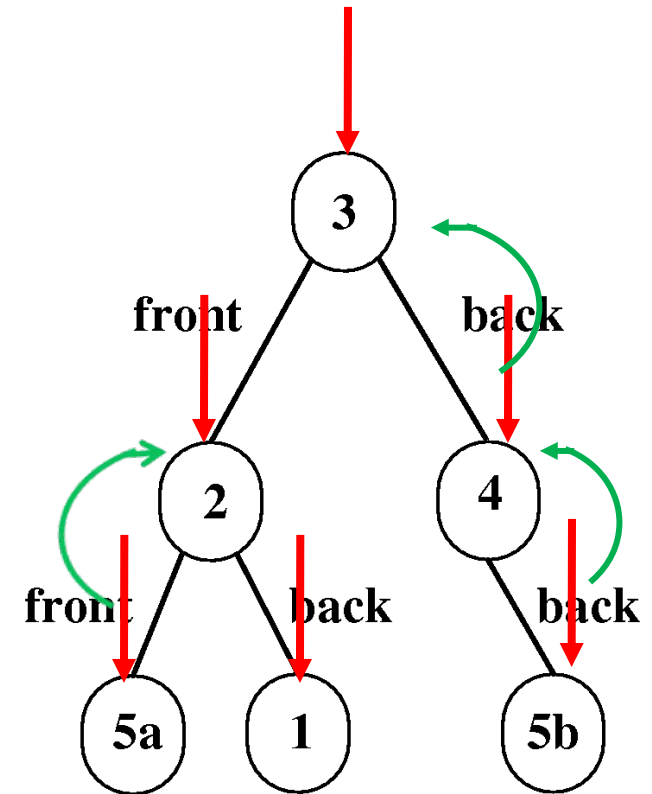
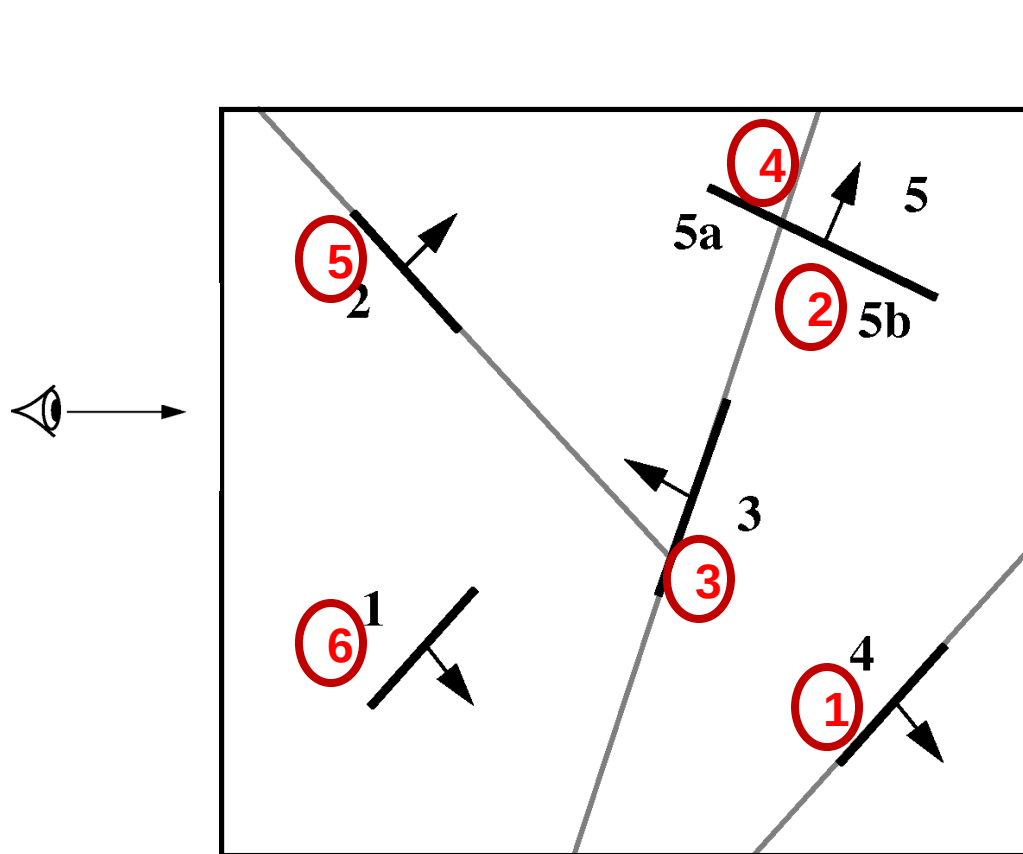
Displaying BSP Tree

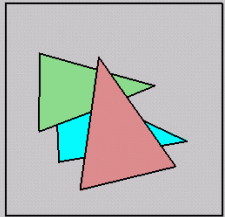
- Given a viewpoint, apply modified in-order tree traversal
- If viewer is in the root polygon's front half space
 - Display all the polygons in the root's rear half space
 - Then display the root
 - Finally display the polygons in its front half space
- If viewer is in the root polygon's rear half space
 - Display all the polygons in the root's front half space
 - Then display the root
 - Finally display the polygons in its rear half space
- If viewer is looking along the edge of the root polygon
 - Follow any display order provided above



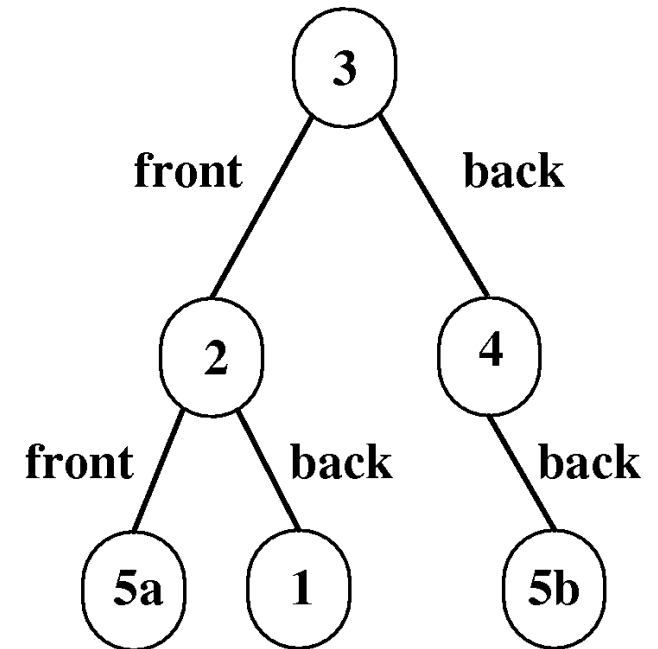
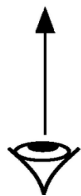
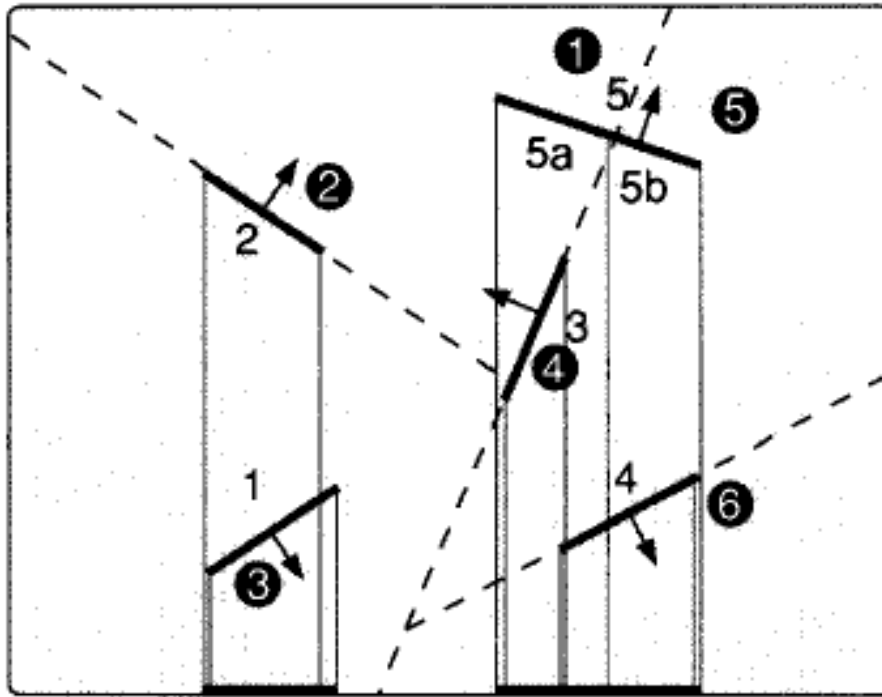


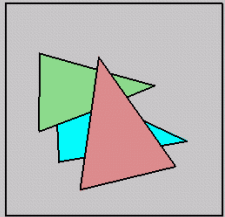
Displaying BSP Tree





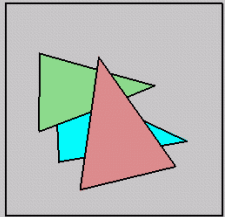
Displaying BSP Tree





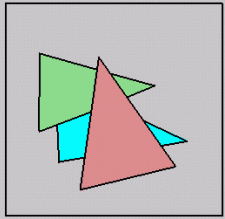
Root of BSP Tree

- Root should be selected such that fewer polygons are split among all of its descendants
 - Easy heuristic: Select the polygon which cause fewest split in its children
 - Testing five / six polygons gives best case



Food for Thought

- BSP Tree vs. Depth Sort
 - Both performs the object splitting and intersections at object precision
 - Both Depends on image precision for pixel overwriting
 - Unlike depth sort, BSP splits and order the polygons during preprocessing step
 - In BSP, more polygon can be split than depth sort
- List Priority vs. Z buffer?
- How BSP tree can assist in 3D Clipping?



Thank you 😊