Hidden Surface Removal

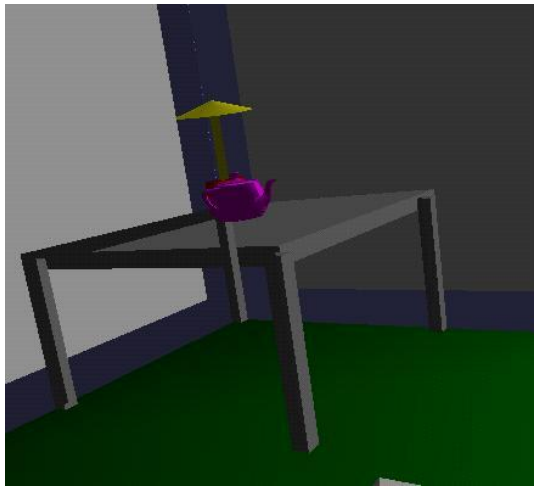
Amartya Kundu Durjoy
Lecturer, CSE, UGV

Hidden Surface Removal

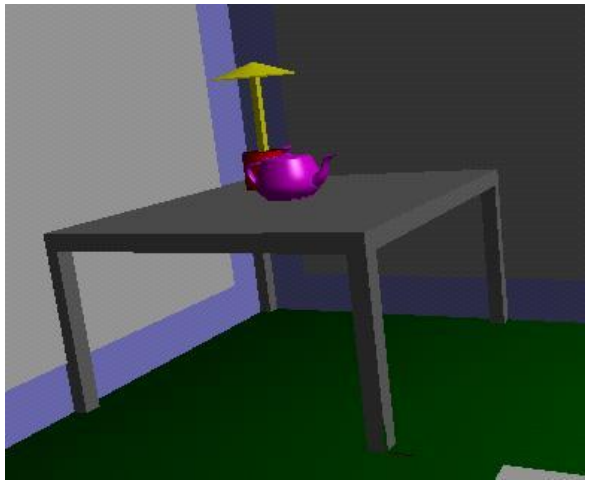
- Drawing polygon faces on screen consumes CPU cycles
- We cannot see every surface in scene
- To save time, draw only surfaces we see
- Surfaces we cannot see and their elimination methods:
  - Occluded surfaces: hidden surface removal (visibility)
  - Back faces: back face culling
  - Faces outside view volume: viewing frustrum culling/clipping
- Definitions:
  - Object space: before vertices are mapped to pixels
  - Image space: after vertices have been rasterized

Visibility (hidden surface removal)

- A correct rendering requires correct visibility calculations
- Correct visibility – when multiple opaque polygons cover the same screen space, only the front most one is visible (remove the hidden surfaces)



wrong visibility



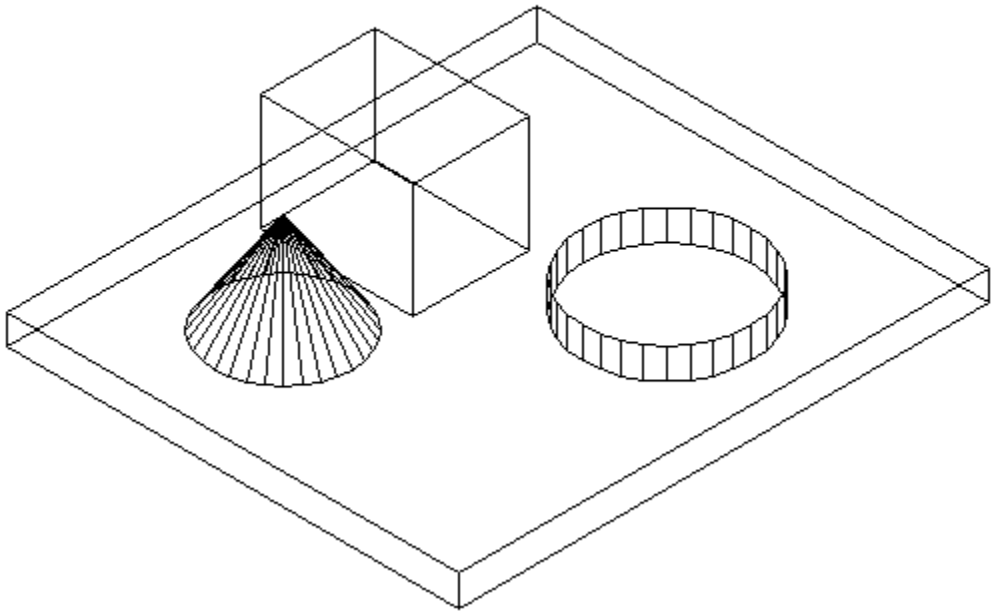
Correct visibility

Visibility (hidden surface removal)

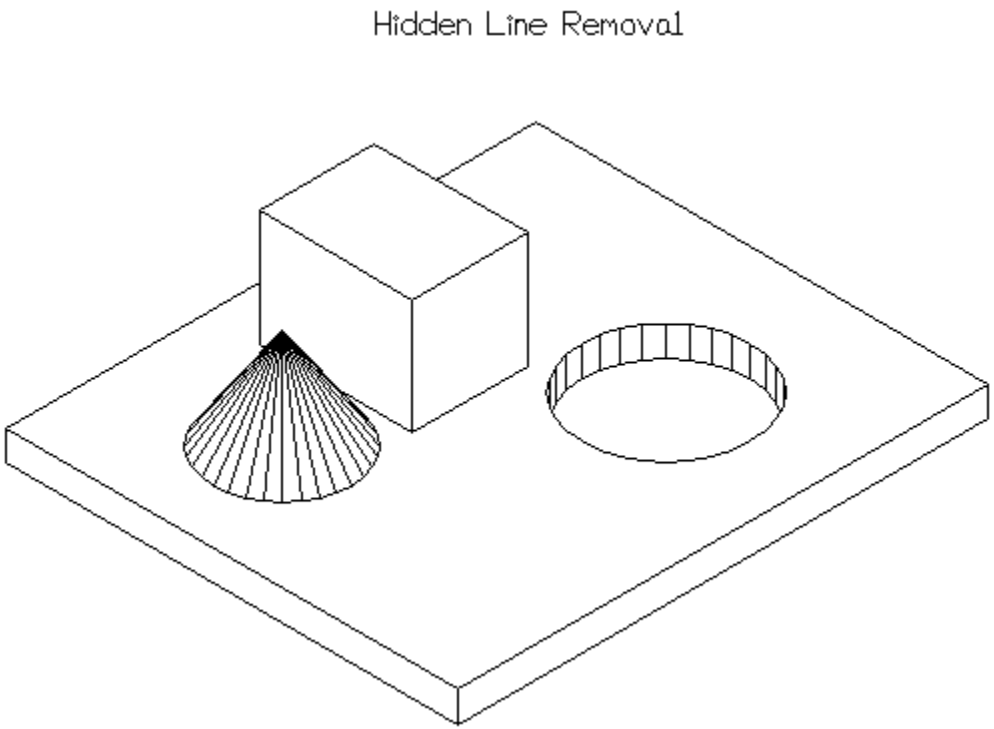
- Goal: determine which objects are visible to the eye
  - Determine what colors to use to paint the pixels
- Active research subject - lots of algorithms have been proposed in the past (and is still a hot topic)

No Lines Removed

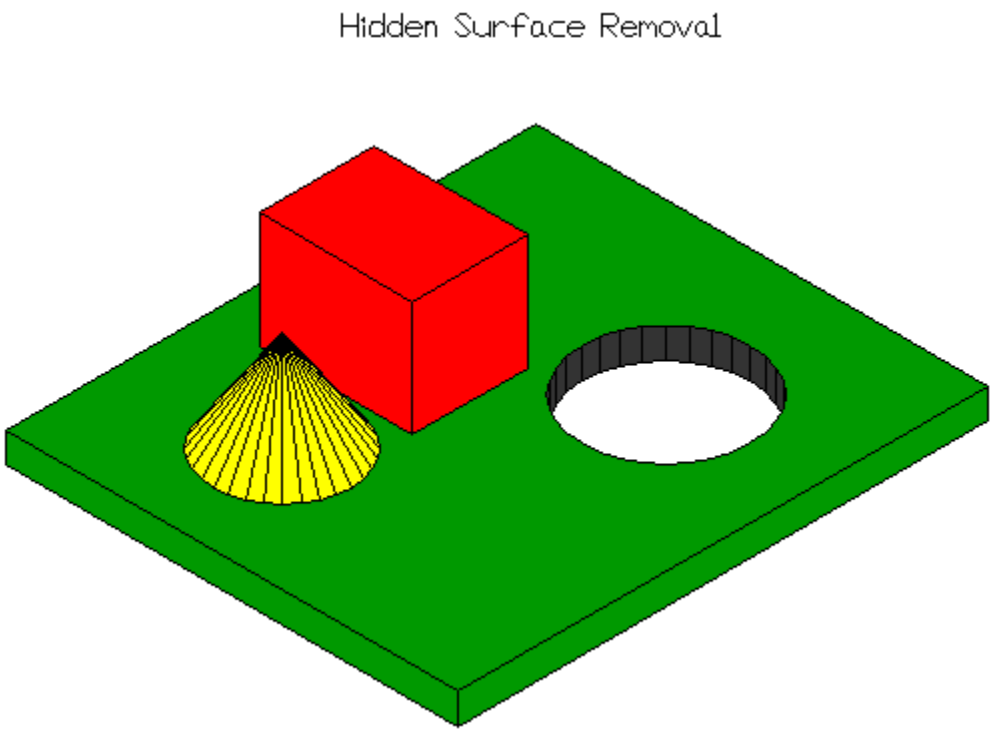
Wireframe



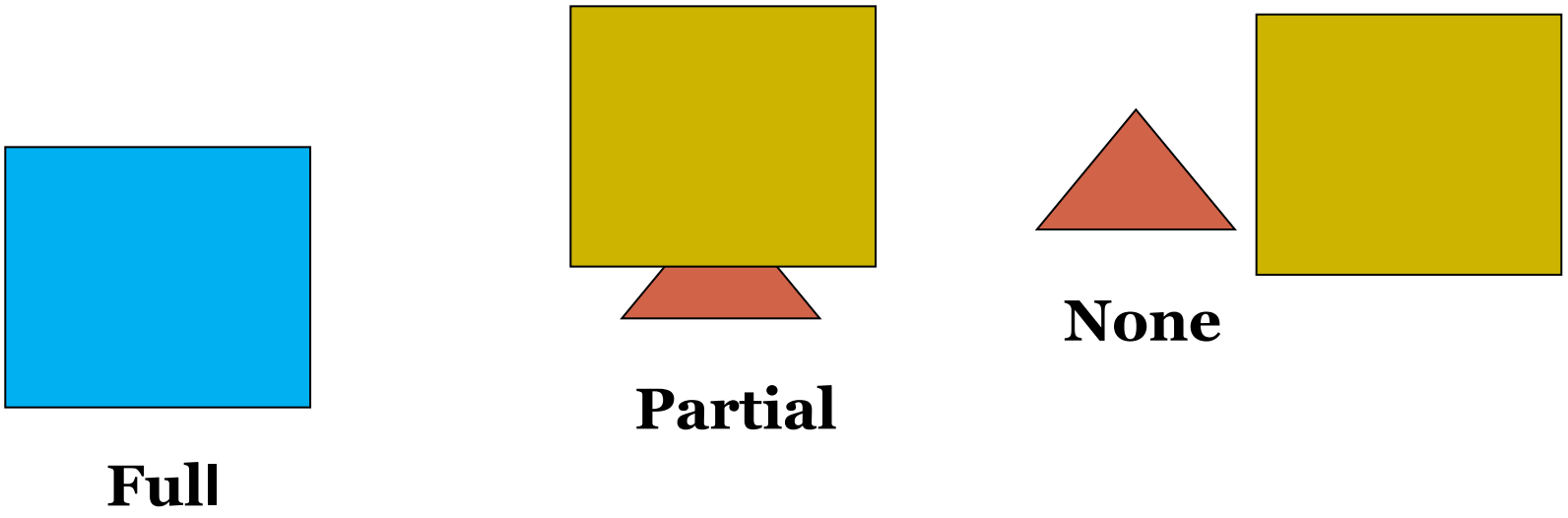
Hidden Lines Removed



Hidden Surfaces Removed



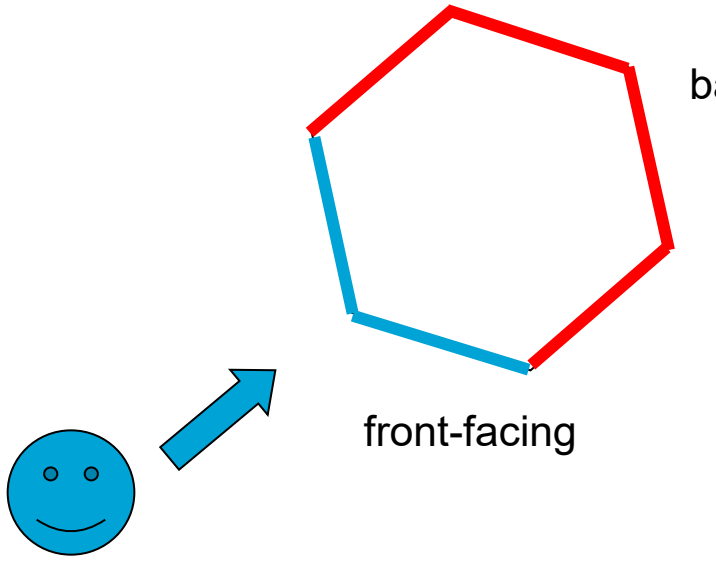
Occlusion: Full, Partial, None



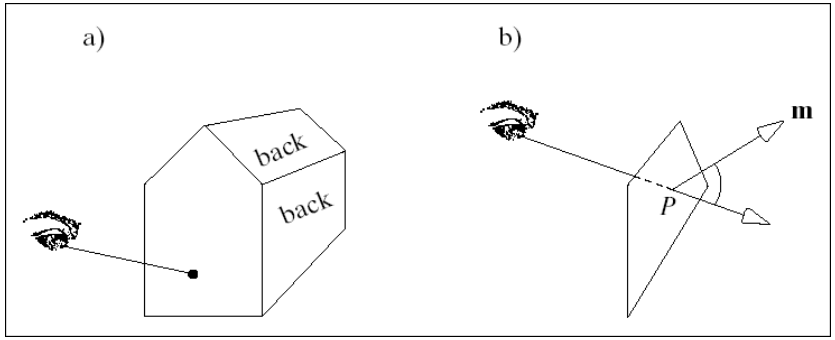
- The rectangle is closer than the triangle
- Should appear in front of the triangle

Backface Culling

- Avoid drawing polygons facing away from the viewer
  - Front-facing polygons occlude these polygons in a closed polyhedron
- Test if a polygon is front- or back-facing?

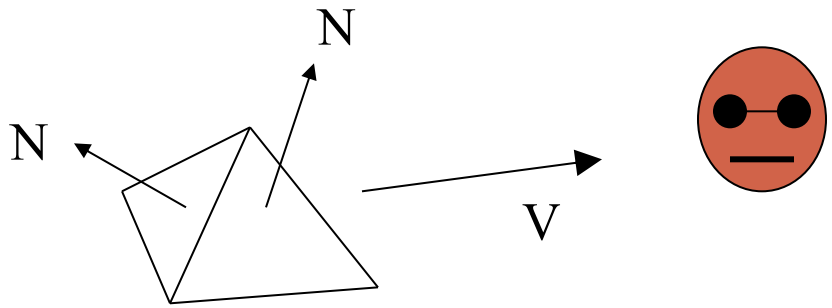


Ideas?



Backface Culling

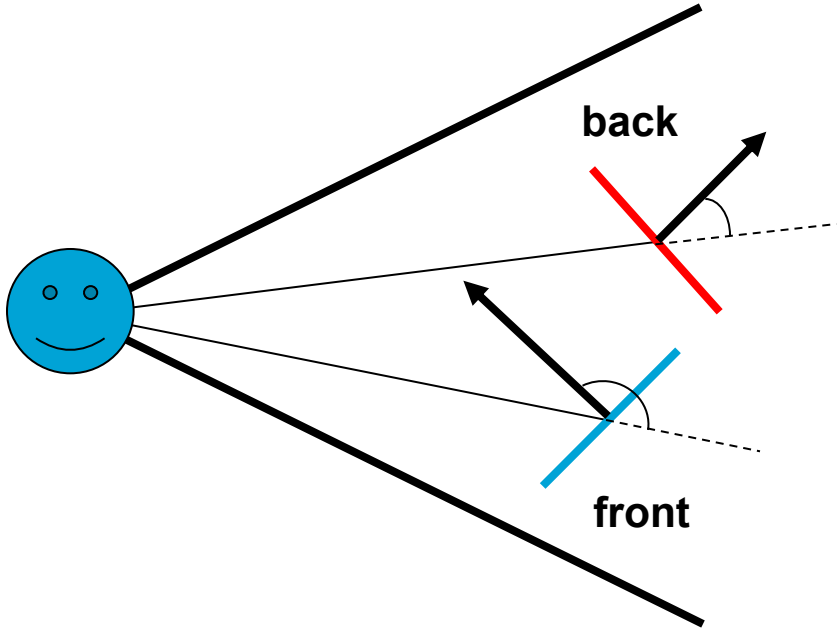
- If we find backface, do not draw, save rendering resources
- There must be other forward face(s) closer to eye
- F is face of object we want to test if backface
- P is a point on F
- Form view vector, V as (eye – P)
- N is normal to face F



Detecting Back-face Polygons

- The polygon normal of a …
  - front-facing polygon points **towards** the viewer
  - back-facing polygon points **away** from the viewer

If $(\mathbf{n} \bullet v) > 0 \Rightarrow$ "back-face"
If $(\mathbf{n} \bullet v) \leq 0 \Rightarrow$ "front-face"
v = view vector



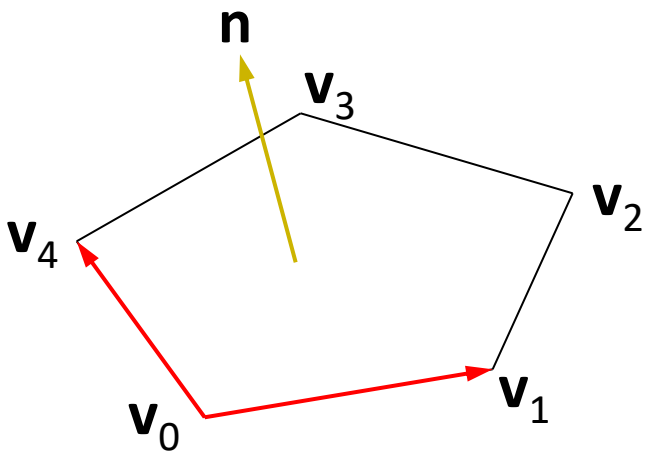
- Eye-space test … EASY!
  - "back-face" if $\mathbf{n}_z < 0$

Polygon Normals

- Let polygon vertices $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_{n-1}$ be in counterclockwise order and co-planar
- Calculate normal with cross product:

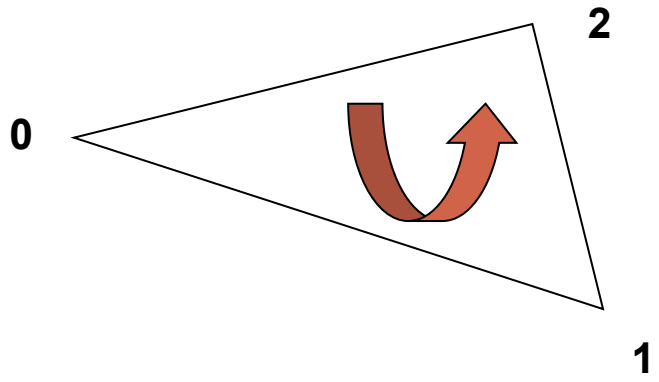
$$\mathbf{n} = (\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_{n-1} - \mathbf{v}_0)$$

- Normalize to unit vector with $\mathbf{n} / \|\mathbf{n}\|$

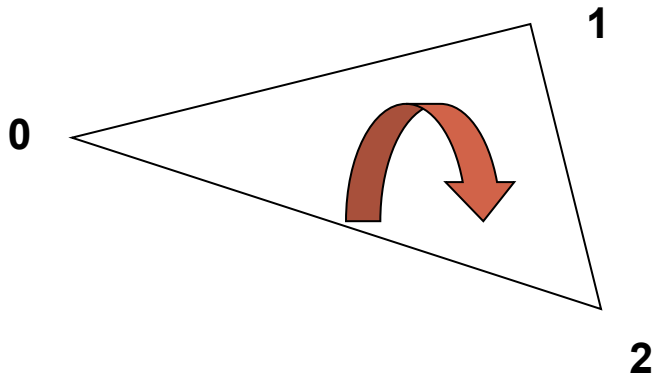


Normal Direction

- Vertices counterclockwise ⇒ Front-facing
- Vertices clockwise ⇒ Back-facing



Front facing



Back facing

Visibility

- How do we ensure that closer polygons overwrite further ones in general?

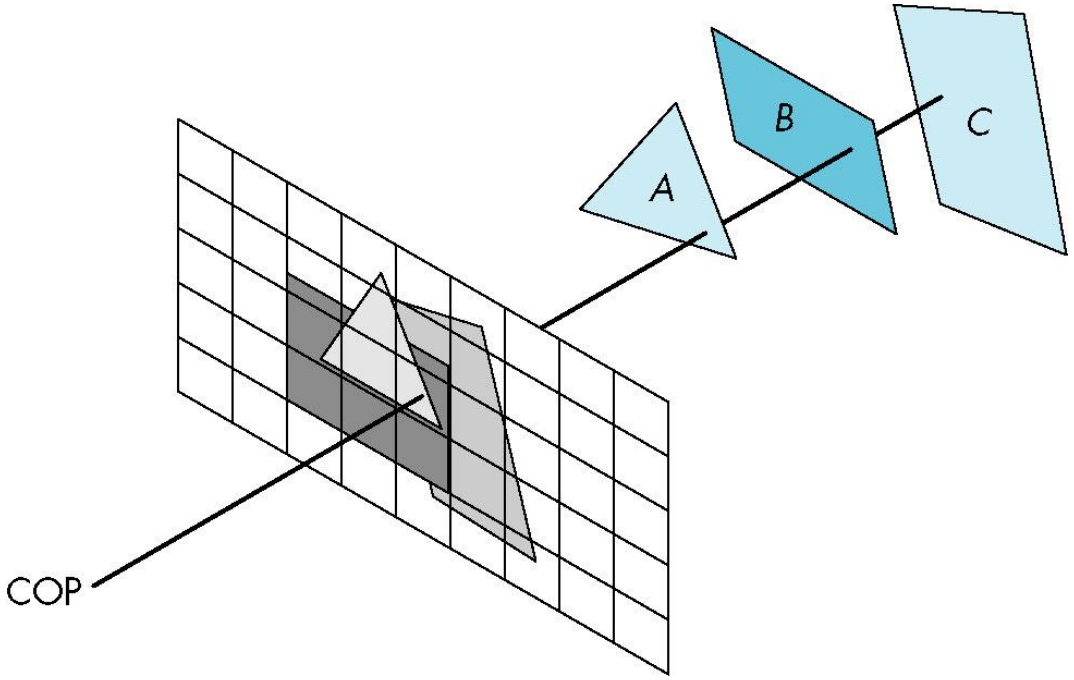


Image Space Approach – Z-buffer

- Method used in most of graphics hardware (and thus OpenGL): Z-buffer algorithm
- Requires lots of memory
- Basic idea:
  - rasterize every input polygon
  - Recall that we have z at polygon vertices
  - Maintains 2 buffers
  - For every pixel in the polygon interior, calculate its corresponding z value (by interpolation) (Z buffer)
  - Choose the color of the polygon whose z value is the closest to the eye to paint the pixel. (Refresh Buffer)

Image Space Approach – Z-buffer

- Recall: after projection transformation
- In viewport transformation
  - x,y used to draw screen image
  - z component is mapped to pseudo-depth with range [0,1]
- However, objects/polygons are made up of vertices
- Hence z is known at vertices
- Point in object seen through pixel may be between vertices
- Need to interpolate to find z

Z (depth) buffer algorithm

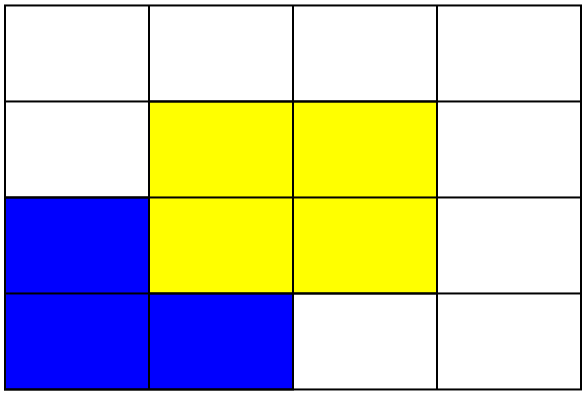
- How to choose the polygon that has the closet Z for a given pixel?
- Assumption for example: eye at z = 0, farther objects have increasingly negative values
  - Initialize (clear) every pixel in the z buffer to a very large negative value
  - Track polygon z's.
  - As we rasterize polygons, check to see if polygon's z through this pixel is less than current minimum z through this pixel
  - Run the following loop:

Z (depth) Buffer Algorithm

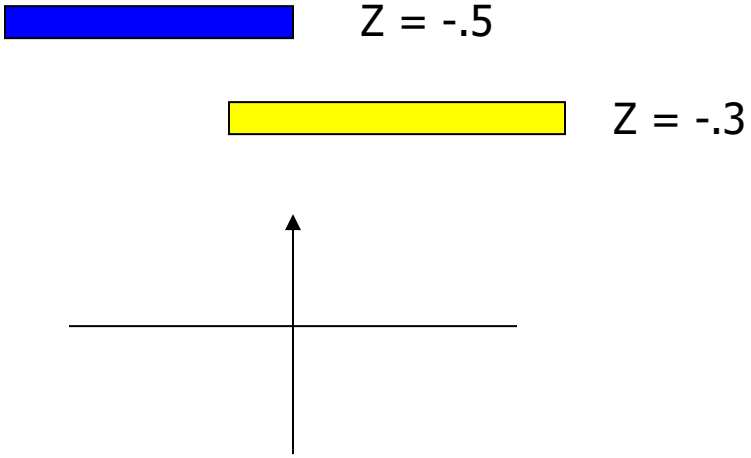

```
For  each polygon  {
   for each pixel (x,y) inside the polygon projection area  {
        if  (z_polygon_pixel(x,y) > depth_buffer(x,y) ) {
             depth_buffer(x,y) = z_polygon_pixel(x,y);
             color_buffer(x,y) = polygon color at (x,y)
        }
   }
}
```

Note: we have depths at vertices. Interpolate for interior depths

Z buffer example



Final image



Top View

Z buffer example

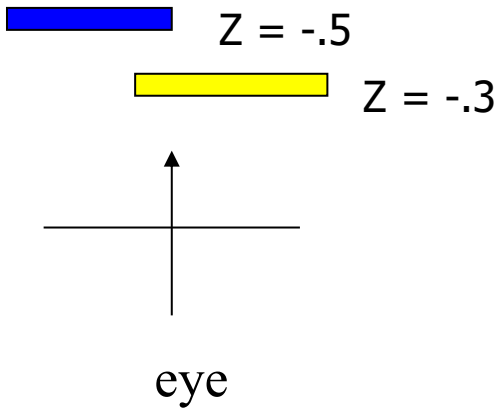
Step 1: Initialize the depth buffer

-999	-999	-999	-999
-999	-999	-999	-999
-999	-999	-999	-999
-999	-999	-999	-999

Z buffer example

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

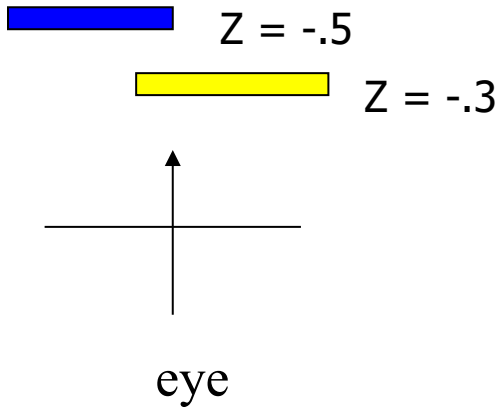
-999	-999	-999	-999
-999	-999	-999	-999
-.5	-.5	-999	-999
-.5	-.5	-999	-999



Z buffer example

Step 3: Draw the yellow polygon

-999	-999	-999	-999
-999	-.3	-.3	-999
-.5	-.3	-.3	-999
-.5	-.5	-999	-999



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:

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

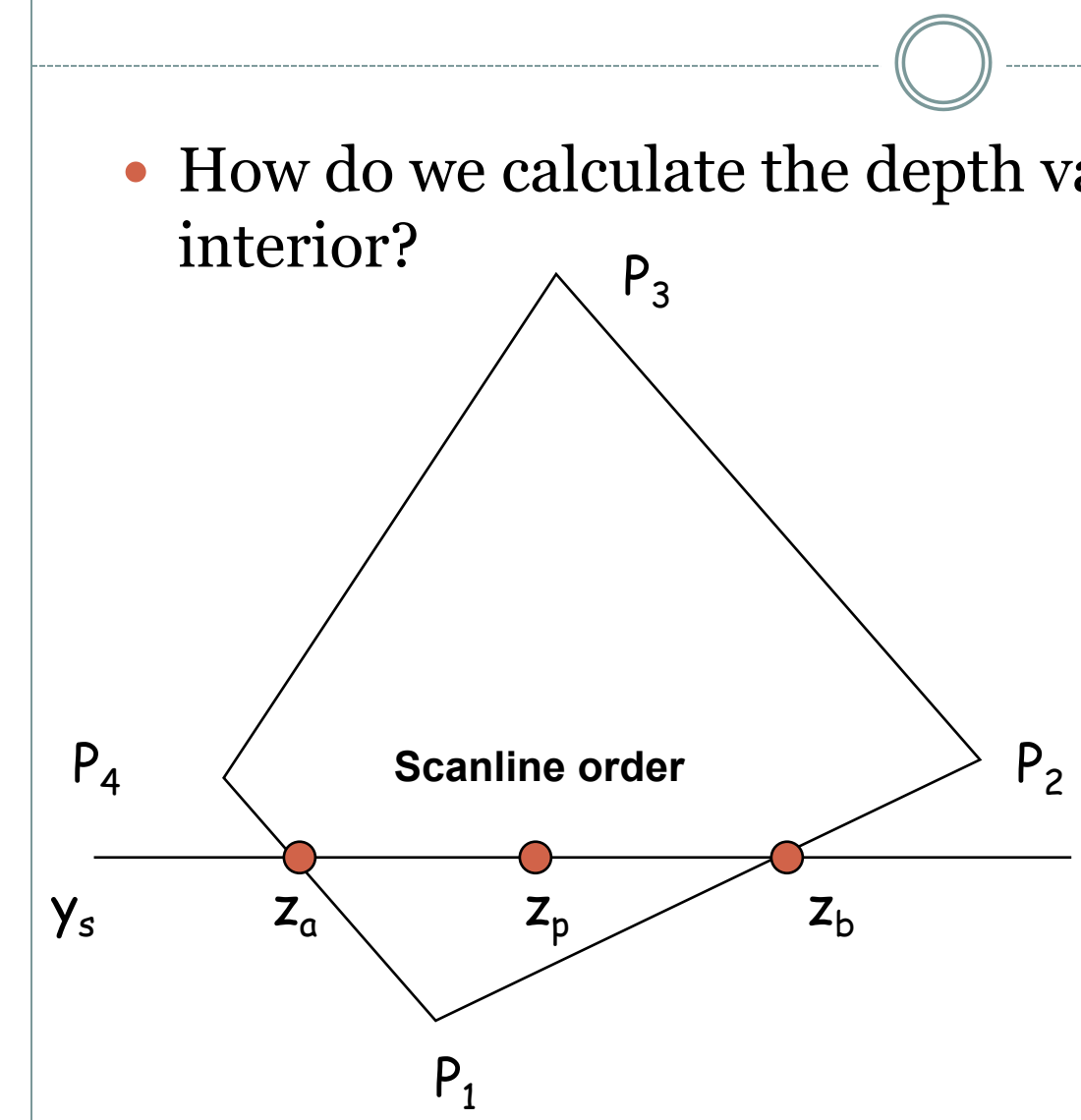
Calculating depth values efficiently

- 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.

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

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

Z-Buffer



- How do we calculate the depth values on the polygon interior?

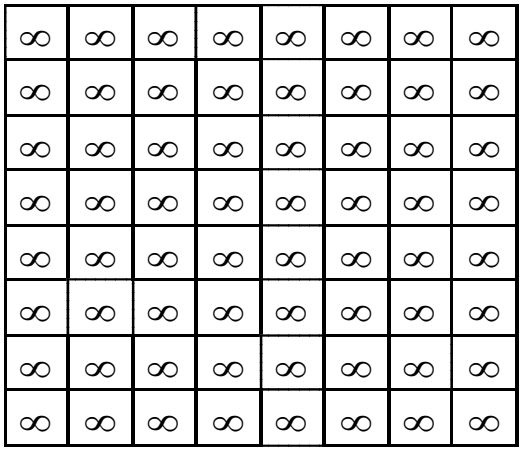
$$z_a = z_1 + (z_4 - z_1)\frac{(y_1 - y_s)}{(y_1 - y_4)}$$

$$z_b = z_1 + (z_2 - z_1)\frac{(y_1 - y_s)}{(y_1 - y_2)}$$

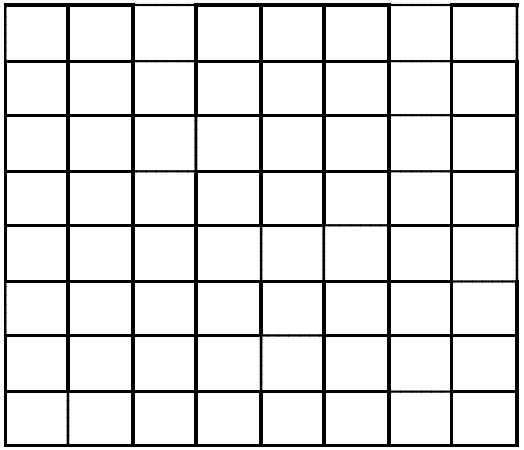
$$z_p = z_a + (z_b - z_a)\frac{(x_a - x_p)}{(x_a - x_b)}$$

Bilinear Interpolation

Z-Buffer Example



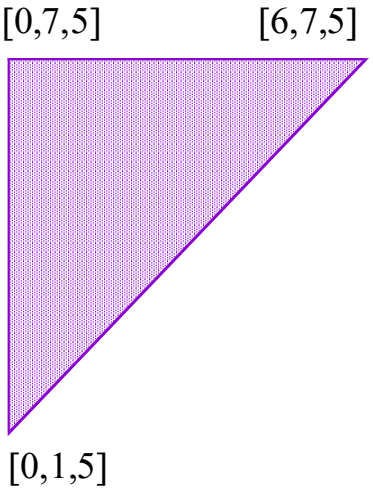
Z-buffer



Screen

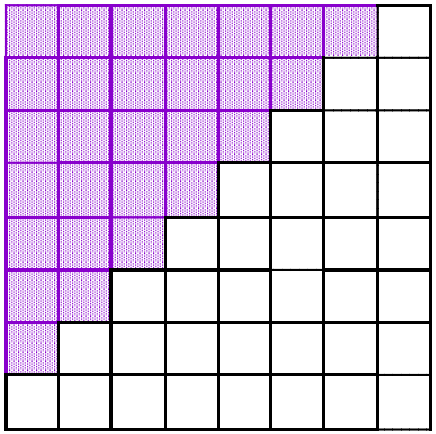
Z-Buffer Example

Parallel with the image plane



5	5	5	5	5	5	5	
5	5	5	5	5	5		
5	5	5	5	5			
5	5	5	5				
5	5	5					
5	5						
5							

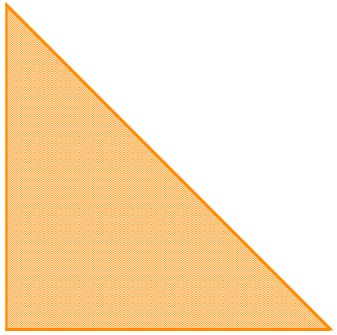
5	5	5	5	5	5	5	∞
5	5	5	5	5	5	∞	∞
5	5	5	5	5	∞	∞	∞
5	5	5	5	∞	∞	∞	∞
5	5	5	∞	∞	∞	∞	∞
5	5	∞	∞	∞	∞	∞	∞
5	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞



Z-Buffer Example

Not Parallel

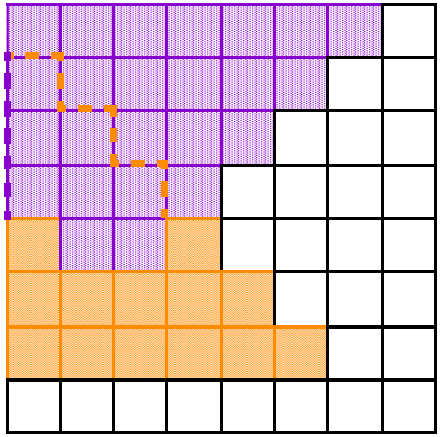
[0,6,7]



7					
6	7				
5	6	7			
4	5	6	7		
3	4	5	6	7	
2	3	4	5	6	7

[0,1,2] [5,1,7]

5	5	5	5	5	5	5	∞
5	5	5	5	5	5	∞	∞
5	5	5	5	5	∞	∞	∞
5	5	5	5	∞	∞	∞	∞
4	5	5	7	∞	∞	∞	∞
3	4	5	6	7	∞	∞	∞
2	3	4	5	6	7	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞



Limitation of Z-buffer Algorithm

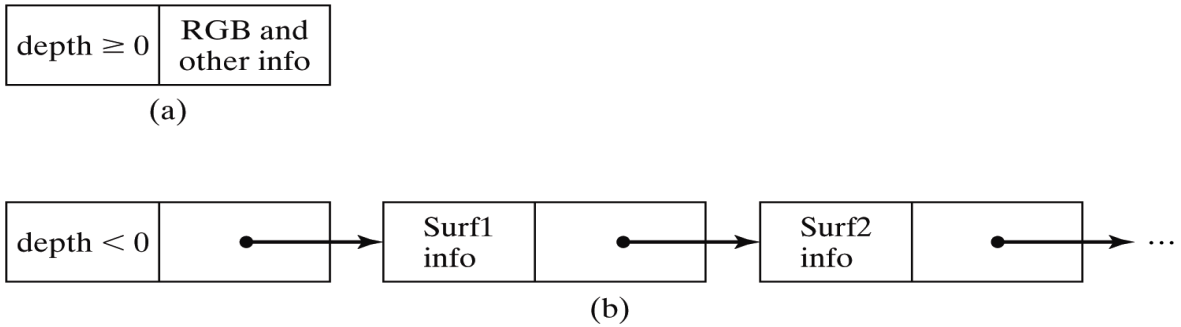
- This algorithm do not work on **transparent surface**, this is only for opaque surface.
- Hence here no method is available to compare the Z values of the surfaces which are transparent.
- For working on transparent surface another method is available which is called A-buffer Algorithm.
- Extra memory is required to store depth value.

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

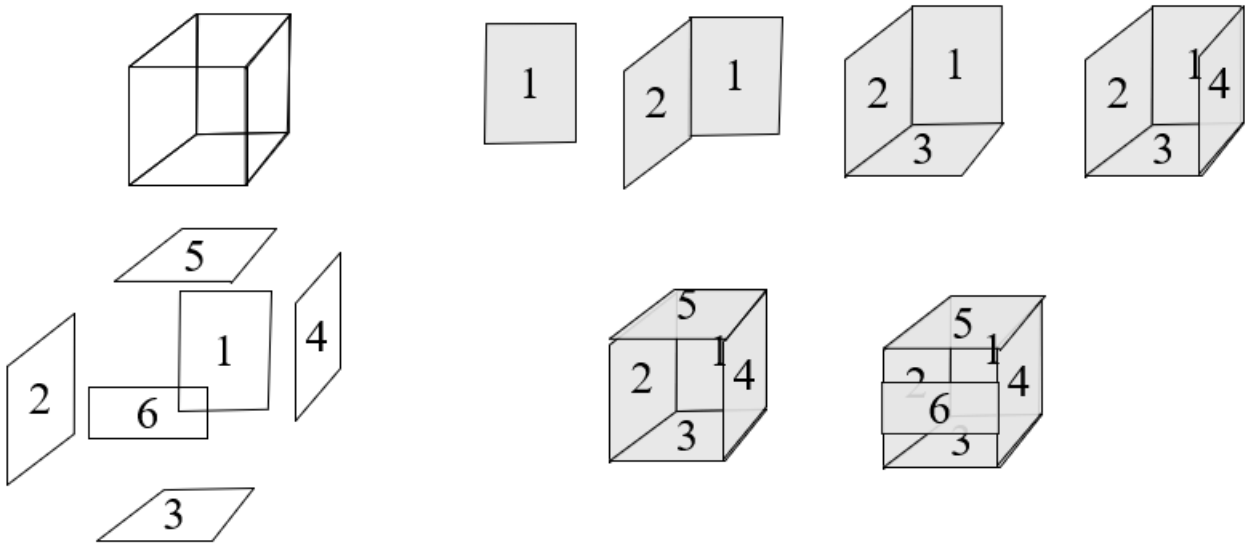
- Each position in the A-buffer has two fields:
  - a depth field
  - surface data field which can be either surface data or a pointer to a linked list of surfaces that contribute to that pixel position
  - Many other information can be stored in the link list such as : % of transparency, % of area coverage, surface rendering factor.



Object Space Methods

Depth Sorting

- Also known as **painters algorithm**. First draw the distant objects than the closer objects. Pixels of each object overwrites the previous objects.



Depth-sort algorithm

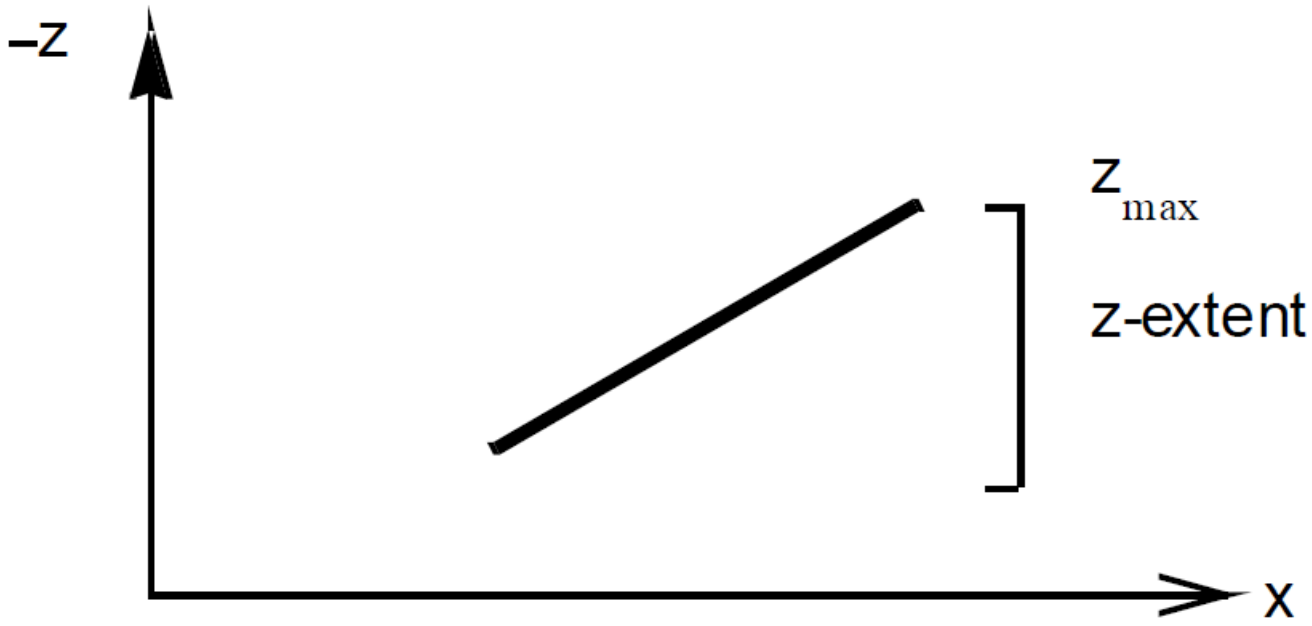
- The idea here is to go back to front drawing all the objects into the frame buffer with nearer objects being drawn over top of objects that are further away.

- Simple algorithm:
  - Sort all polygons based on their farthest z coordinate
  - Resolve ambiguities
  - Draw the polygons in order from back to front

- This algorithm would be very simple if the z coordinates of the polygons were guaranteed never to overlap. Unfortunately that is usually not the case, which means that step 2 can be somewhat complex.

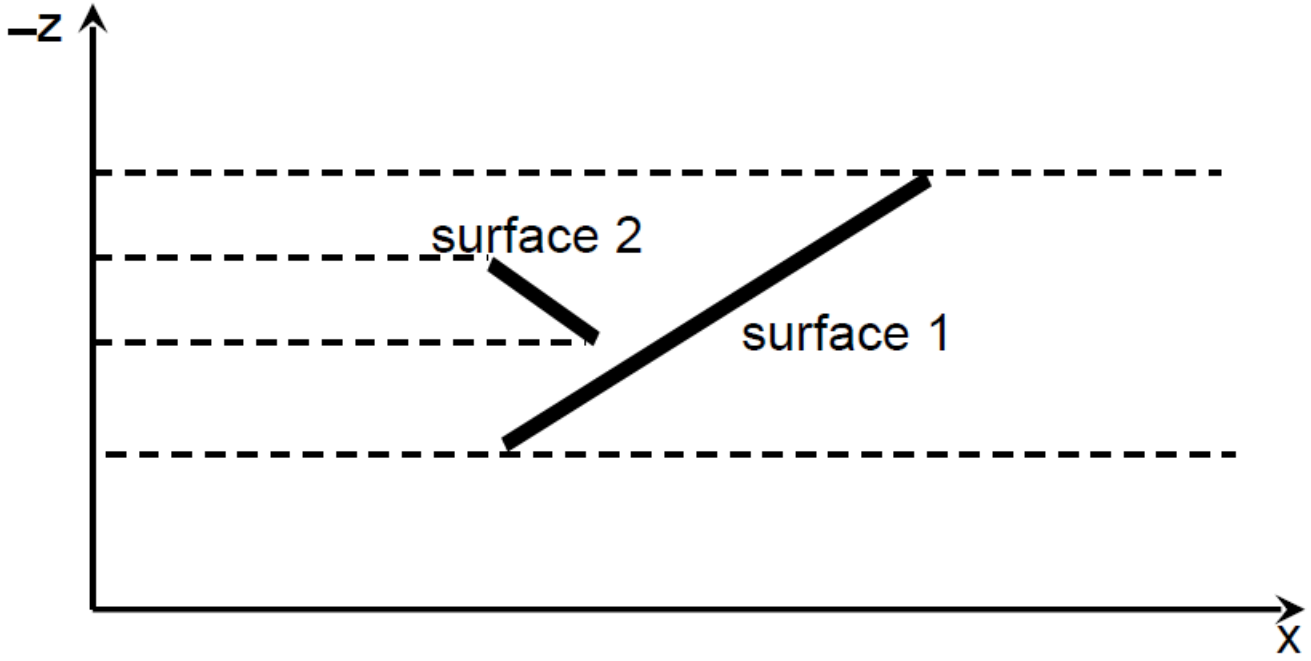
Depth-sort algorithm

- First must determine z-extent for each polygon

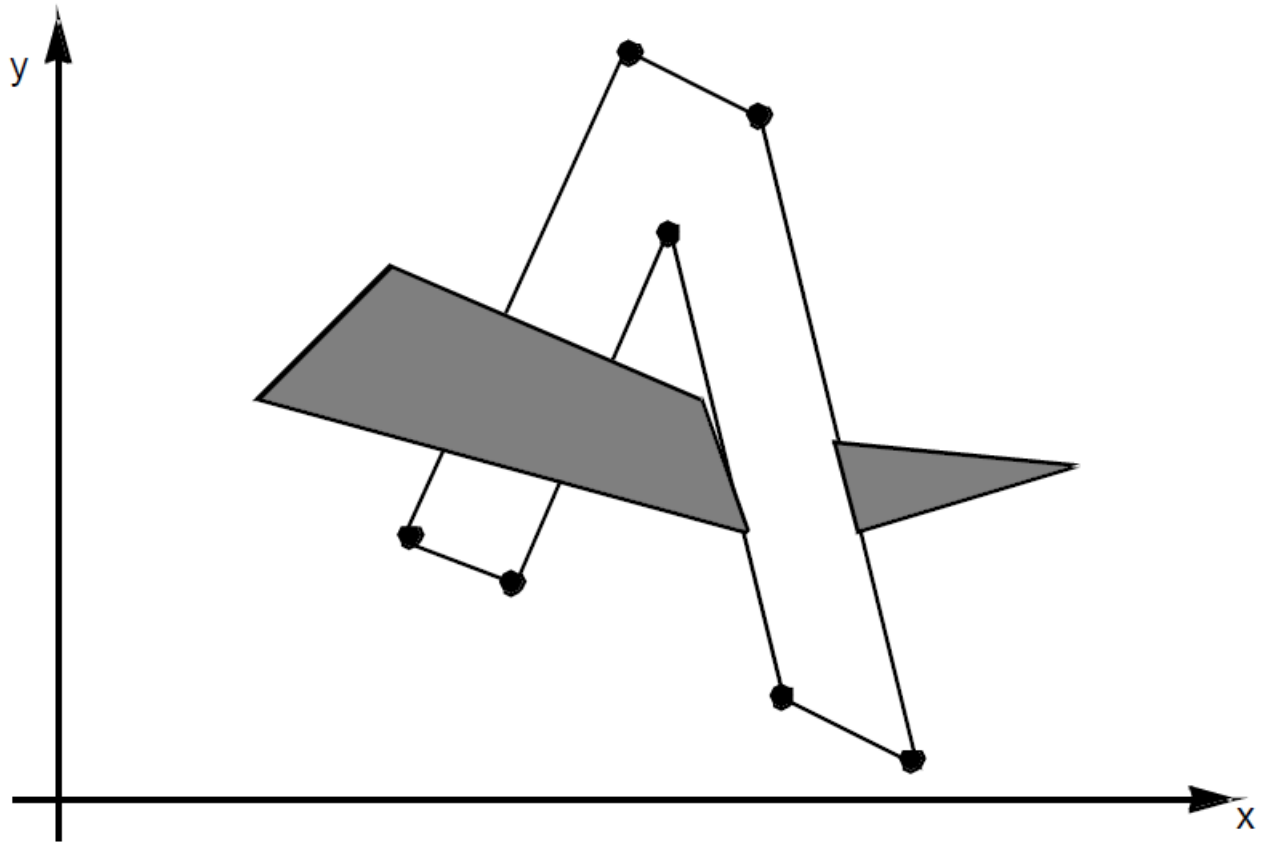


Depth-sort algorithm

- Ambiguities arise when the z-extents of two surfaces overlap.



Depth-sort algorithm



Depth-sort algorithm

- All polygons whose z extents overlap must be tested against each other.

- We start with the furthest polygon and call it P. Polygon P must be compared with every polygon Q whose z extent overlaps P's z extent. 5 comparisons are made. If any comparison is true then P can be written before Q. If at least one comparison is true for each of the Qs then P is drawn and the next polygon from the back is chosen as the new P.

Depth-sort algorithm

1. Do P and Q's x-extents not overlap?
2. Do P and Q's y-extents not overlap?
3. Is P entirely on the opposite side of Q's plane from the viewport?
4. Is Q entirely on the same side of P's plane as the viewport?
5. Do the projections of P and Q onto the (x,y) plane not overlap?

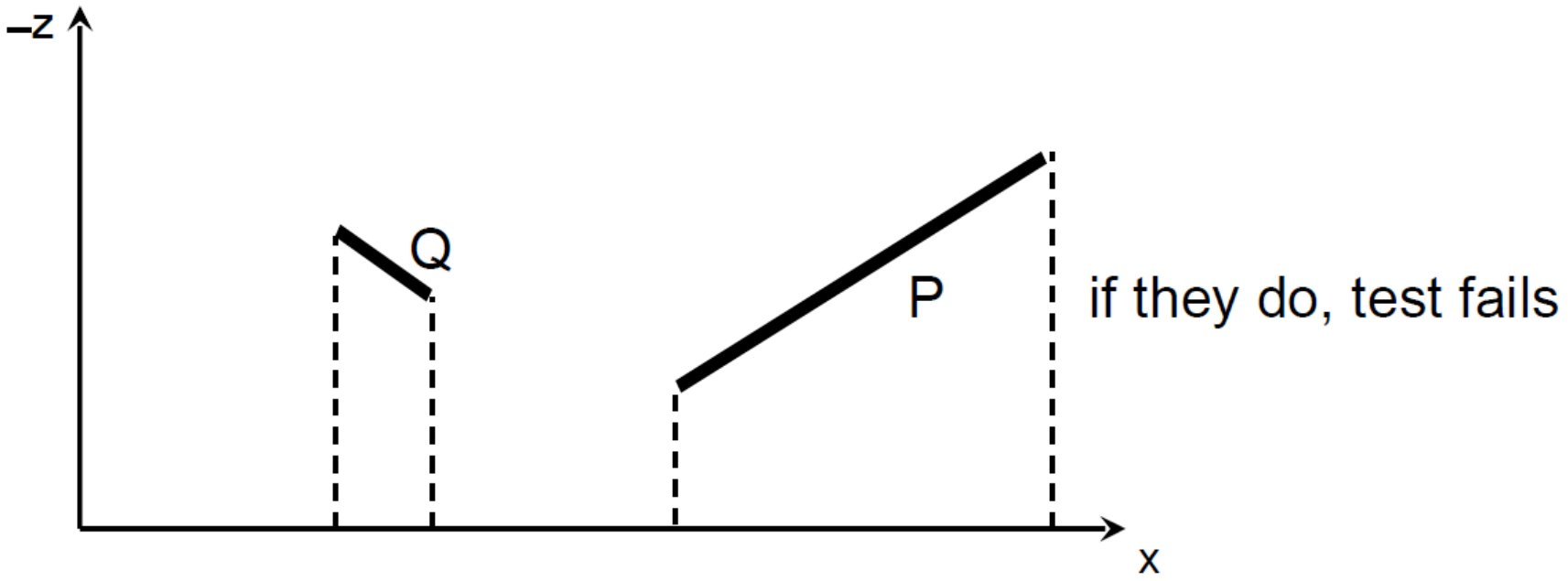
- If all 5 tests fail we quickly check to see if switching P and Q will work. Tests 1, 2, and 5 do not differentiate between P and Q but 3 and 4 do. So we rewrite 3 and 4 as:

3'. Is Q entirely on the opposite side of P's plane from the viewport?

4'. Is P entirely on the same side of Q's plane as the viewport?

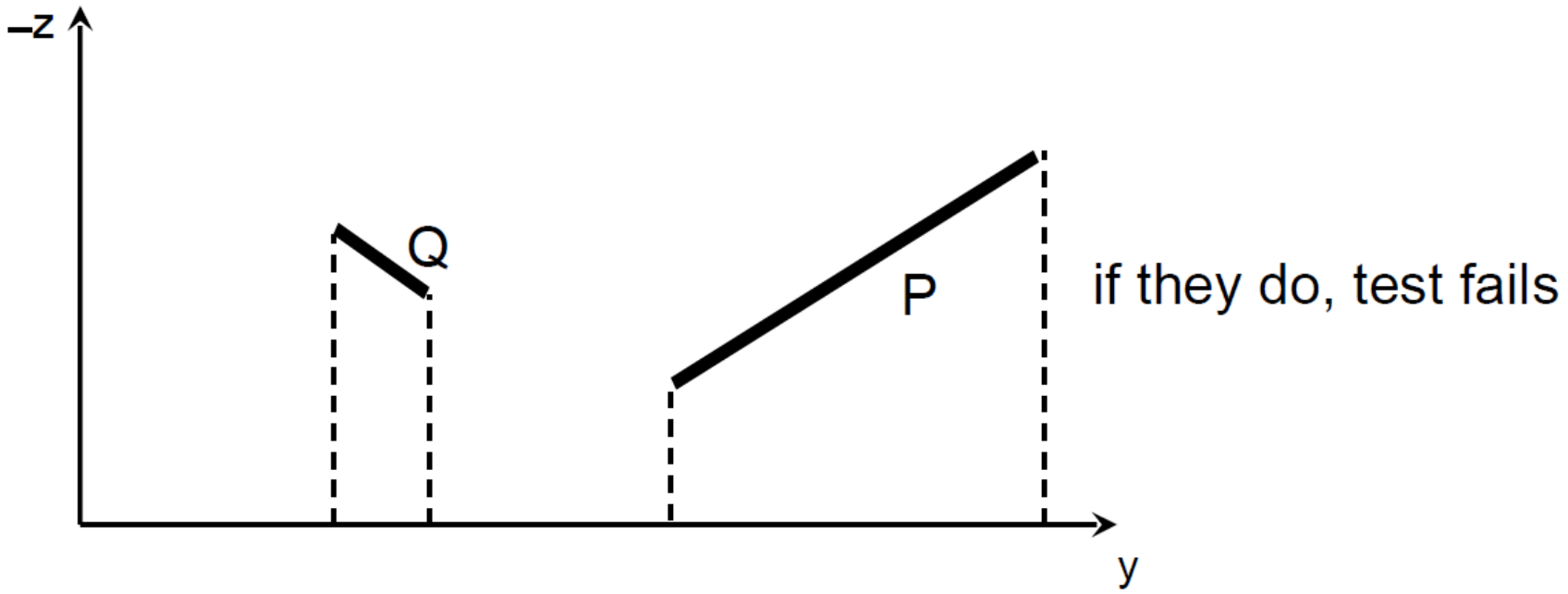
Depth-sort algorithm

- x - extents not overlap?



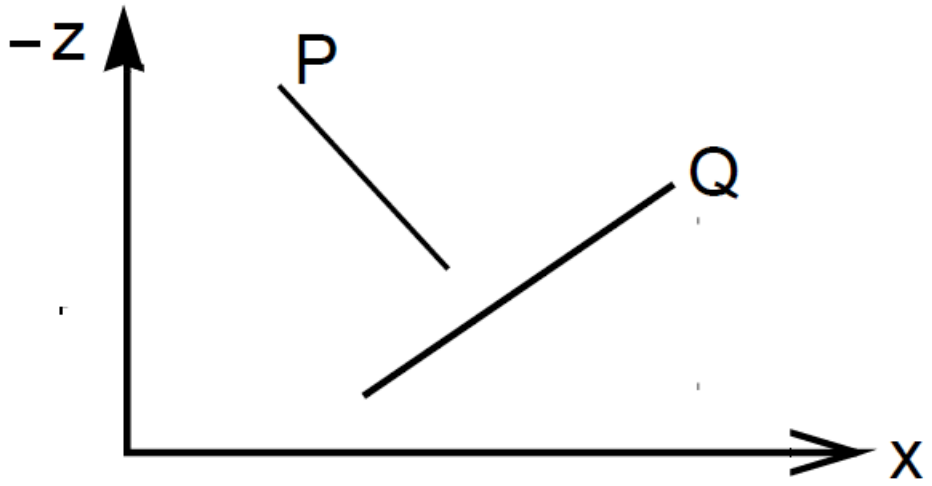
Depth-sort algorithm

- y - extents not overlap?



Depth-sort algorithm

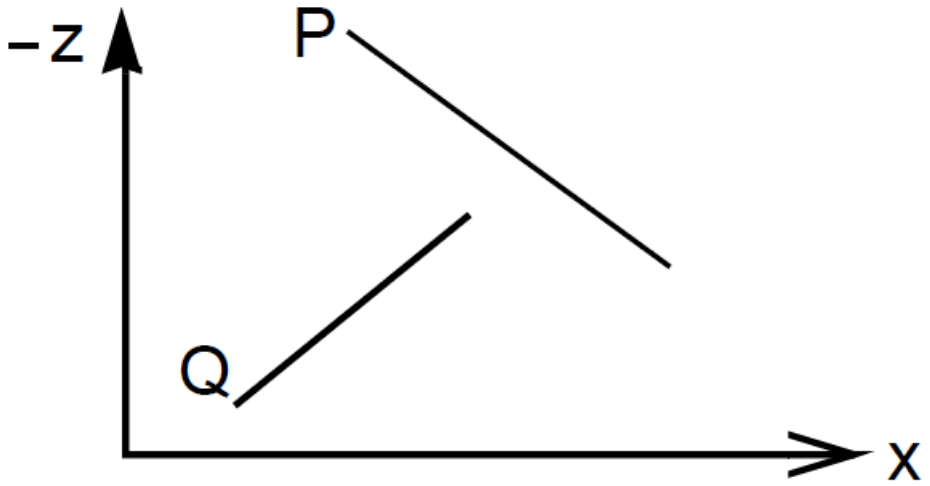
- Is P entirely behind the surface Q relative to the viewing position (i.e., behind Q's plane with respect to the viewport)?



Test is true…

Depth-sort algorithm

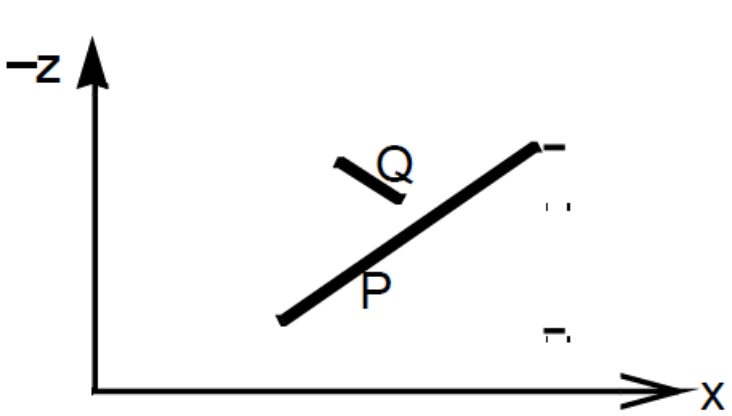
- Is Q entirely in front of P's plane relative to the viewing position (i.e., the viewport)?

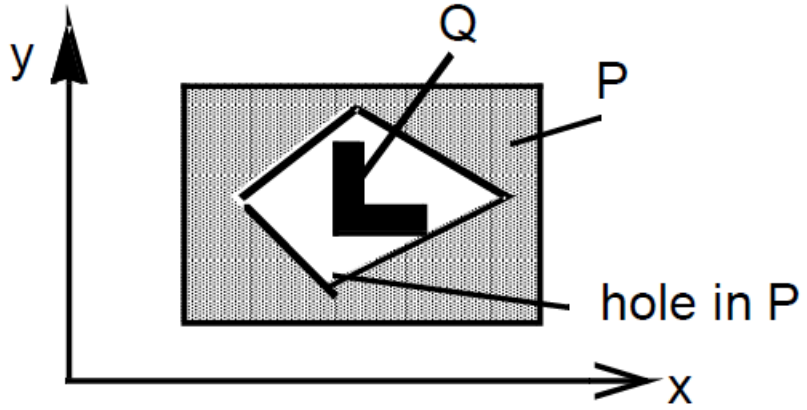


Test is true…

Depth-sort algorithm

- Do the projections of P and Q onto the (x,y) plane not overlap?





Test is true…

Depth-sort algorithm

- If all tests fail…

– … then reverse P and Q in the list of surfaces sorted by maximum depth

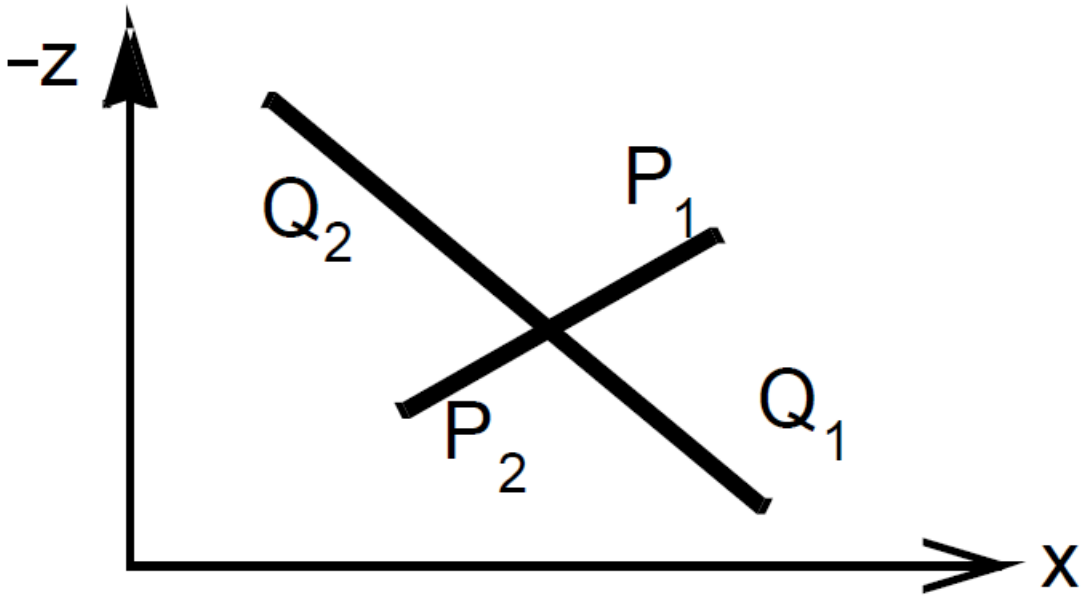
– set a flag to say that the test has been performed once.

– If the tests fail a second time, then it is necessary to split the surfaces and repeat the algorithm on the 4 new split surfaces

Depth-sort algorithm

- Example:

– We end up processing with order Q2,P1,P2,Q1



Reference

- **Computer Graphics**
  - Donald Hearn, M Pauline Baker
  - Pearson Education