Computer Graphics (CS 543) Lecture 12 (Part 2): Viewport Transformation, Hidden Surface Removal and Rasterization

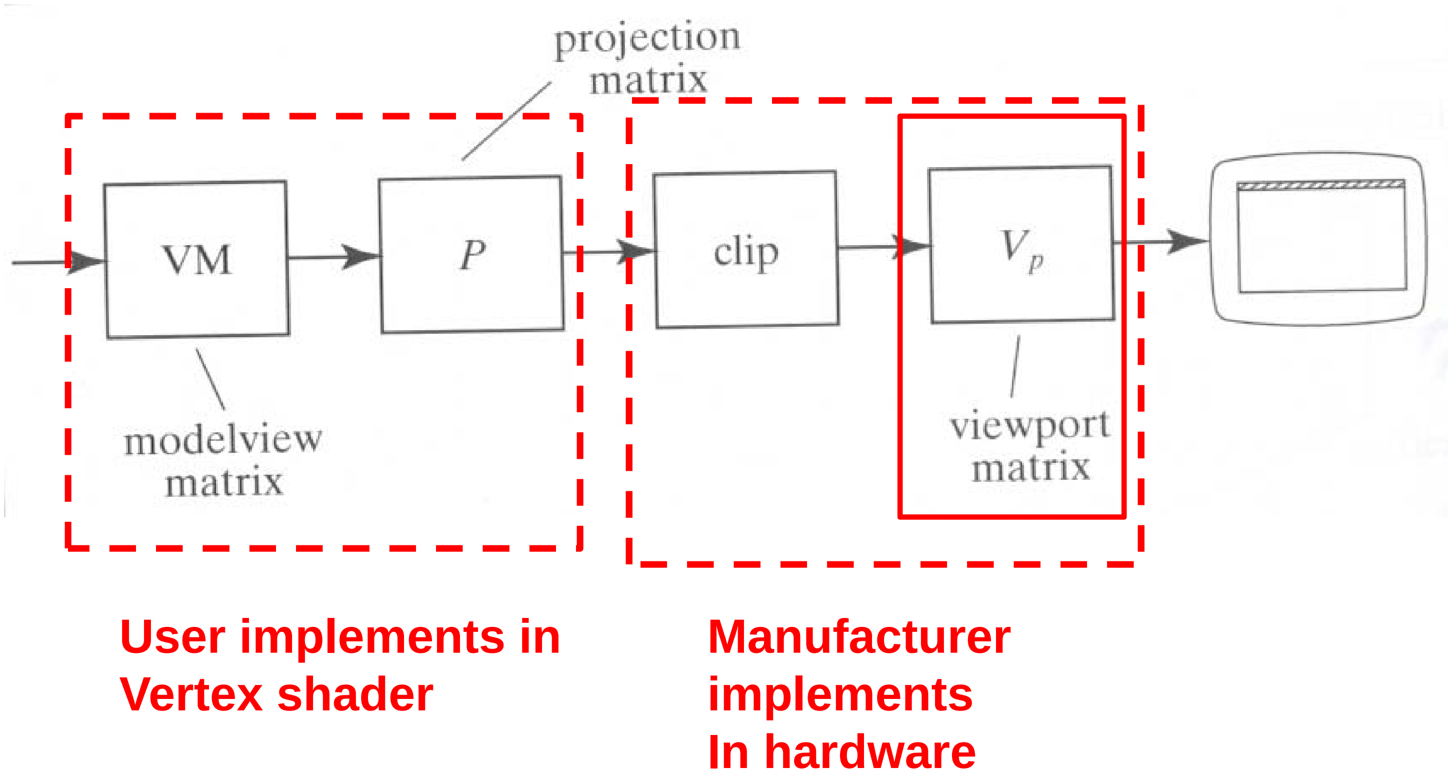
Prof Emmanuel Agu

Computer Science Dept.

Worcester Polytechnic Institute (WPI)

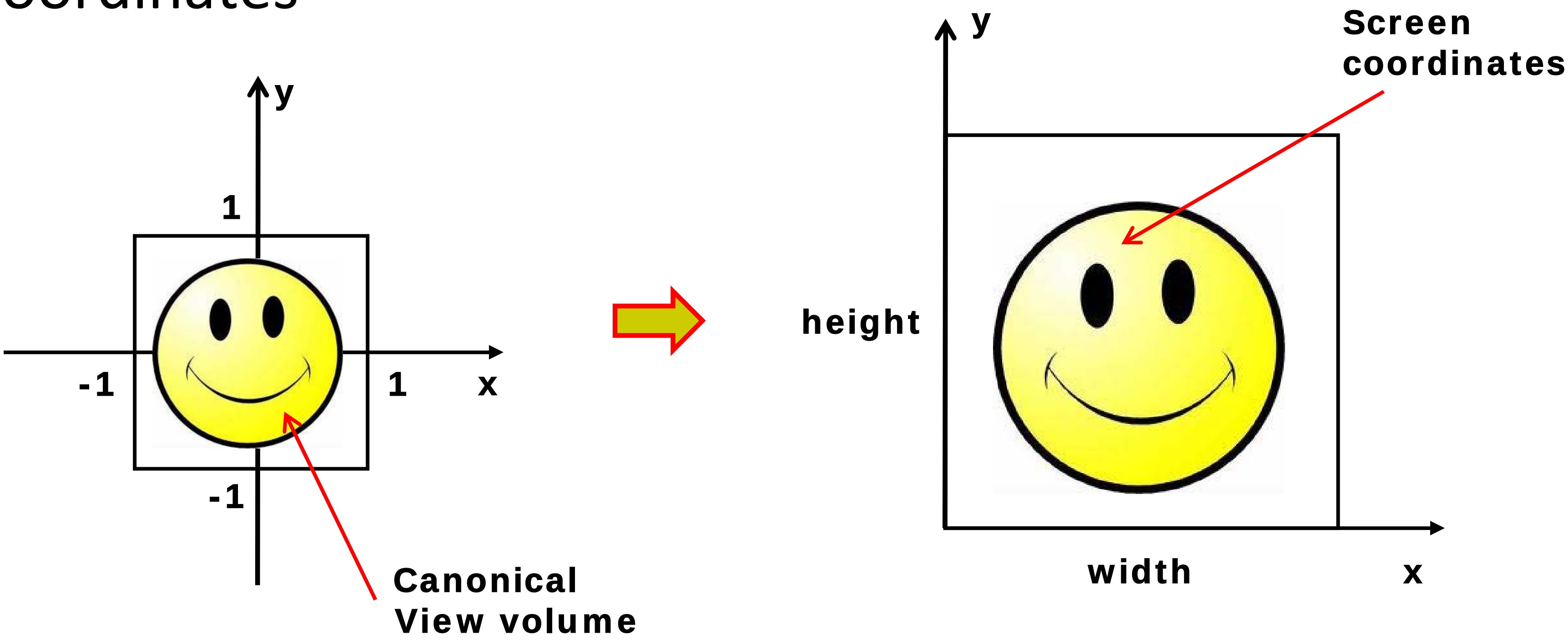
Viewport Transformation

- After clipping, do viewport transformation



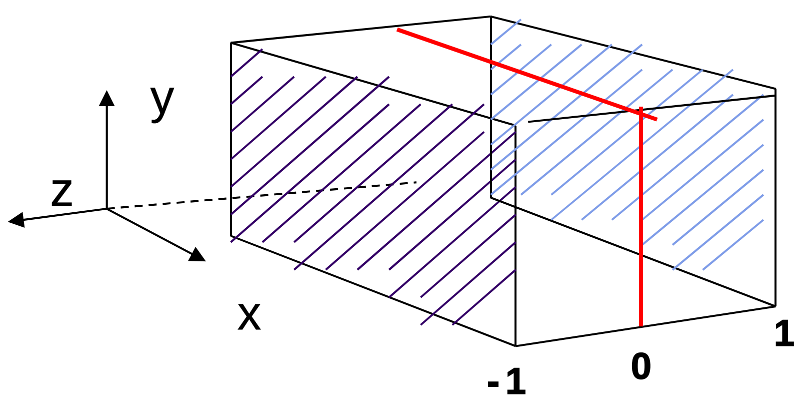
Viewport Transformation

- Command to set viewport: **glViewport(x,y, wid, ht)**
- **x,y, wid, ht** in screen coordinates (pixels)
- Viewport transformation shifts x, y to screen (x, y) coordinates



Viewport Transformation

- Also maps z values (pseudo-depth) from range [-1,1] to [0,1]
- Pseudo-depth stored in depth buffer, used for Depth testing (Hidden Surface Removal)

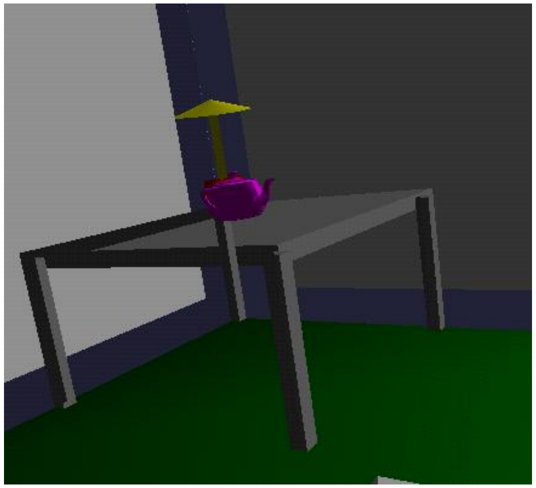


Hidden surface Removal

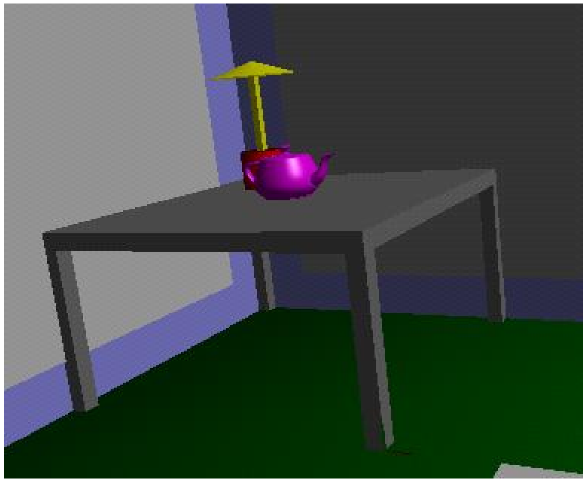
- Drawing polygonal faces on screen consumes CPU cycles
- We cannot see every surface in scene
- To save time, draw only surfaces we see
- Surfaces we cannot see and elimination methods:
  - **Occluded surfaces:** hidden surface removal (visibility)
  - **Back faces:** back face culling
  - **Faces outside view volume:** viewing frustrum culling
- Classification of techniques:
  - **Object space techniques:** applied before rasterization
  - **Image space techniques:** applied after vertices have been rasterized

Visibility (hidden surface removal)

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



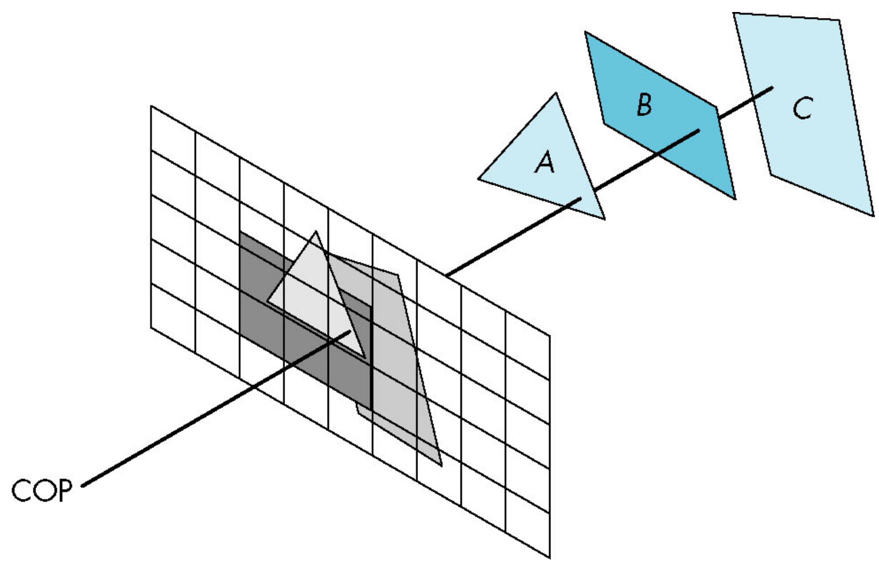
wrong visibility



Correct visibility

Image Space Approach

- Through each pixel, (nm for an $n \times m$ frame buffer) find closest of k polygons
- Complexity $O(nmk)$
- Ray tracing
- z-buffer : OpenGL



OpenGL - Image Space Approach

- Paint pixel with color of **closest** object

```
for (each pixel in image) {
   determine the object closest to the pixel
   draw the pixel using the object's color
}
```

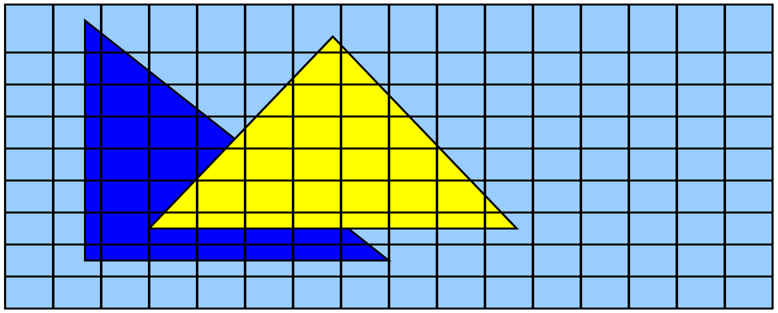


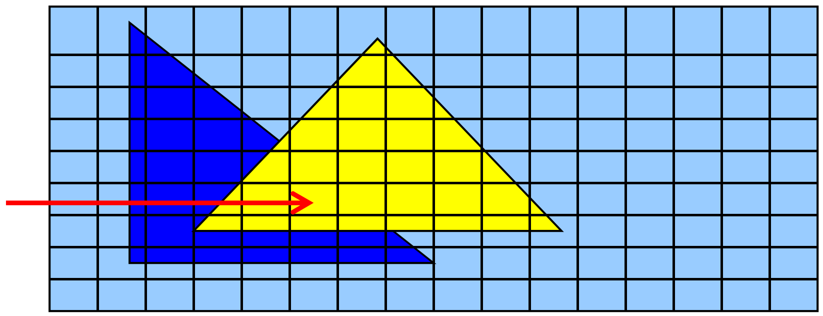
Image Space Approach – Z-buffer

- Z-buffer (or depth buffer) algorithm: Method used in most of graphics hardware (and OpenGL):
- Requires lots of memory
- Recall: during viewport transformation
  - x,y mapped to screen coordinates, used to draw screen
  - z component mapped to range [0,1]
  - Larger z values: Further away from viewer
- Hence, we know depth z at polygon vertices
- During rasterization, object depth between vertices interpolated so we know depth at all pixels

Z-buffer Algorithm

- Basic Z-buffer idea:
  - rasterize every input polygon
  - For every pixel in 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.

Find depth (z) of every polygon at each pixel



Z (depth) buffer algorithm

- Note: eye at z = 0, farther objects have larger values of z (between 0 and 1)
  1. Initialize (clear) every pixel in the z buffer to 1.0
  2. Track polygon z's.
  3. As we rasterize polygons, check to see if polygon's z through this pixel is less than current minimum z through this pixel
  4. Run the following loop:

Z (depth) Buffer Algorithm

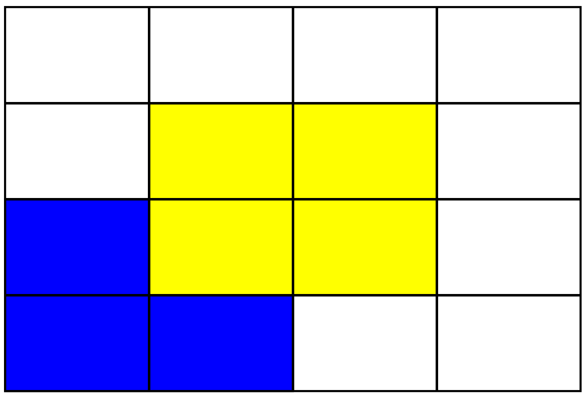
Depth of polygon being rasterized at pixel (x, y)

Largest depth seen so far Through pixel (x, y)

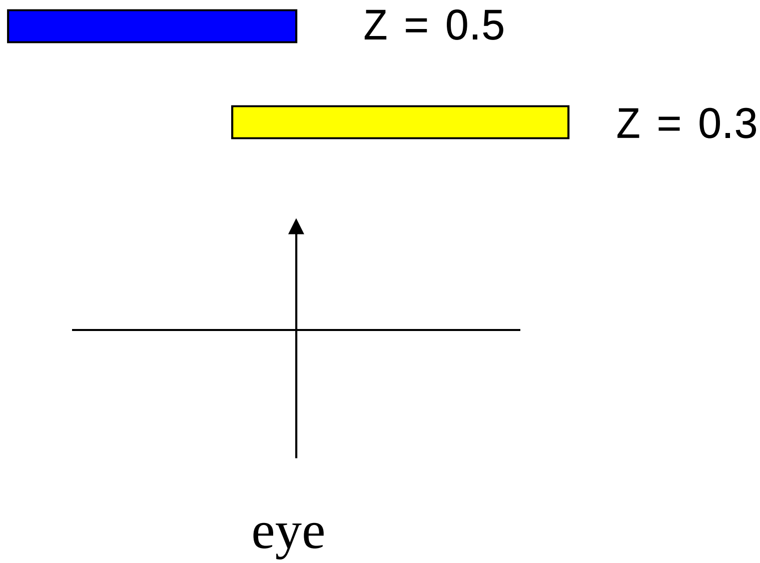
```
For  each polygon  {

   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: know depths at vertices. Interpolate for interior z_polygon_pixel(x, y) depths

Z buffer Illustration



Correct Final image



Top View

Z buffer Illustration

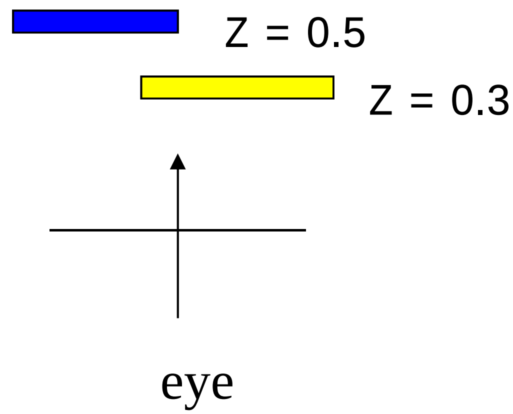
Step 1: Initialize the depth buffer

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 Illustration

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

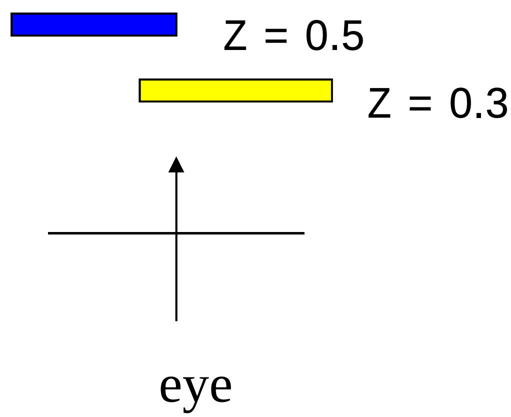
1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0
0.5	0.5	1.0	1.0
0.5	0.5	1.0	1.0



Z buffer Illustration

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

Z-Buffer Depth Compression

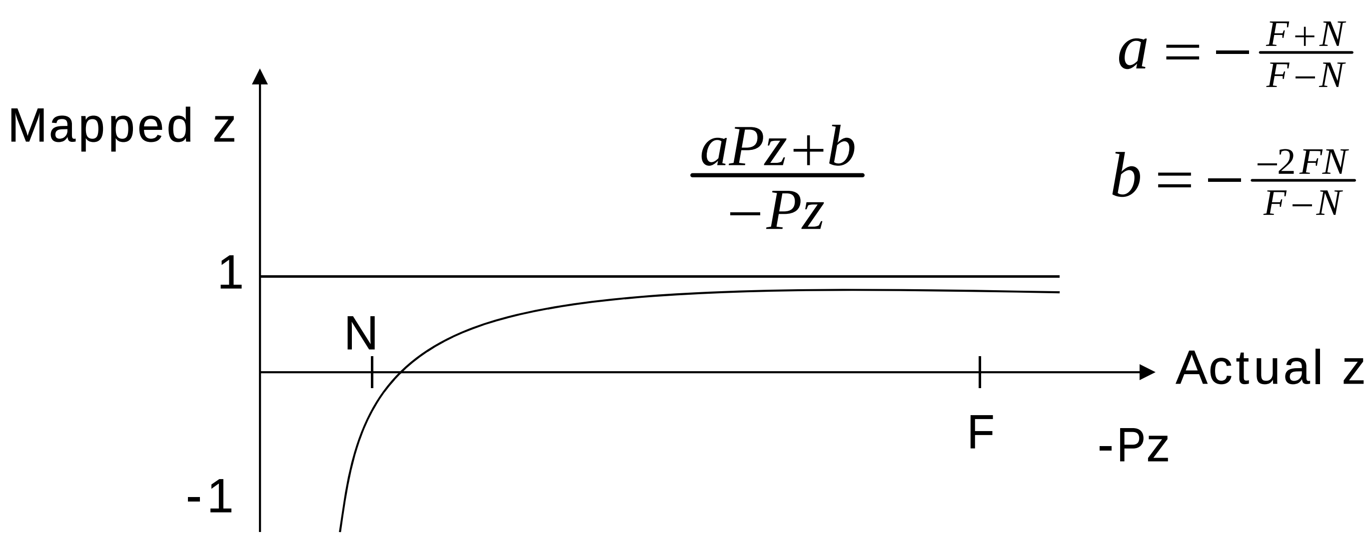
- **Pseudodepth calculation:** Recall that we chose parameters (a and b) to map z from range [near, far] to **pseudodepth** range[-1,1]

$$\begin{pmatrix} \frac{2N}{x\max - x\min} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2N}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & \frac{-(F+N)}{F-N} & \frac{-2FN}{F-N} \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

These values map z values of original view volume to [-1, 1] range

Z-Buffer Depth Compression

- This mapping is almost linear close to eye
- Non-linear further from eye, approaches asymptote
- Also limited number of bits
- Thus, two z values close to far plane may map to same pseudodepth: ***Errors!!***

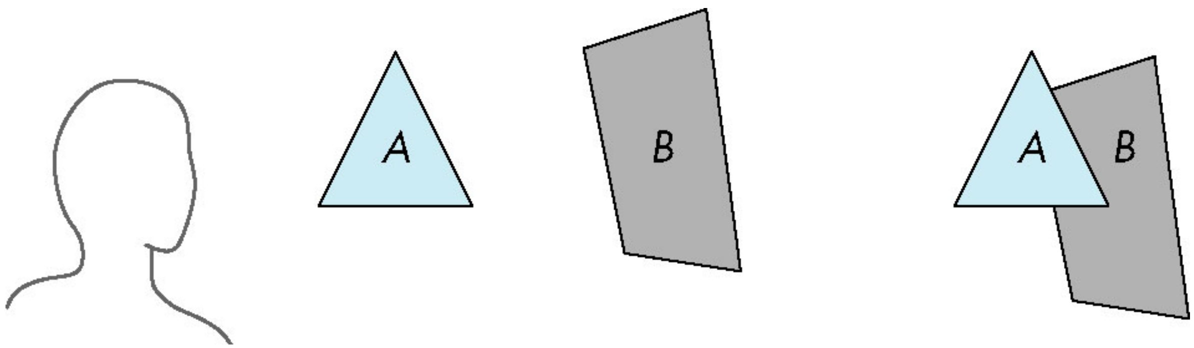


OpenGL HSR Commands

- 3 main commands to do HSR
- **glutInitDisplayMode(GLUT_DEPTH | GLUT_RGB)** instructs openGL to create depth buffer
- **glEnable(GL_DEPTH_TEST)** enables depth testing
- **glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)** initializes depth buffer every time we draw a new picture

Painter's HSR Algorithm

- Render polygons in back to front order so that polygons behind others are simply painted over



B behind A as seen by viewer

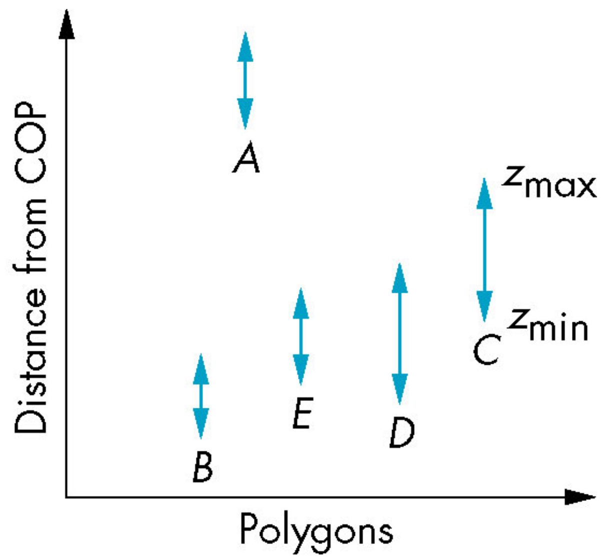
Fill B then A

Depth Sort

- Requires sorting of polygons (based on depth) first
  - O(n log n) calculation to sort polygon depths
  - Not every polygon is clearly in front or behind all other polygons

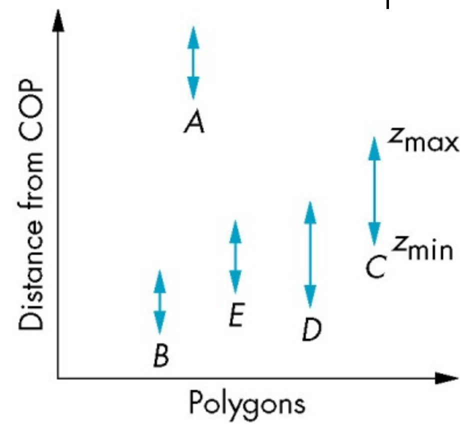
- Order polygons and deal with easy cases first, harder later

Polygons sorted by distance from COP

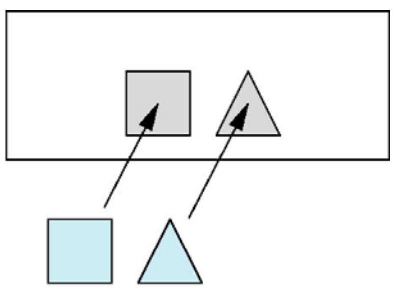


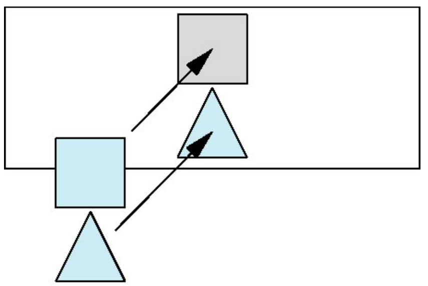
Easy Cases

- A lies behind all other polygons
  - Can render

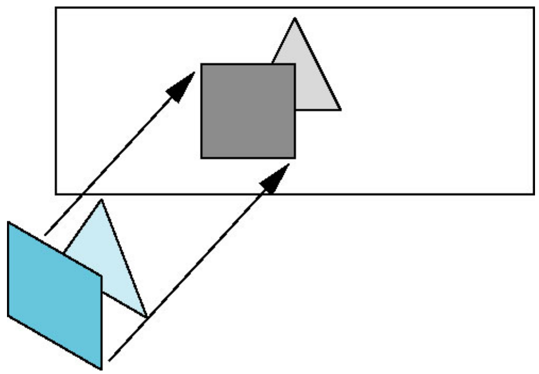


- Polygons overlap in z but not in either x or y
  - Can render independently

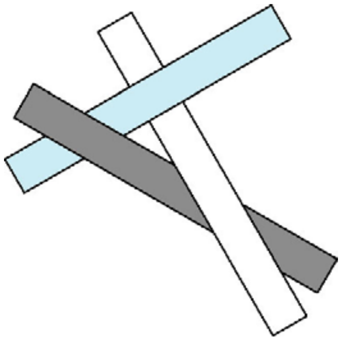




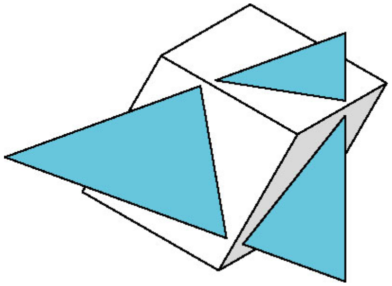
Hard Cases



Overlap in both (x,y) and z ranges



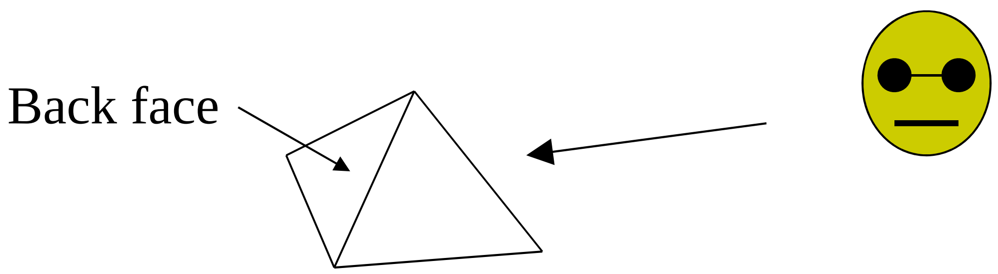
cyclic overlap



penetration

Back Face Culling

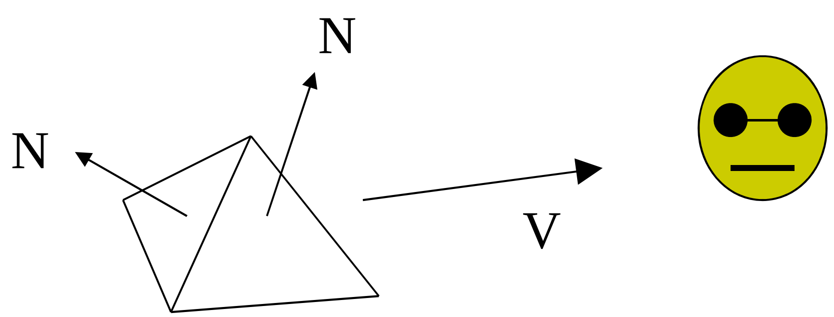
- Back faces: faces of opaque object that are "pointing away" from viewer
- **Back face culling:** remove back faces (supported by OpenGL)



- How to detect back faces?

Back Face 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



Backface test: F is backface if N.V < 0 why??

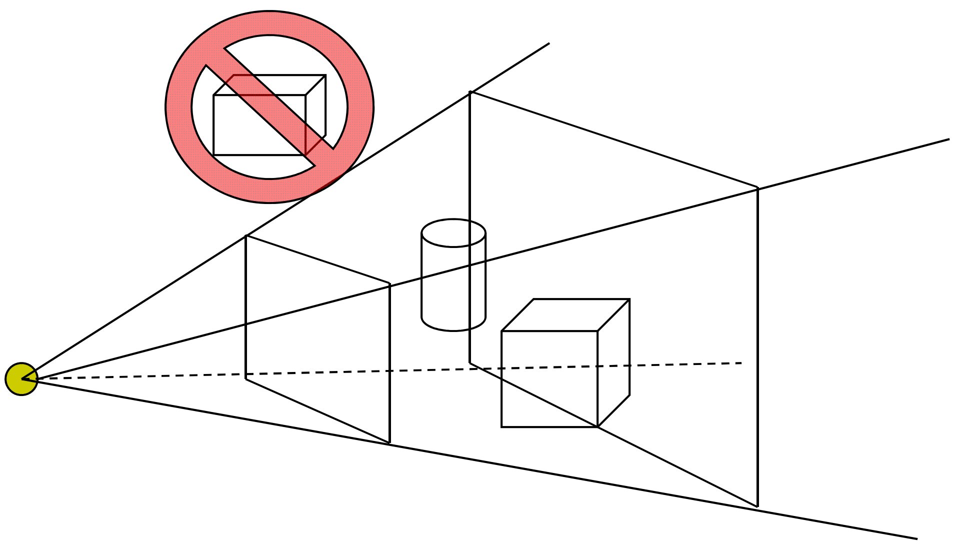
Back Face Culling: Draw mesh front faces

```
void drawFrontFaces( )
{
    for(int f = 0;f < numFaces; f++)
    {
        if(isBackFace(f, ….) continue;
        glDrawArrays(GL_POLYGON, 0, N);
    }
```

Note: In OpenGL we can simply enable culling but may not work correctly if we have nonconvex objects

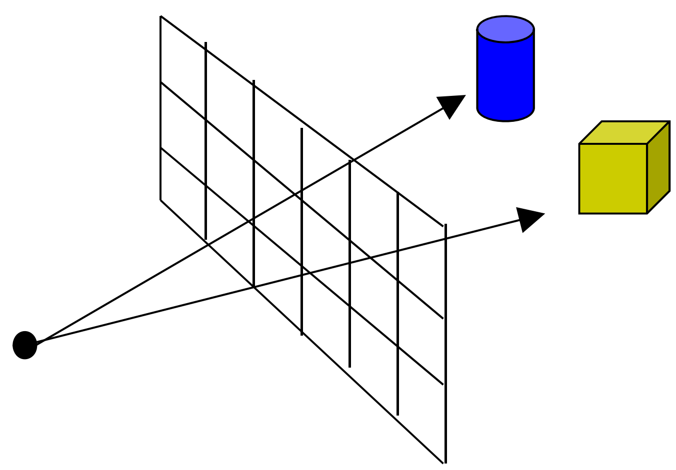
View-Frustum Culling

- Remove objects that are outside view frustum
- Done by 3D clipping algorithm (e.g. Liang-Barsky)



Ray Tracing

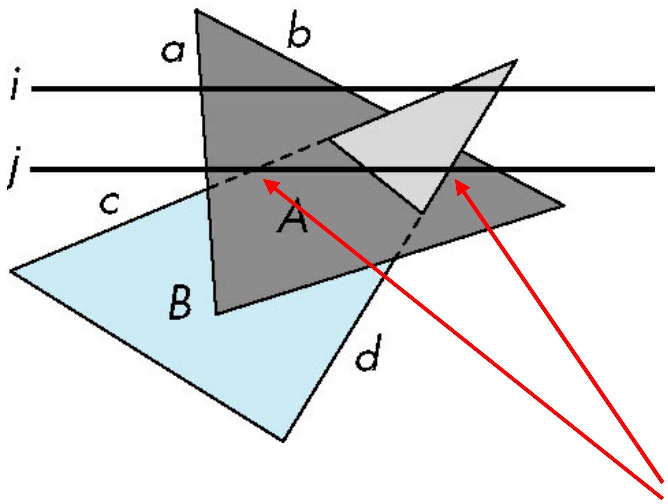
- Ray tracing is another image space method
- Ray tracing: Cast a ray from eye through each pixel to the world.
- Question: what does eye see in direction looking through a given pixel?



More on this topic later

Scan-Line Algorithm

- Can combine shading and hsr through scan line algorithm

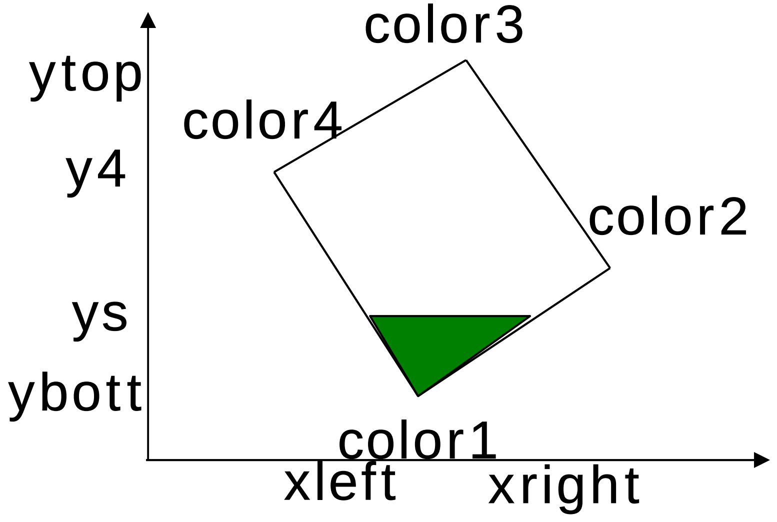


scan line i: no need for depth information, can only be in no or one polygon

scan line j: need depth information only when in more than one polygon

Combined z-buffer and Gouraud Shading (Hill)

```
for(int y = ybott; y <= ytop; y++)  // for each scan line
{
    for(each polygon){
    find xleft and xright
    find dleft,  dright, and dinc
    find colorleft and colorright, and colorinc
    for(int x = xleft, c = colorleft, d = dleft; x <= xright;
                                x++, c+= colorinc, d+= dinc)
    if(d < d[x][y])
    {
       put c into the pixel at (x, y)
       d[x][y] = d; // update closest depth
    }}
}
```



References

- Angel and Shreiner, Interactive Computer Graphics, 6th edition
- Hill and Kelley, Computer Graphics using OpenGL, 3rd edition, Chapter 9