CS 543 - Computer Graphics: Hidden Surface Removal

by

Robert W. Lindeman

gogo@wpi.edu

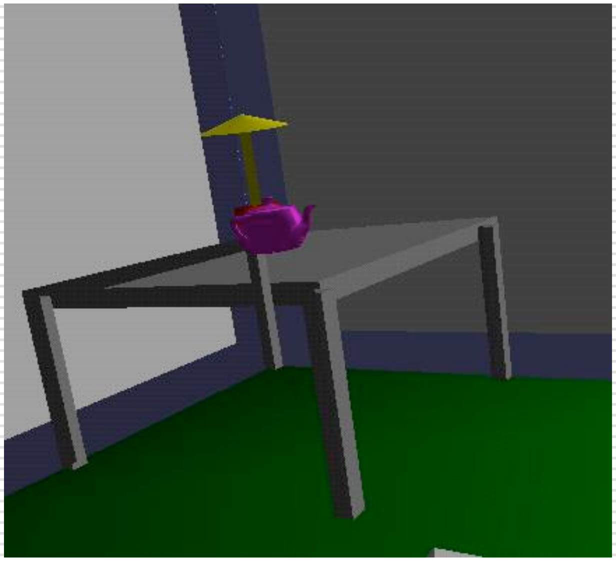
(with help from Emmanuel Agu ;-)

Hidden Surface Removal

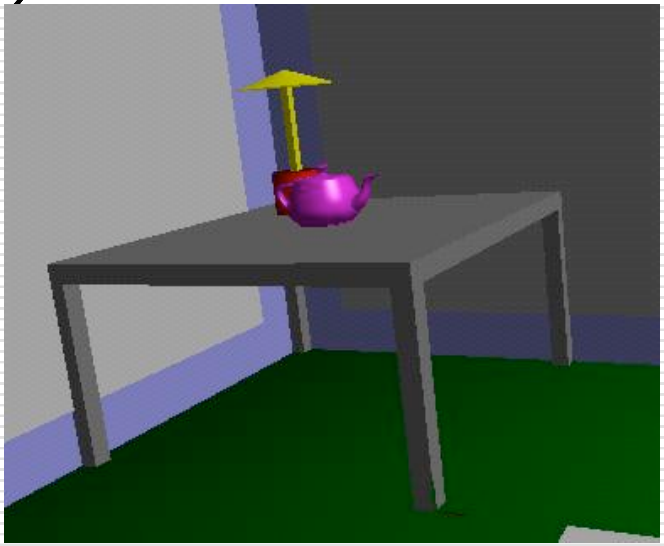
- Drawing polygonal faces on screen consumes CPU cycles
- We cannot see every surface in scene
- To save time, draw only surfaces we see
- Surfaces we cannot see, and their elimination methods
  - **Occluded surfaces:** hidden surface removal (visibility)
  - **Back faces:** back face culling
  - **Faces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

Visibility
Hidden Surface Removal

- A correct rendering requires correct visibility calculations
  - When multiple opaque polygons cover the same screen space, only the closest one is visible (remove the other hidden surfaces)

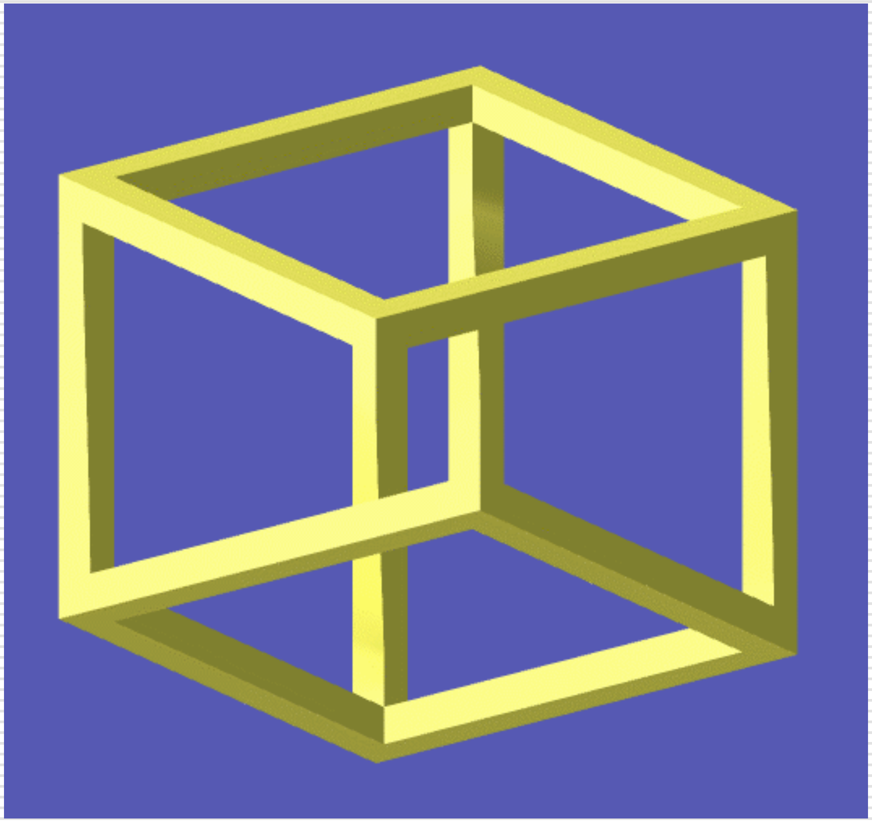


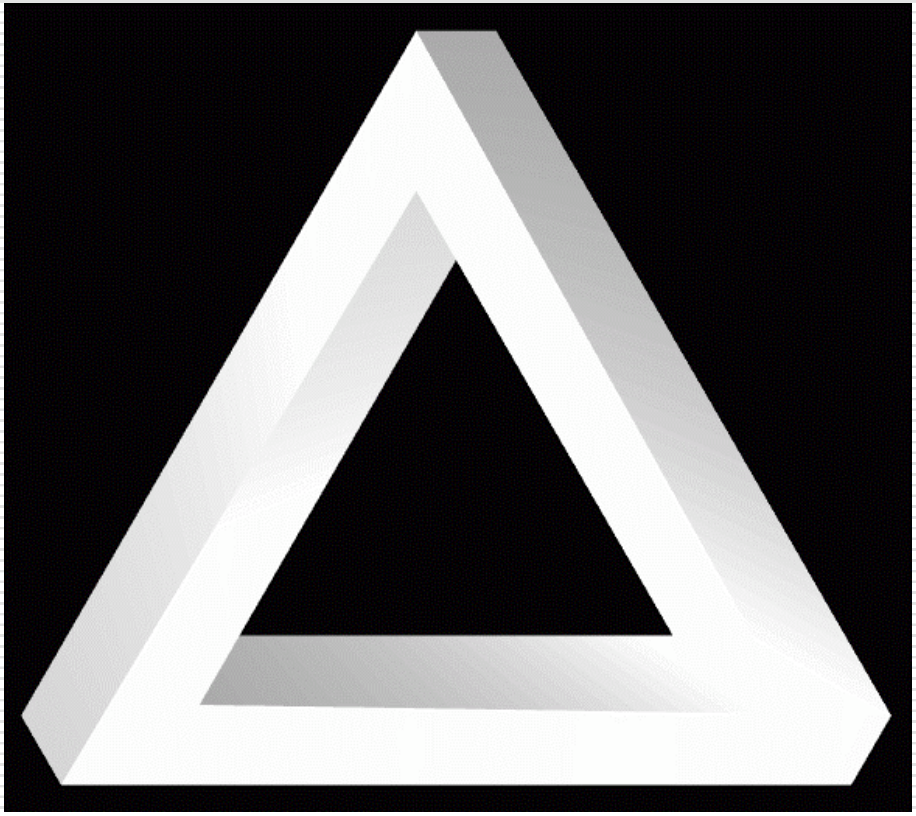
Wrong visibility



Correct visibility

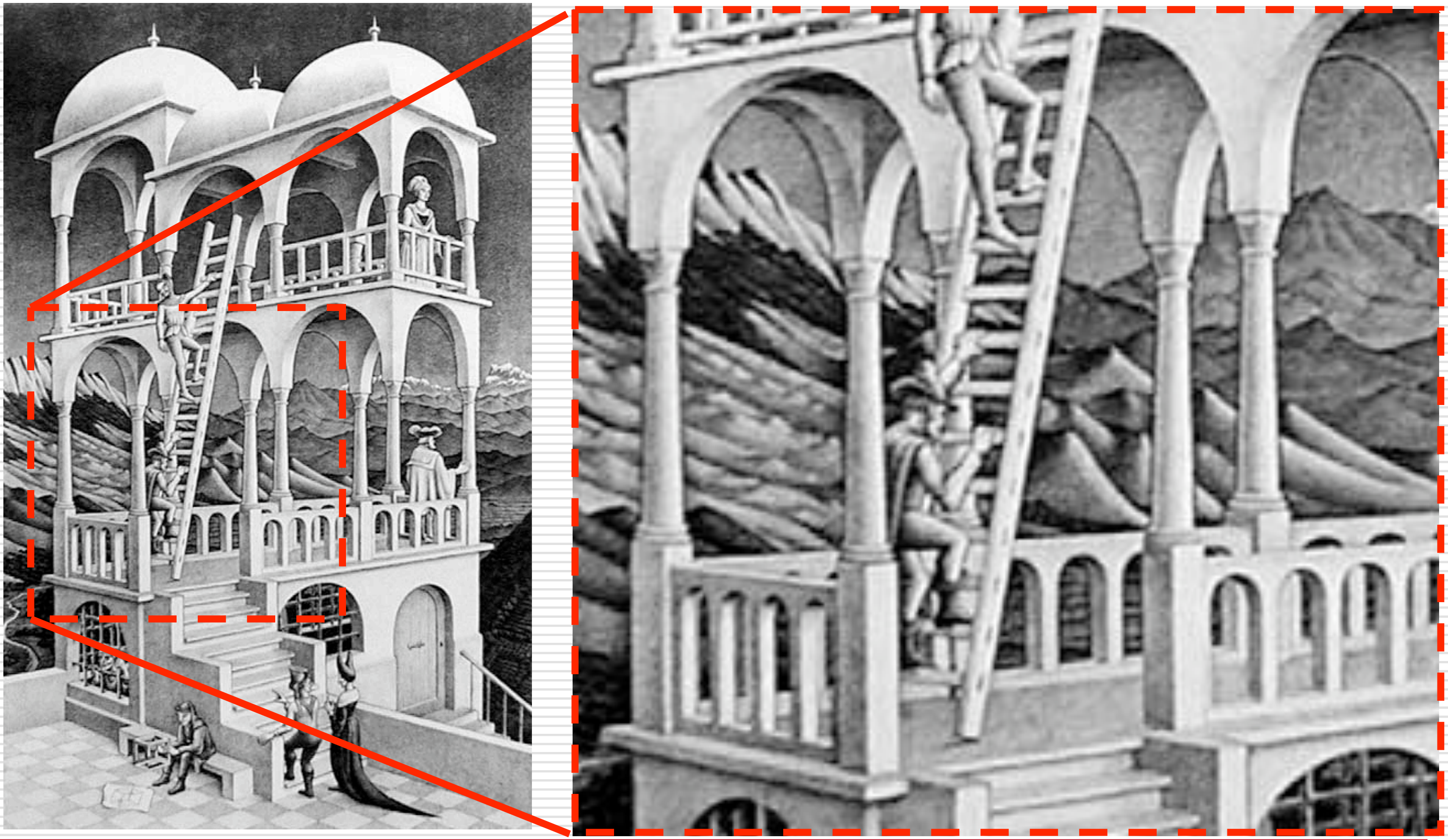
Visibility
Hidden Surface Removal (cont.)





http://www.worldofescher.com/

Visibility
Hidden Surface Removal (cont.)

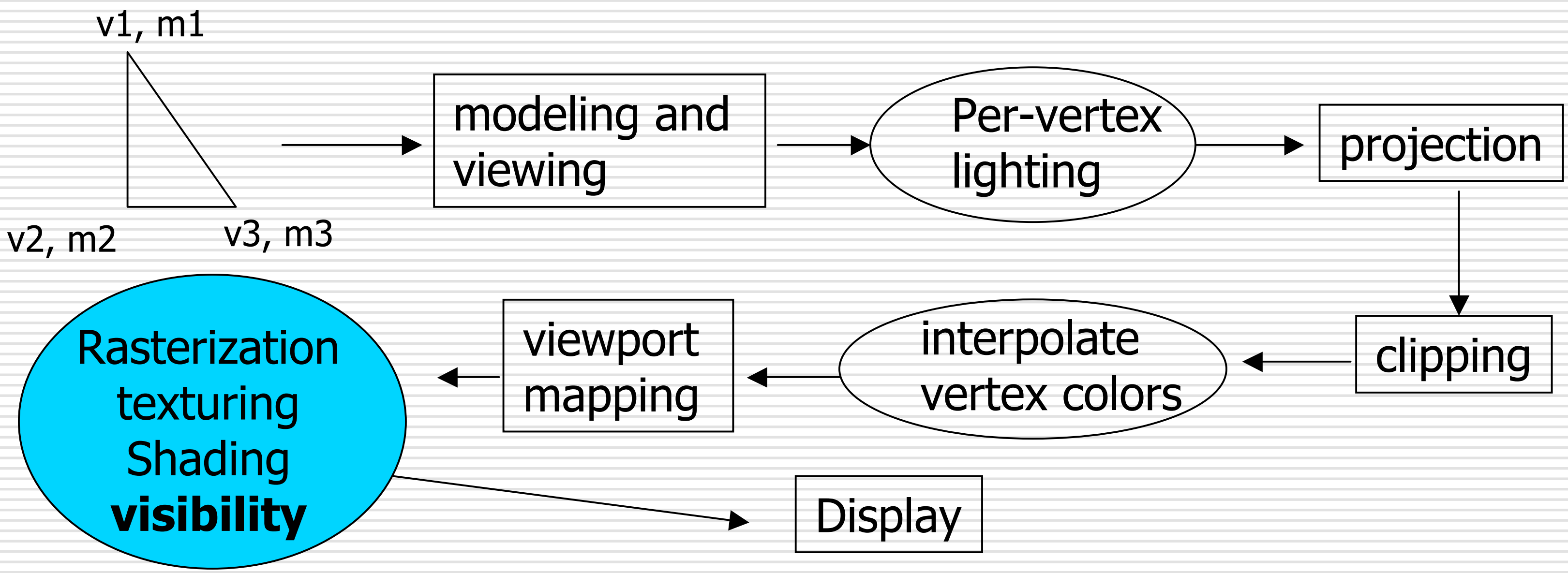


Visibility
Hidden Surface Removal (cont.)

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

Visibility
Hidden Surface Removal (cont.)

- Where is visibility performed in the graphics pipeline?



Note: Map (x,y) values to screen (draw) and use z value for depth testing

OpenGL: Image-Space Approach

- Determine which of the ***n*** objects is visible to each pixel on the image plane

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

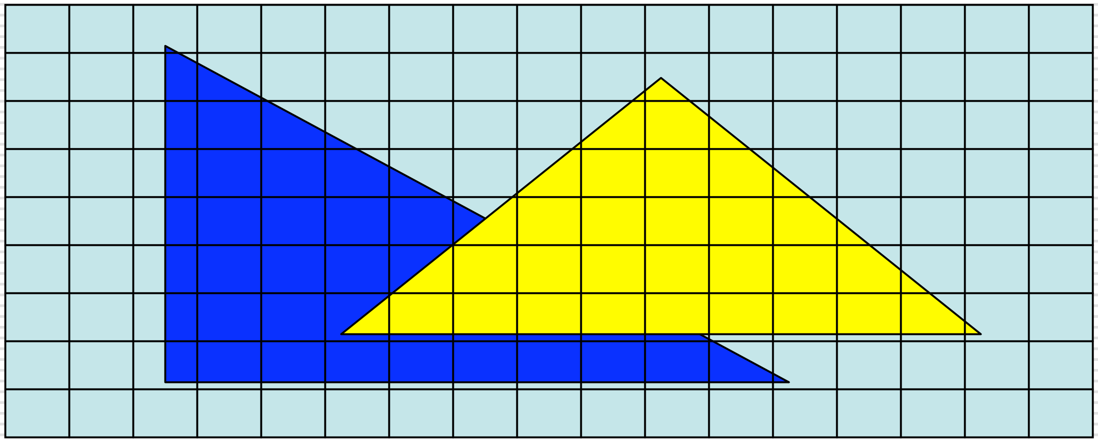


Image Space Approach
Z-buffer

- Method used in most of graphics hardware (and thus OpenGL)
  - Z-buffer (or depth buffer) algorithm
- Requires lots of memory
- Recall
  - After projection transformation, in viewport transformation
    - x,y used to draw screen image, mapped to viewport
    - z component is mapped to pseudo-depth with range [0,1]
- Objects/polygons are made up of vertices
- Hence, we know depth z at polygon vertices
- Point on object seen through pixel may be between vertices
- Need to interpolate to find z

Image Space Approach Z-buffer (cont.)

- 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

Z (Depth) Buffer Algorithm

- How do we choose the polygon that has the closet Z for a given pixel?
- Example: eye at Z = 0, farther objects have increasingly positive values, between 0 and 1
  1. Initialize (clear) every pixel in the Z buffer to 1.0
  2. Track polygon Zs
  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:

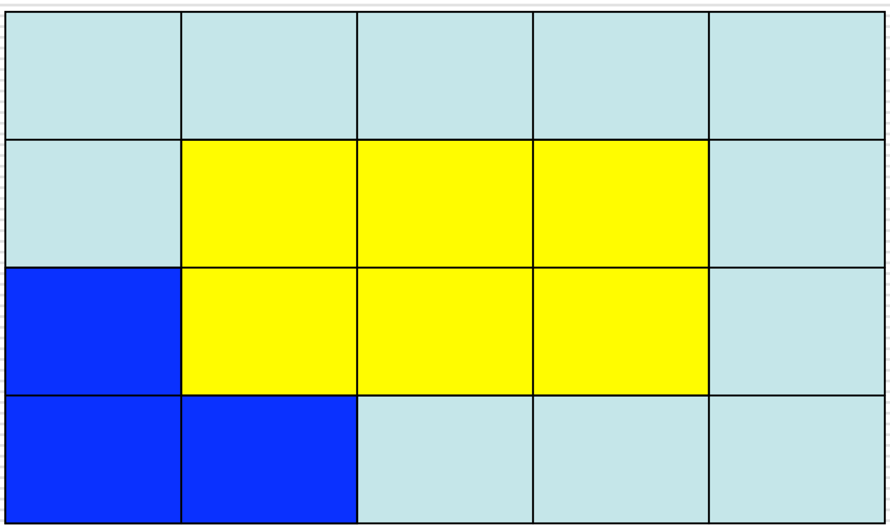
Z (Depth) Buffer Algorithm (cont.)

```
foreach polygon in scene  {
  foreach pixel (x,y) inside the polygon projection  {
    if( z_polygon_pixel( x, y ) < z_buffer( x, y ))  {
      z_buffer( x, y ) = z_polygon_pixel( x, y );
      color_buffer( x, y ) = polygon color at ( x, y )
    }
  }
}
```

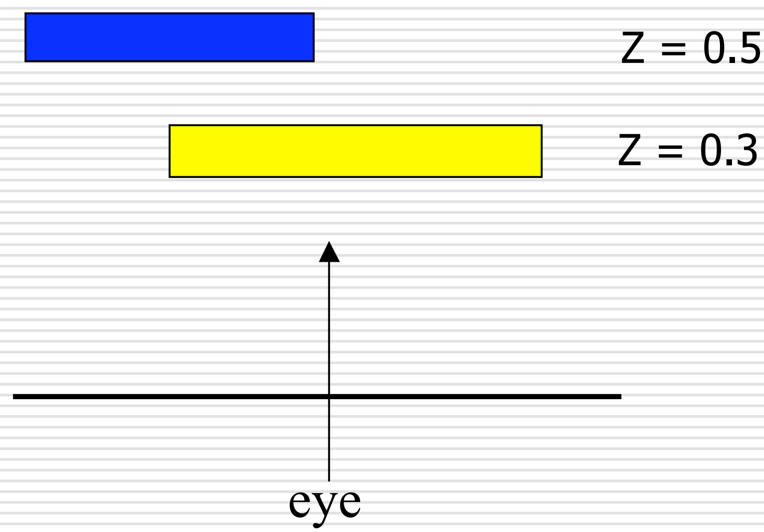
Note: We know the depths at the vertices. Interpolate for interior z_polygon_pixel(x, y) depths

Z-Buffer Example

Correct Final image



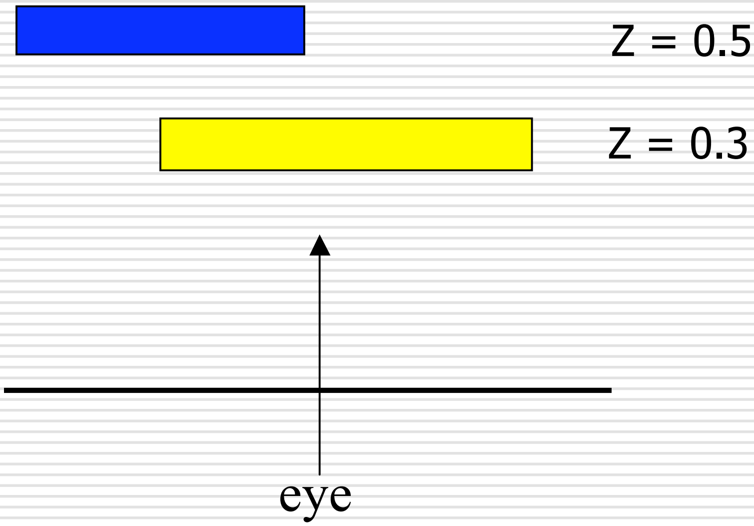
Top View



Z-Buffer Example (cont.)

- **Step 1**: Initialize the depth buffer

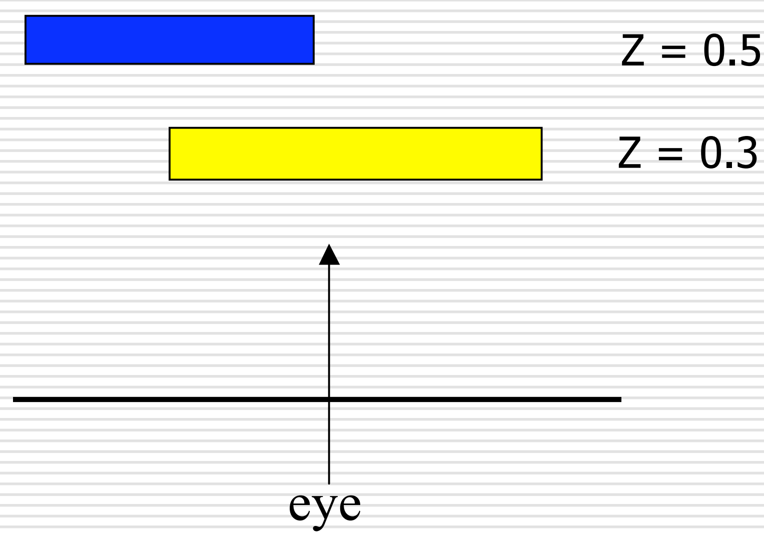
1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0



Z-Buffer Example (cont.)

- **Step 2**: Draw the blue polygon, assuming the program draws blue polygon first (the order does not affect the final result anyway)

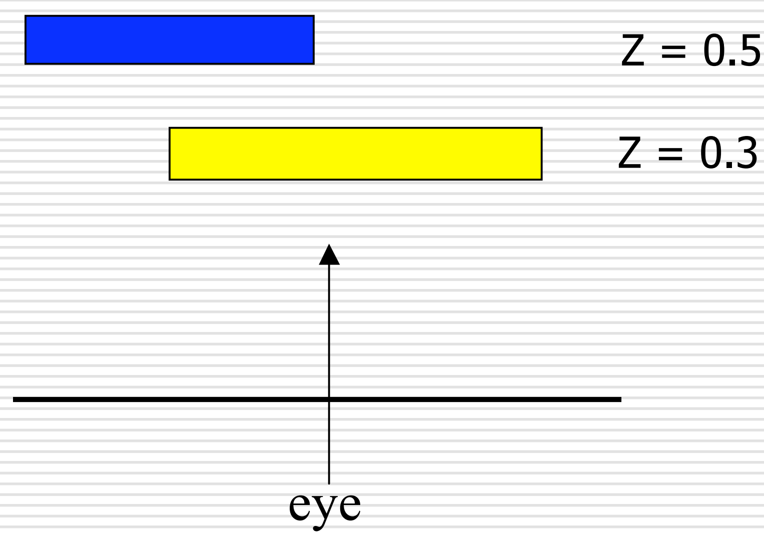
1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0
0.5	0.5	1.0	1.0	1.0
0.5	0.5	1.0	1.0	1.0



Z-Buffer Example (cont.)

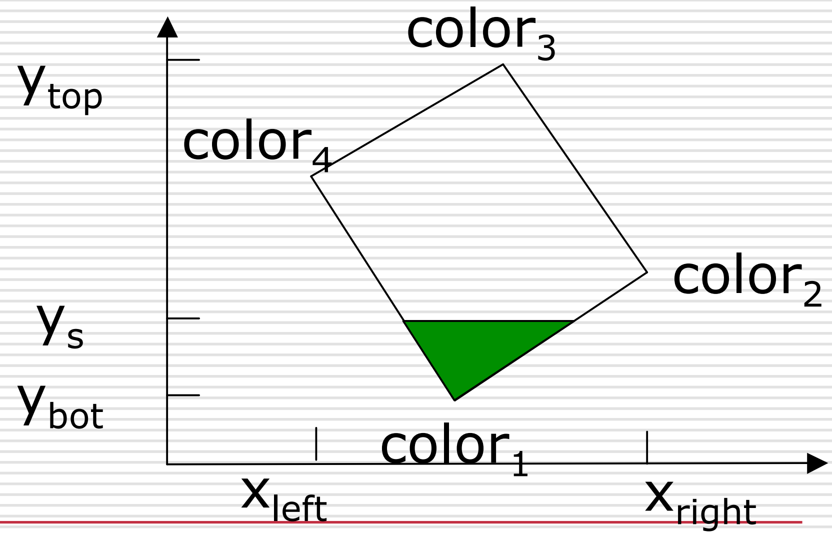
- **Step 3**: Draw the yellow polygon
  - Z-buffer drawback: wastes resources by rendering a face, and then drawing over it

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



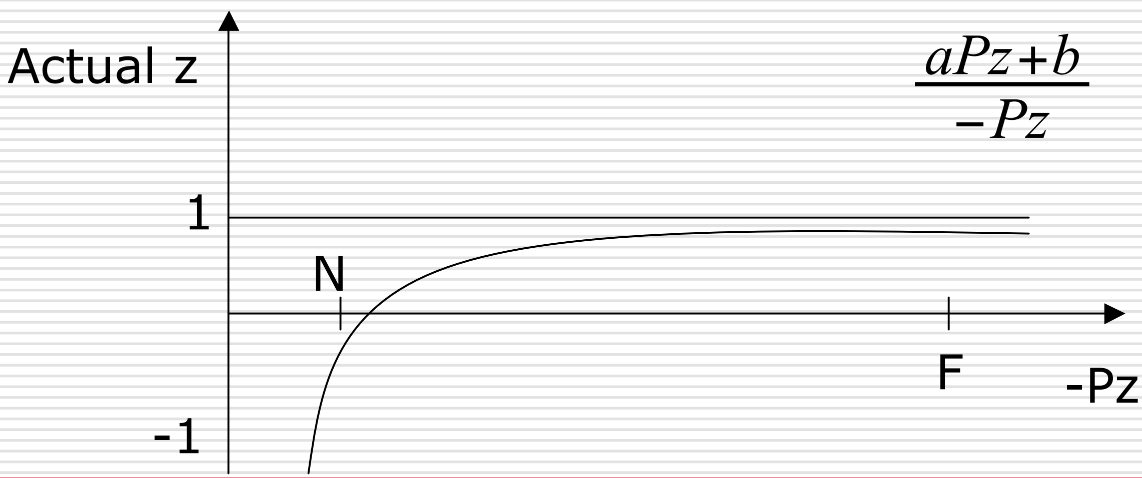
Combined Z-buffer and Gouraud Shading

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



Z-Buffer Depth Compression

- Recall that we chose parameters **a** and **b** to map z from range [near, far] to **pseudodepth** range[0,1]
- This mapping is almost linear close to the eye, but is non-linear further from the eye, approaches asymptote
- Also, limited number of bits
- Thus, two z values close to far plane may map to same pseudodepth: ***Errors!!***



$$\frac{aPz+b}{-Pz}$$

$$a = -\frac{F+N}{F-N}$$

$$b = -\frac{-2FN}{F-N}$$

OpenGL Hidden-Surface Removal (HSR) Commands

- Primarily three commands to do HSR
  - Instruct OpenGL to create a depth buffer

```
glutInitDisplayMode(GLUT_DEPTH | GLUT_RGB)
```

  - Enable depth testing

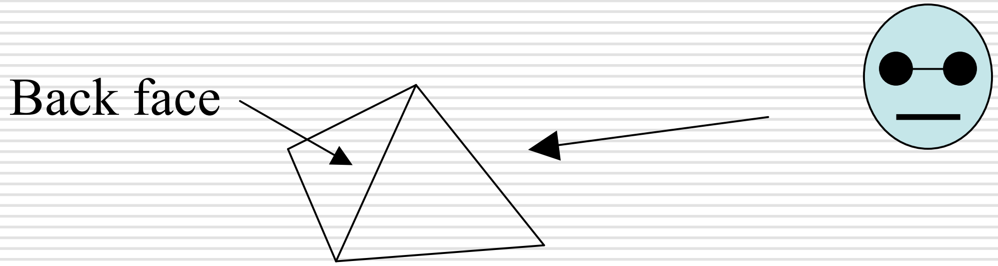
```
glEnable(GL_DEPTH_TEST)
```

  - Initialize the depth buffer every time we draw a new picture

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
```

Back Face Culling

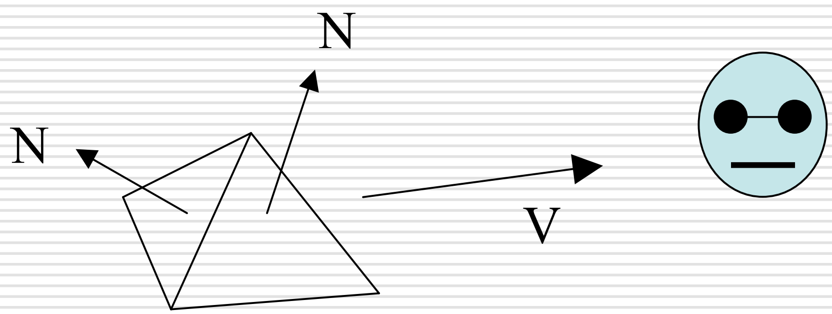
- Back faces: faces of an opaque object which are "pointing away" from the viewer
- Back face culling
  - Remove back faces (supported by OpenGL)



- How can we detect back faces?

Back Face Culling (cont.)

- If we find a back face, 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 back face
- **P** is a point on **F**
- Form view vector, **V** as (**eye** – **P**)
- **N** is normal to face **F**



- Back face test: **F** is back face if **N.V** < **0**

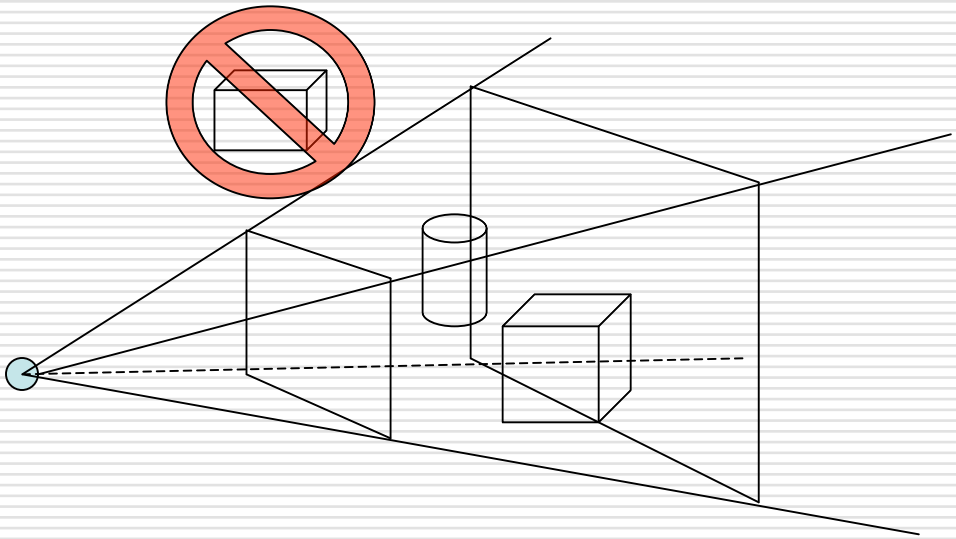
Back Face Culling: Draw Front Faces of a Mesh

```
void Mesh::drawFrontFaces( void )  {
  for( int f = 0; f < numFaces; f++ )  {
    if( isBackFace( f, ... ) continue;
    glBegin( GL_POLYGON );
      int in = face[f].vert[v].normIndex;
      int iv = face[v].vert[v].vertIndex;
      glNormal3f( norm[in].x, norm[in].y, norm[in].z);
      glVertex3f( pt[iv].x, pt[iv].y, pt[iv].z );
    glEnd( );
  }
}
```

Refer to: Case 7.3, p. 375 in Hill

View-Frustum Culling

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



Ray Tracing

- Ray tracing is another example of image space method
- Ray tracing
  - Cast a ray from eye through each pixel to the world
- Answers the question:
  - What does eye see in direction looking through a given pixel?

Painter's Algorithm

- A depth-sorting method
- Surfaces are sorted in the order of decreasing depth
- Surfaces are drawn in the sorted order, and overwrite the pixels in the frame buffer
- Subtle difference from depth buffer approach
  - Entire face drawn
- Two problems
  - It can be nontrivial to sort the surfaces
  - There can be no solution for the sorting order

Painter's Algorithm (cont.)