CS 4731: Computer Graphics
Lecture 17: Hidden Surface Removal

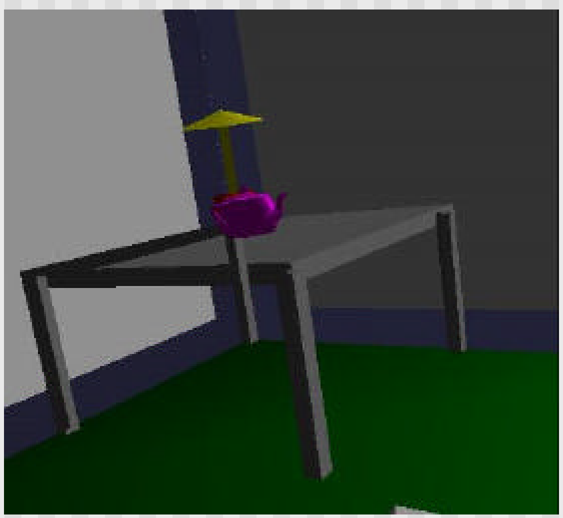
Emmanuel Agu

Hidden surface Removal

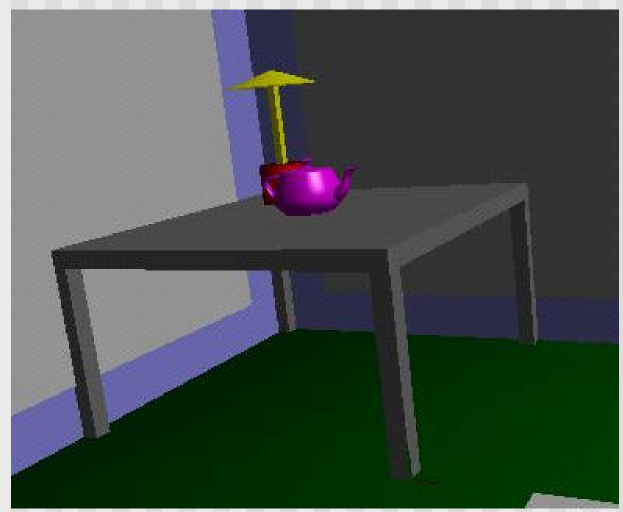
- Drawing polygonal faces on screen consumes CPU cycles
- We cannot see every surface in scene
- To save time, draw only surfaces we see
- Surfaces we cannot see and their elimination methods:
  - **Occluded surfaces:** hidden surface removal (visibility)
  - **Back faces:** back face culling
  - **Faces outside view volume:** viewing frustrum culling
- Definitions:
  - **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
- Correct visibility – when multiple opaque polygons cover the same screen space, only the closest one is visible (remove the other hidden surfaces)



wrong visibility



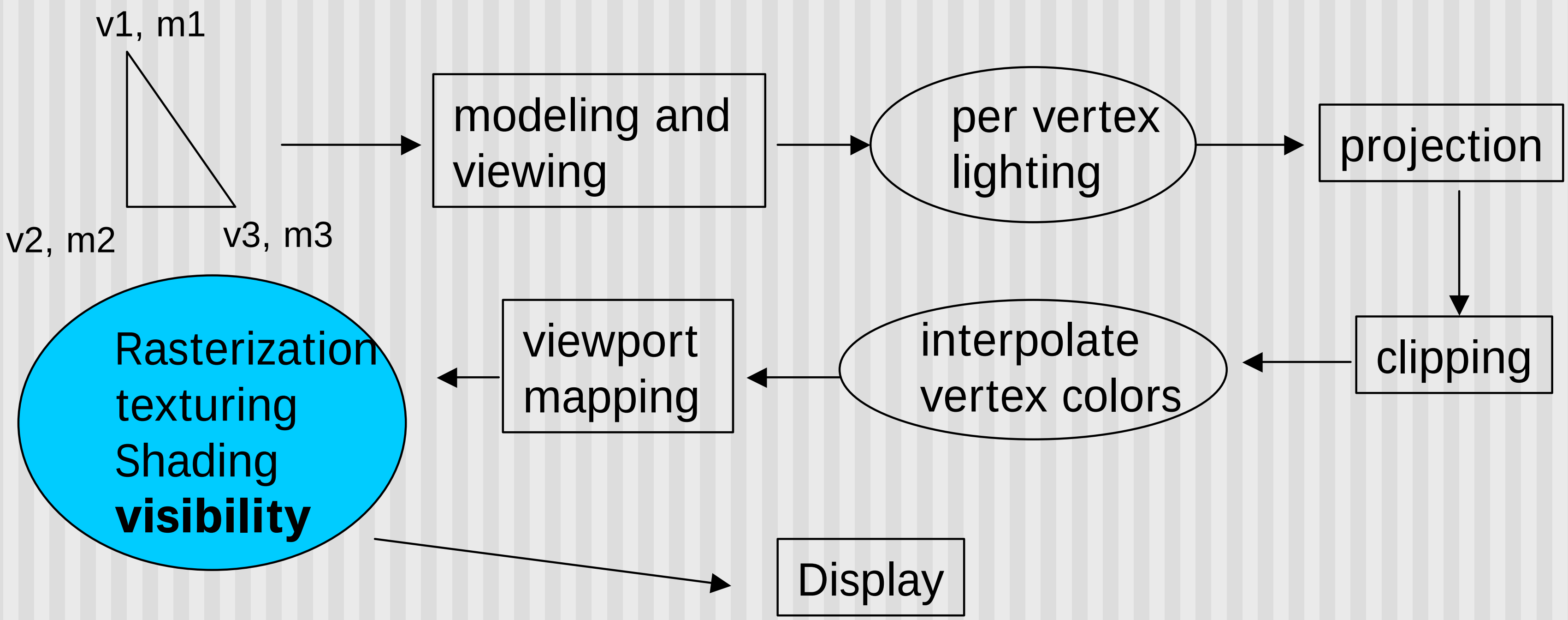
Correct visibility

Visibility (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)

Visibility (hidden surface removal)

- Where is visiblity 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
}
```

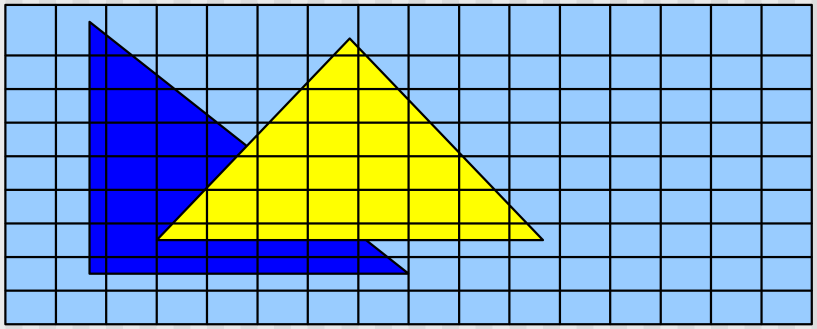


Image Space Approach – Z-buffer

- 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

- 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 to 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

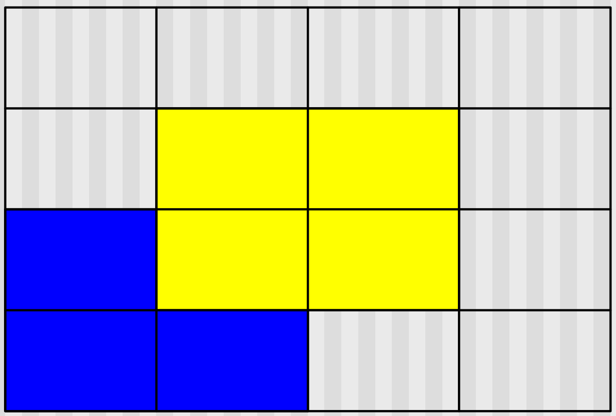
1. Initialize (clear) every pixel in the z buffer to 1.0
2. Track polygon z's.
3. As we rasterize polygons, check to see if polygon's z through this pixel is less than current minimum z through this pixel
4. Run the following loop:

Z (depth) Buffer Algorithm

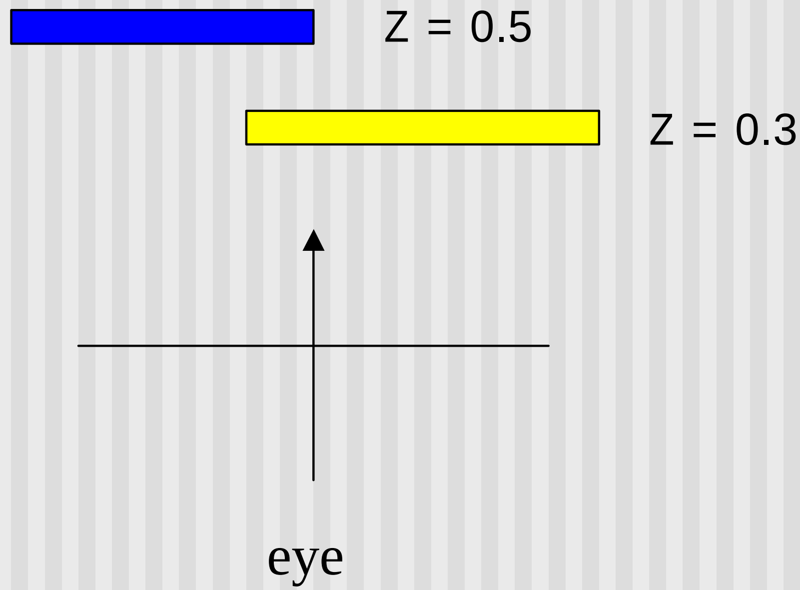
```
For  each polygon  {
   for each pixel (x,y) inside the polygon projection area  {
         if  (z_polygon_pixel(x,y) < depth_buffer(x,y) ) {
               depth_buffer(x,y) = z_polygon_pixel(x,y);
               color_buffer(x,y) = polygon color at (x,y)
         }
     }
  }
```

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

Z buffer example



Correct Final image



Top View

Z buffer example

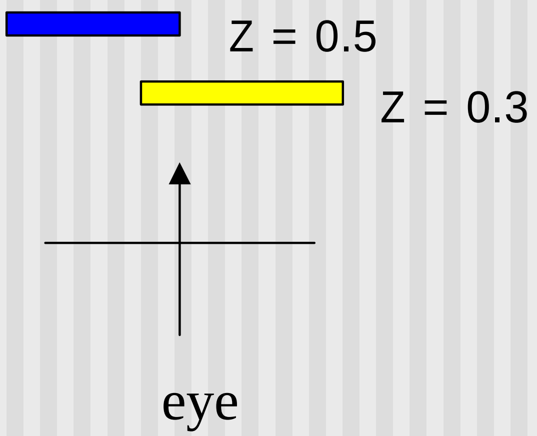
Step 1: Initialize the depth buffer

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

Z buffer example

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

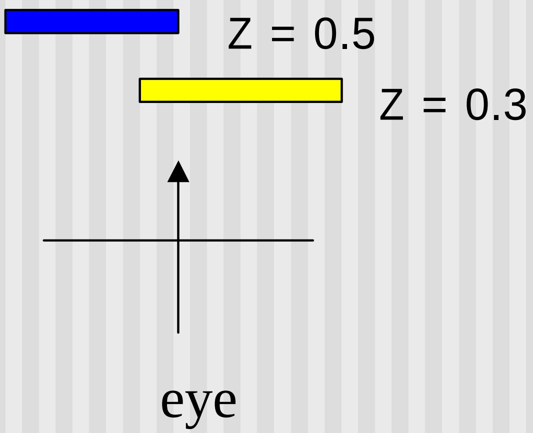
1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0
0.5	0.5	1.0	1.0
0.5	0.5	1.0	1.0



Z buffer example

Step 3: Draw the yellow polygon

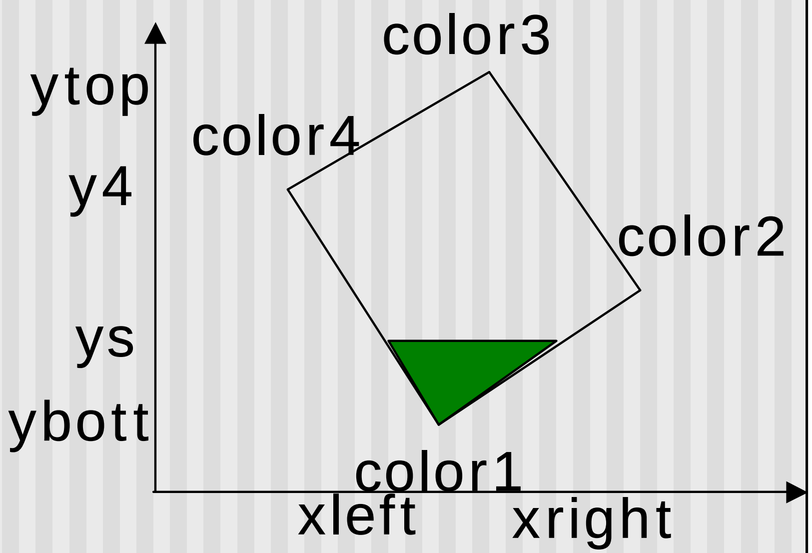
1.0	1.0	1.0	1.0
1.0	0.3	0.3	1.0
0.5	0.3	0.3	1.0
0.5	0.5	1.0	1.0



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

Combined z-buffer and Gouraud Shading (fig 8.31)

```
for(int y = ybott; y <= ytop; y++)  // for each scan line
{
    for(each polygon){
    find xleft and xright
    find dleft and 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
    }}
}
```



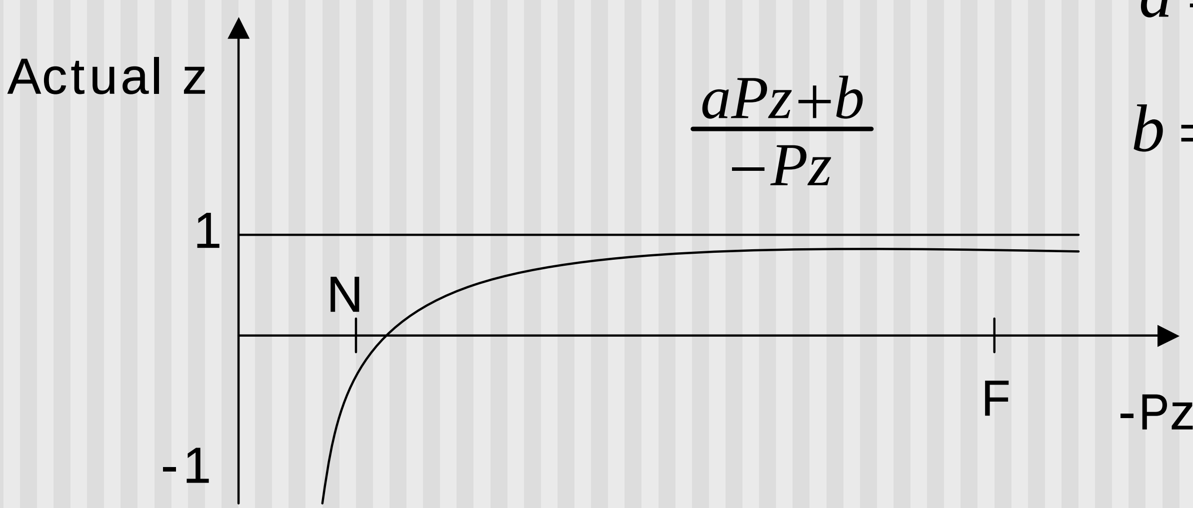
Z-Buffer Depth Compression

- 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 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!!***

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

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

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

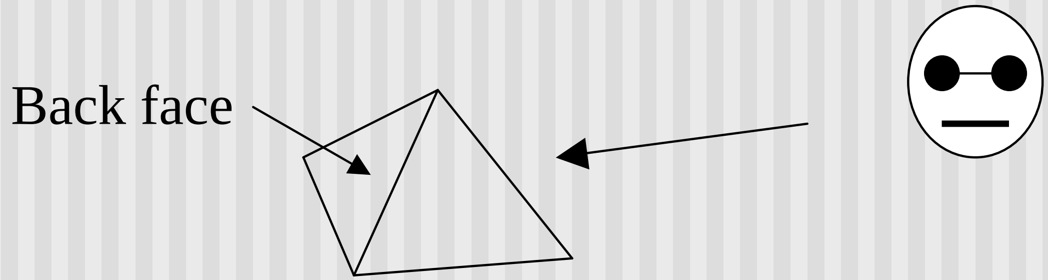


OpenGL HSR Commands

- Primarily three 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 the depth buffer every time we draw a new picture

Back Face Culling

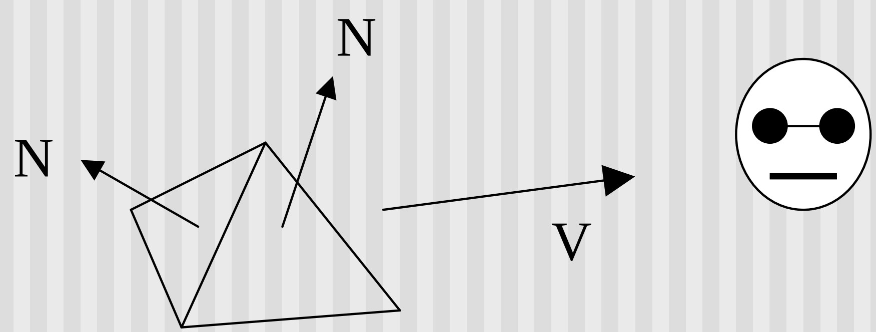
- Back faces: faces of opaque object which are “pointing away” from viewer
- Back face culling – remove back faces (supported by OpenGL)



- 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??

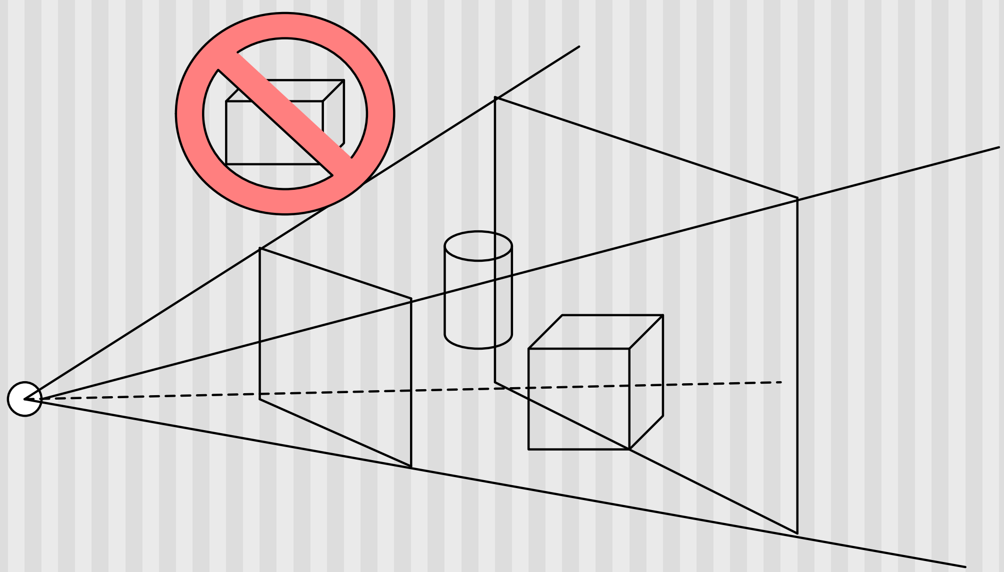
Back Face Culling: Draw mesh front faces

```
void Mesh::drawFrontFaces( )
{
    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( );
}
```

Ref: case study 7.5, pg 406, Hill

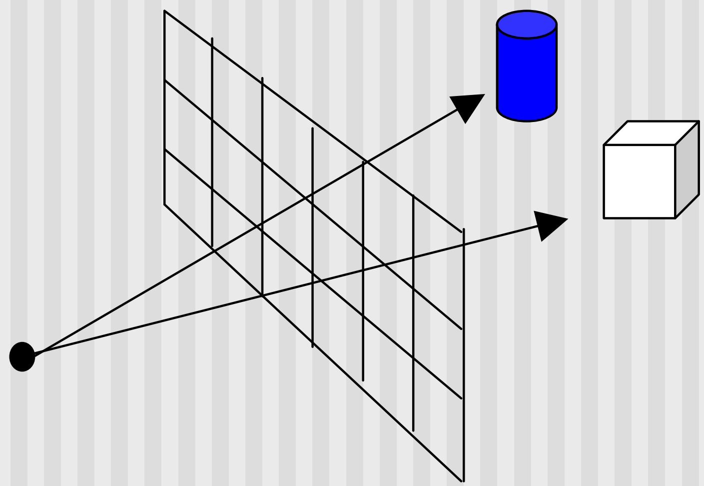
View-Frustum Culling

- 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.
- Question: what does eye see in direction looking through a given pixel?



Will discuss more later

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

References

- Hill, section 8.5