

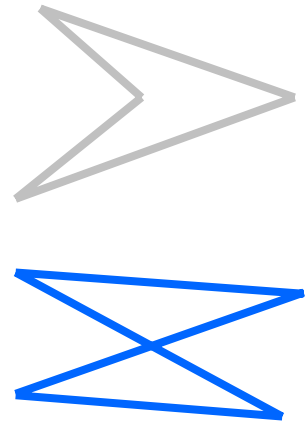
# COMP 371 -- Winter 2012

## Computer Graphics

3D Object Representation: Triangle Mesh  
Normals  
Hidden Surface Removal  
Visual Realism and Visibility  
BSP Trees

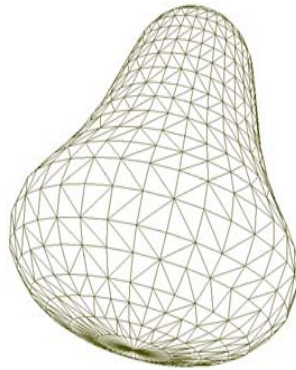
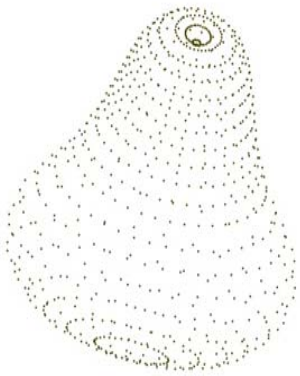
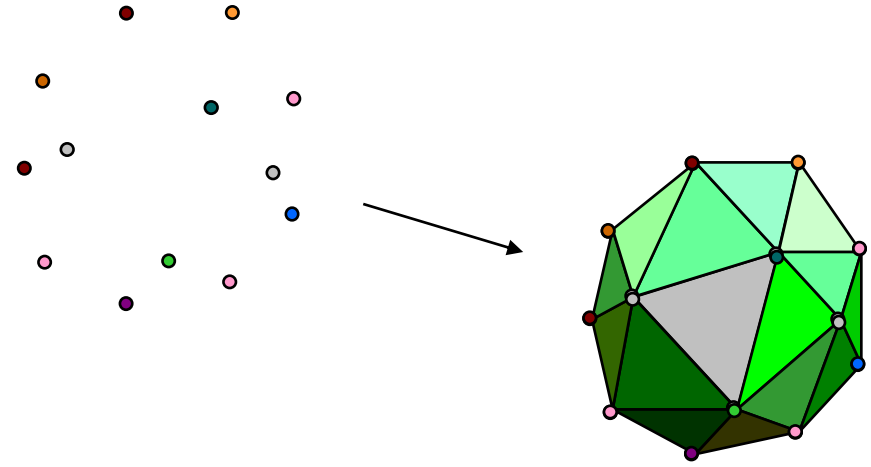
# Polygon Meshes

- Any shape can be modeled out of polygons
  - if you use enough of them...
- Polygons with how many sides?
  - Can use triangles, quadrilaterals, pentagons, ... n-gons
  - Triangles are most common.
  - When  $> 3$  sides are used, ambiguity about what to do when polygon nonplanar, or concave, or self-intersecting.
- Polygon meshes are built out of
  - *vertices* (points)
  - *edges* (line segments between vertices)
  - *faces* (polygons bounded by edges)



# Samples and connectivity

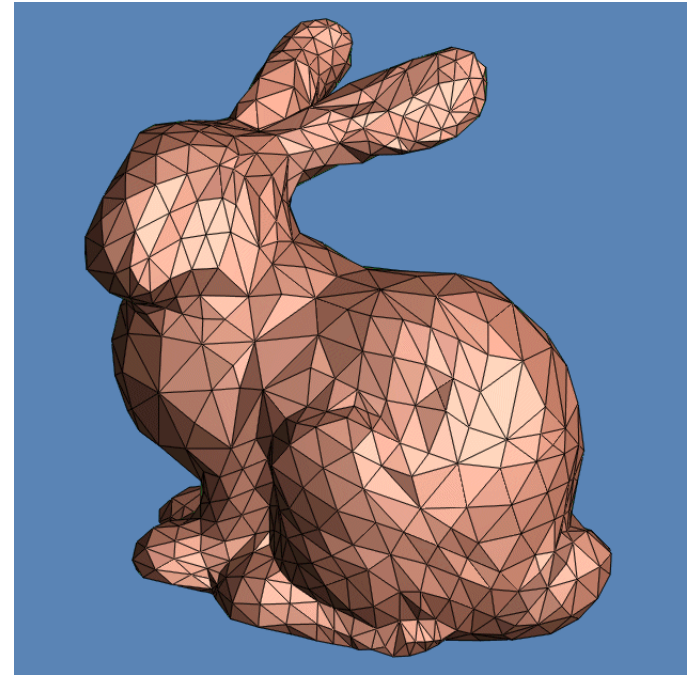
- Samples (“vertices”)
  - Location  $(x,y,z)$
- Connectivity (“triangles”)
  - Define how surface interpolates samples
  - Specifies surface as a set of triangles
  - Associates each triangle with 3 samples (called corners)



# 3D objects: Wavefront OBJ file format

OBJ mesh file format is very commonly used, and is quite simple. Each line starting with a character indicating what type of information it is. Here is one example of an OBJ file describing a tetrahedron:

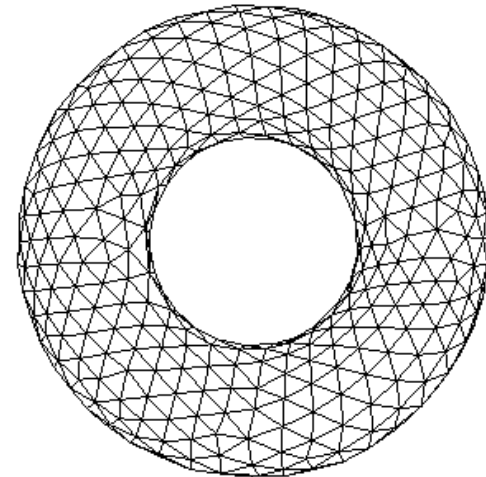
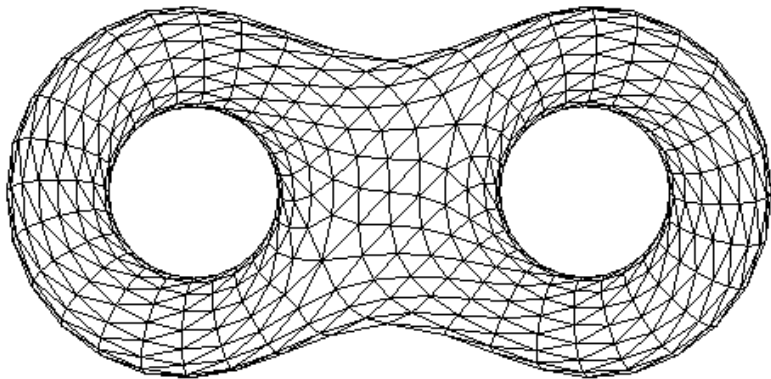
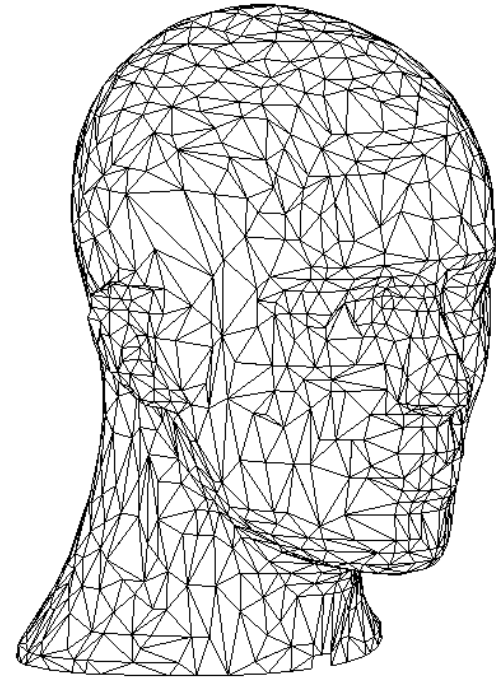
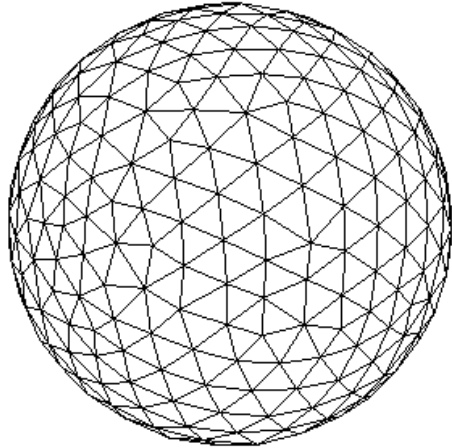
```
# OBJ file format with ext .obj
v 1.0 0.0 0.0
v 0.0 1.0 0.0
v 0.0 0.0 1.0
v 0.0 0.0 0.0
f 2 4 3
f 4 2 1
f 3 1 2
f 1 3 4
```



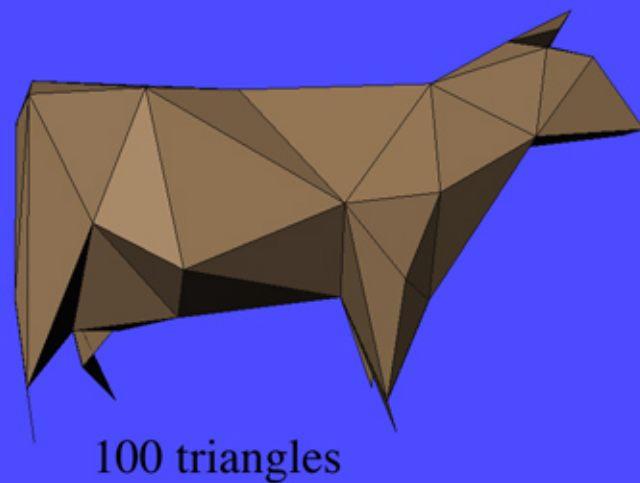
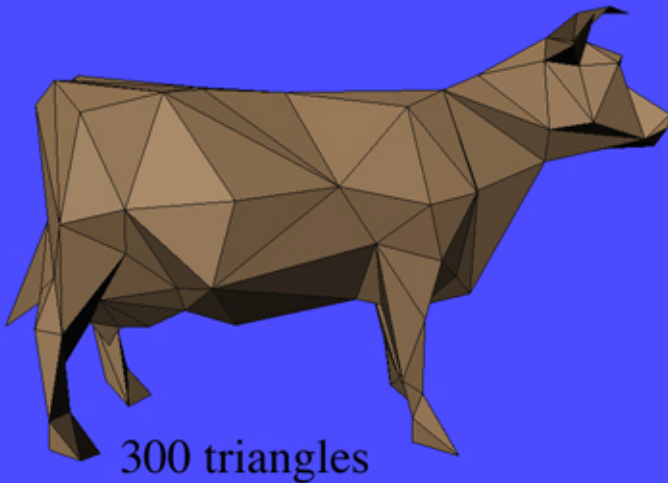
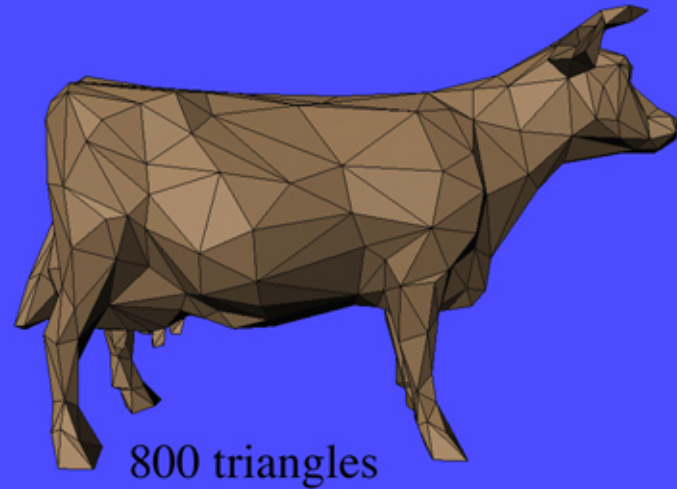
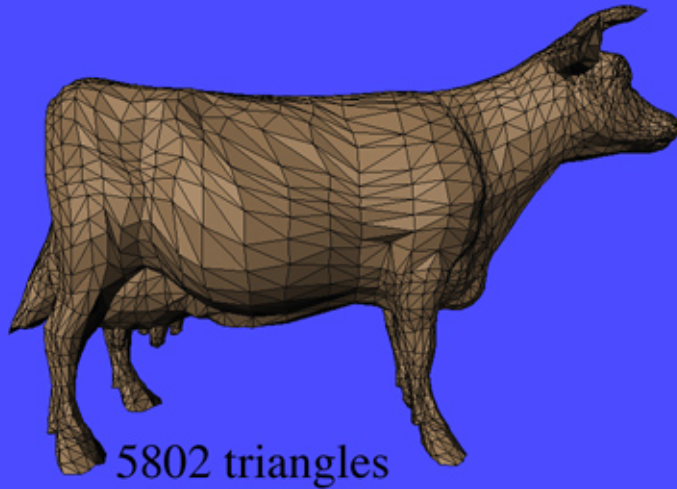
The vertices are indexed starting from 1. The faces are given as triples of vertex indices. There are three kinds of lines allowed. Comment is any line starting with ' # '

- Vertex lines start with 'v' and give three coordinates of a vertex:  $v \ x \ y \ z$
- Face lines start with 'f' and give three vertex indices that form a face:  $f \ v1 \ v2 \ v3$

# Examples: Triangle Mesh



# How Many Triangles to Use?

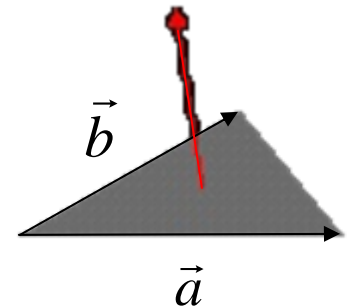
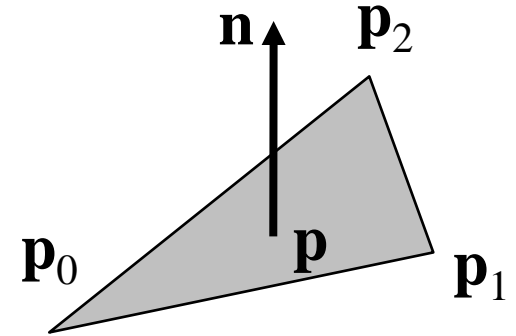


# Surface Normal

$$\text{plane } \mathbf{n} \cdot (\mathbf{p} - \mathbf{p}_0) = 0$$

$$\mathbf{n} = (\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_2 - \mathbf{p}_0)$$

$$\text{normalize } \mathbf{n} \leftarrow \mathbf{n} / |\mathbf{n}|$$



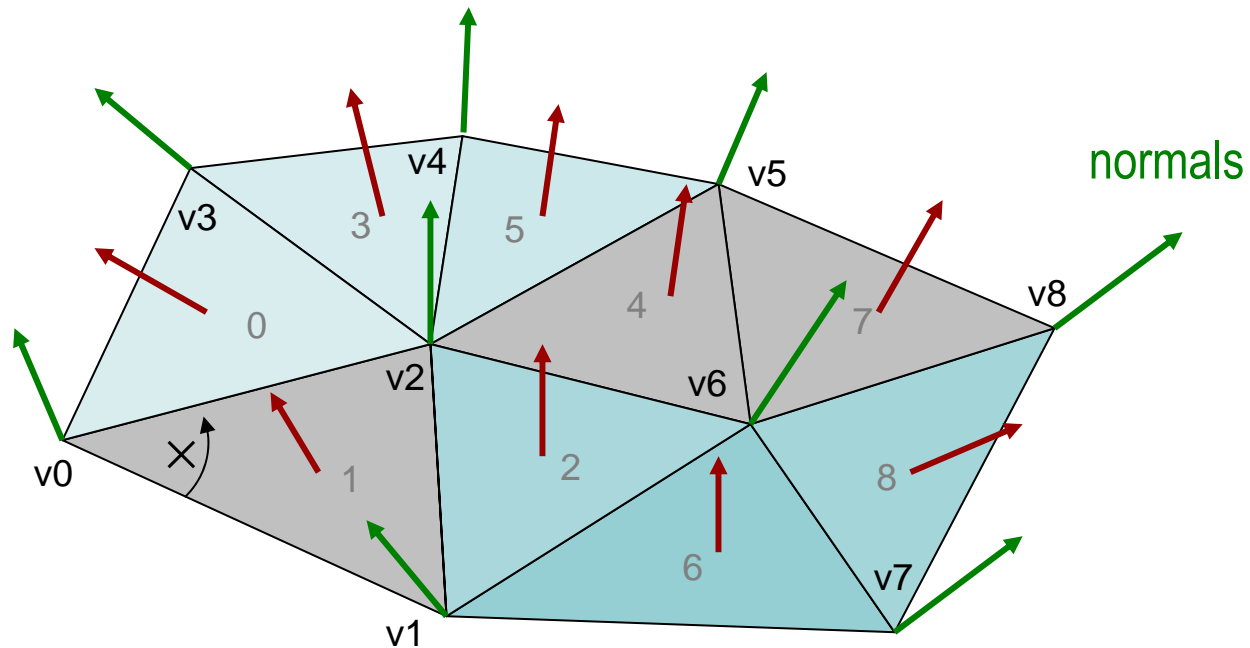
$$\vec{a} \times \vec{b} = \begin{vmatrix} \hat{x} & \hat{y} & \hat{z} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix}$$

$$\hat{n} = \frac{\vec{a} \times \vec{b}}{\|\vec{a} \times \vec{b}\|}$$

```
glNormal3f( nx, ny, nz );
glBegin( GL_TRIANGLES );
    glVertex3fv( v1 );
    glVertex3fv( v2 );
    glVertex3fv( v3 );
glEnd();
```

# Finding Normals

- Find triangle normals by simple cross product

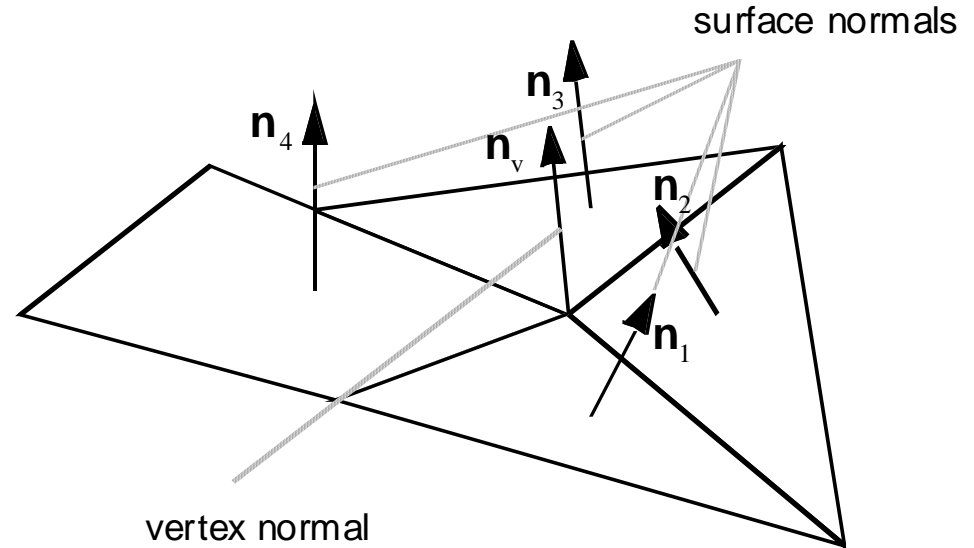


- Find vertex normals (shared normals) by averaging neighboring triangle normals

# Normals at Vertices

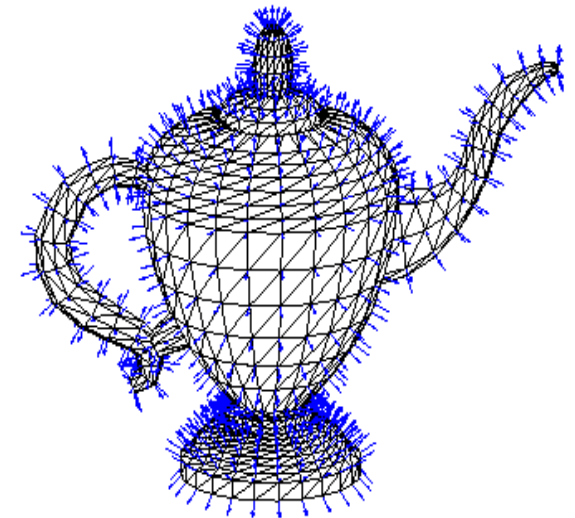
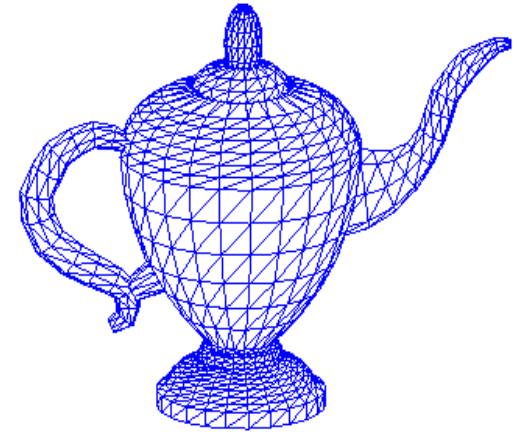
- Compute normals of triangles using cross product formula

$$\mathbf{n}_v = \frac{\sum_{i=1}^N \mathbf{n}_i}{\left| \sum_{i=1}^N \mathbf{n}_i \right|}$$



# Normal Vector

- Normals define how a surface reflects light
  - `glNormal3f( x, y, z )`
    - Current normal is used to compute vertex's color
    - Use *unit* normals for proper lighting
      - scaling affects a normal's length
      - `glEnable( GL_NORMALIZE )`
- or
- `glEnable( GL_RESCALE_NORMAL )`

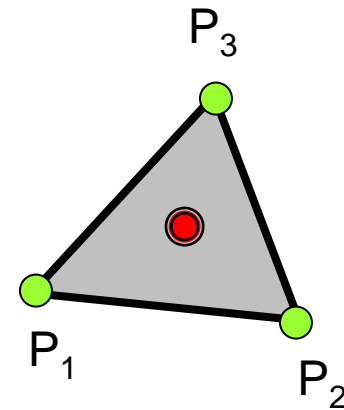
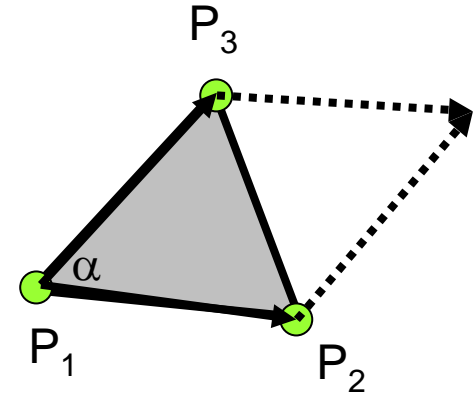


# Triangle Area and Centroid

$$\text{Area} = \frac{\| (P_3 - P_1) \times (P_2 - P_1) \|}{2}$$

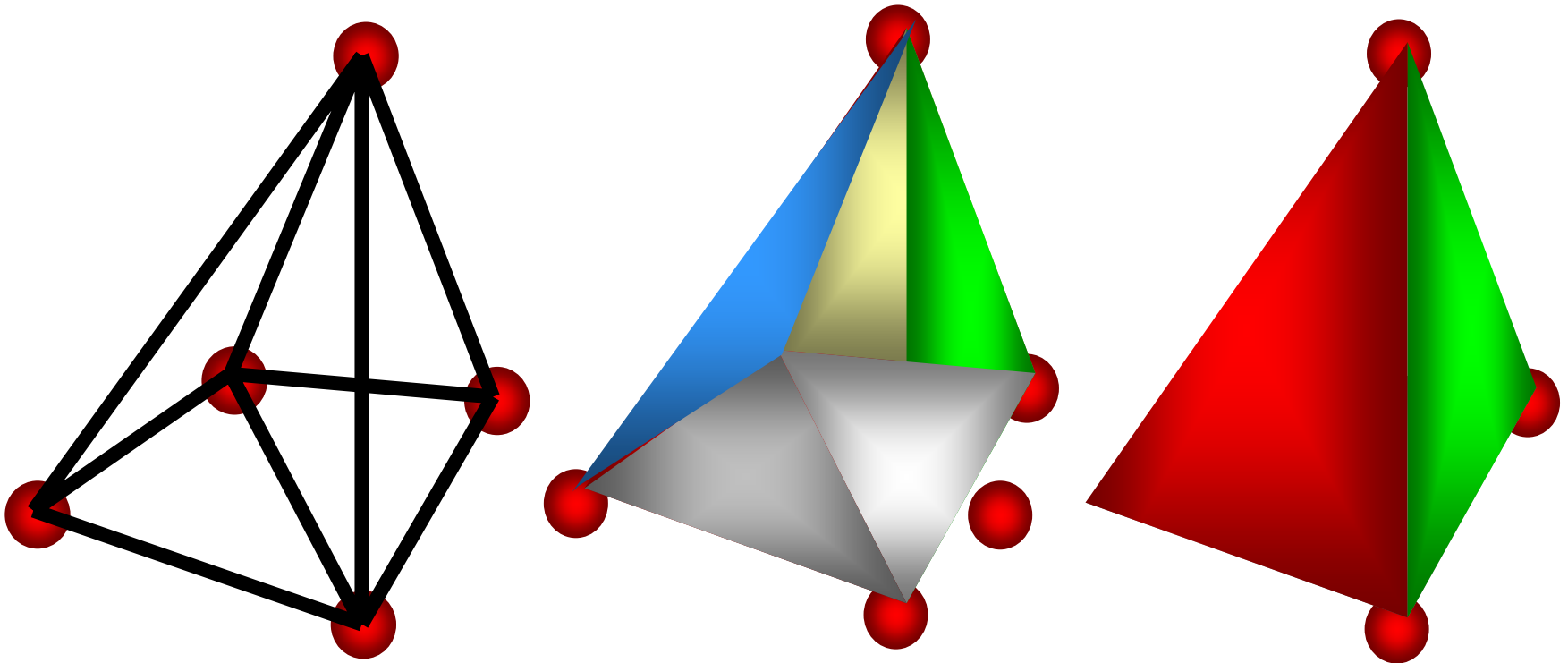
- The determinant is twice the area of the triangle whose vertices are the rows of the matrix.
- The centroid is the average of the vertices:

$$\text{Centroid} = \frac{P_1 + P_2 + P_3}{3}$$



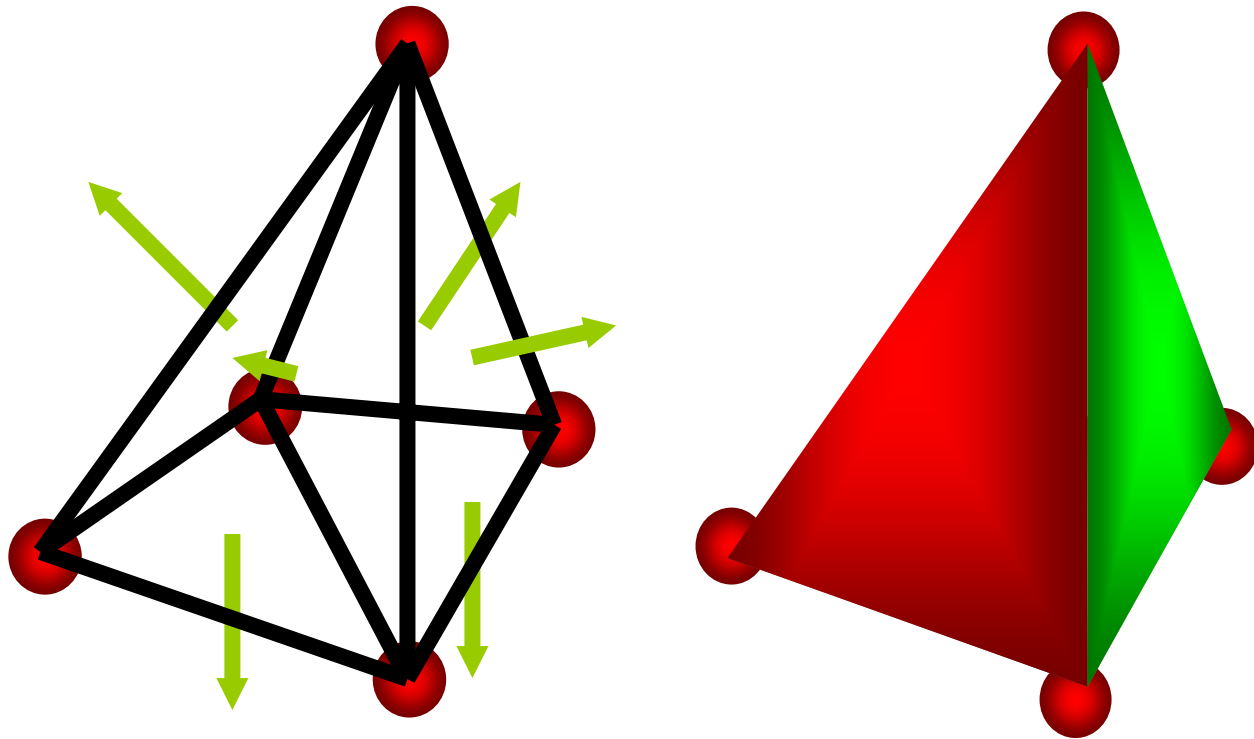
# Goal of Visible Surface Determination

To draw only the surfaces (triangles) that are visible, given a view point and a view direction

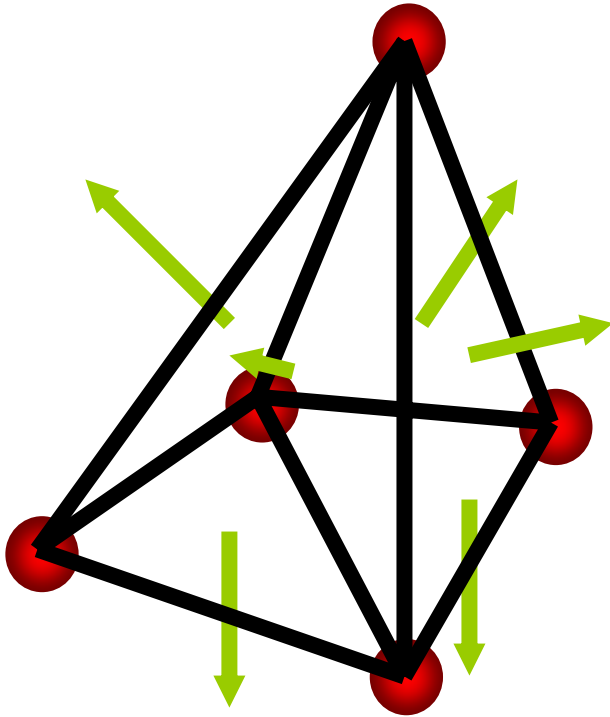


# What do the normals tell us?

Q: How can we use normals to tell us which “face” of a triangle we see?



# Viewing Coordinates

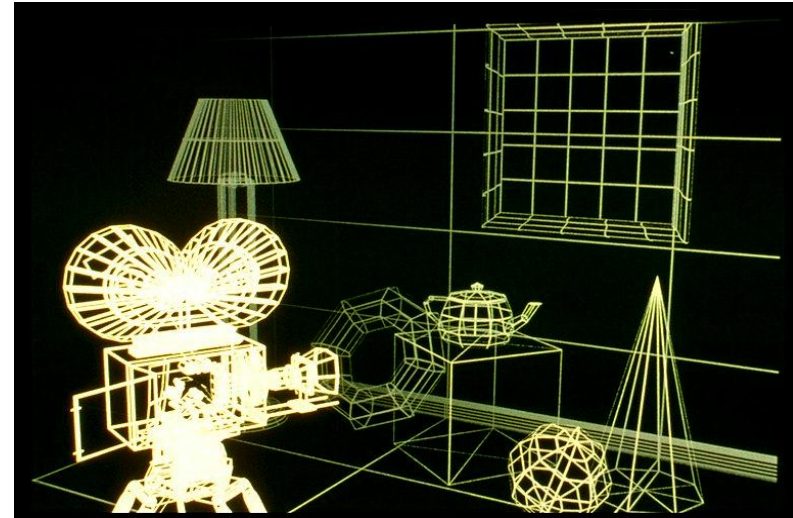


If we are in viewing coordinates, how can we simplify our comparison?

Think about the different components of the normals you want and don't want.

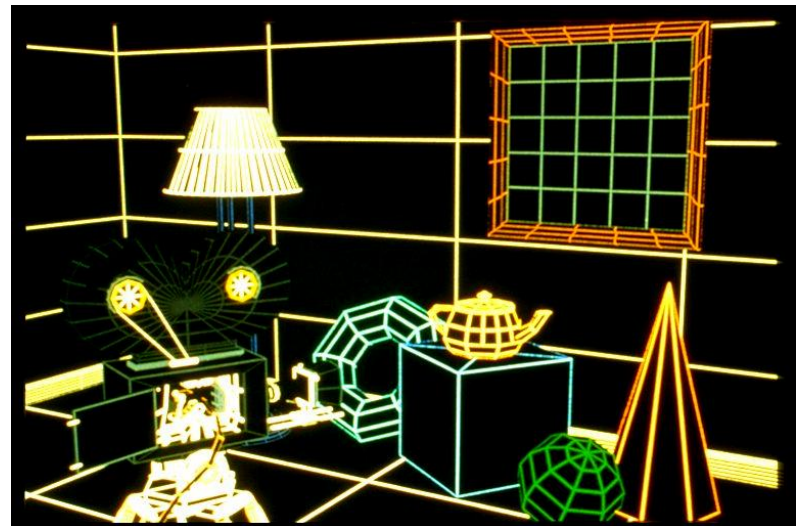
# Visibility of primitives

- We don't want to waste time rendering primitives which don't contribute to the final image.
- A scene primitive can be invisible for 3 reasons:
  - Primitive lies outside field of view
  - Primitive is *back-facing* (under certain conditions)
  - Primitive is occluded by one or more objects nearer the viewer
- How do we remove these efficiently?
- How do we identify these efficiently?



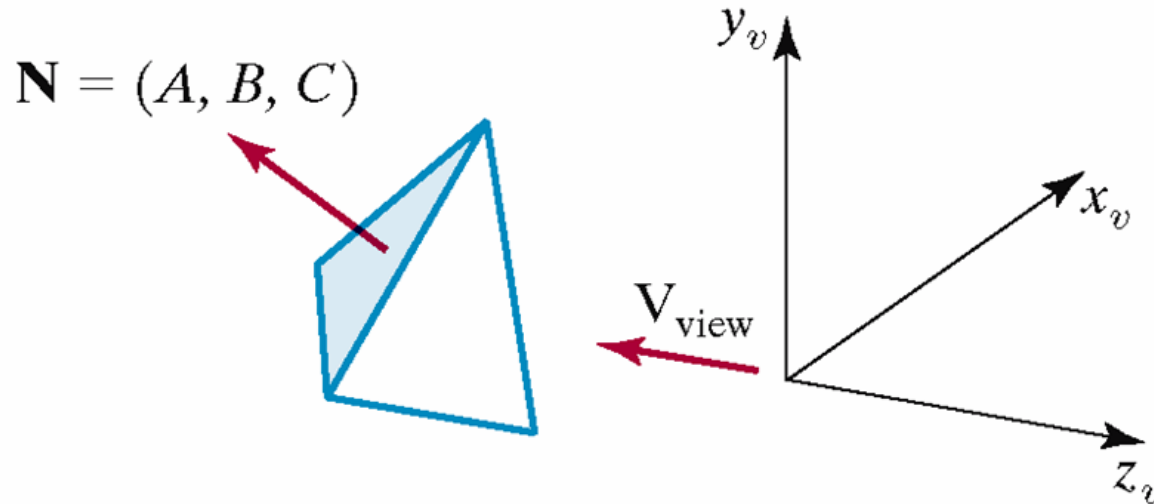
## The visibility problem.

- Removal of faces facing away from the viewer.
- Removal of faces obscured by closer objects.



# Back Face Culling

- **Definition:** a *back face* is any surface of an object which faces away from the eye.
- A back face can't be seen, and so does not need to be processed if we can identify it.
- Elimination of back faces is called back face culling.
- In a closed polygonal surface, back faces can be determined by the fact that their outward normals point away from the viewer.

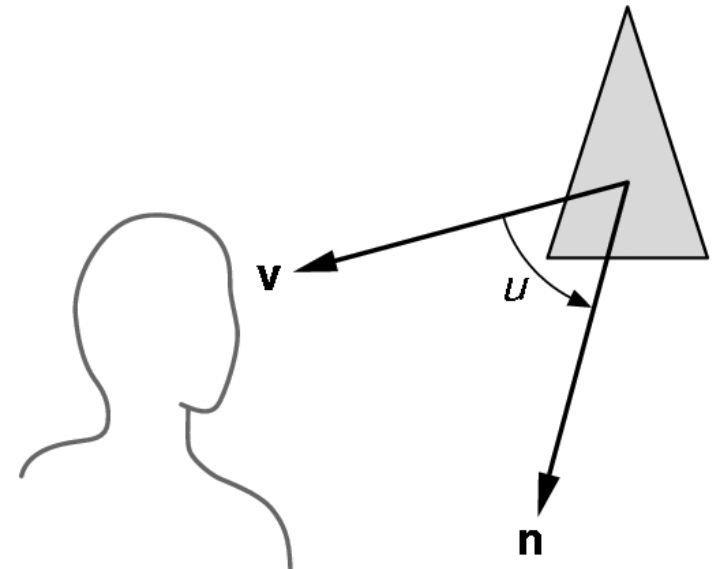
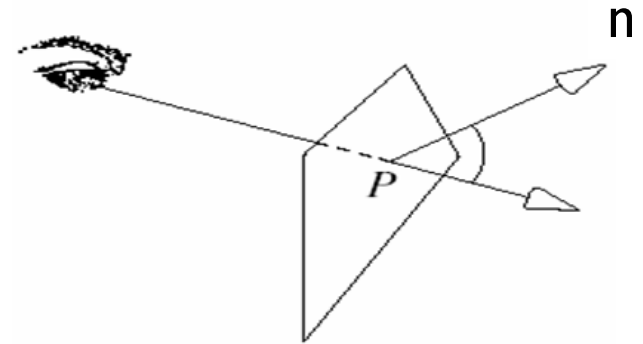


A polygon is back face if:

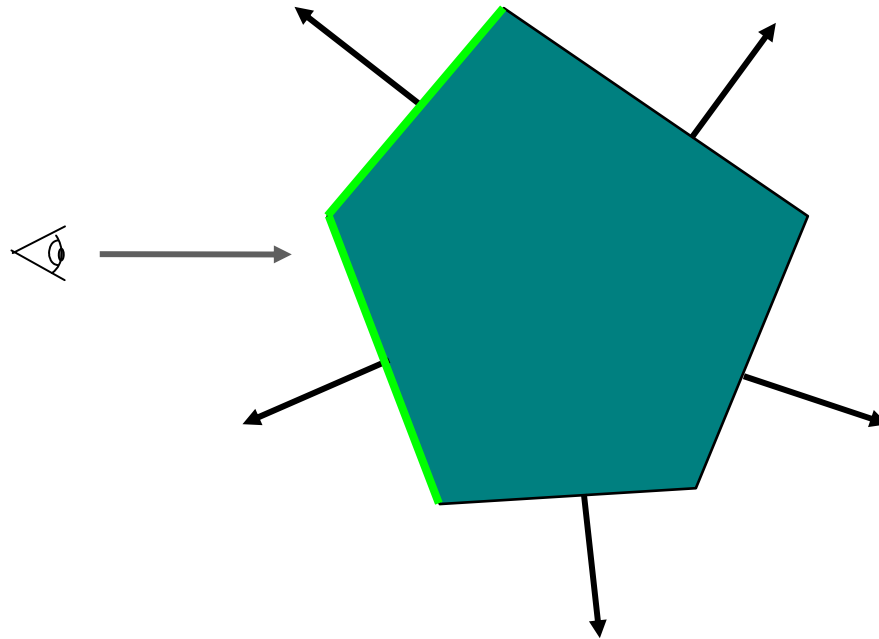
$$V_{view} \cdot N > 0$$

# Backface culling: Test

- Polygon with normal  $\mathbf{n}$  is back-facing relative to view direction vector  $\mathbf{v}$  (the Z-axis) if  $\mathbf{n} \cdot \mathbf{v} > 0$  (because that means angle is less than 90 degrees)
  - Can also just use point on polygon  $\mathbf{p}$  to define  $\mathbf{v} = \mathbf{p} - \mathbf{eye}$  in world coordinates
- To suppress the display of back surfaces, call `glCullFace( GL_BACK );`  
`glEnable( GL_CULL_FACE );` // hide all CCW faces
- Front face culling is also possible  
`glCullFace( GL_FRONT );`  
`glEnable( GL_CULL_FACE );` // hide all CW faces



# Back Face Culling in Action

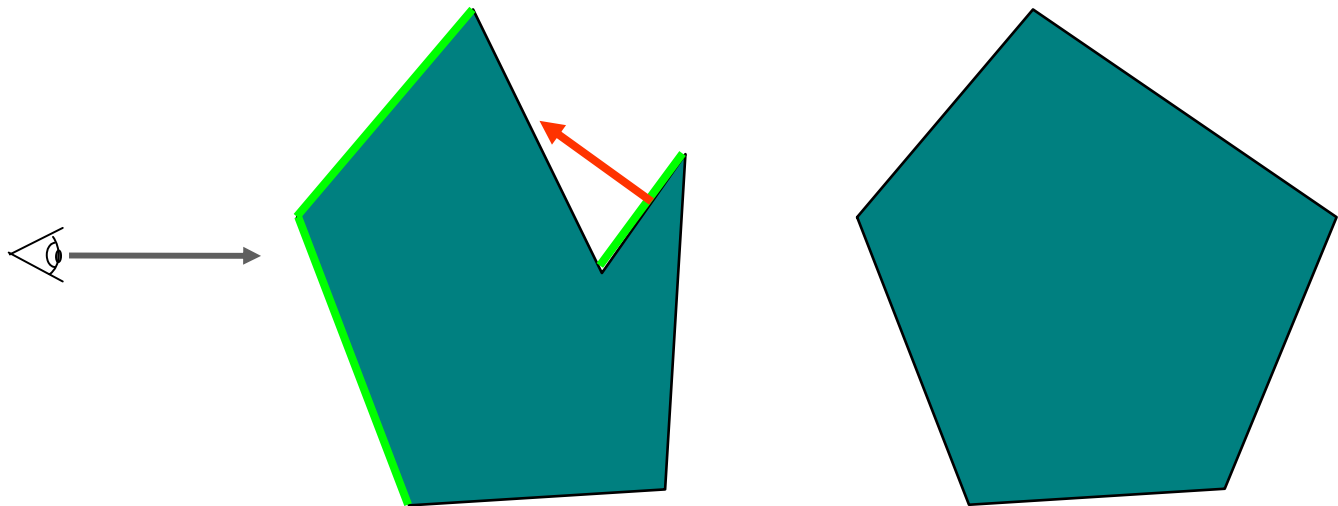


## Algorithm:

1. Find the angle between the eye-vector and the surface normal.
2. If it's in the range from  $0$  to  $90^\circ$  inclusive, discard the face.

# Back Face Culling is Only Part of the Story

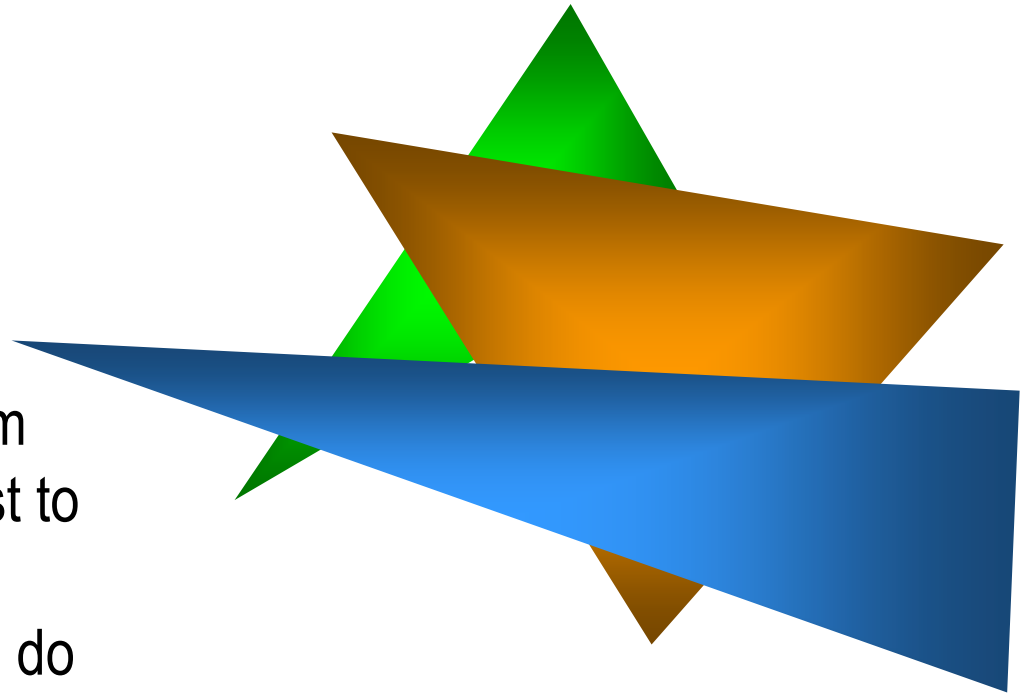
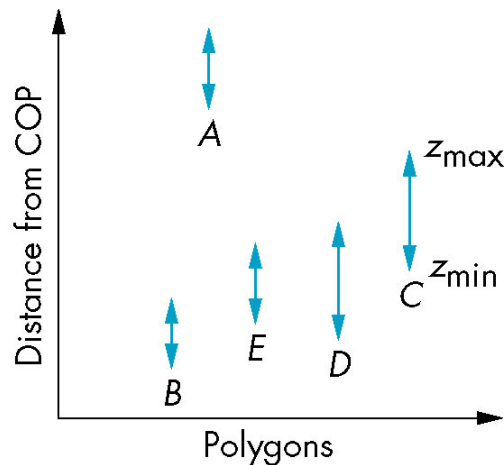
Back face culling alone does not solve all visibility problems.



In general, back face removal can be expected to eliminate approximately half the polygon surfaces in a scene.

# Multiple Objects

- If we want to draw:
- **Idea:** Sort primitives by minimum depth, then rasterize from furthest to nearest
- When there are depth overlaps, do more tests of bounding areas, etc. to see one actually occludes the other

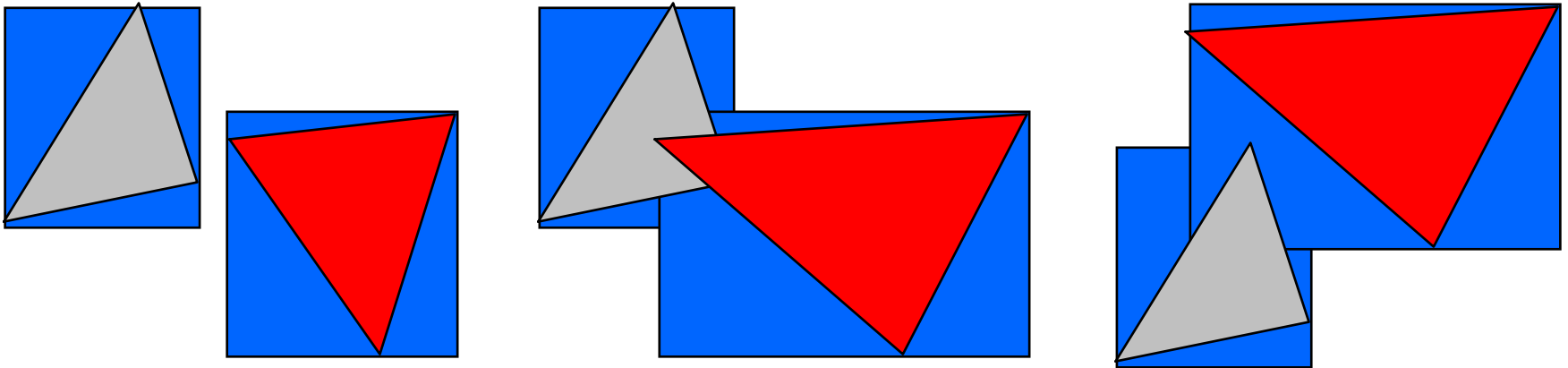


We can sort in  $z$ . What are the advantages?  
Disadvantages?

Called Painter's Algorithm or *splatting*.

# Extents and Bounding Volumes

- To test if two parts are overlapping
- To see if a ray intersects an object



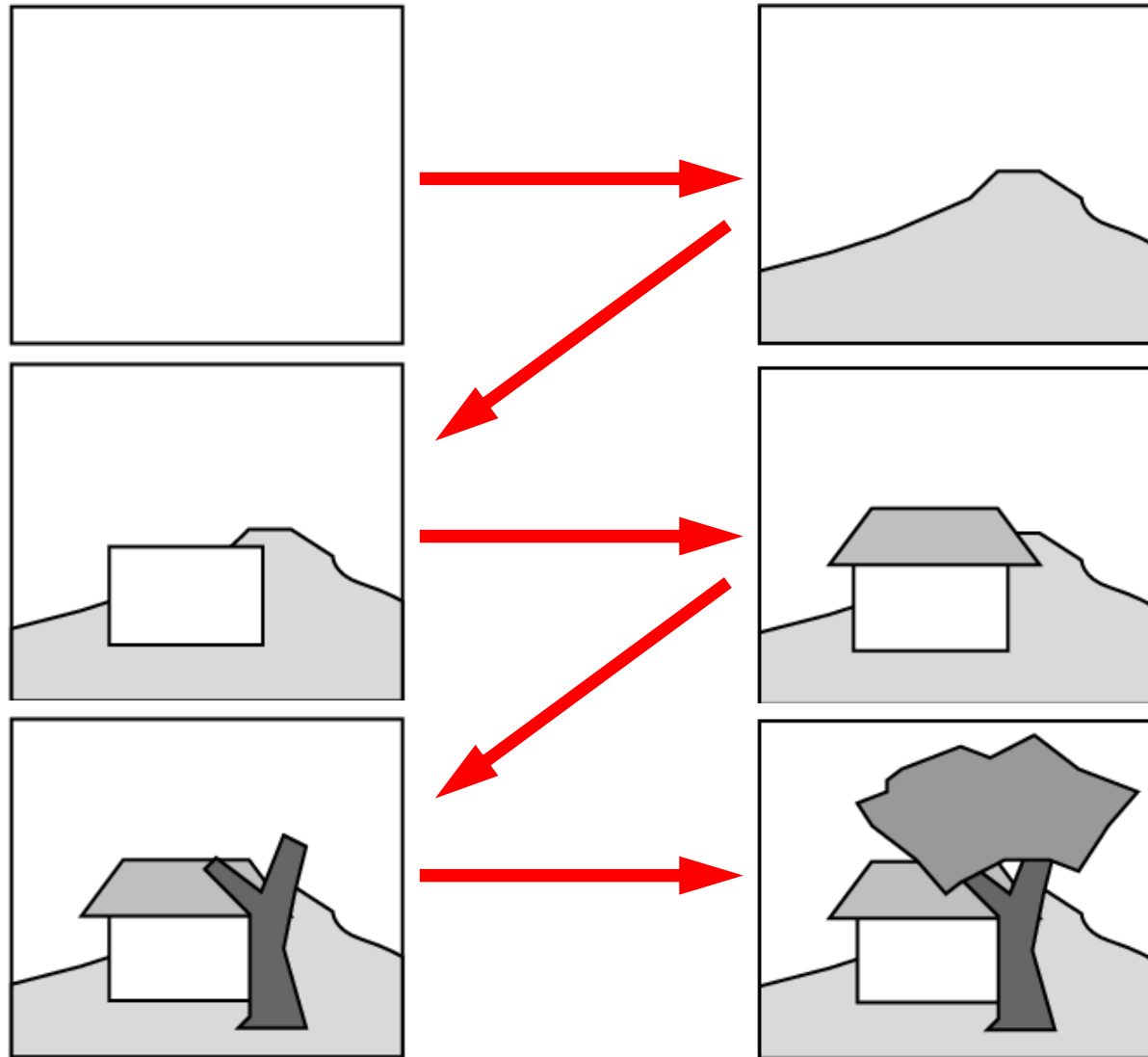
# Painter's Algorithm Subtleties

- What do we mean sort in  $z$ ? That is for a triangle, what is its representative  $z$  value?
  - Minimum  $z$
  - Maximum  $z$
  - Polygon's centroid
- Work cost = sort + draw



# Painter's algorithm

Draw primitives  
from back to  
front to avoid  
need for depth  
comparisons



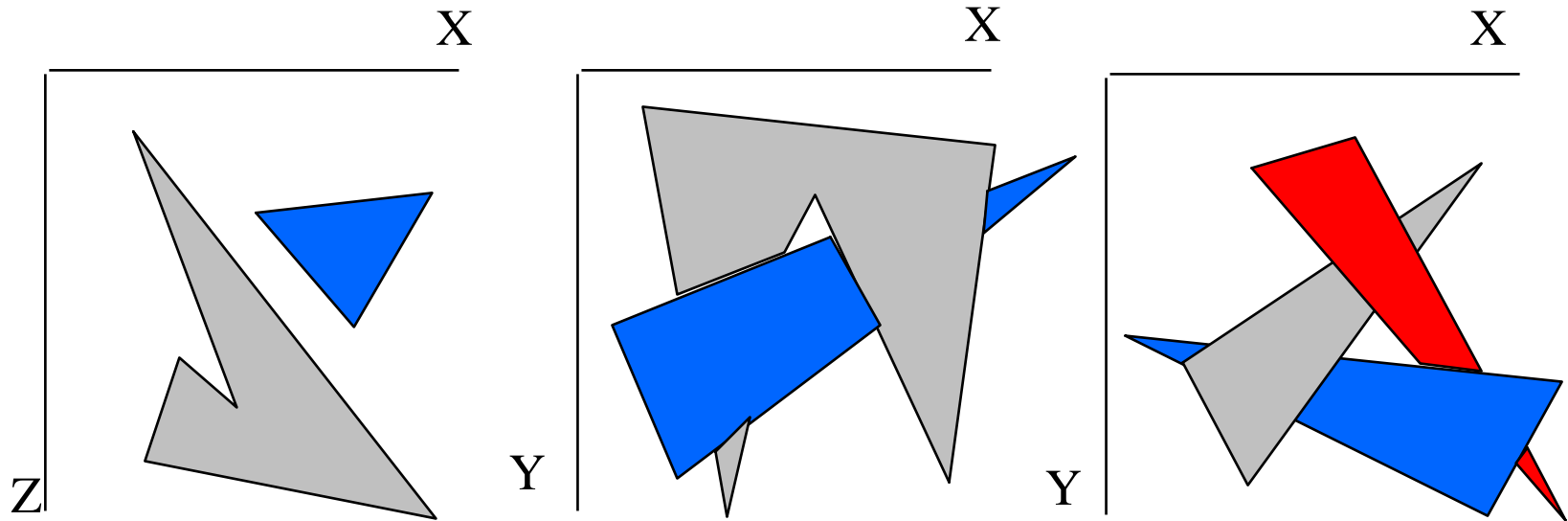
# Painter's Algorithm

- Sort all polygons according to the (farthest) z coordinates of each
  - When the polygon's z extents overlap, resolve the sorting ambiguities by splitting the polygons
  - Draw (fill) the polygons in the sorted order
    - —back to front
- 
- *Also known as Depth-sort algorithm*
  - *When can we skip step 2 above?*

# Painter's Algorithm...

- Does polygon P obscure polygon Q?
  - Do the polygons' x extents overlap?
  - Do the polygons' y extents overlap?
  - Is P entirely on the opposite side of Q's plane from the viewpoint?
  - Is Q entirely on the same side of P's plane from the viewpoint?
  - Do the projections of the polygons onto screen not overlap?

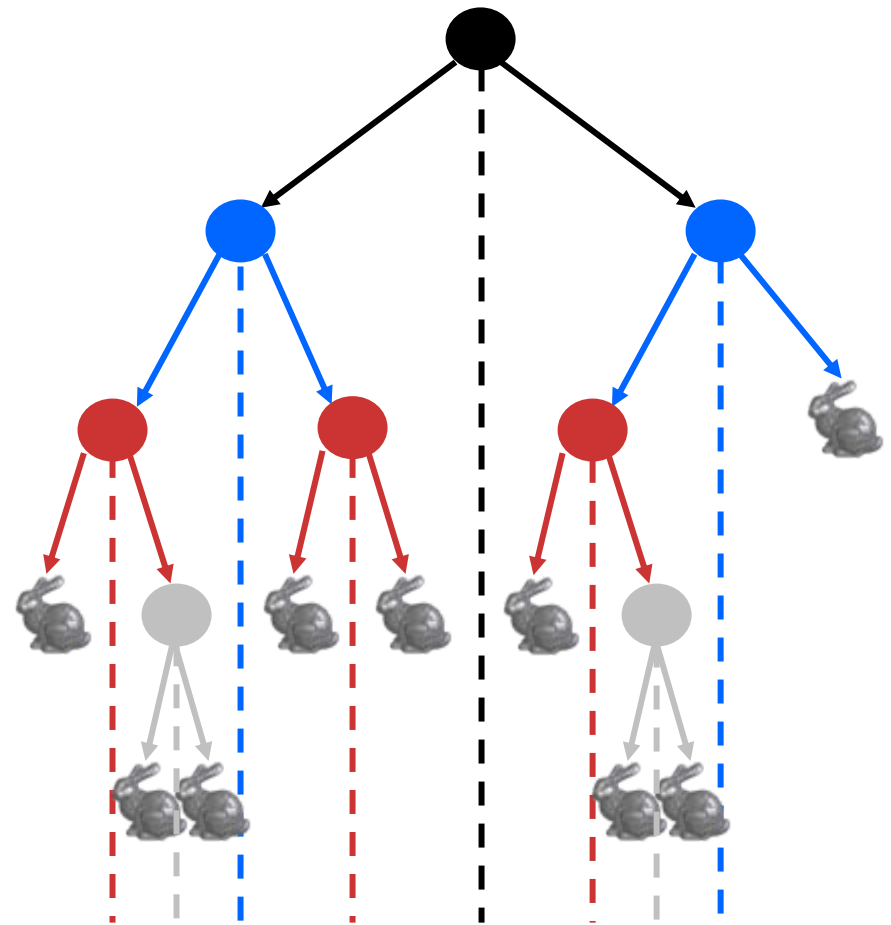
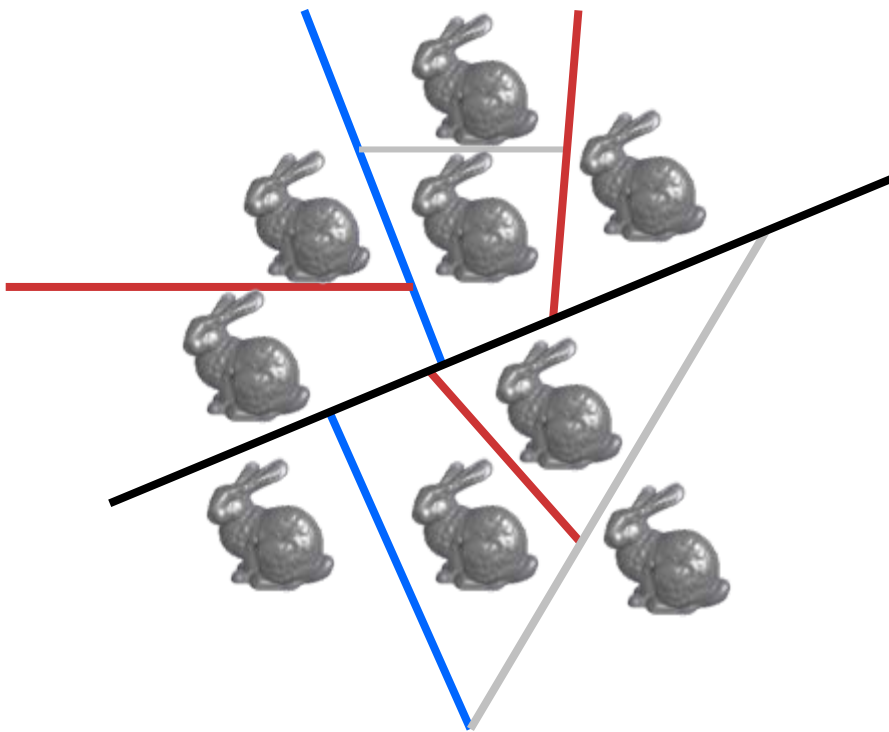
# Painter's Algorithm...



Some cases in which Z extents of polygons overlap

# BSP Trees

- preprocess: create binary tree
  - recursive spatial partition



# Binary Space Partitioning Trees

- Very efficient for a static group of 3D polygons as seen from an arbitrary viewpoint
- Correct order for Painter's algorithm is determined by a suitable traversal of the binary tree of polygons: BSP Tree

# BSP Tree

- Code works for any view
- Tree can be pre-computed
- Requires evaluation of

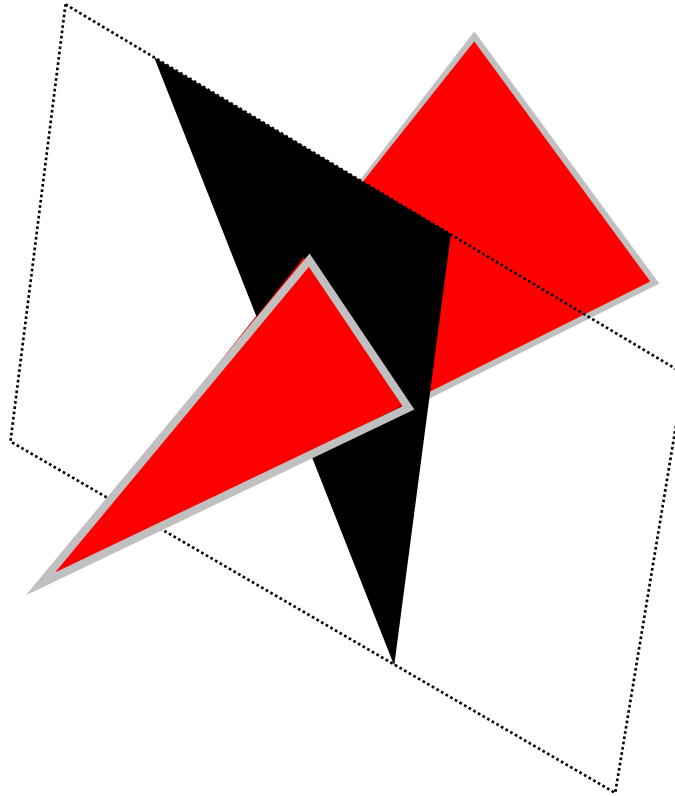
$$f_{\text{plane of the triangle}}(\text{eye})$$

# BSP Tree Construction

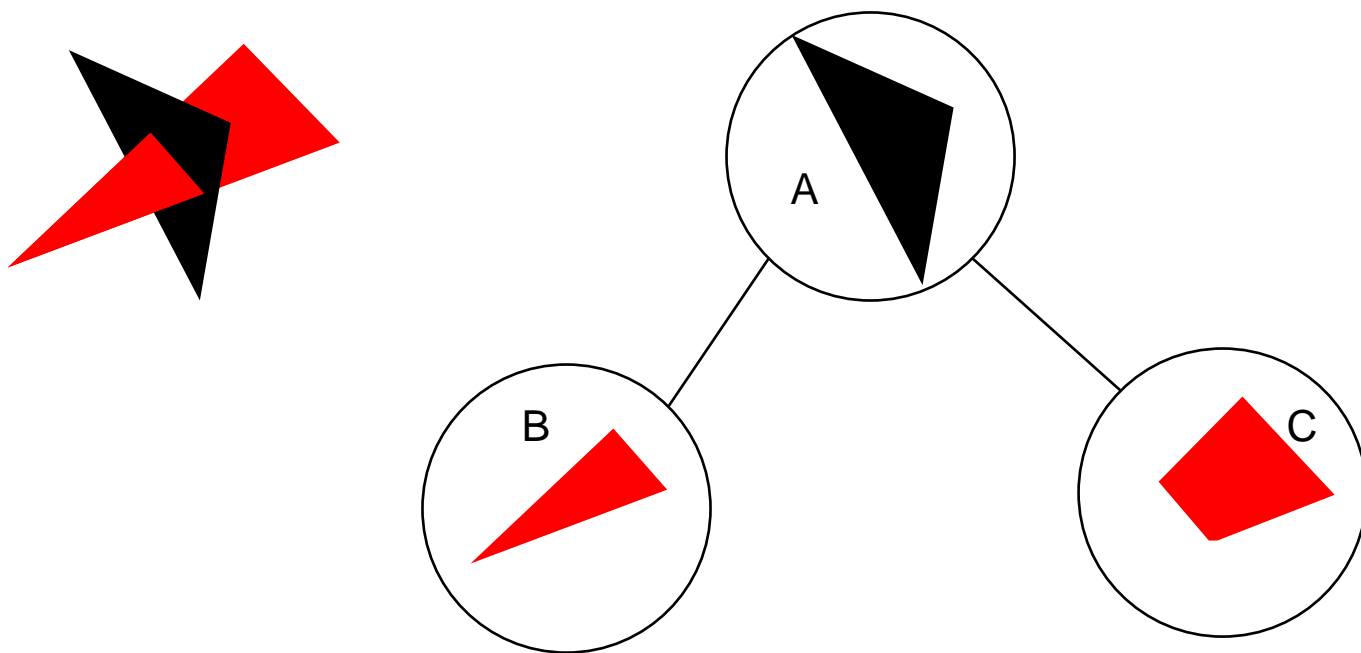
- The binary tree is constructed using the following principle:
  - For each polygon, we can divide the set of other polygons into two groups
  - One group contains those lying in front of the plane of the given polygon
  - The other group contains those in the back
  - The polygons intersecting the plane of the given polygon are split by that plane

# BSP Tree

- Split Triangle:
  - How to?



# BSP Tree



# BSP Tree

## Draw BSP Tree

```
function draw(bsptree tree, point eye)
```

```
if tree.empty then
```

```
    return
```

```
if  $f_{\text{tree.root}}(\text{eye}) < 0$ 
```

```
    draw (tree.right)
```

```
    rasterize(tree.root)
```

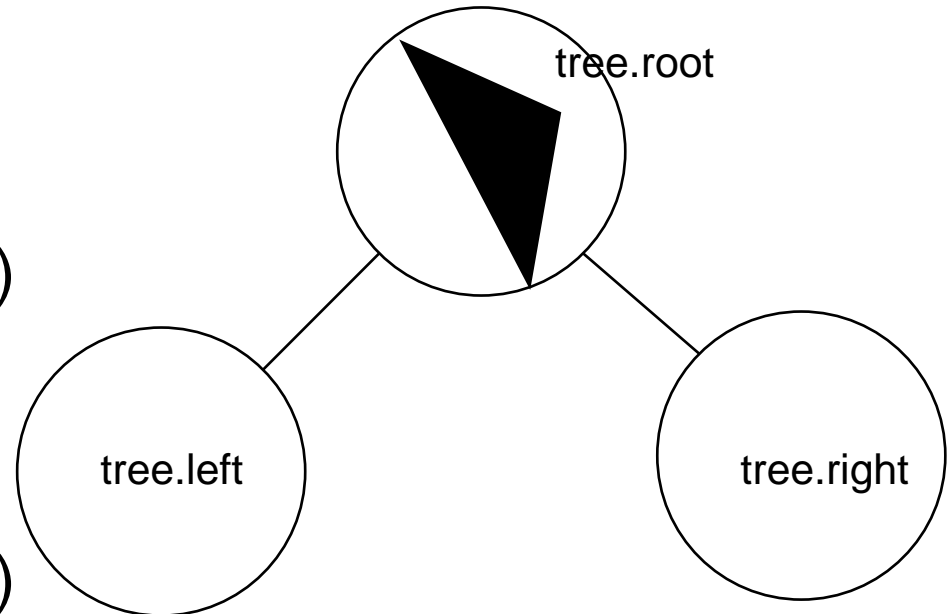
```
    draw(tree.left)
```

```
else
```

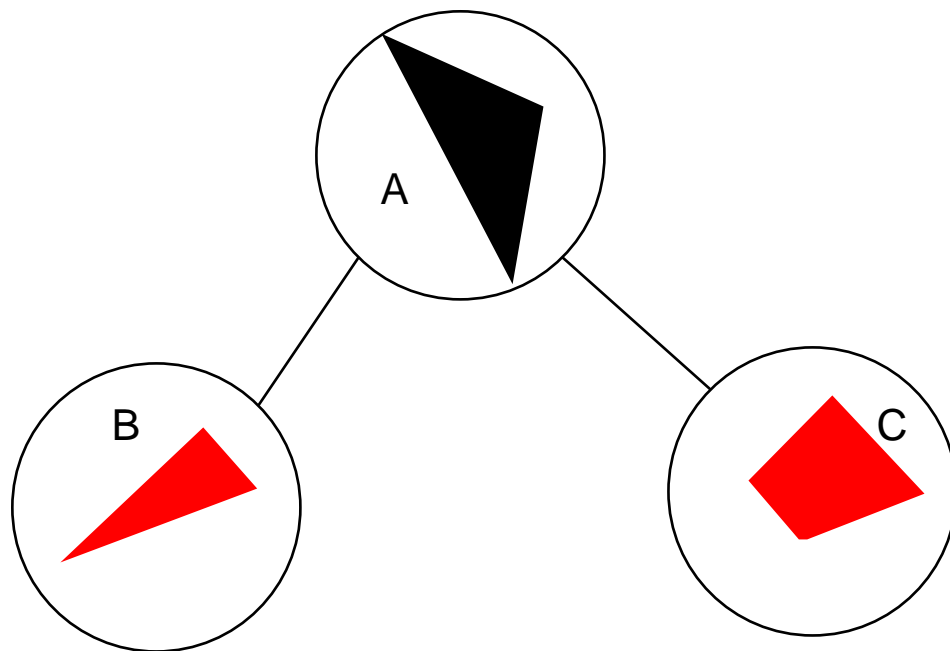
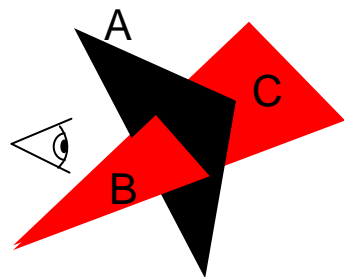
```
    draw (tree.left)
```

```
    rasterize(tree.root)
```

```
    draw(tree.right)
```

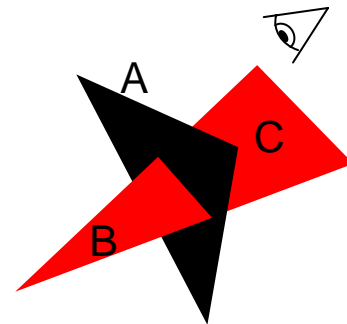


# BSP Tree



rasterize(C)  
rasterize(A)  
rasterize(B)

rasterize(B)  
rasterize(A)  
rasterize(C)



# BSP Tree

- Code works for any view
- Tree can be pre-computed
- Requires evaluation of

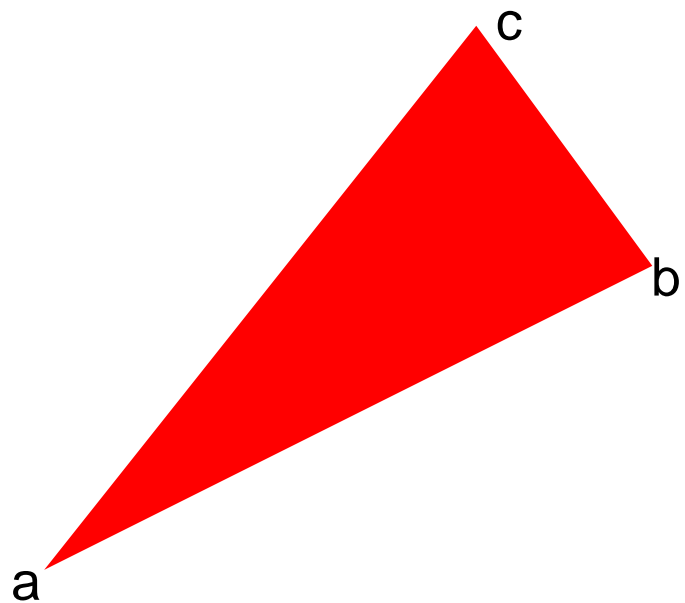
$$f_{\text{plane of the triangle}}(\text{eye})$$

# BSP Tree

- Equation of the Plane
  - $Ax + By + Cz + D = 0$
- How to compute A, B, C and D ?
  - $(A, B, C)$  is the normal to the plane.

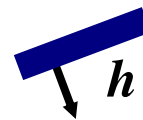
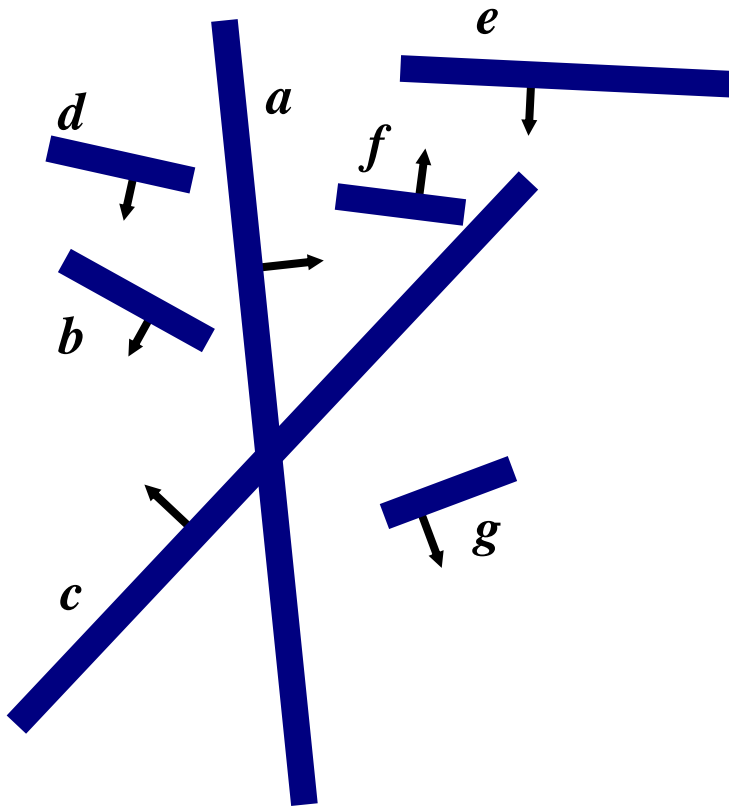
$$\text{normal} = (b-a) \times (c-a)$$

$$D = -(Ax_a + By_a + Cz_a) \\ = -\text{normal} \cdot a$$



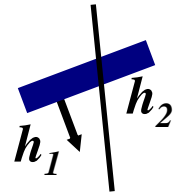
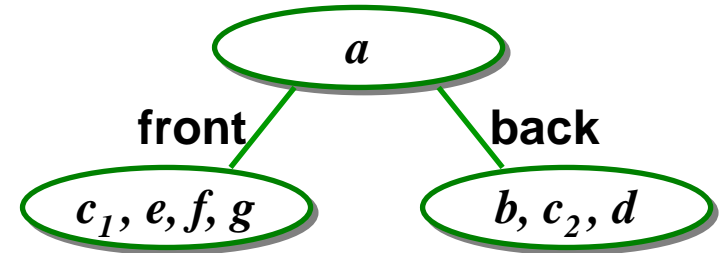
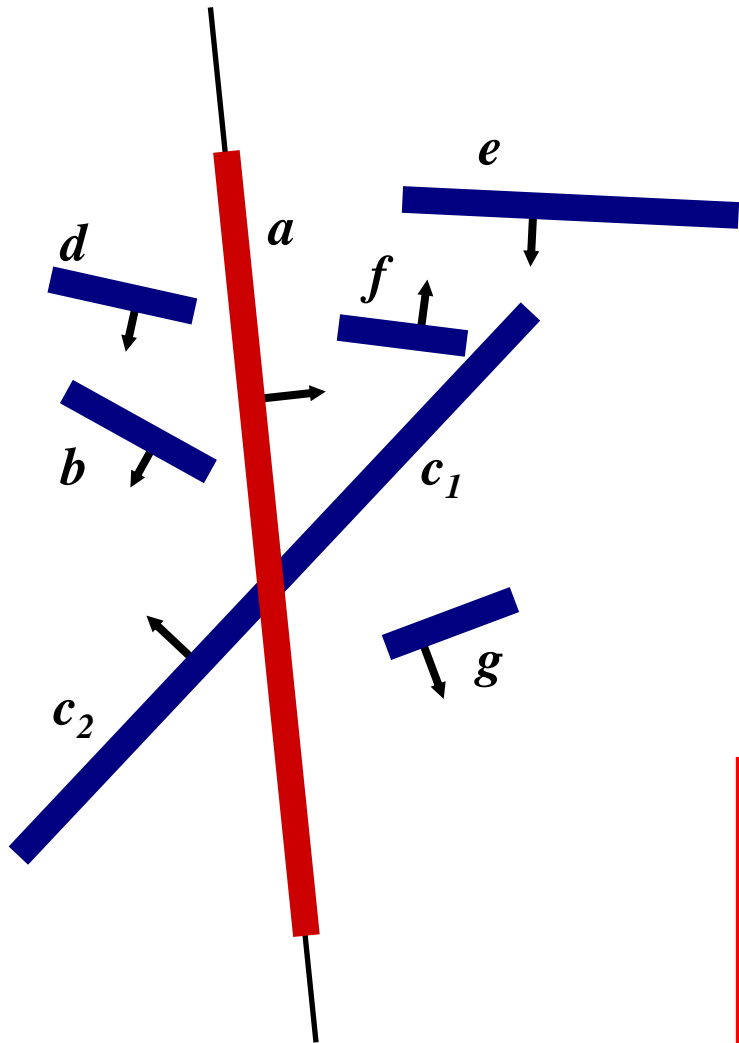
# BSP-Trees Example

*a,b,c,d,e,f,g*



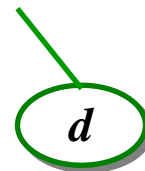
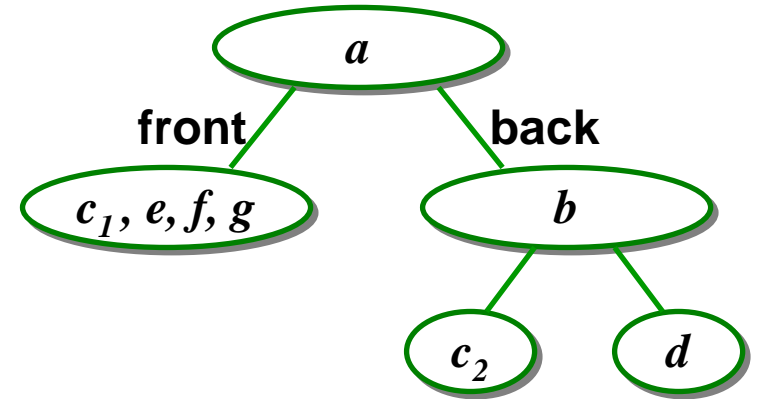
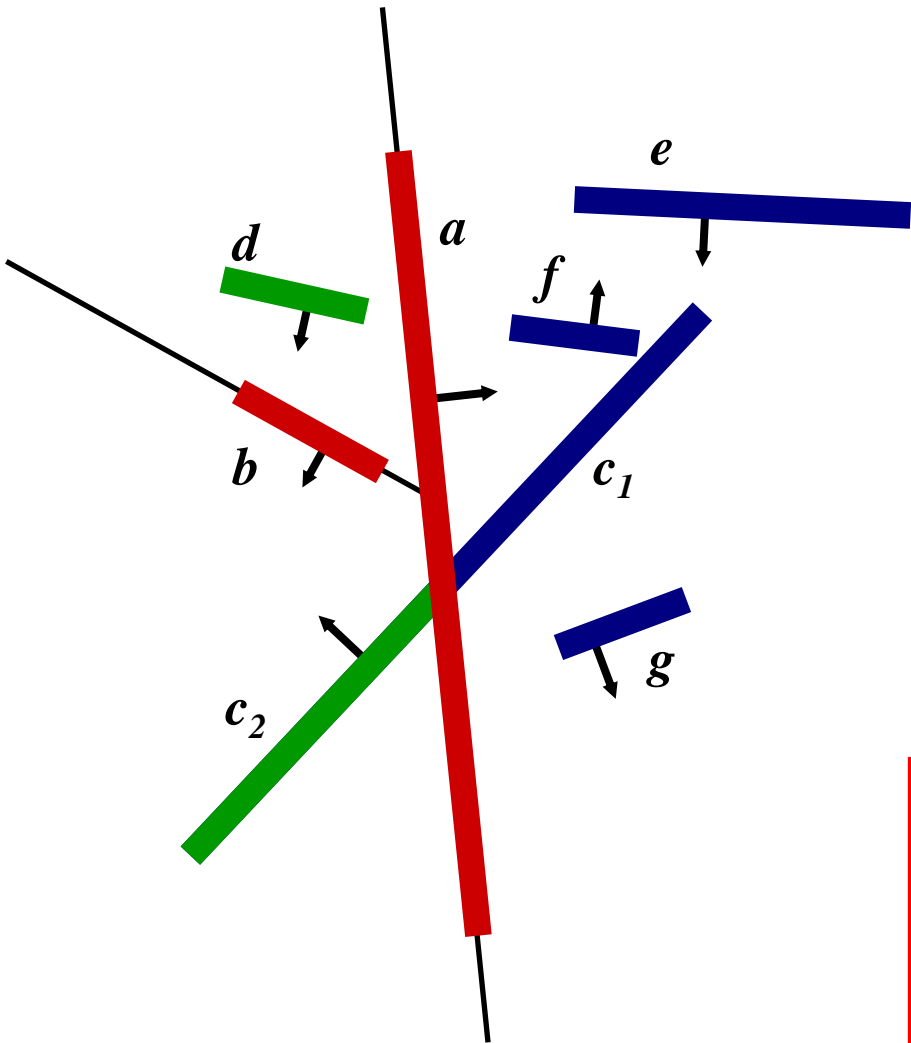
Each line represents the side view of a polygon with a normal indicated by the arrow.

# BSP-Trees Example



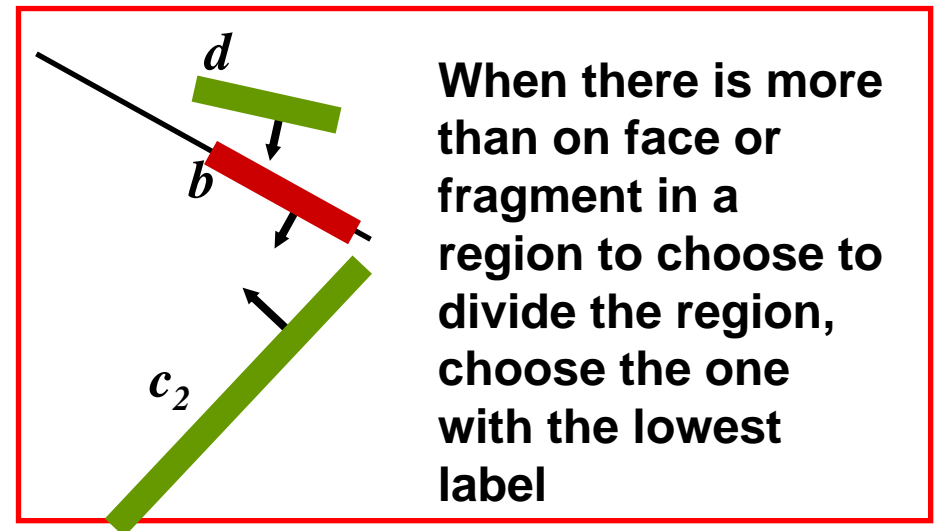
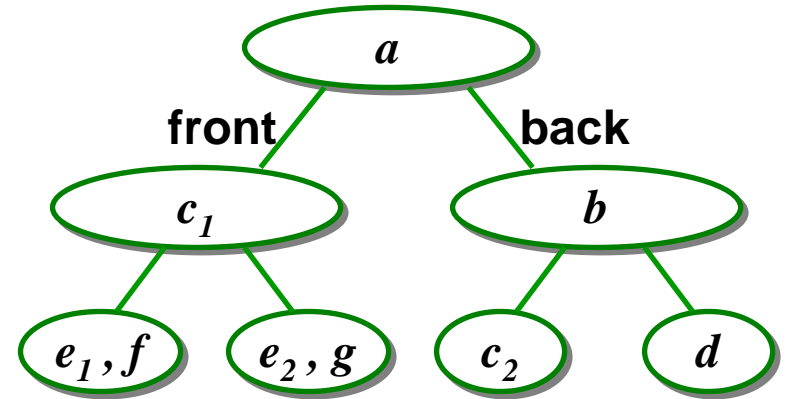
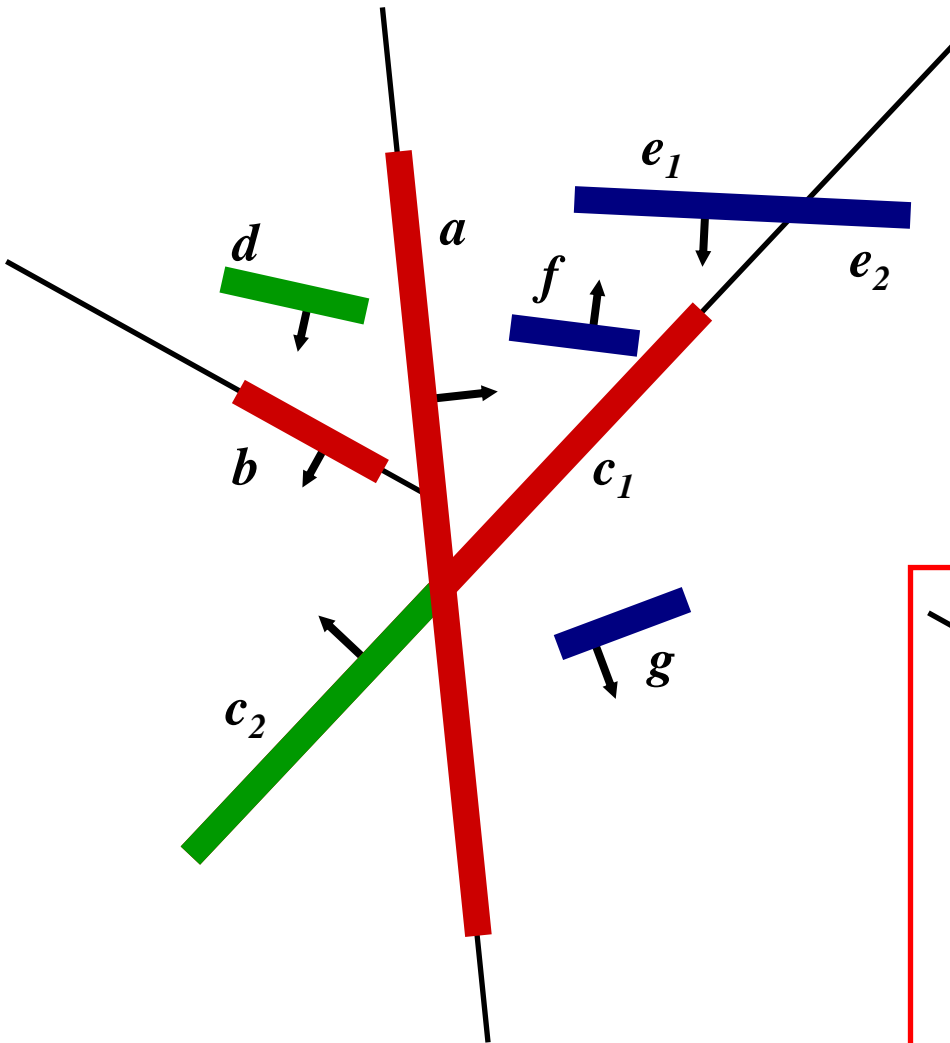
The line passes through a face and divides it into two fragments which represent subfaces

# BSP-Trees Example

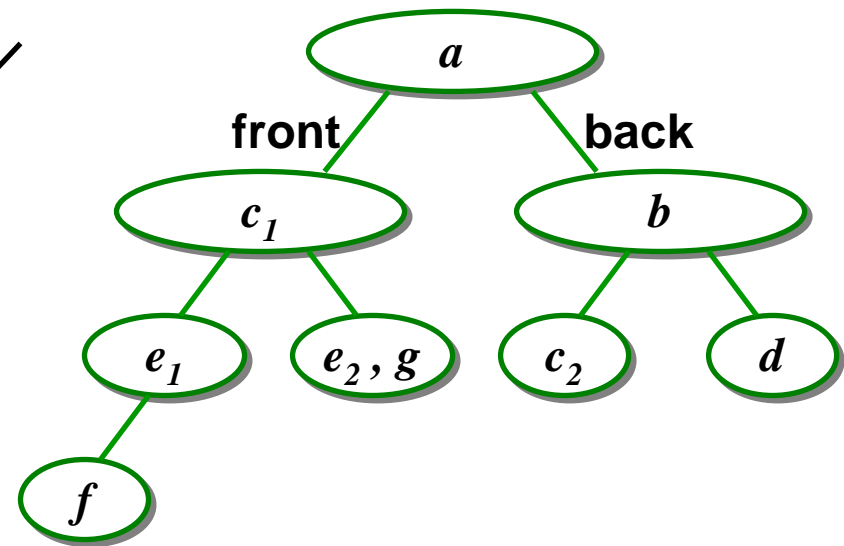
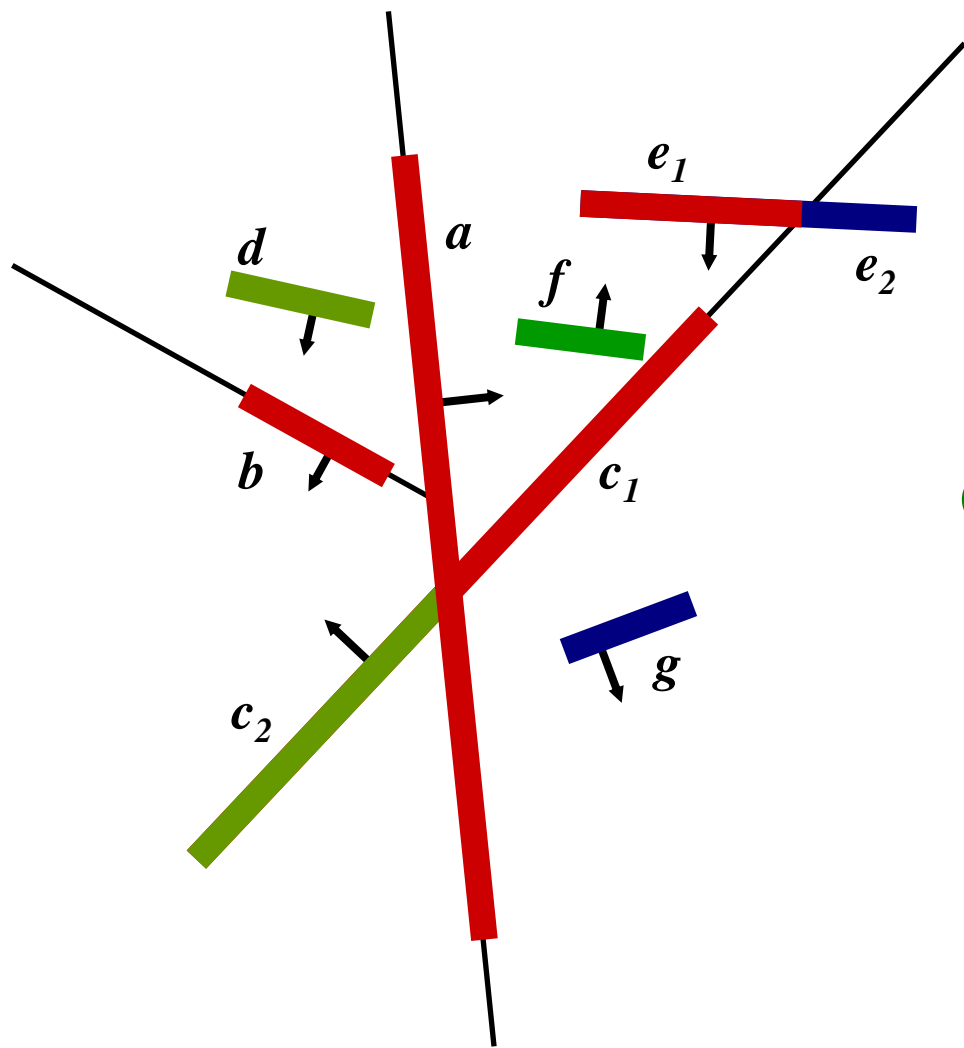


When there is a single face or fragment in a region, that face is stored in a leaf in the tree

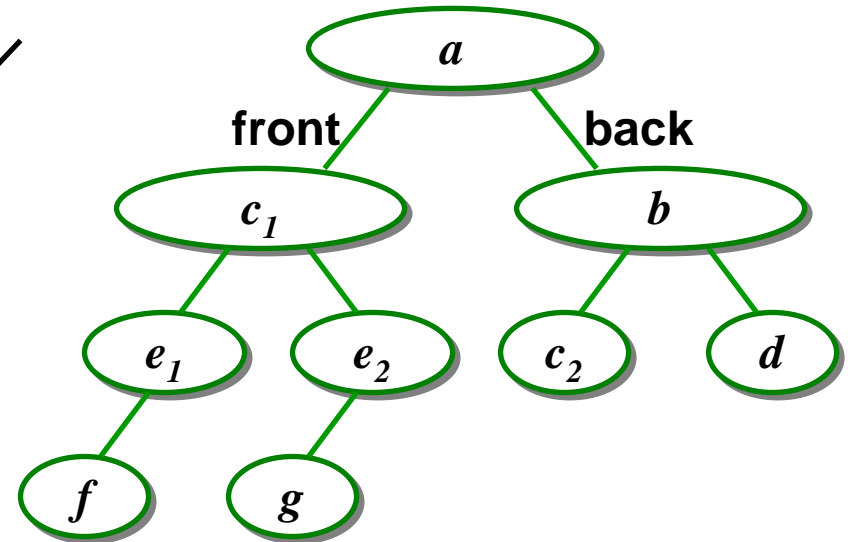
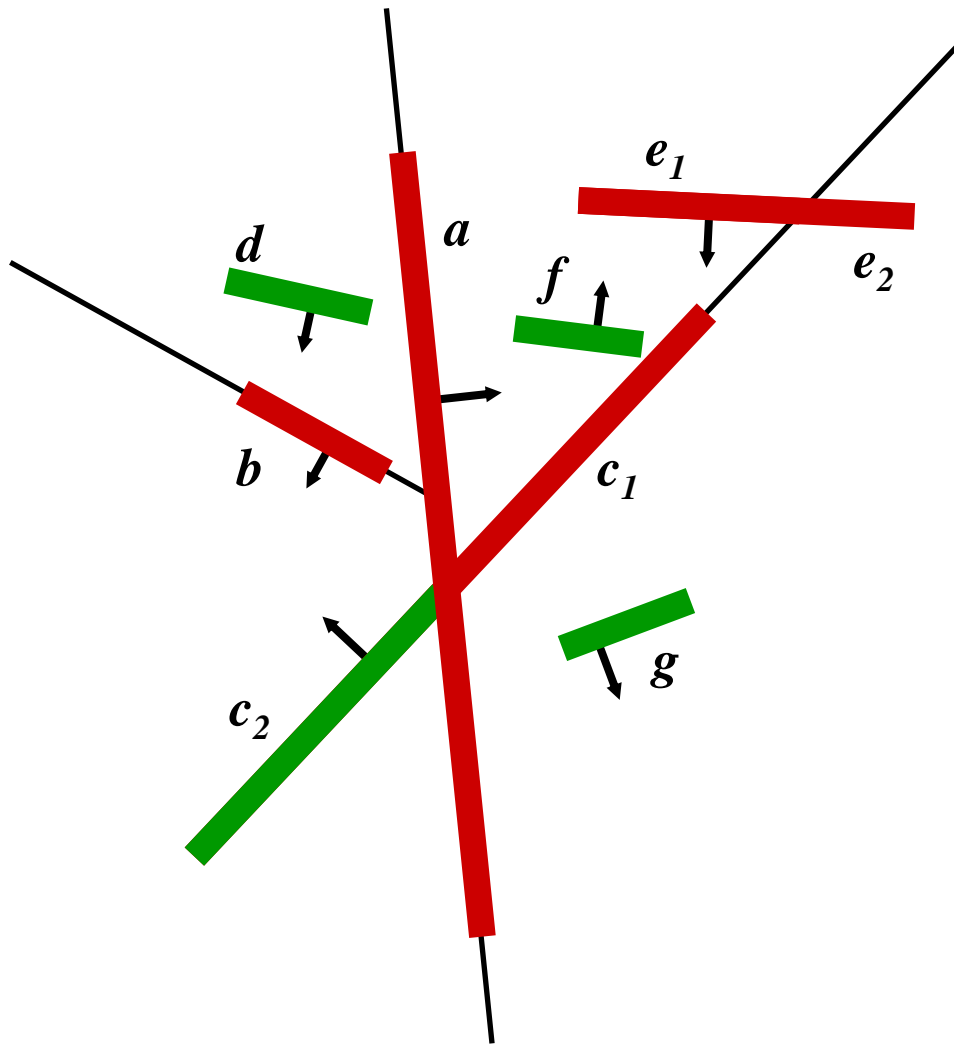
# BSP-Trees Example



# BSP-Trees Example



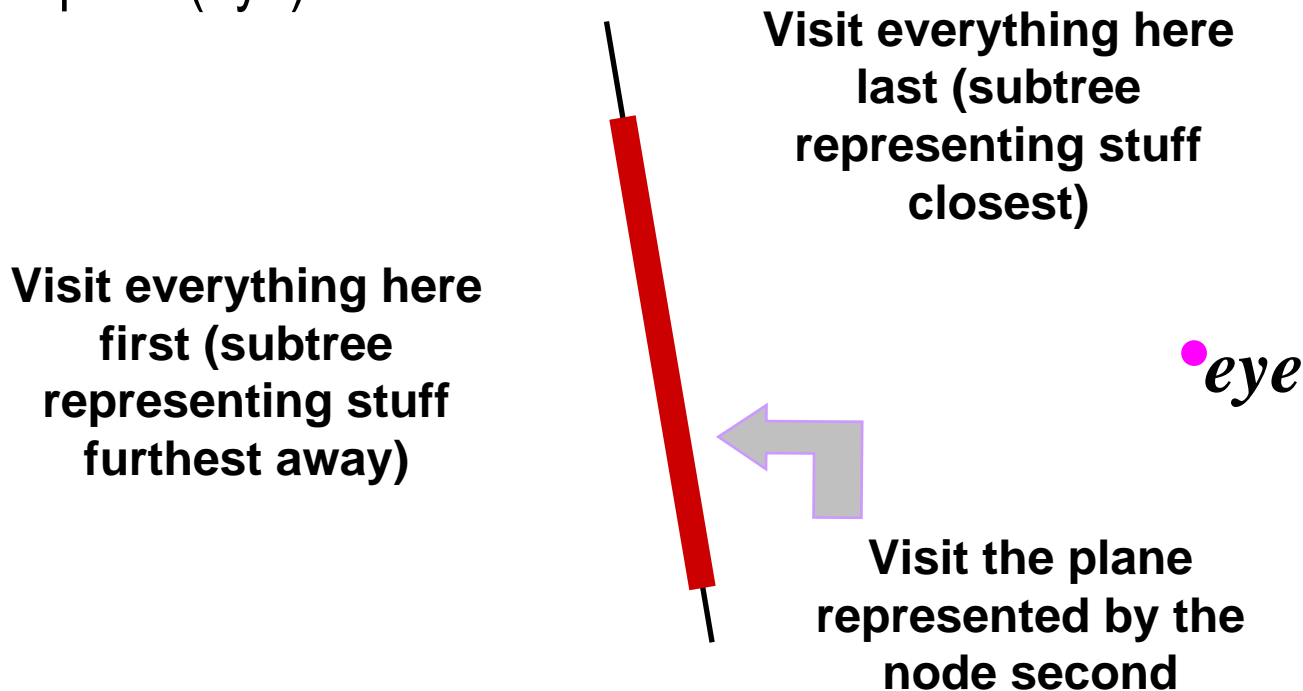
# BSP-Trees Example



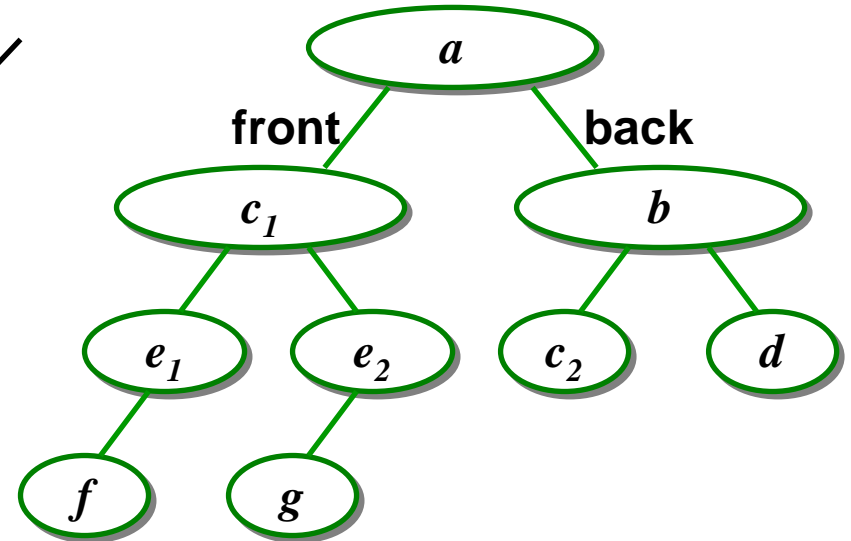
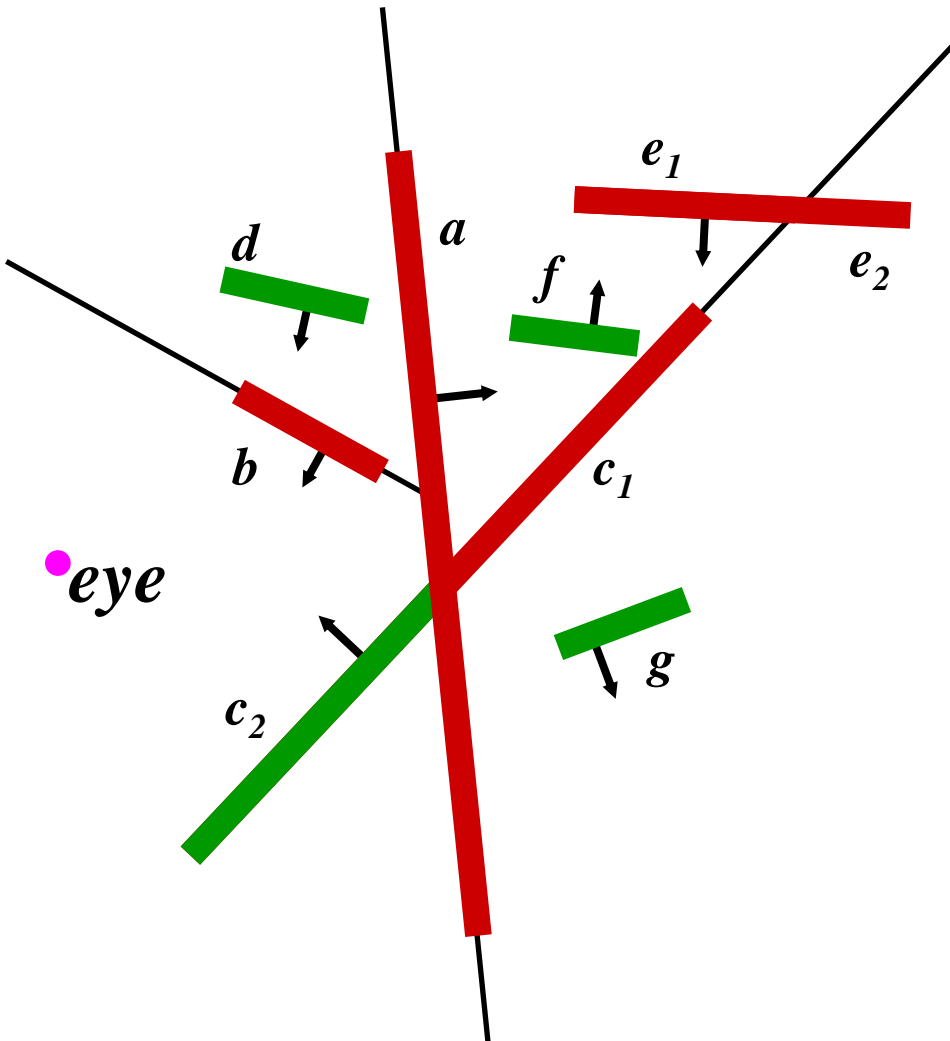
# Visibility Order Traversal

## Variation of in-order-traversal

1. Visit the child that represents the side of the plane farthest away from the viewpoint (eye).
2. Visit the sub-tree root
3. Visit the child that represents the side of the plane closest to the viewpoint (eye).



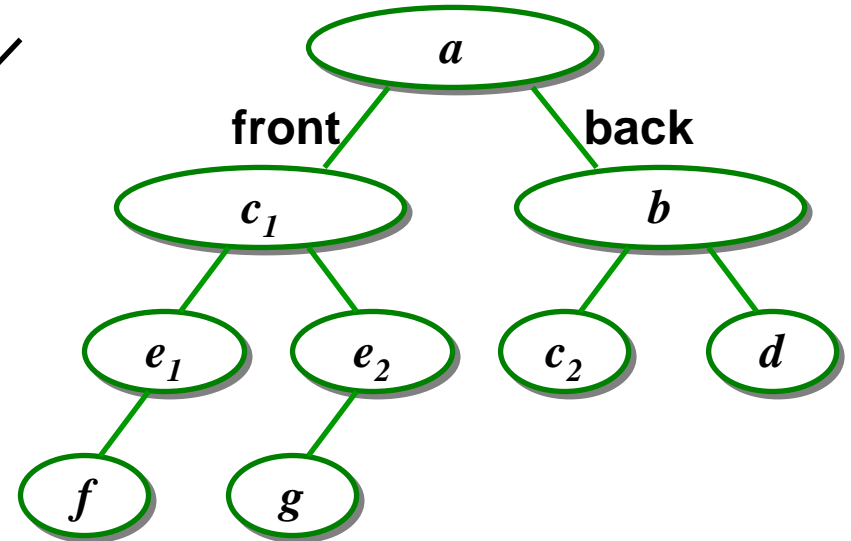
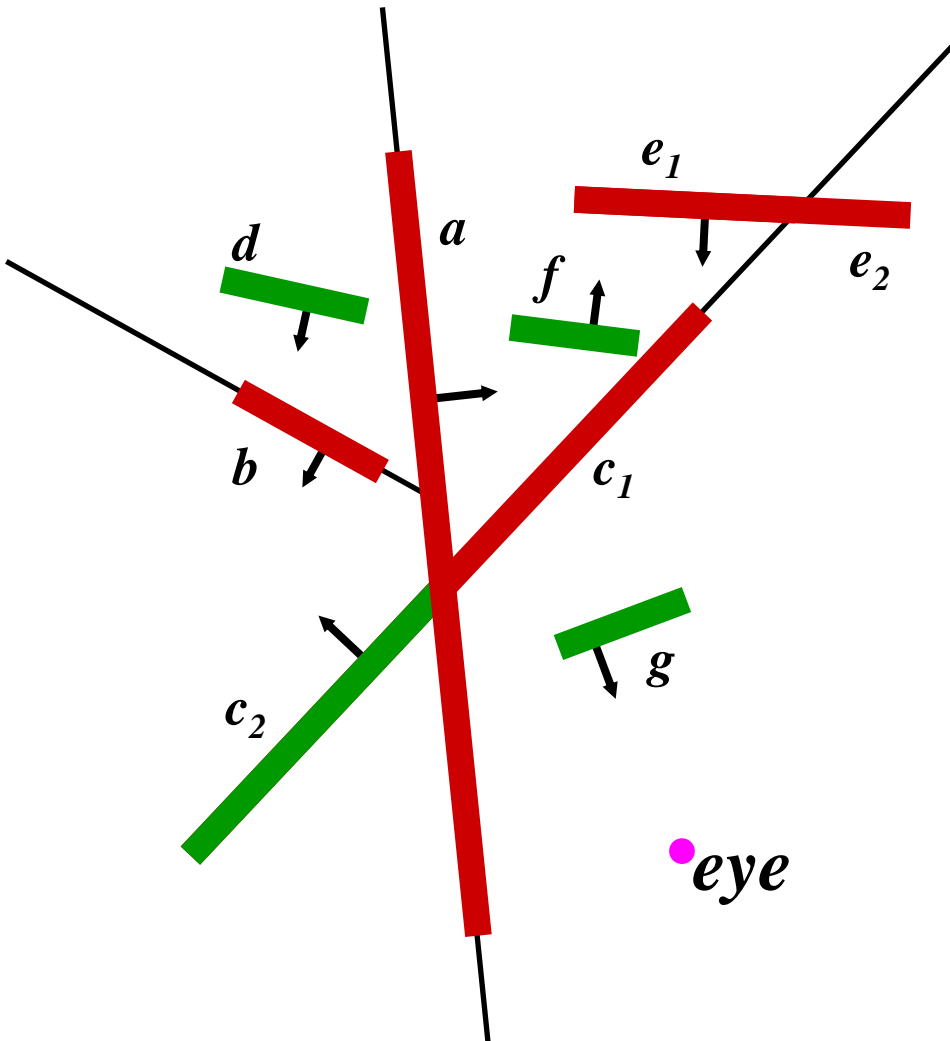
# BSP-Trees: Traversal



Order Visited:

$e_2, g, c_1, e_1, f, a, d, b, c_2$

# BSP-Trees: Traversal



Order Visited:

$d, b, c_2, a, e_1, f, c_1, e_2, g$

# BSP Demo

- Good demo:
  - <http://symbolcraft.com/graphics/bsp/>

The image displays the BSP Tree Visualizer 2.0 interface, which is used for visualizing and editing Binary Space Partitioning (BSP) trees. The interface is divided into three main sections:

- Top Left:** A 2D diagram showing five lines representing planes in a 2D space. The lines are labeled 0, 1, 2, 3, and 4. A pink arrow points from line 2 towards line 3.
- Top Right:** A 3D diagram showing the same five planes, now colored (green, blue, and cyan) and labeled with their front and back faces (e.g., 2f, 2b, 3f, 3b, 4f, 4b). A pink arrow points from the front face of plane 3 towards the back face of plane 4.
- Middle:** A tree diagram representing the BSP tree structure. The root node is 1 (green). Node 1 has two children: 2f (green) and 2b (blue). Node 2b has two children: 3f (green) and 3b (blue). Node 3b has two children: 4ff (green) and 4b (blue). A pink arrow points from node 3b towards node 4b.
- Bottom:** A 3D rendered view of the scene. The scene is a 2D plane with a gray background and a cyan foreground. The cyan area is divided into several regions by the planes, corresponding to the nodes in the tree. A pink arrow points from the cyan region towards the gray region.

At the bottom of the interface, there are control buttons and text:

- Buttons:  Create/Move,  Delete,  Drive Mode, and a **Reset** button.
- Text: BSP Tree Visualizer 2.0 Copyright © 1996-2002 Paton J. Lewis

# Summary: BSP Trees

## pros:

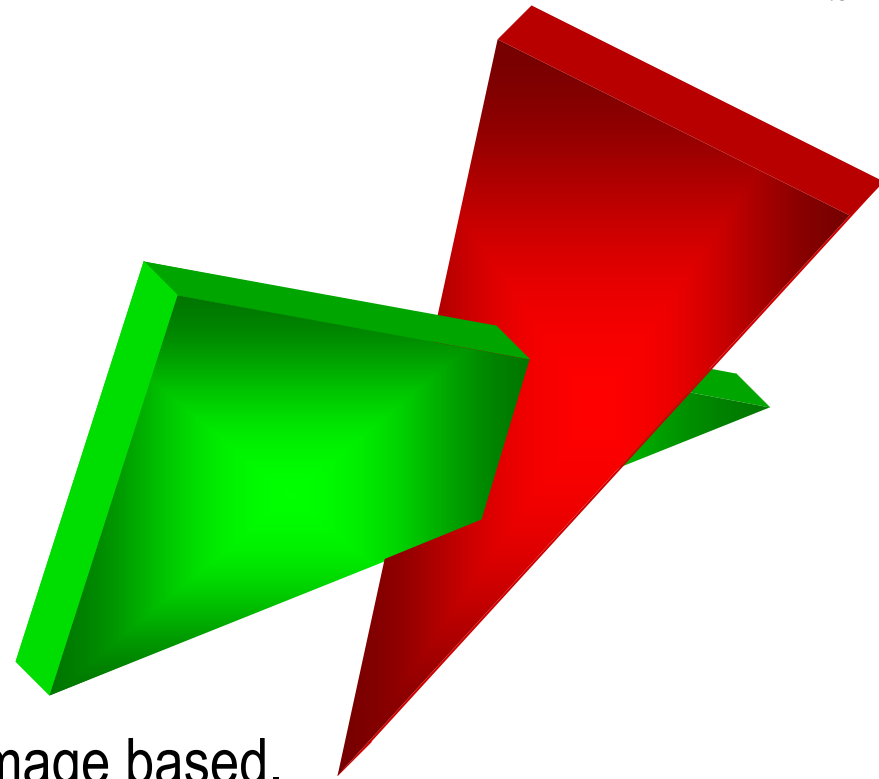
- simple, elegant scheme
- only writes to framebuffer (no reads to see if current polygon is in front of previously rendered polygon, i.e., painters algorithm)
  - thus very popular for video games (but getting less so)

## cons:

- computationally intense preprocess stage restricts algorithm to static scenes
- slow time to construct tree:  $O(n \log n)$  to split, sort
- splitting increases polygon count:  $O(n^2)$  worst-case

# Depth Buffers

- **Goal:** We want to only draw something if it appears *in front* of what is already drawn.
- What does this require? Can we do this on a per object basis?
- We can't do it object based, it must be image based.
- What do we know about the  $x,y,z$  points where the objects overlap?
- Remember our “eye” or “camera” is at the origin of our view coordinates.
- What does that mean need to store?



# The Z-Buffer Algorithm

- idea: retain depth (Z in eye coordinates) through projection transform
  - use canonical viewing volumes
  - each vertex has z coordinate (relative to eye point) intact
- augment color framebuffer with **Z-buffer** or *depth buffer* which stores Z value at each pixel
  - at frame beginning, initialize all pixel depths to  $\infty$
  - when rasterizing, interpolate depth (Z) across polygon and store in pixel of Z-buffer
  - suppress writing to a pixel if its Z value is more distant than the Z value already stored there

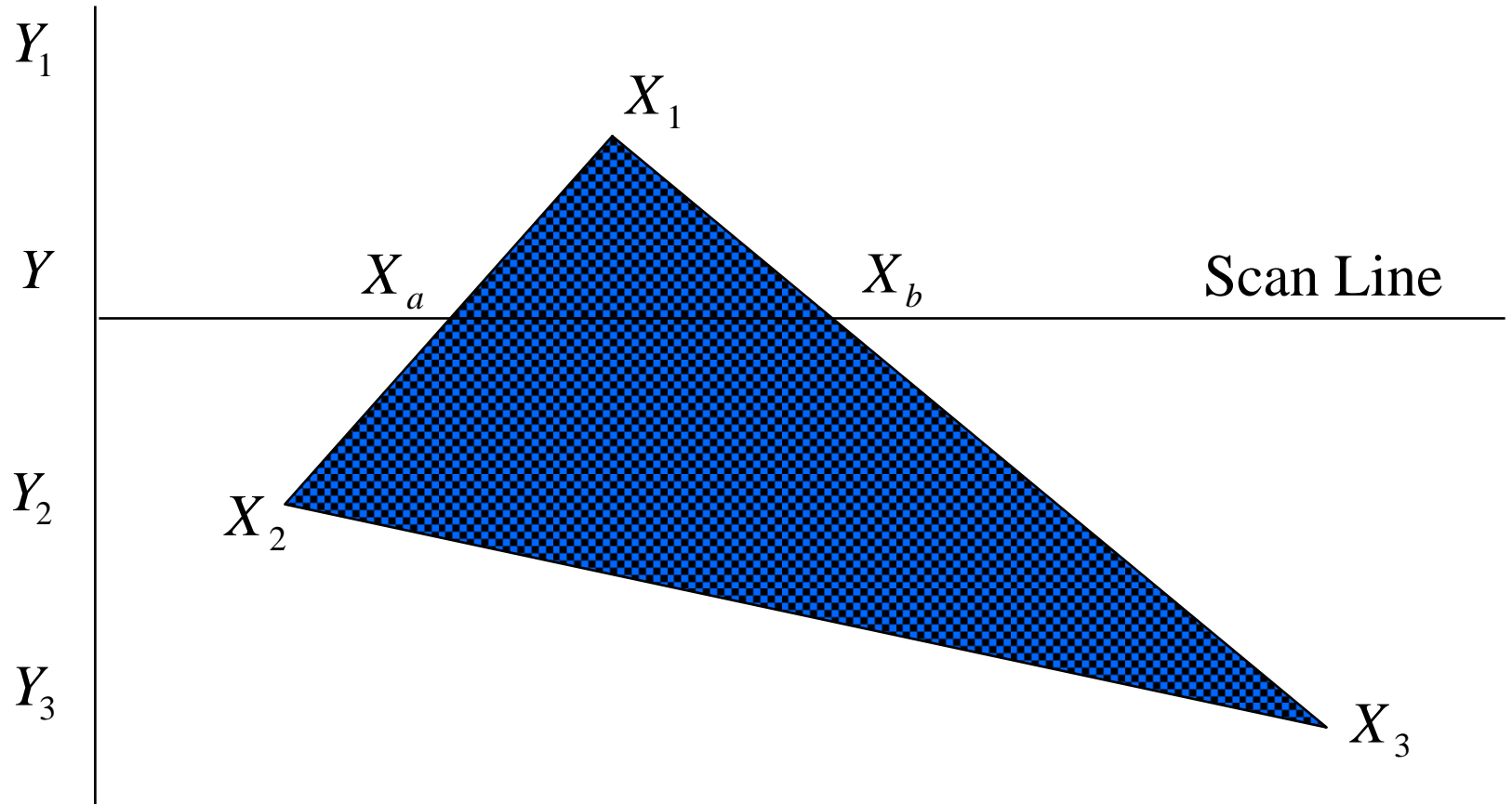
## Z-buffering in OpenGL

- Create depth buffer by setting `GLUT_DEPTH` flag in `glutInitDisplayMode()`
- Enable per-pixel depth testing with `glEnable(GL_DEPTH_TEST)`
- Clear depth buffer by setting `GL_DEPTH_BUFFER_BIT` in `glClear()`

# Polygon Scan Conversion

- Order of scan conversion is not important
- During the scan conversion, for a pixel position  $(i,j)$ , the z-buffer and the frame buffer record the information associated with the largest z encountered thus far for the pixel
- Scan conversion is done one scan-line at a time, as done for filling polygons

# Polygon Scan Conversion...



# z-Buffer Algorithm

- Suited to pipelining and hardware
- Works in image space but loops over polygons
- z-buffer:
  - Holds current closest intersection depth
  - Same resolution as frame buffer
  - Number of bits per pixel gives depth

- Algorithm:

- $\forall x, y: \text{zbuffer}(x, y) = \text{MAX}$

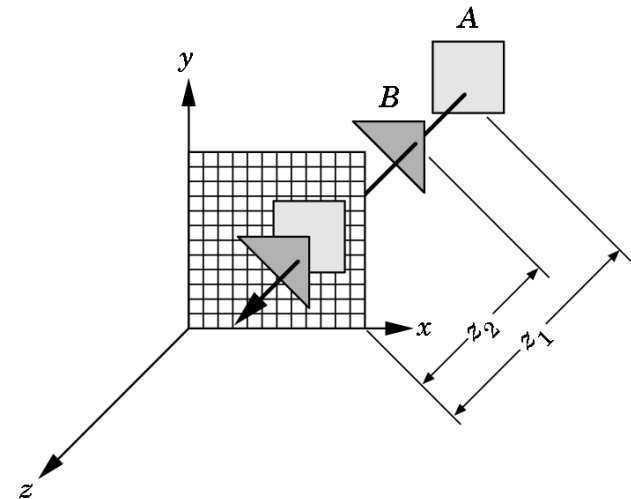
FOR each polygon for each pixel in polygon's projection

$p_z = \text{polygons } z\text{-value at pixel coord } (x,y)$

    IF  $p_z \leq \text{zbuffer}(x, y)$  THEN

$\text{zbuffer}(x, y) = p_z$

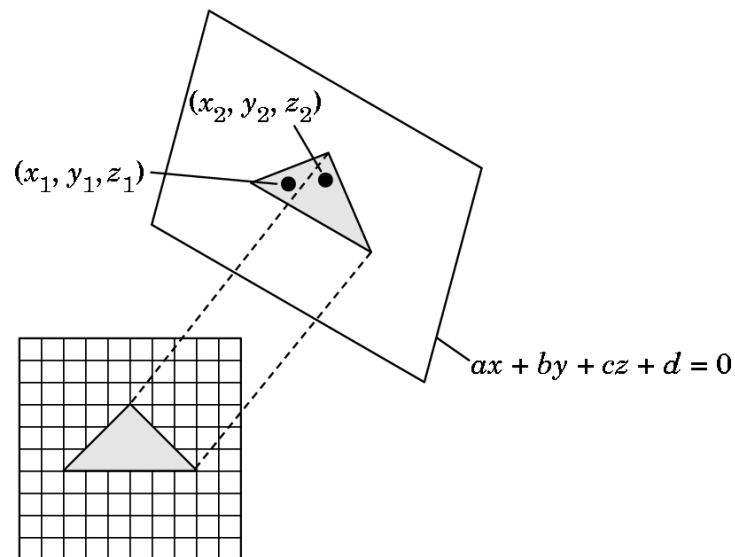
$\text{framebuffer}(x, y) = \text{polygons colour at pixel coord } (x,y)$



# z-Buffer Algorithm



- Amount of incremental work is small
- Rasterizing a polygon scan line by scan line
  - Polygon is in a plane  $ax + by + cz + d = 0$
  - Take two points  $(x_1, y_1, z_1)$  and  $(x_2, y_2, z_2)$
  - Rewrite in differential form  
 $a \Delta x + b \Delta y + c \Delta z = 0$   
 $\Delta x = x_2 - x_1, \Delta y, \Delta z$  similar
  - Across a scan line  $\Delta y = 0$  and  $\Delta x$  is a constant step
  - $\Delta z = -c/a \cdot \Delta x$



No pre-sorting needed or object-object comparisons needed

# Ray Casting

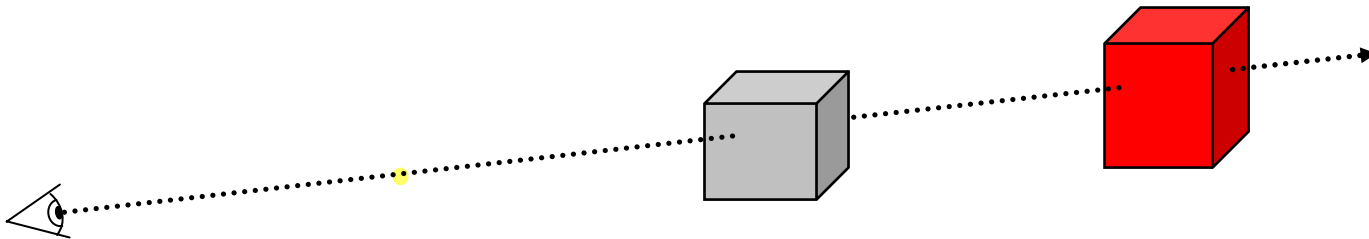
## Cast ray(*ray*)

for every polygon in the scene

- intersect *ray* with polygon
- store the *color* at point of intersection
- and the *distance* from ray origin

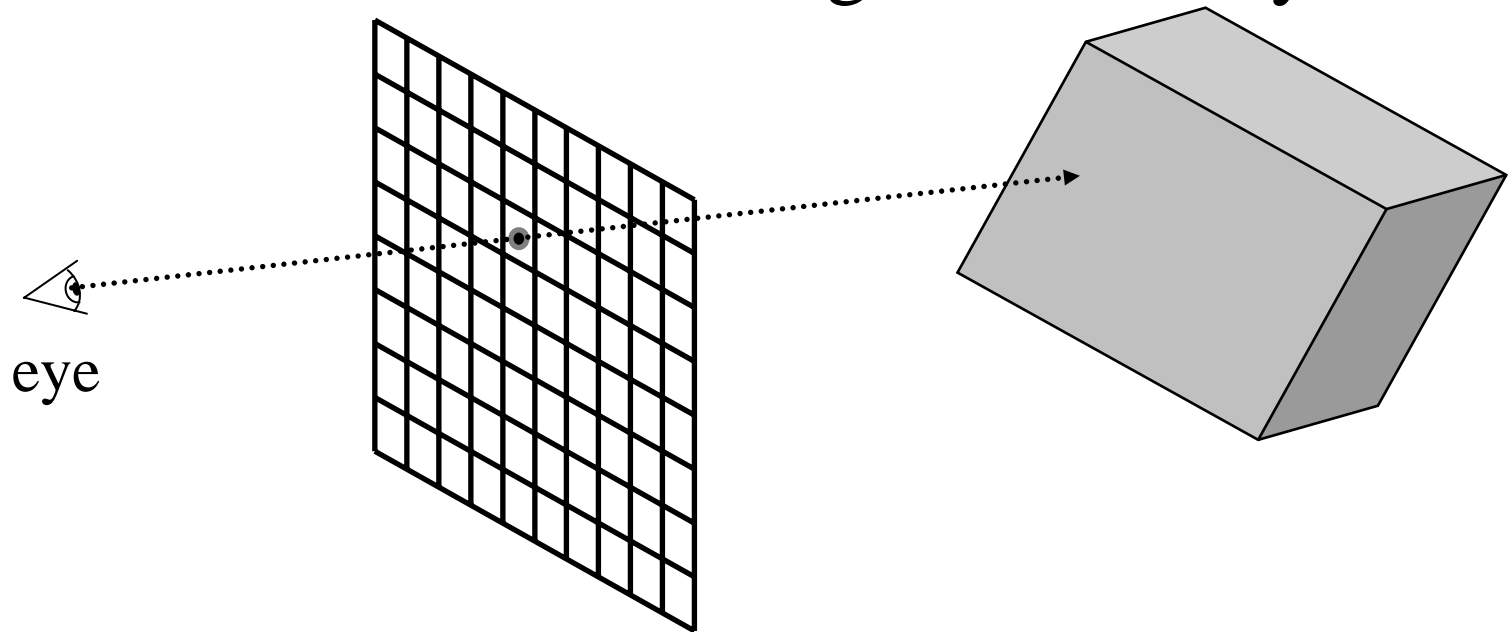
sort the intersection points according to *distance*

draw\_pixel(*color*)



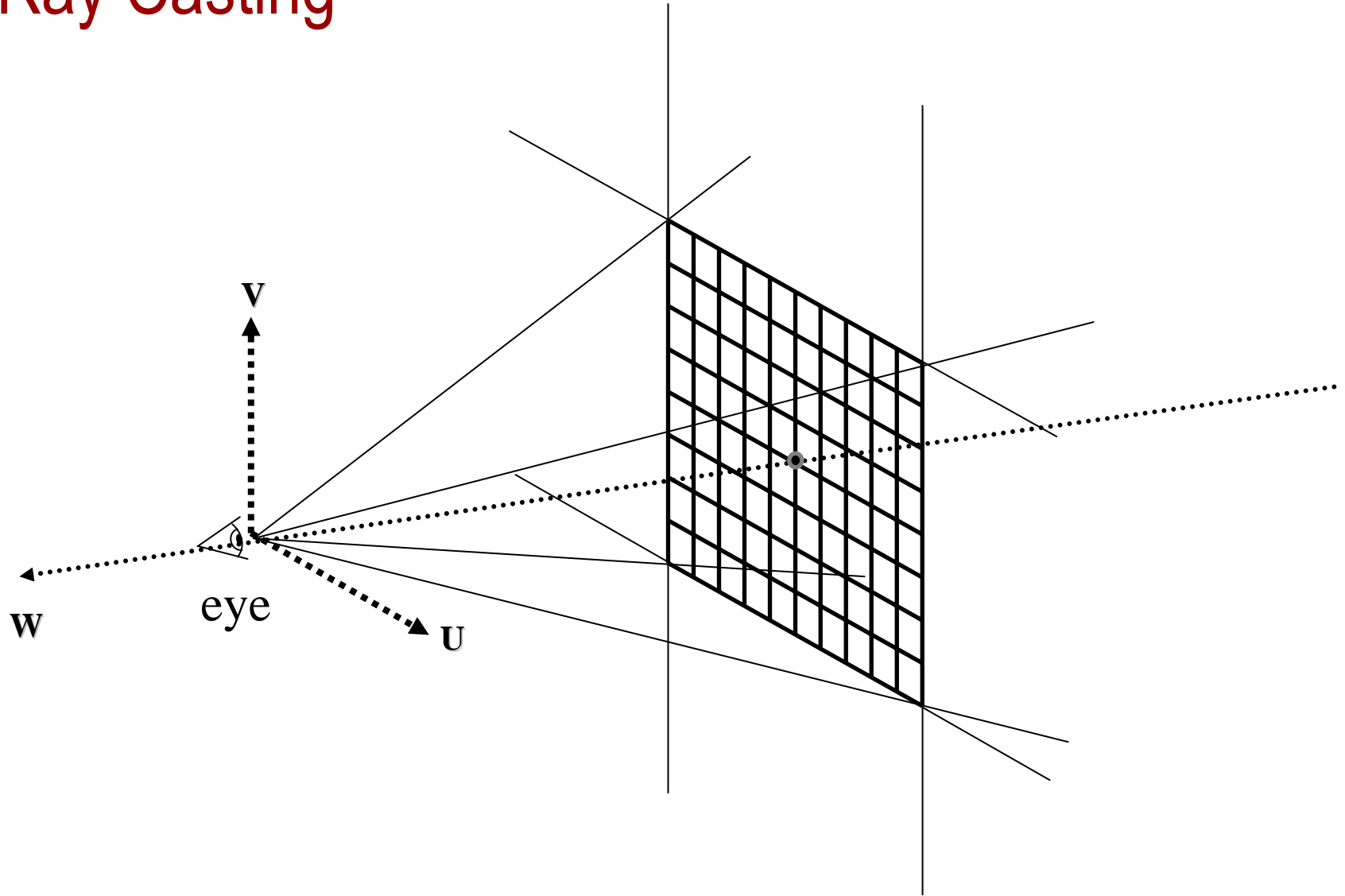
# Ray Casting

★ Find nearest surface along the view ray.

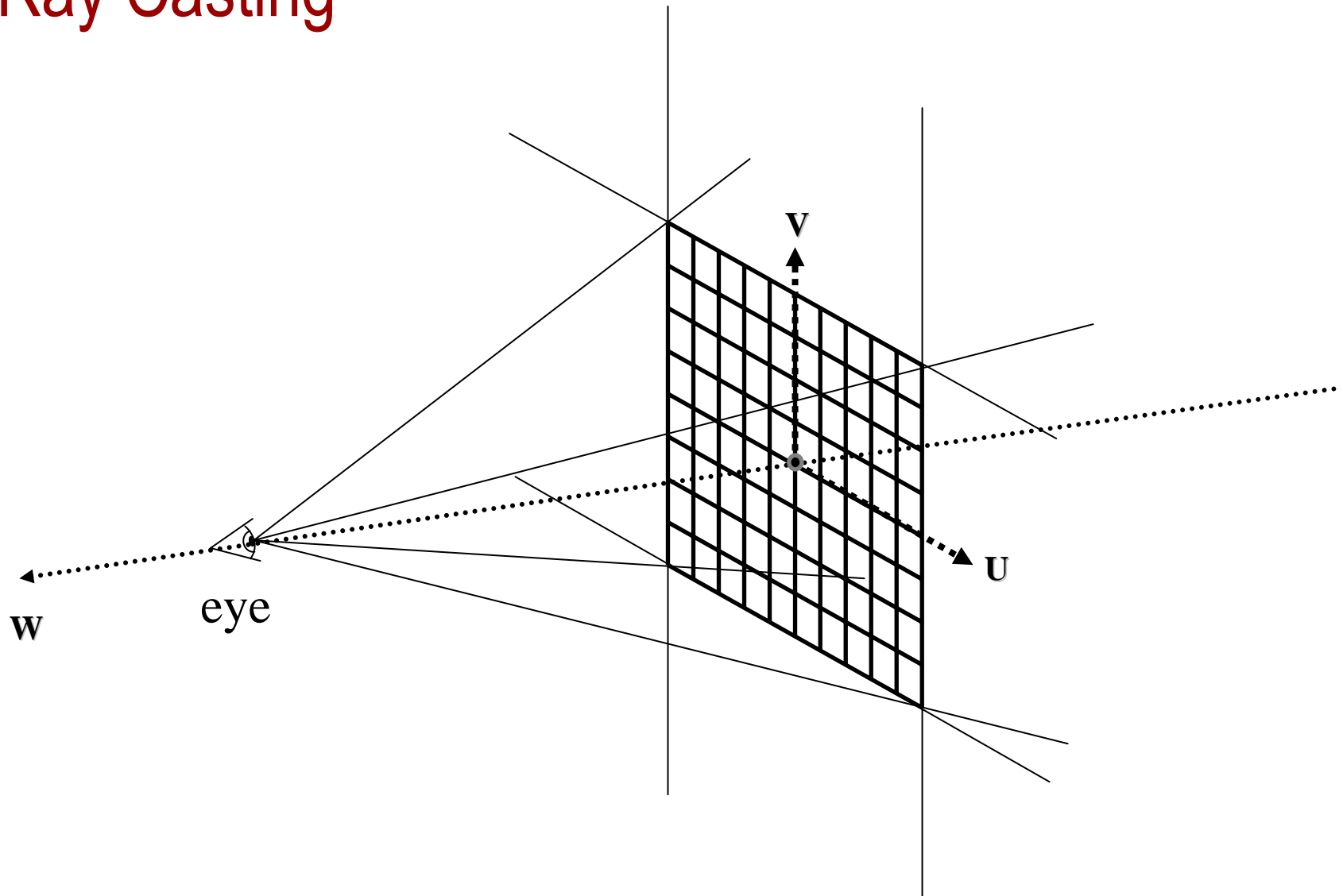


```
for i = 0 to nRows-1
  for j = 0 to nCols-1
    ray = genRay(eye, pixel(i,j))
    castRay(ray)
```

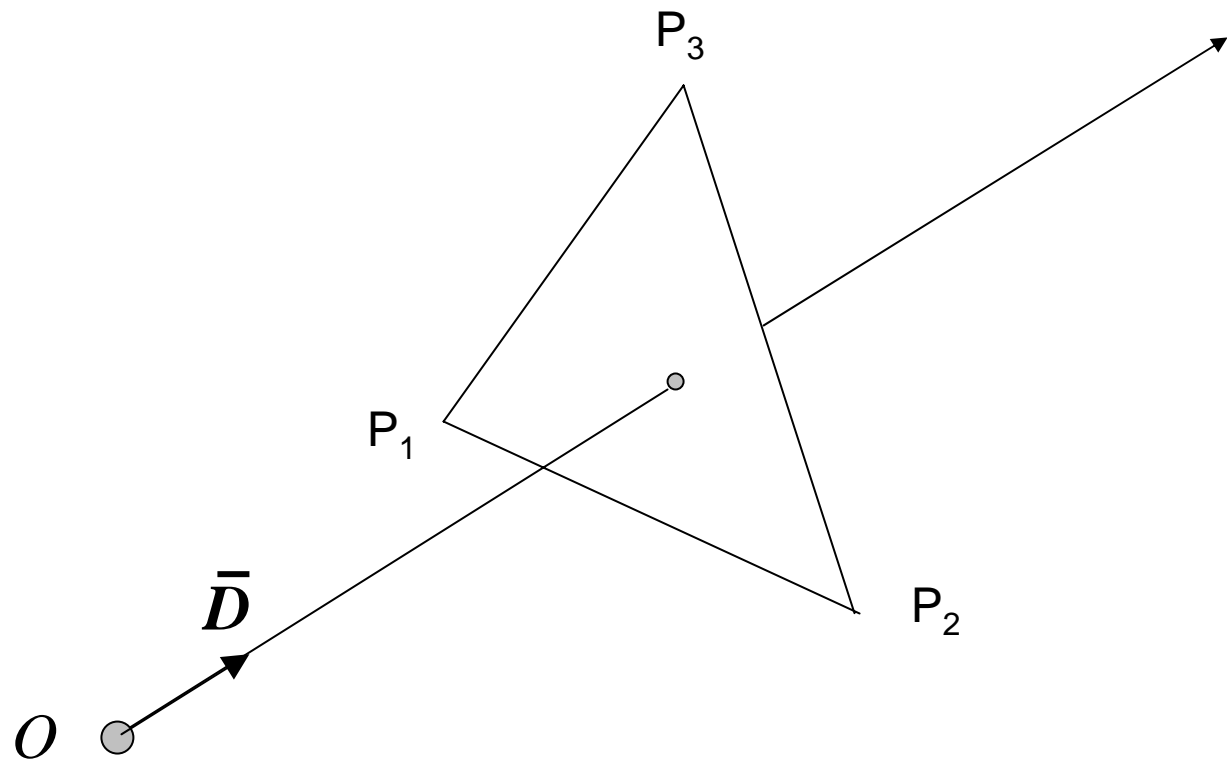
# Ray Casting



# Ray Casting



# Ray-Triangle Intersection



# Ray-Triangle Intersection

- *Step I:* Compute intersection of the Ray with the plane containing the triangle. Let it be  $P$ .
- *Step II:* Find whether point  $P$  is inside or outside the triangle.

