# Visibility

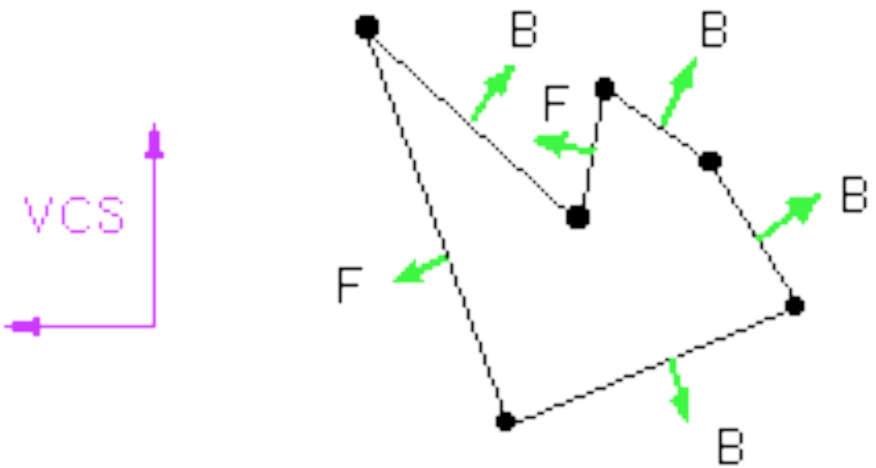
***Do not draw what is not visible***

- Self occlusions
- Object to object occlusion

# Back Face culling

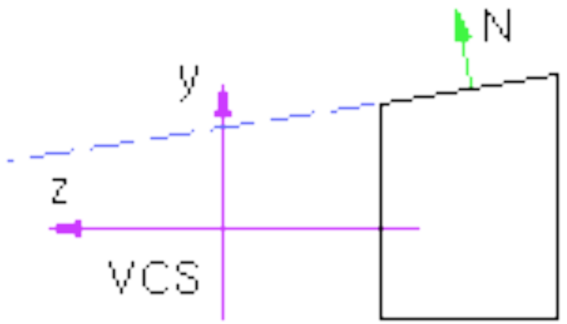
**[Hill: 406.407. Foley & van Dam: p. 663-664]**

***Remove back facing polygons***



# Back facing culling in VCS

*Can we use the z-component of the normal?*

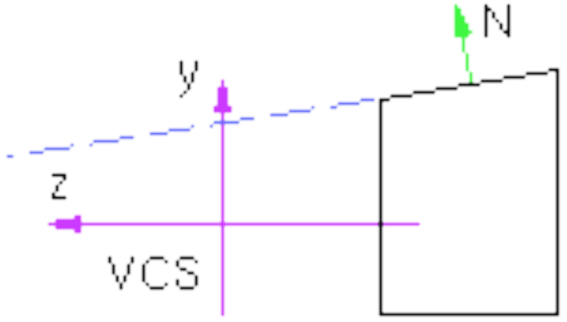


# Back facing culling in VCS

*Can we use the z-component of the normal?*

*Yes, if the projection is orthorgaphic.*

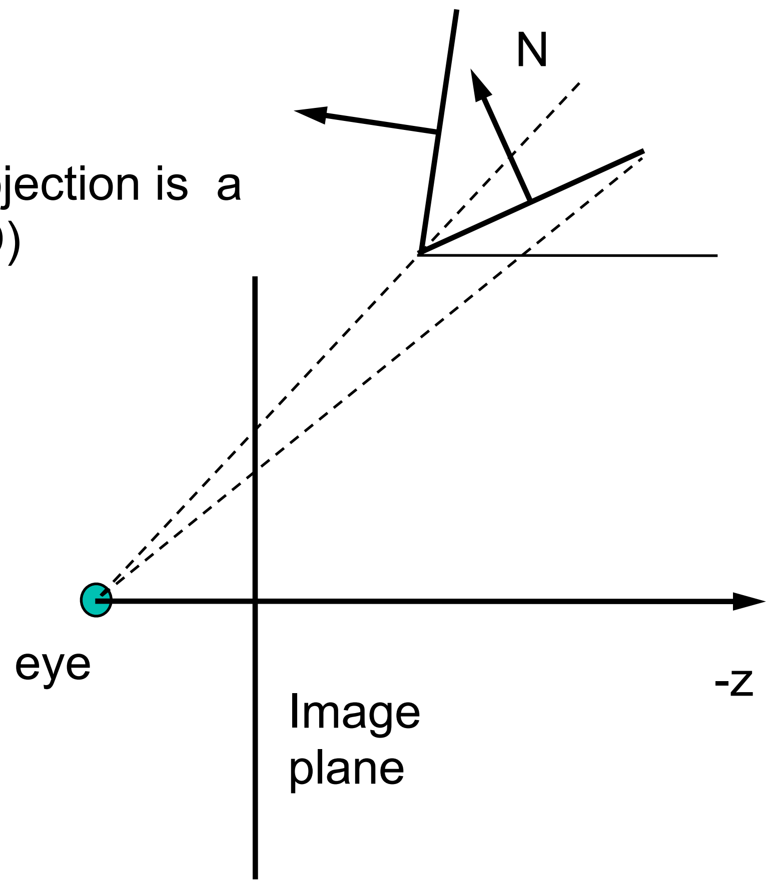
*What about perspective?*



# Back facing culling in VCS

***No!***

“Zero”point: projection is a point (line in 3D)



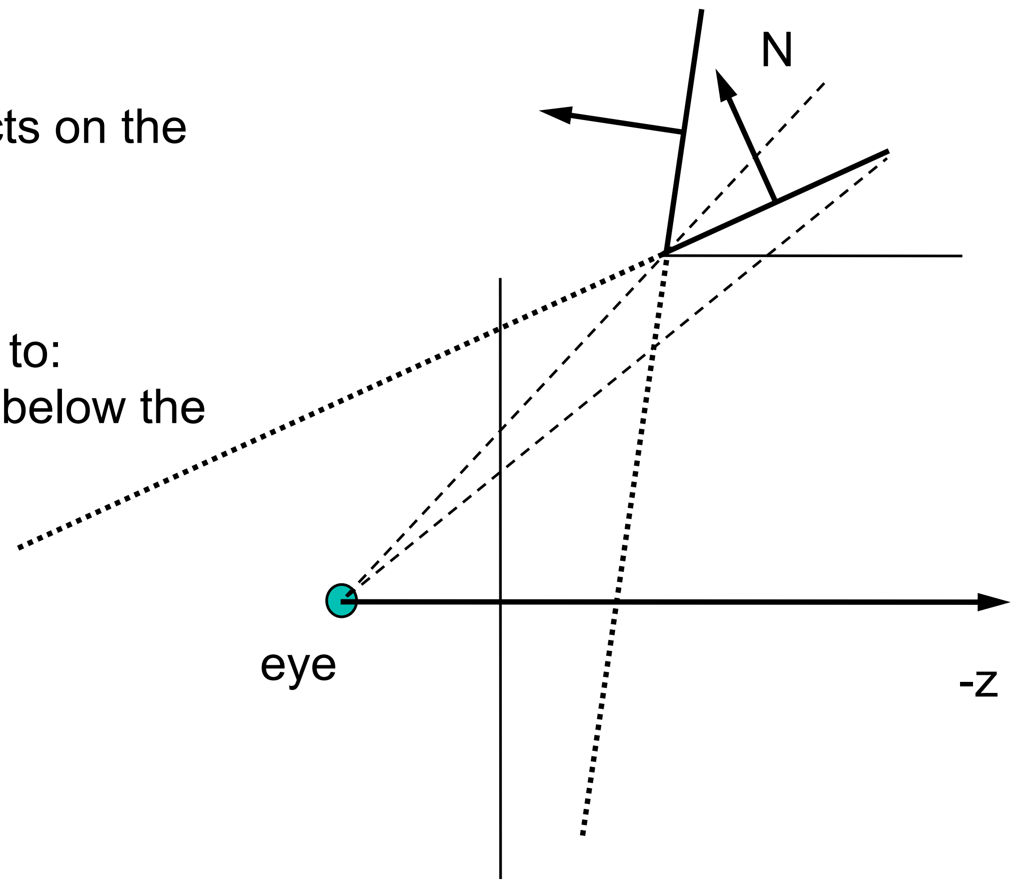
# Back facing culling in VCS

***What do we really need to look at?***

Answer:

The face that projects on the
Image plane.

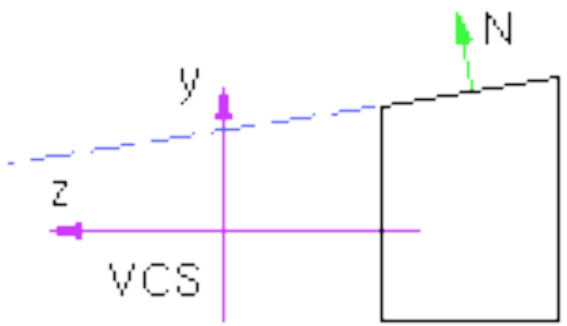
Which comes down to:
Is the eye above or below the
polygon?



# Back facing culling in VCS

## *How do we do that?*

- Calculate surface normal $N=(A,B,C)$
- Compute D in plane equation using any of the vertices of the polygon $Plane(x,y,z) = Ax+By+Cz+D$.
- Compute $Plane(eye)$ and check the sign
  > 0 above → front facing
  < 0 below (behind) → back facing



# Back face culling in NDCS

In NDCS, the z-component of the surface normal does reflect the true visibility, as desired. If the z-component is positive, the normal points away from the eye and the polygon should thus be culled.

Reminder: In NDCS the camera is pointing towards the positive z-axis.

# Back face culling in OpenGL

*glCullFaceI(GLenum mode)*

*mode: GL_FRONT*

*GL_BACK*

*GL_FRONT_AND_BACK*

*glEnable(GL_CULL_FACE) ;*

*glDisable(GL_CULL_FACE) ;*

# Visibility Algorithms

**[Hill: Chapter 13. Foley & van Dam: p. 649-651]**

***Visibility algorithms are needed to determine which objects in the scene are obscured by other objects. They are typically classified into two groups:***

image-space algorithms

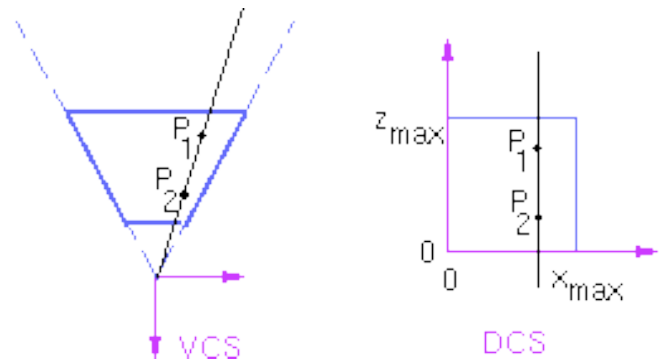
- *operate on display primitives. e.g., pixels, scan-lines*
- *visibility resolved to the precision of the display*
- *e.g.: z-buffer, Watkin's, ray-tracing*

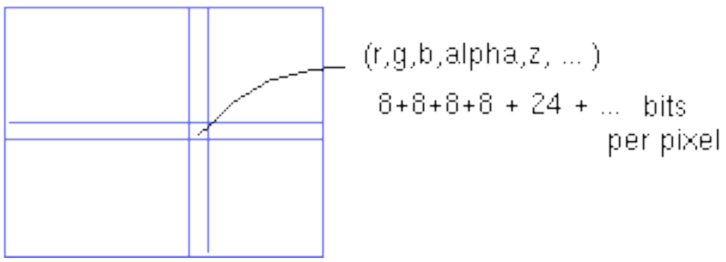
object-space algorithms

- *BSP: binary-space partitions*
- *variations on painter's algorithm*
- *worst case: creation of $O(N^2)$ primitives from N original primitives*

# Z-buffer[Hill: 436-439. Foley & van Dam: p. 668-672]

***The z-buffer keeps depth information about each pixel.***



# Z-buffer algorithm

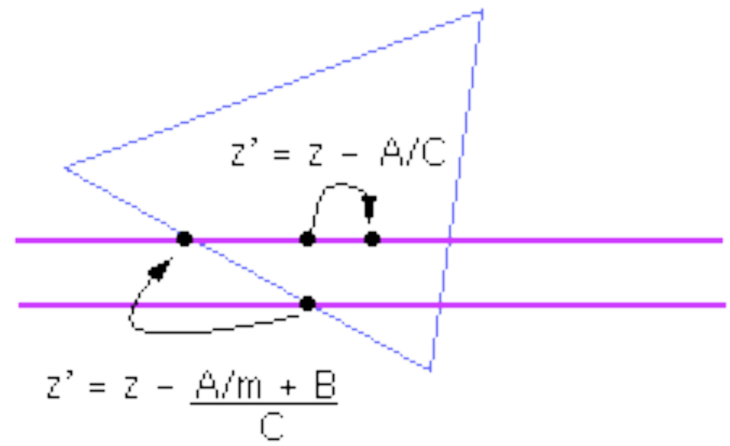
```
for all i,j {
  Depth[i,j] = MAX_DEPTH
  Image[i,j] = BACKGROUND_COLOUR
}
for all polygons P {
        for all pixels in P {
                if (Z_pixel < Depth[i,j]) {
                        Image[i,j] = C_pixel
                        Depth[i,j] = Z_pixel
                }
        }
}
```

# Characteristics of the z-buffer algorithm:

- *Commonly used*
- *Memory intensive*
- *Hardware implementation common*
- *Handles polygon interpenetration*
- *Jaggies!*

# Generating z values during scan conversion.

*Method A: From the plane equation*




We want z=f(x,y).

Plane equation: $0 = A x + B y + C z + D$

Solving for z: $z(x,y) = ( - A x - B y - D ) /C$

So $z(x+1,y) = ( - A (x+1) - B y - D ) /C$

$= z(x+1) = z(x,y) - A/C$

So along a scanline $\boxed{z(x+1,y) = z(x,y) – A/C}$

Similarly from scanline to scanline $(x,y) \rightarrow (x+1/m,y+1)$ and

$\boxed{Z(x+1/m,y+1) = z(x,y) - (A/m + B )/C.}$

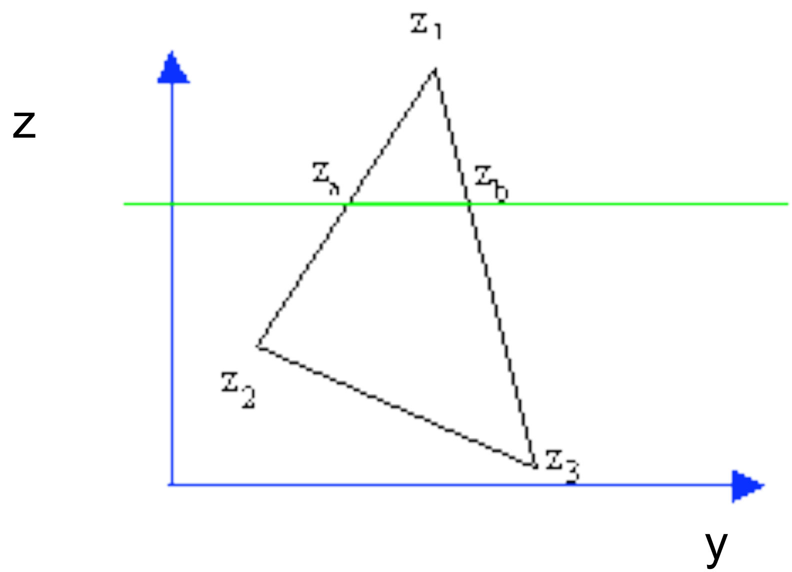
# Method B: Bilinear interpolation

***Equivalent to method A, without having to solve for plane equation***

- Incrementally interpolates any type of quantity between known values at the vertices
  - *colours -- Gouraud shading*
  - *texture coordinates*
  - *surface normals*

# Bilinear interpolation of z-coordinates

***It can also be done incrementally***




$$\frac{z_a - z_2}{z_1 - z_2} = \frac{y_a - y_2}{y_1 - y_2}$$

$$z_a = z_2 + (z_1 - z_2)\,\frac{(y_a - y_2)}{(y_1 - y_2)}$$

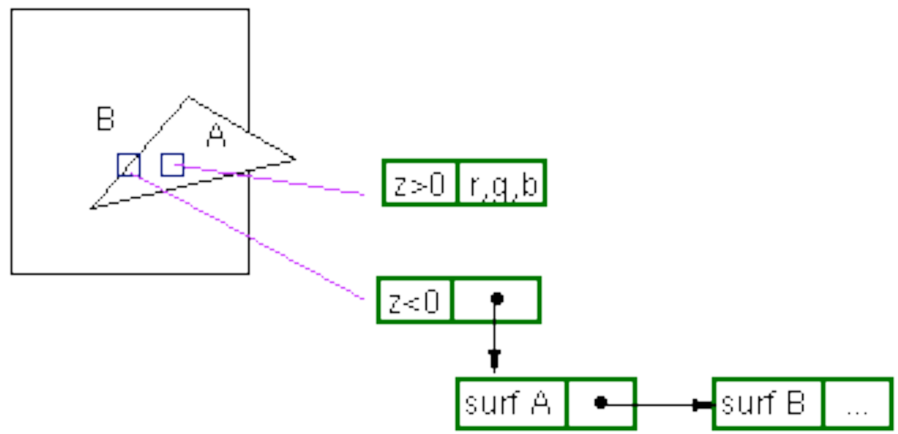
# A-buffer [Not covered in Hill]

*z-buffer: only one visible surface per pixel*

*A-buffer: linked list of surfaces*

*Antialiased, area-averaged, accumulation buffer*

# A-buffer linked list of surfaces



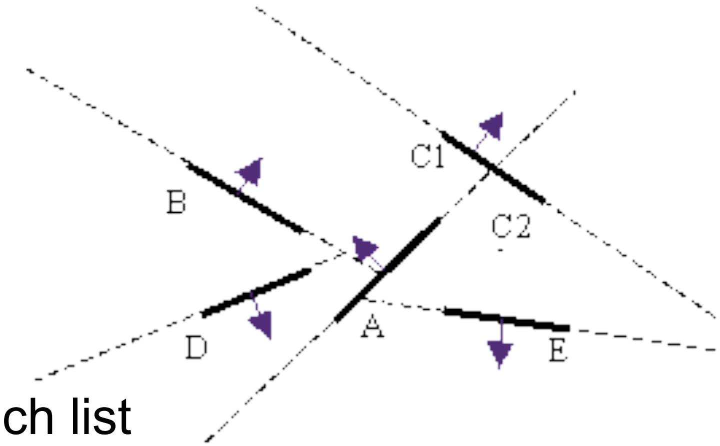

The data for each surface includes:

RGB
Z
alpha
area coverage percentage
other surface parameters

# BSP trees[Hill: 707-711. Foley & van Dam: p. 675-680]

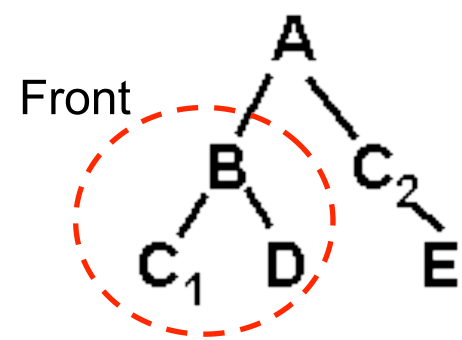
- Binary space partition
- Object space, produces back-to-front ordering
- Preprocess the scene once to build BSP tree
- Traversal of BSP tree is view dependent

# Building a BSP tree




```
BSPtree *BSPmaketree(polygon list) {
   choose a polygon as the tree root
   for all other polygons
      if polygon is in front, add to front list
      if polygon is behind, add to behind list
      else split polygon and add one part to each list
   BSPtree = BSPcombinetree(BSPmaketree(front list),
     root,  BSPmaketree(behind list) )
}
```

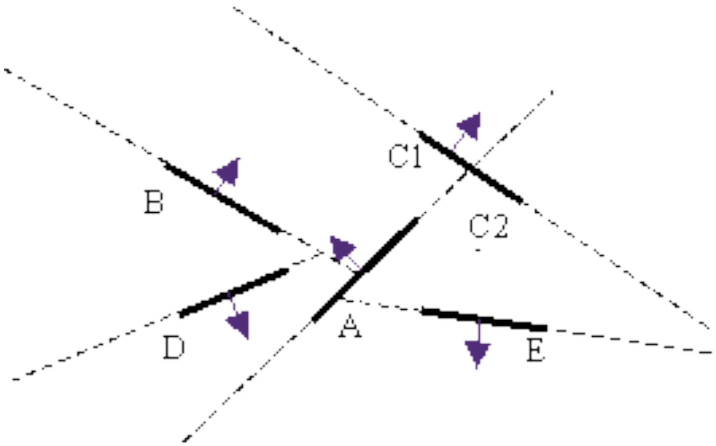
Example tree with A chosen as the root:

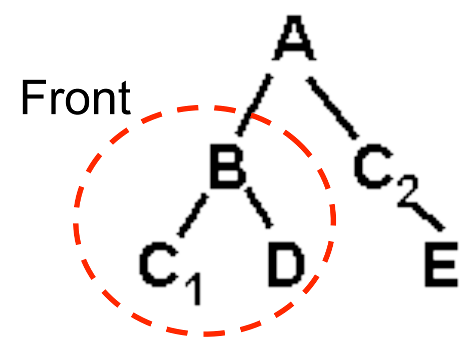


# Drawing using the BSP tree

***View dependent***

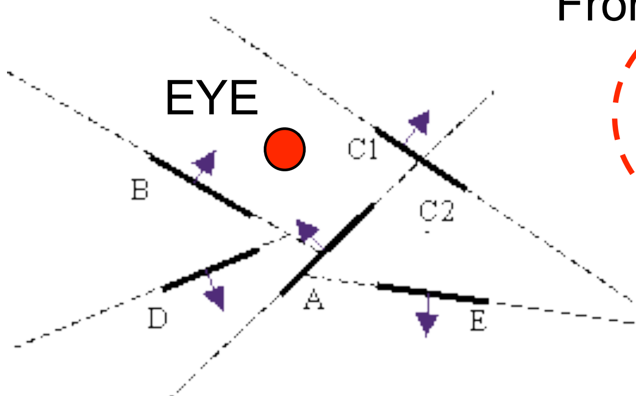
```
DrawTree(BSPtree) {
    if (eye is in front of root) {
            DrawTree(BSPtree->behind)
            DrawPoly(BSPtree->root)
            DrawTree(BSPtree->front)
      } else {
            DrawTree(BSPtree->front)
            DrawPoly(BSPtree->root)
            DrawTree(BSPtree->behind)
      }
    }
}
```

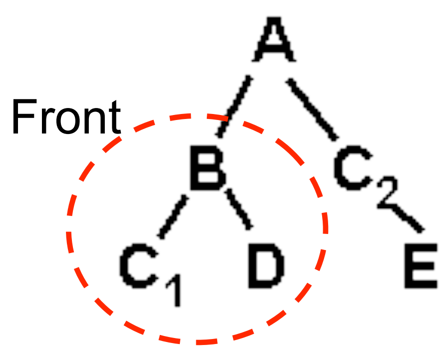


# Example

```
DrawTree(BSPtree) {
        if (eye is in front of root) {
              DrawTree(BSPtree->behind)
              DrawPoly(BSPtree->root)
              DrawTree(BSPtree->front)
         } else {
              DrawTree(BSPtree->front)
              DrawPoly(BSPtree->root)
              DrawTree(BSPtree->behind)
         }
      }
}
```




Execution:
Eye in front of A,
draw A->behind,
eye behind C2,
draw C2->front,
draw C2,
draw C2→behind,

draw E,
draw A,
draw A→front,
Eye in front of B,
draw B→behind,

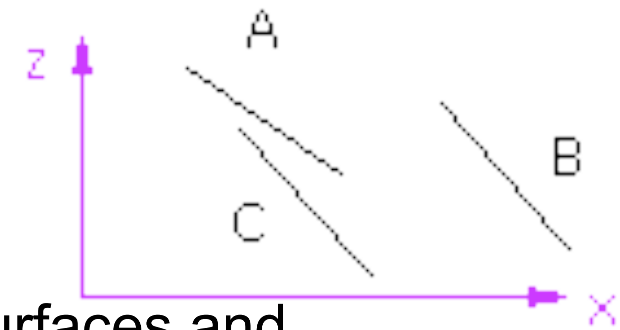
# Depth sorting algorithms [Hill: 706,711-713. Foley & van Dam: p. 672-675]

***Several object-space algorithms achieve a front-to-back ordering in other ways. These depth-sort algorithms have the following basic steps:***

- Sort polygons by z
- Resolve ambiguities where z-extents overlap
- Scan-convert polygons in back-to-front order

# Resolving ambiguities

- Bounding rectangles do not overlap in xy-plane
- A is completely behind C
- C is completely in front of A
- Projections on xy-plane do not overlap



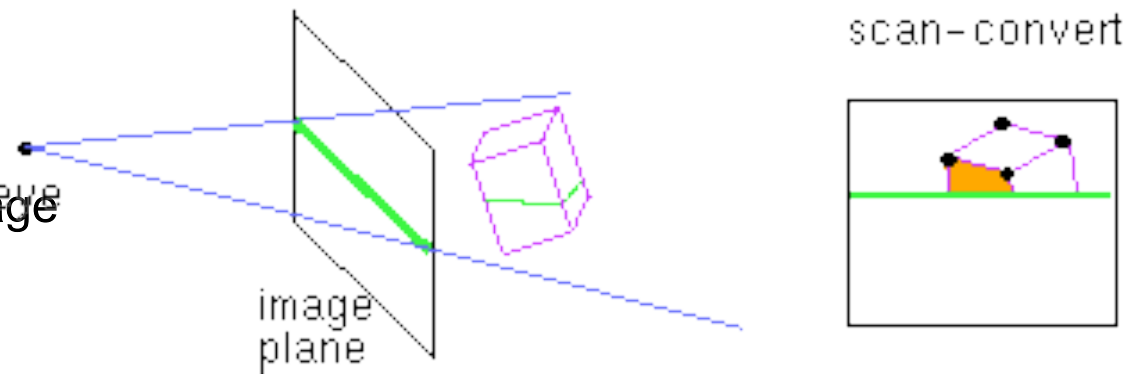

If these fail, exchange the order of the surfaces and repeat. If this still fails, the polygons can be intersected to split one of the polygons if necessary. In the worst case, the algorithm can generate O(n^2) new polygons.

# Scanline Algorithms

**[Hill: 713-716. Foley & van Dam: p. 680-684]**

modify scan-conversion to handle multiple polygons

- priority according to Z
- resolve visibility one scanline at a time
- less memory




for each scanline (row) in image
  for each pixel in scanline
    determine closest object
    calculate pixel colour, draw pixel
  end
end

# Raytracing

for each pixel on screen

determine ray from eye through pixel

find closest intersection of ray with an object

cast off reflected and refracted ray, recursively

calculate pixel colour, draw pixel

End

- *rays cast through image pixels*
- *solves visibility, some global illumination*
- *requires efficient intersection tests O(mnN): m x n pixels, N objects*