

## Visibility

### ***Do not draw what is invisible***

- Due to being outside the view volume
- Due to self-occlusion
- Due to object-to-object occlusion

## Visibility

### ***Do not draw what is invisible***

- Outside the view volume
  - *Clipping, culling*
- Self-occlusion
- Object-to-object occlusion

# 3D Clipping

Keep only what is visible

We can clip in 3 different CSs

1. In the VCS

What are the six plane equations?

2. In the CCS

Clipping in homogeneous coordinates

4D volume bounded by 3D planes

Still simple and efficient

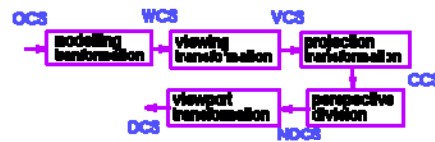
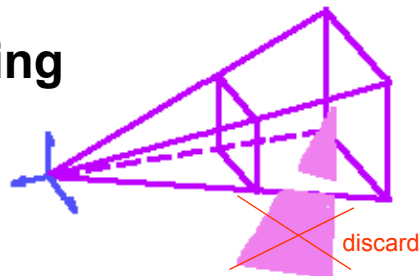
Usually done here (!)

(a.k.a. Clipping Coordinate System)

3. In the NDCS

Singularity at  $P_z = 0$

In any case, we must clip lines against planes



## Reminder: Planes

### Plane equations

- Explicit

$$z = \alpha x + \beta y + \gamma$$

$$\alpha = -a / c$$

$$\beta = -b / c$$

$$\gamma = -d / c$$

- Implicit

$$c \neq 0$$

$$F(x, y, z) = ax + by + cz + d = \mathbf{n} \cdot \mathbf{P} + d$$

$$\text{Points on Plane: } F(x, y, z) = 0$$

- Parametric

$$\text{Plane}(s, t) = P_0 + s(P_1 - P_0) + t(P_2 - P_0)$$

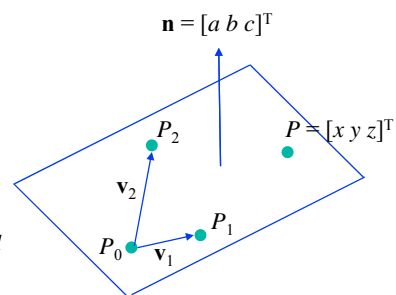
$P_0, P_1, P_2$  are not collinear

or

$$\text{Plane}(s, t) = P_0 + s\mathbf{v}_1 + t\mathbf{v}_2, \text{ where } \mathbf{v}_1, \mathbf{v}_2 \text{ are basis vectors}$$

Convex combination defines a triangle:

$$\text{Triangle}(s, t) = (1 - s - t)P_0 + sP_1 + tP_2, \text{ with } s, t \in [0, 1]$$



## Intersection of Line and Plane

Implicit equation for the plane:

$$F(P) = \mathbf{N} \cdot \mathbf{P} + D = 0$$

Parametric equation for the line from  $P_a$  to  $P_b$ :

$$L(t) = P_a + t(P_b - P_a)$$

Plug  $L(t)$  into  $F(P)$  and solve for  $t = t_i$ :

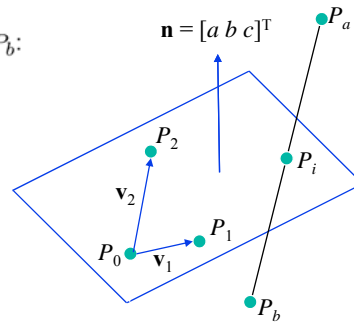
$$\mathbf{N} \cdot [P_a + t_i(P_b - P_a)] = -D$$

Therefore,

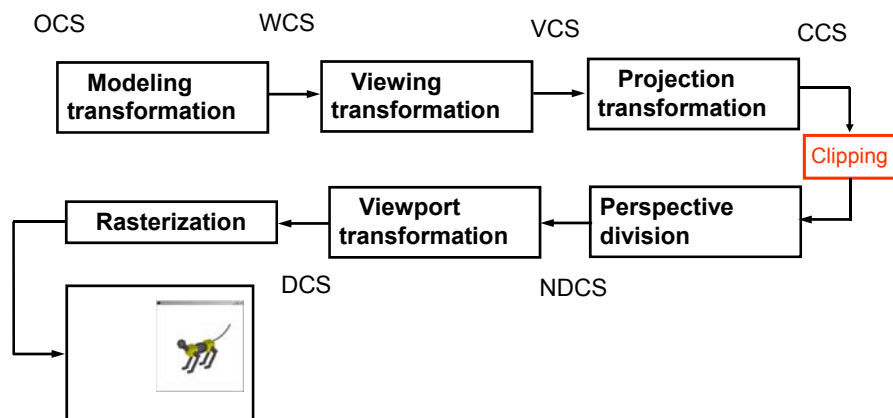
$$t_i = \frac{-D - \mathbf{N} \cdot P_a}{\mathbf{N} \cdot P_b - \mathbf{N} \cdot P_a} = \frac{-F(P_a)}{F(P_b) - F(P_a)}$$

Finally, evaluate  $L(t_i)$  for intersection point  $P_i$ :

$$P_i = P_a - \frac{F(P_a)(P_b - P_a)}{F(P_b) - F(P_a)} = \frac{P_a F(P_b) - P_b F(P_a)}{F(P_b) - F(P_a)}$$



## Z-Buffer Graphics Pipeline



## Polygon Culling

***When an entire polygon lies outside the view volume, it can be “culled”***

- I.e., eliminated from the pipeline
- Especially helpful when many triangles are grouped into an object with an associated bounding volume (e.g., a bounding sphere) which lies outside the view volume (signed distance to plane > sphere radius)

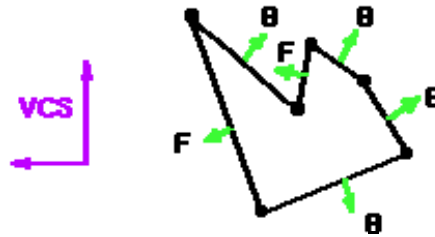
## Visibility

***Do not draw what is invisible***

- Outside the view volume
  - *Clipping, culling*
- Self-occlusion
  - *Backface culling*
- Object-to-object occlusion

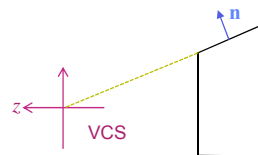
## Backface Culling

*Remove backfacing polygons*



## Backface Culling in the VCS

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



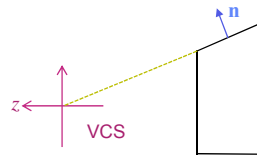
## Backface Culling in the VCS

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

*Yes, if the projection is orthographic*

*What about perspective?*

*No!*

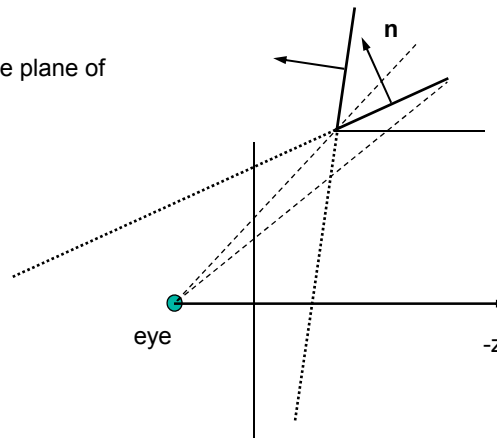


## Backface Culling in the VCS

*What do we really need to look at?*

Answer:

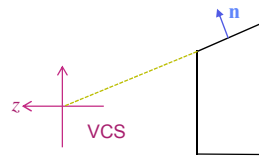
Is the eye above or below the plane of the polygon?



## Backface Culling in the VCS

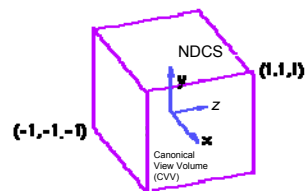
### How do we do that?

- Calculate the normal  $\mathbf{n} = [a, b, c]^T$
- Compute  $d$  in plane equation using any of the vertices of the polygon  
 $Plane(x, y, z) = ax + by + cz + d = 0$
- Compute  $Plane(eye-position)$  and check the sign  
 $> 0$  above  $\rightarrow$  front facing  
 $< 0$  below (behind)  $\rightarrow$  back facing



## Backface Culling in the 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 the NDCS, the camera is pointing towards the positive  $z$ -axis.

# Visibility

## ***Do not draw what is invisible***

- Outside the view volume
  - *Clipping, culling*
- Self-occlusion
  - *Backface culling*
- Object-to-object occlusion
  - *Visibility algorithms*

# Visibility Algorithms

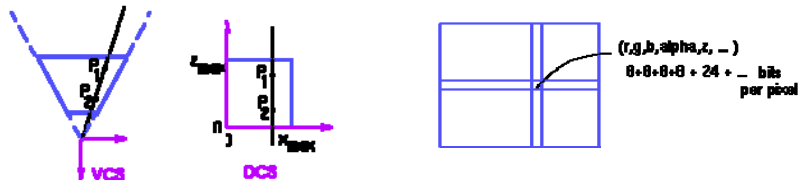
***Visibility algorithms are needed to determine which objects in the scene are obscured by other objects. They are typically classified into two groups:***

- 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 algorithm, ray-tracing
- Object-space algorithms
  - BSP: binary-space partitions
  - variations on the Painter's Algorithm
  - worst case: creation of  $O(N^2)$  primitives from  $N$  original primitives



# Z-Buffer

*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_COLOR
}
for all polygons P {
    for all pixels in P {
        if ( $Z_{\text{pixel}} < \text{Depth}[i,j]$ ) {
            Image[i,j] = Color_pixel
            Depth[i,j] =  $Z_{\text{pixel}}$ 
        }
    }
}

```

## 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:  $ax + by + cz + d = 0$

Solving for  $z$ :  $z(x, y) = (-ax - by - d) / c$

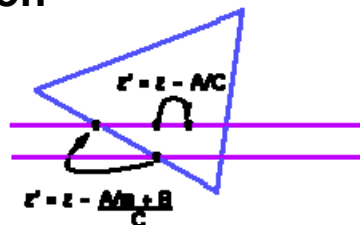
So  $z(x+1, y) = (-a(x+1) - by - d) / c$

$$= z(x, y) - a/c$$

So along a scanline  $z(x+1, y) = z(x, y) - a/c$

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

$$z(x+1/m, y+1) = z(x, y) - (a/m + b)/c.$$



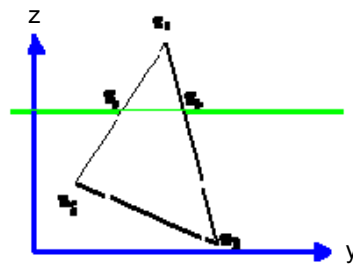
## Method B: Bilinear Interpolation

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

- Incrementally interpolates any type of quantity between known values at the vertices
  - colors (Gouraud shading)
  - texture coordinates
  - surface normals

## Bilinear Interpolation of Z-Coordinates

*It can also be done incrementally*



Observe that

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

Therefore

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

## A-Buffer

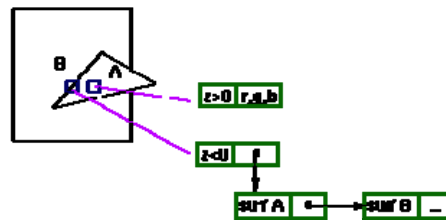
***Z-buffer: only one visible surface per pixel***

***A-buffer: linked list of surfaces***

- Antialiased, area-averaged, accumulation buffer

The data for each surface includes:

RGB  
z  
alpha  
area coverage percentage  
other surface parameters



## Binary Space Partition (BSP) Trees

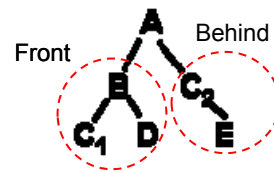
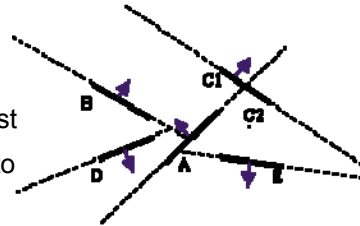
- Object space algorithm
- Produces back-to-front ordering
- Preprocess the scene once to build BSP tree
- Traversal of BSP tree is view dependent

## 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) )
}
    
```



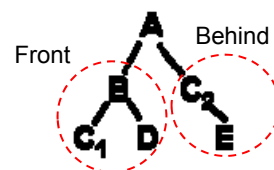
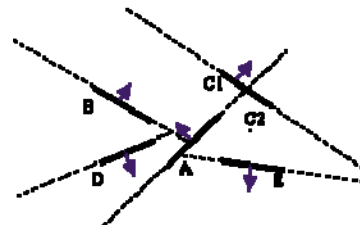
Example tree with A chosen as the root:

## 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)
    }
}
    
```

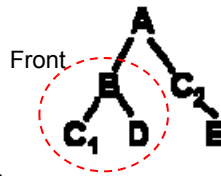
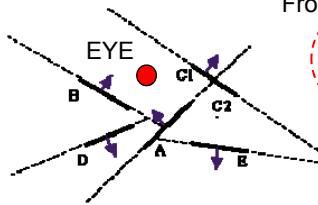


## Example

```

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 C<sub>2</sub>,  
 draw C<sub>2</sub>->front,  
 draw C<sub>2</sub>,  
 draw C<sub>2</sub>->behind,  
 draw E,

draw A,  
 draw A->front,  
 Eye in front of B,  
 draw B->behind,  
 draw D,  
 draw B,  
 draw B->front,  
 draw C<sub>1</sub>

Drawing order: C<sub>2</sub>, E, A, D, B, C<sub>1</sub>

## Depth Sorting Algorithms

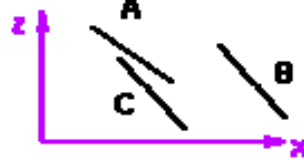
**Several object-space algorithms achieve a front-to-back ordering in other ways**

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

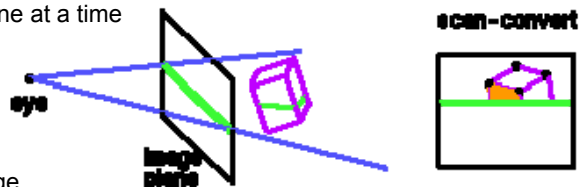
If this still fails, the polygons can be intersected to split one of the polygons if necessary

In the worst case, the algorithm can generate  $O(n^2)$  new polygons

## Scanline Algorithms

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 color, draw pixel

end

end

# Ray Tracing

*We will study this algorithm later*

for each pixel on the screen

determine a **view ray** from the camera through the pixel

find the closest intersection of the view ray with an object

cast off other rays, recursively

calculate pixel color and draw pixel

End

- View rays cast through image pixels
- Solves visibility, some global illumination
- Requires efficient intersection tests  
 $O(mnN)$ :  $m \times n$  pixels,  $N$  objects

