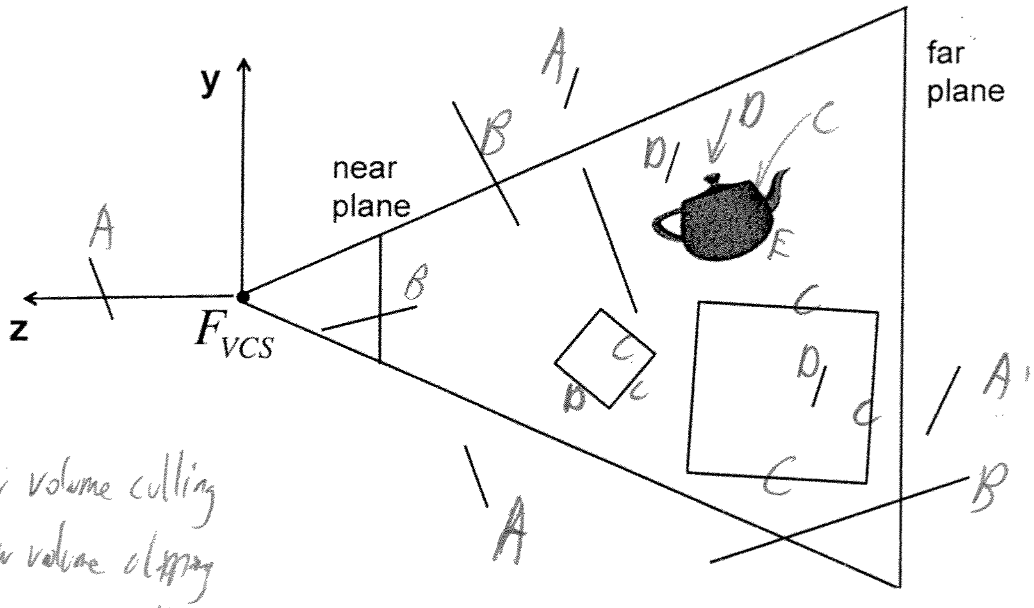
Ref

Visibility

Determining which objects / triangles / pixels can be seen



(A) view volume culling
(B) view volume clipping
(C) backface culling
(D) occlusion: z-buffer (pixel level)
(E) occlusion culling (object level)

Visibility

Methods

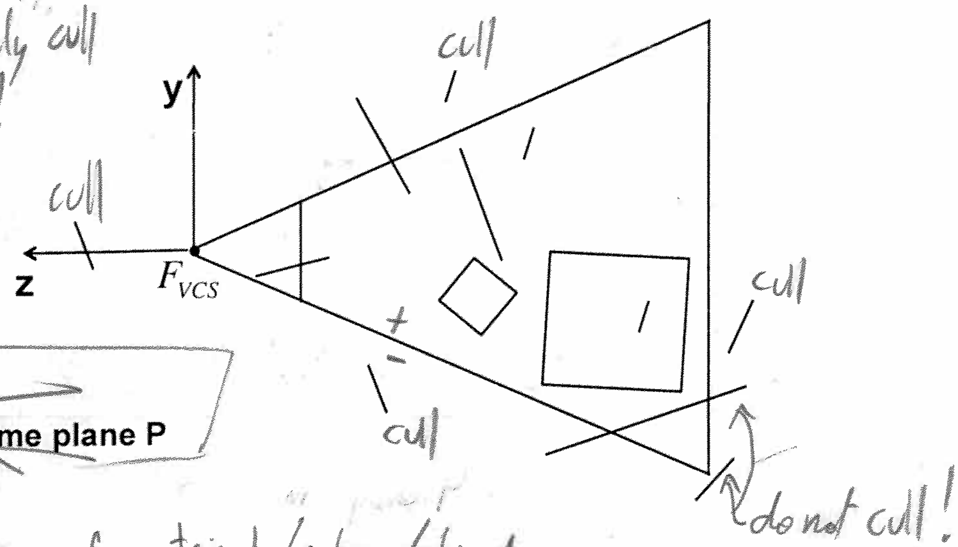
- view volume culling
- view volume clipping
- backface culling
- z-buffer occlusion test

- painter's algorithm & BSP trees
- occlusion culling
- raycasting (and raytracing)

View Volume Culling (for triangles)

Idea: cull a triangle if all vertices are outside the view volume

No! This will falsely "cull" some triangles!




```
cull = true
for each vertex V,
  for each view-volume plane P
```

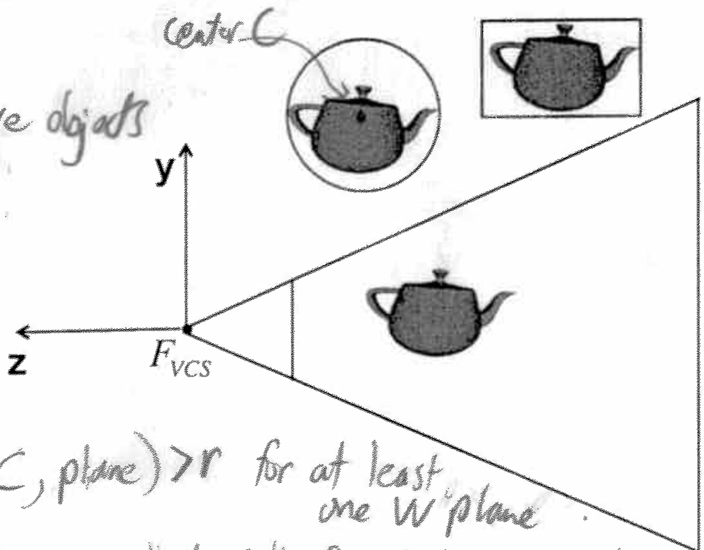
cull if all vertices of a triangle/polygon/object are "outside" wrt at least one of the view volume planes

This happens in hardware for triangles, "fixed function" part of pipeline.

View Volume Culling (for objects)

Software!

Idea: fast cull of entire objects



bounding sphere:

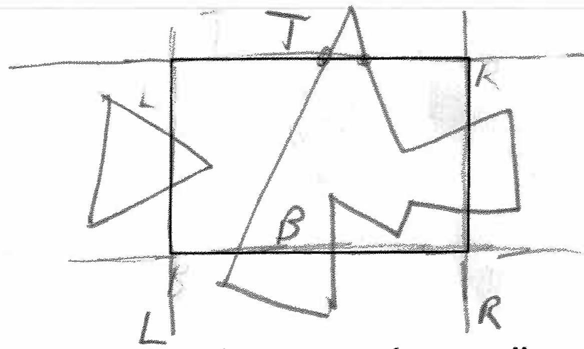
cull if dist(C, plane) > r for at least one VV plane

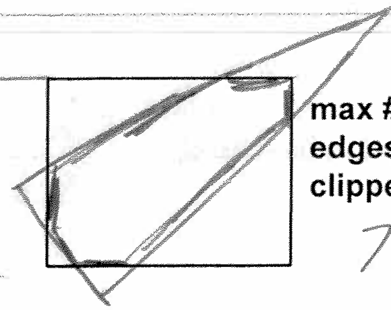
bounding box:

cull if all vertices are "outside" for at least one VV plane.

View Volume Clipping

2D clipping





max # of edges for a clipped triangle?

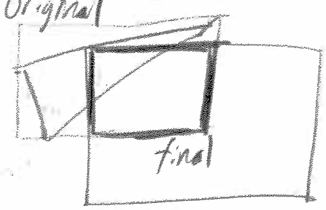
7 sided polygon!

general procedure to produce a clipped polygon: (Sutherland-Hodgeman) (next slide)

original vertex list → clip near → clip far → clip top → clip bot → clip left → clip right → clipped vertex list.

procedure for triangles with bounding-box scan conversion:

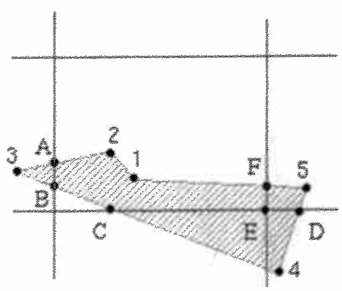
vertex list → clip near → clip far → clip polygon bounding box → clipped



(skip)

View Volume Clipping

Sutherland Hodgeman clipping



repeat starting vertex

original: 1, 2, 3, 4, 5, 1

clip L: 1, 2, A, B, 4, 5, 1

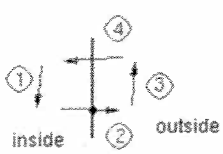
clip B: 1, 2, A, B, C, D, 5, 1

clip R: 1, 2, A, B, C, E, F, 1

clip T: (unchanged)

```
for each side of clipping window
   for each edge of polygon
      output points based upon the following table
```

also add: clip near, clip far.



case #	first point	second point	output point(s)
1	inside	inside	second point
2	inside	outside	intersection point
3	outside	outside	none
4	outside	inside	intersection point and second point

How to compute 3D intersection points?

⇒ in a few slides

Clipping in VCS

Works!

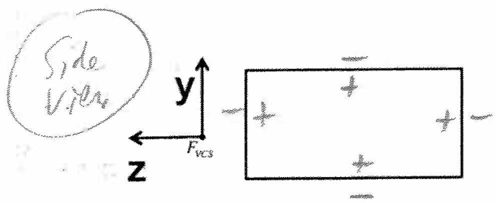
Plane equations $F(x,y,z) > 0$: "inside"

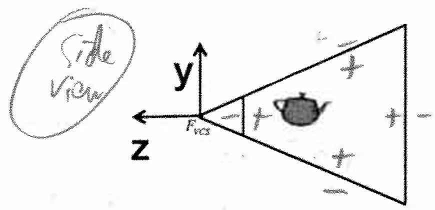
Othographic View Volume

```
left:     x - left = 0
right:  -x + right = 0
bottom:  y - bottom = 0
top:    -y + top = 0
front:  -z - near = 0
back:     z + far = 0
```

Perspective View Volume

```
left:     x + left*z/near = 0
right:   -x - right*z/near = 0
top:     -y - top*z/near = 0
bottom:   y + bottom*z/near = 0
front:   -z - near = 0
back:     z + far = 0
```





Note: clipping in VCS works fine, but the actual plane equations depend on the view volume parameters.
Why not just clip to ±1 for x,y,z in NDCS?

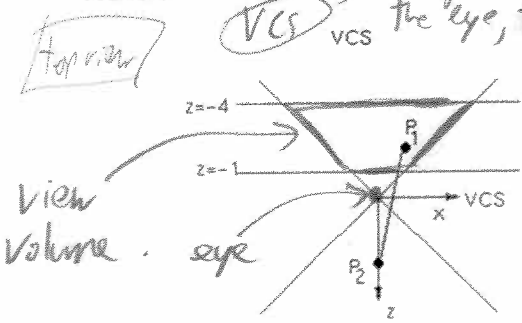
Clipping in NDCS (?)

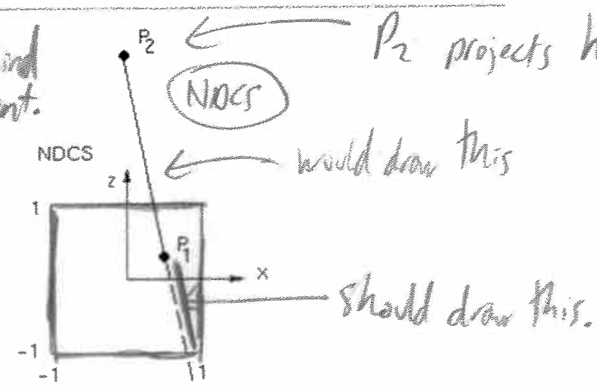
(no!)

+ canonical plane equations
− problematic with segments that cross z=0 plane.

NDCS

Top view, line goes from P_2, behind the eye, to P_1, in front.





$$\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & -5/3 & -8/3 \\ & & -1 & \end{bmatrix}$$

	P_1	P_2
VCS	(1, 0, -2)	(0,0,2)
CCS	(1, 0, 2/3, 2)	(0,0,-6,-2)
NDCS	(1/2, 0, 1/3)	(0, 0, 3)

Clipping.

Clipping in CCS

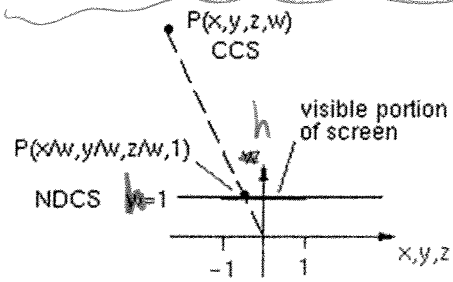
$$P_{VCS} \rightarrow \boxed{M_{proj}} \rightarrow P_{CCS} \rightarrow \boxed{/h} \rightarrow P_{NDCS}$$

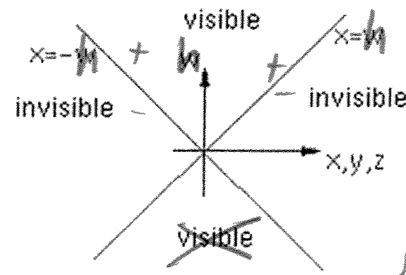
Works! Usually where clipping is done.

NDCS: $-1 \le \frac{x_{ccs}}{h_{ccs}} \le 1$

CCS:

$-h_{ccs} \le x_{ccs} \le +h_{ccs}$ ← use this instead.





h<0 means point is behind the eye.

canonical plane equations:

left: $x + h = F(P_{ccs})$

right: $-x + h = F(P_{ccs})$

bottom: $y + h = F(P_{ccs})$

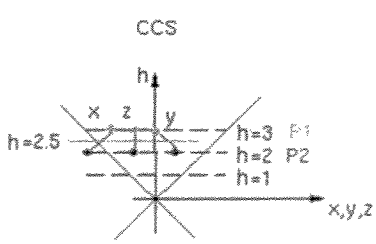
top: $-y + h = F(P_{ccs})$

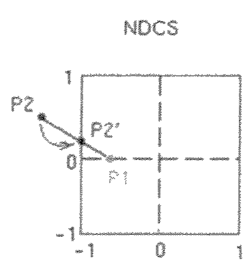
near: $z + h = F(P_{ccs})$

far: $-z + h = F(P_{ccs})$.

Clipping in CCS

example (skip)





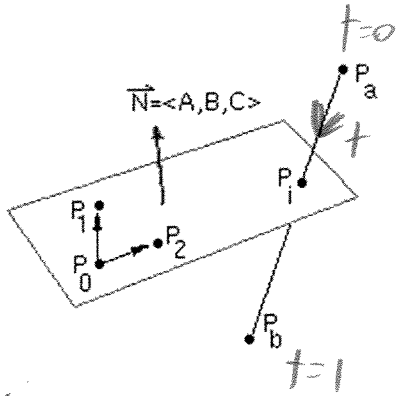
unclipped CCS
P1(-2,0,-1,3)
P2(-3,1,-1,2)

clipped CCS
P1(-2,0,-1,3)
P2(-2.5,0.5,-1,2.5)

unclipped NDCS
P1(-0.67,0,-0.33)
P2(-1.5,0.5,-0.5)

clipped NDCS
P1(-0.67,0,-0.33)
P2(-1,0.2,-0.4)

Line-Plane intersection



Computing a normal to a plane:

$$\vec{N} = (P_2 - P_0) \times (P_1 - P_0)$$

Plane equation:

$$Ax + By + Cz + D = 0$$

$$\langle A, B, C \rangle \cdot \langle x, y, z \rangle + D = 0.$$

$$N \cdot P + D = 0$$

To solve for D; substitute any point:

e.g. $N \cdot P_1 + D = 0 \Rightarrow D = -N \cdot P_1$

Line eq'n:

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

Substitute into plane eq'n:

$$N \cdot (P_a + t(P_b - P_a)) + D = 0$$

$$N \cdot P_a + t(N \cdot P_b - N \cdot P_a) + D = 0$$

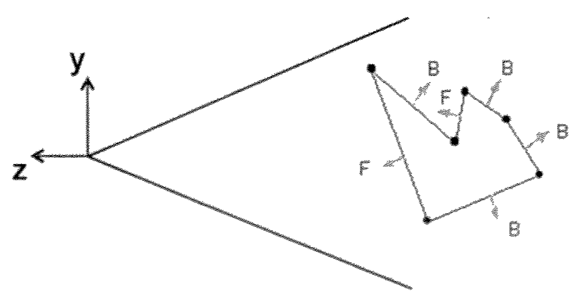
$$t = \frac{-N \cdot P_a - D}{N \cdot P_b - N \cdot P_a} = \frac{-F(P_a)}{F(P_b) - F(P_a)}$$

$\Rightarrow$ to find intersection point

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

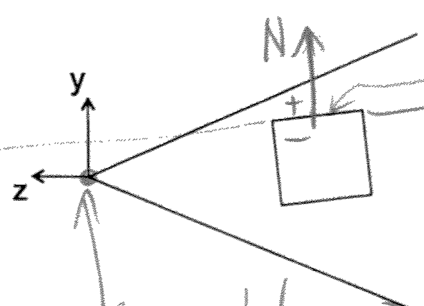
↑ now known.

Backface Culling in VCS



Idea: cull if $N_z < 0$

Works, but it will miss culling some faces. e.g.



eye is below $\Rightarrow$ cull!

Better: cull if the eye is below plane

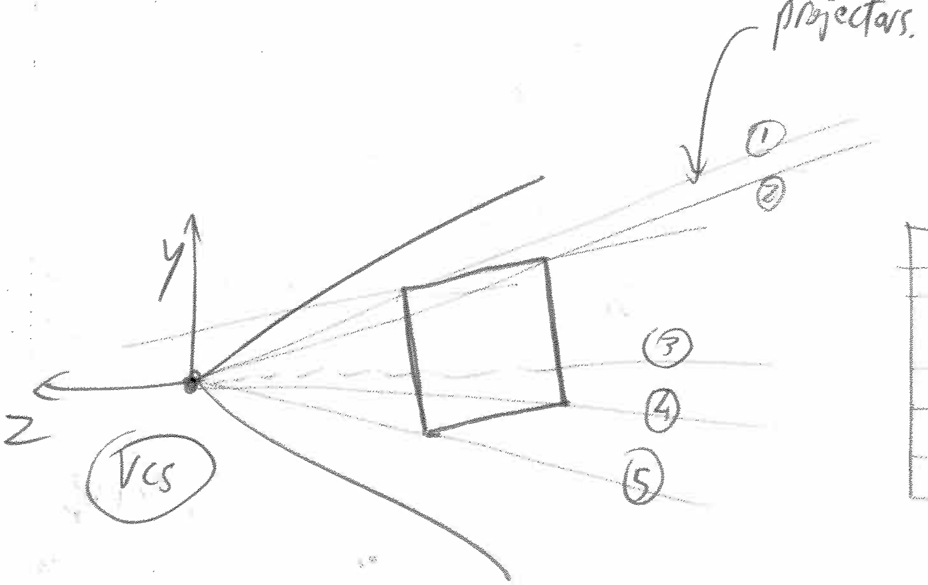
Math: $N \cdot P + D = F(x, y, z)$

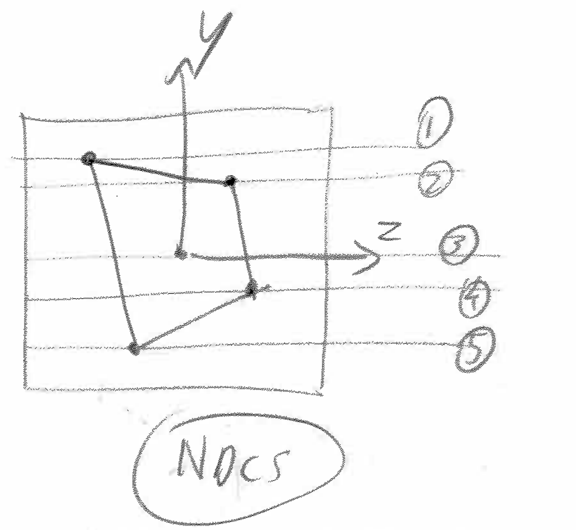
where $D = -N \cdot P_1$

$P_{eye} = (0, 0, 0)$

$\Rightarrow -N \cdot P_1 < 0$? cull!

Backface Culling in NDCS

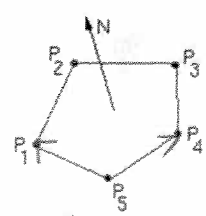




cull if eye is below plane.

cull if the normal points away, i.e., $N_z > 0$.

Computing Surface Normals



Method 1 Take cross product

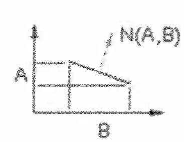
$$\vec{N} = (P_4 - P_5) \times (P_1 - P_5)$$

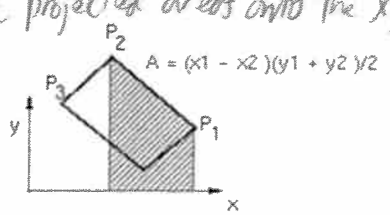
Usually rely on CCW "winding order", i.e., vertices listed in CCW order when seen from above.

Or, normals may be given explicitly.

ok to skip.

Method 2 Use projected areas onto the xy, yz, xz planes.





$$N_x = \sum_{i=1}^{n}(y_i - y_j)(z_i + z_j)$$
$$N_y = \sum_{i=1}^{n}(z_i - z_j)(x_i + x_j)$$
$$N_z = \sum_{i=1}^{n}(x_i - x_j)(y_i + y_j)$$

Transforming Normals

rotate, scale / translate

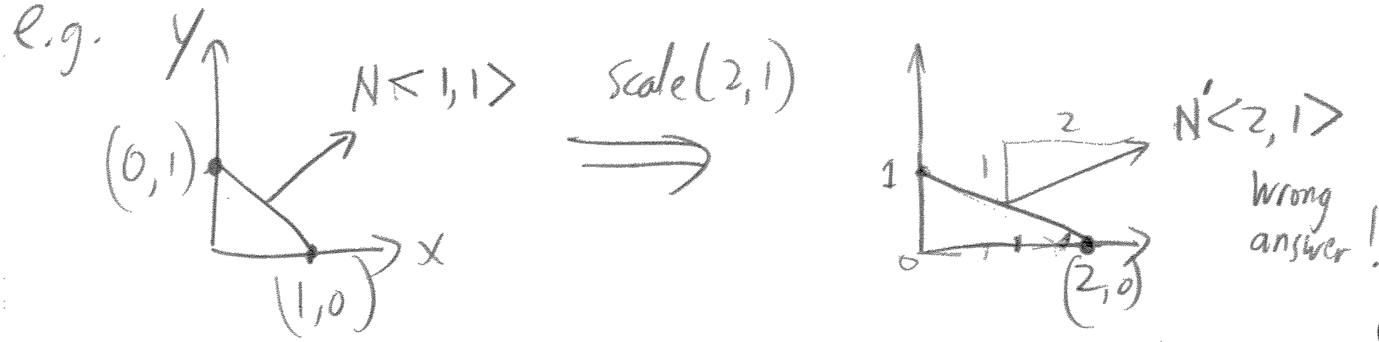
Using h=0

$$\begin{bmatrix} i & j & k & t \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} N_x \\ N_y \\ N_z \\ 0 \end{bmatrix}$$

← skip the translation.

Problem with non-uniform scaling:

e.g.

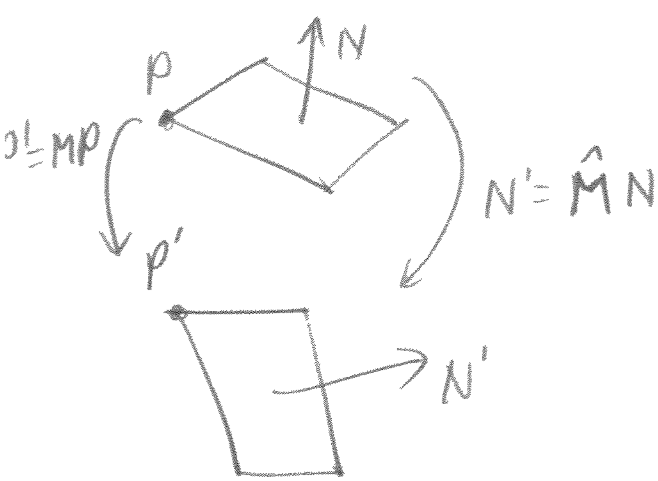


Transforming Normals

develop a normal transformation matrix:

Consider a plane: $Ax + By + Cz + D = 0$

$$\begin{bmatrix} A & B & C & D \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0$$



write this as

$N^T p = 0$

$N'^T p' = 0$

$(\hat{M} N)^T (M p) = 0$

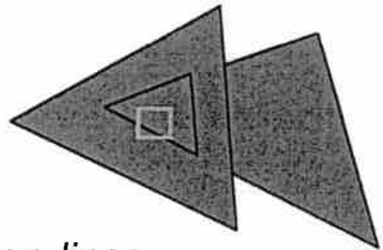
$N^T \hat{M}^T M p = 0$

→ $\hat{M}^T M = I$

$\hat{M} = (M^{-1})$

Occlusion

view occluded by objects in front of a given pixel or polygon ?



- image space algorithms:
  - *operate on pixels or scan-lines*
  - *visibility resolved to the precision of the display*
  - *e.g.: Z-buffer*
- object space algorithms:
  - *explicitly compute visible portions of polygons*
  - *painter's algorithm: depth-sorting, BSP trees*

Z-buffer

store (r,g,b,z) for each pixel

Most Commonly Use

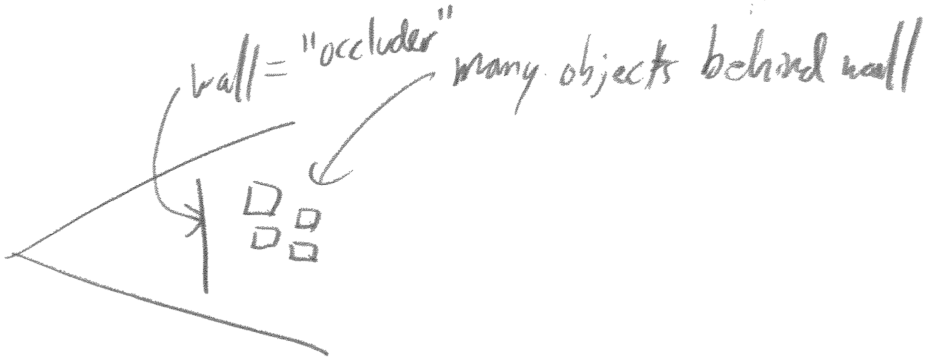
$$Z_{DCS} = \frac{Z_{NDCS}+1}{2} \in [0,1]$$

near far

```
for all i,j {
 Depth[i,j] = MAX_DEPTH
 Image[i,j] = BACKGROUND_COLOUR
}
for all polygons P {
  project vertices into screen-space, i.e., DCS
  for all pixels in P {
    if (Z_pixel < Depth[i,j]) {    // closer?
      Image[i,j] = C_pixel  // overwrite pixel
      Depth[i,j] = Z_pixel  // overwrite z
    }
  }
}
```

Z-buffer

- hardware support
- extra memory
- jaggies, i.e., steps along intersections
- poor performance for high depth complexity scenes;
  - use occlusion culling to mitigate this

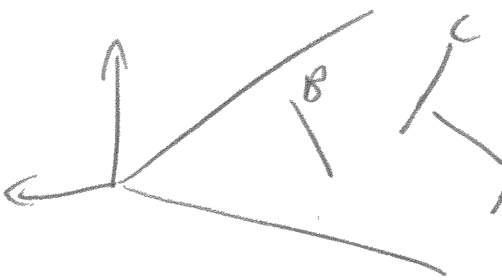


Occlusion Culling

- occlusion queries
  - virtual render of bounding box : test to see if any pixels "would have passed z-buffer test" for the bounding box.
- precomputed visibility tables
  - *store a list of visible cells* → from a given room, which other rooms can be seen?
- horizon maps
  - *for terrain models*

Painter's Algorithm

- draw polygons from back-to-front, i.e., sorted by z e.g. draw A, C, B
- problems:
  1. Which z value to use?
  2. What about cyclic overlap?
  3. What about intersecting geometry?
- fix this by cutting polygons as needed, e.g. BSP trees. (see upcoming slides)



Binary Space Partition (BSP) trees

recursively divides 3D spaces into half-spaces

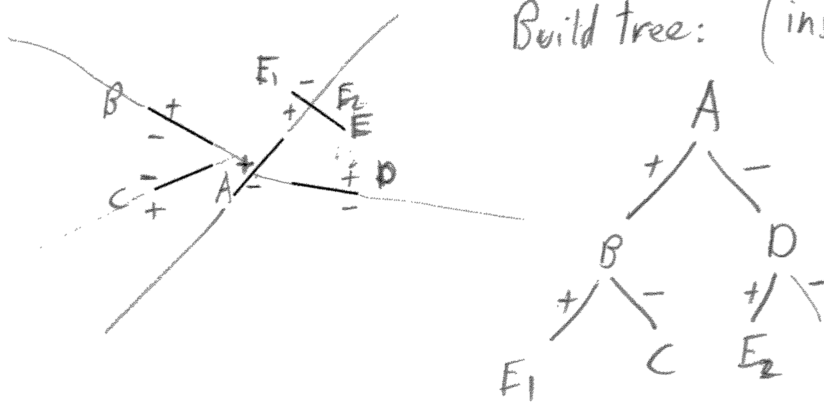
- object-space method
- cuts intersecting polygons
- produces a back-to-front ordering

Steps:

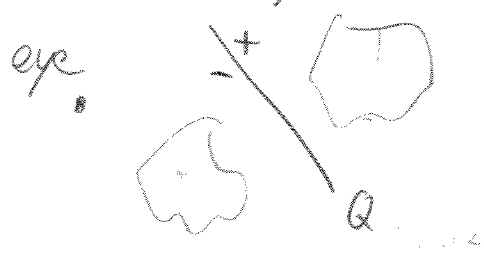
- build the BSP tree (do this only once)
- for each render, traverse the BSP in a view-dependent fashion

BSP trees (example)

Build tree: (insertion in alphabetical order)



Using a BSP tree



eye is in Q−, therefore render in order:

Q+, Q, Q−

back ⟶ front.

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) )
}
```

Using a BSP tree

producing a back-to-front ordering

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

Visibility in Practice: WebGL, OpenGL

Commonly supported by hardware & OpenGL / DirectX
- view volume culling (for triangles)
- view volume clipping
- backface culling
- z-buffer occlusion test

Software, i.e., on your own
- view volume culling (for objects)
- painter's algorithm & BSP trees
- occlusion culling

Raycasting and Raytracing

alternative to projective rendering

- for each pixel p
  - *construct ray r from eye through p*
  - *intersect r with all polygons or objects*
  - *color p according to closest surface*