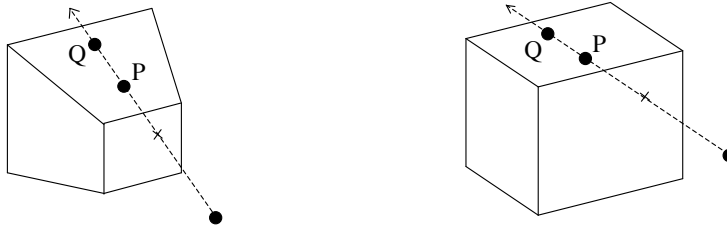


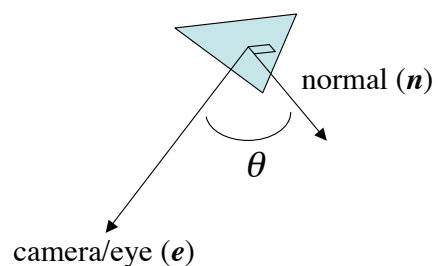
## Hidden Surface Removal

- Hidden surface removal (HSR) determines which polygons are nearest to the viewer at a given pixel
- Key criterion: a point P occludes a point Q (and thus Q is “hidden”) if P and Q lie on the same ray (line) from the camera or eye and P is between the camera location and Q
- Calculating this ray is tough with a frustum, but normalizing that frustum to a cube (which the projection matrix does) transforms the oblique rays to straightforward parallelism with the  $z$  axis:



## Backface Culling

- An excellent preprocessing step to speed up HSR is *backface culling* or *backface removal*. Backface culling checks the normal vector of every surface that we are rendering and throw away any surface whose vector points away from us (the viewer)
- This is easier (and faster) than it may seem, which is why it makes for an excellent preprocessing step:



## How to Cull Backfaces

- Note how a polygon is front-facing if the angle  $\theta$  between its normal and the vector toward the camera/viewer is between  $-90$  and  $90$  degrees. In other words, this means:

$$\cos \theta \geq 0$$

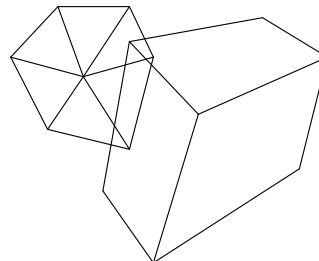
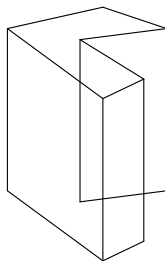
- Recall from our study of vectors that  $\cos \theta = (\mathbf{u} \cdot \mathbf{v}) / \|\mathbf{u}\| \|\mathbf{v}\|$ . So, given  $\mathbf{e}$  is the vector toward the camera and  $\mathbf{n}$  is the normal vector,  $\cos \theta \geq 0$  is the same as saying:

$$\mathbf{e} \cdot \mathbf{n} \geq 0$$

- But, if we perform this calculation *after* transforming to normalized device coordinates (our  $2 \times 2 \times 2$  cube),  $\mathbf{e}$  is merely  $\langle 0, 0, 1, 0 \rangle$  in homogeneous coordinates
- Thus, backface culling is a matter of checking if the  $z$  component of  $\mathbf{n}$  is greater than or equal to zero after NDC transformation!

## Backface Culling == HSR When...

- If everything we are only rendering a single, convex polyhedron, then backface culling is equivalent to HSR
- Backface culling is a straightforward switch in OpenGL — turn it on or off using `GL_CULL_FACE`
- If our polyhedron is not convex, or if there is more than one polyhedron involved, we need to do more work:



## HSR Algorithms

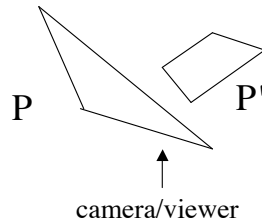
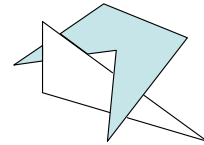
- A number of HSR algorithms exist. There is no absolute “best” algorithm, although due to the state of computing today, there *is* a “winner” in the HSR algorithm sweepstakes
- The algorithms trade off on the following:
  - Memory required
  - Accuracy vs. speed
  - Effect of increasing scene complexity on performance
  - Hardware capabilities/limitations
- Stop for a moment and think...what type of algorithm is likely the “choice algorithm” in this day and age?

## Depth Sorting

- By Newell, Newell, & Sancha, 1972
- Paint each polygon in the scene in order, from the most distant to the nearest
- A “painter’s algorithm” that “naturally” performs HSR by progressively painting over the farthest polygons
- Two primary steps:
  - Sort the polygons in *occlusion-compatible order*. That is, a sequence  $P_1, P_2, P_3, \dots, P_n$  such that for any polygon  $P_i, 1 \leq i \leq n, P_i$  hides (occludes) polygons  $P_{i+1}$  to  $P_n$
  - Scan convert (paint) each polygon from  $P_n$  down to  $P_1$

# Depth Sorting Implementation

- The sort's the thing!
  - Useful preprocess: decompose polygons into triangles to simplify depth comparisons (*tessellation*) — because of polygons arranged like...
- A polygon sort algorithm would be:
  - Determine a maximum  $z$  for each polygon  $P$
  - Sort the polygons according to this maximum  $z$
  - For each polygon, make sure that all of the polygons that are “behind” it according to maximum  $z$  are indeed hidden — we need to do this because sometimes maximum  $z$  doesn't imply occlusion:



Note how P's maximum  $z$  is greater than P' but it is actually P that is occluding P'

## Overlap Testing for Depth Sort

The catch in this algorithm is indeed in the overlap testing component — how do we do this accurately and quickly? There are 5 cases to test — if *any* succeeds, then we can move on. Otherwise, we swap polygons:

1. minimax depth test: minimum  $z$  of one polygon is less than the maximum  $z$  of the other polygon
2. minimax  $x$ - $y$  test: polygons do not overlap in the  $x$  and  $y$  directions
3. behind-plane test: all vertices of one polygon are behind the plane defined by the other polygon (derive plane equation to do this)
4. in-front-of-plane test: all vertices of one polygon are in front of the plane defined by the other polygon (plane equation again)
5. full overlap test: check for overlap in either the  $x$  or  $y$  directions and determine the respective  $z$  values at the overlapping area

## Warnock Algorithm

Takes advantage of *area coherence*: divide the display area into successively smaller rectangles until the entire rectangle can be filled with a single color:

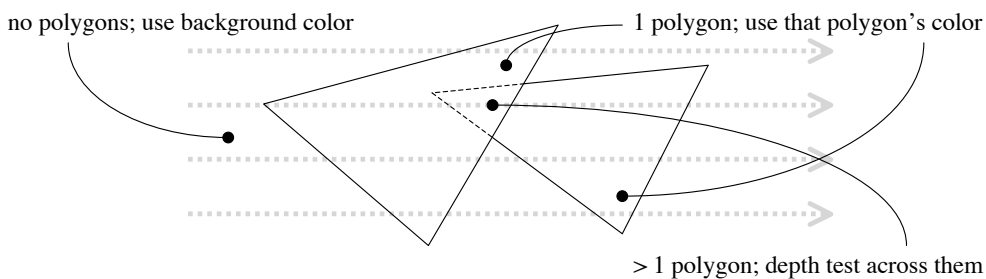
```
doWarnock(x1, y1, x2, y2) {
  if rectangle is a pixel then
    if no polygons map to this pixel then
      set pixel to background color
    else
      set pixel to the color of the polygon closest to this pixel
  else
    if no polygons overlap this rectangle then
      set rectangle to background color
    else if polygon(s) completely overlap this rectangle then
      set rectangle to the color of the closest of these polygons
    else
      doWarnock(x1, y1, (x1 + x2) / 2, (y1 + y2) / 2)
      doWarnock(x1, (y1 + y2) / 2, (x1 + x2) / 2, y2)
      doWarnock((x1 + x2) / 2, y1, x2, (y1 + y2) / 2)
      doWarnock((x1 + x2) / 2, (y1 + y2) / 2, x2, y2)
    end if
  end if
end if
```

## Scan-Line Algorithm

- a.k.a. “scan coherence” algorithm — not to be confused with  $z$ -buffer on one scan line (we’ll talk about  $z$ -buffer next)
- Most efficient algorithm prior to lower-cost memory and specialized graphics hardware
- This algorithm also benefits from sorted surfaces and tessellation

## Scan-Line Algorithm Implementation

- Traverse the display device one scan line at a time, left-to-right, top-to-bottom
- Check polygon list to see which ones intersect the current pixel
  - Once we are “in” a polygon, we know that we will stay “in” until we hit another one of that polygon’s edges (this is the core of “scan coherence”)
- When the list of “in” polygons  $> 1$ , we perform a depth check, and choose the color for the polygon that “wins” that check



## Incremental $z$ Calculation

- Two observations/assumptions:
  - Polygons are flat — e.g. they lie on a plane
  - As we traverse a polygon one scan line at a time, the  $z$  value changes at a constant rate (since the polygon is flat)
- Thus, we can have *incremental* calculation of  $z$ , which is faster than passing through the projection matrix (another use of the coherence concept):
  - Calculate the “slope” of the polygon upon entry (based on the plane’s equation  $Ax + By + Cz + D = 0$  — note how  $\langle A, B, C, D \rangle$  is the normal vector in homogeneous coordinates)
  - Calculate the initial value of  $z_i$  at the entry point:
 
$$z_i = (-Ax_i - By_i - D) / C$$
  - Since we are scanning along the  $x$ -axis, the only difference between the entry point and the next point is  $+1$  along that axis:
 
$$z_{i+1} = (-A(x_i + 1) - By_i - D) / C = (-Ax_i - By_i - D - A) / C$$

$$z_{i+1} = z_i - (A / C)$$
  - $A / C$  is constant per polygon — so, calculating the next  $z$  is a single addition!

## Z-Buffer Algorithm

- Very general and powerful technique — works for all kinds of polygons and all forms of occlusion (including cyclical)
- The key trick — maintain a separate, parallel buffer that stores the depth ( $z$  component) of the closest polygon that is currently at that point (thus, the synonymous name, “depth buffer”)

red	blue	red	brown	black
white	green	red	red	black
white	green	red	brown	black
blue	blue	blue	blue	blue
green	white	red	red	brown

frame buffer — display device

0.5	0.8	0.5	0.2	1.0
0.35	0.44	0.5	0.5	1.0
0.35	0.44	0.5	0.2	1.0
0.8	0.8	0.8	0.8	0.8
0.44	0.35	0.5	0.5	0.2

$z$  or depth buffer

## Z-Buffer Implementation

```

clear frame buffer viewport to background color
clear depth buffer zbuffer to 1.0
for each polygon P
  for each pixel  $(x_{ndc}, y_{ndc})$  to which P projects
    if  $z_{ndc} < zbuffer[x_{ndc}, y_{ndc}]$  then
       $zbuffer[x_{ndc}, y_{ndc}] := z_{ndc}$ 
       $viewport[x_{ndc}, y_{ndc}] := \text{color of P at } (x_{ndc}, y_{ndc})$ 
    endif
  endfor
endfor

```

- Note how the initial value of the depth buffer is 1.0 because in NDC, that is the maximum  $z$  value
- Exact coordinates in viewport are derived from the NDC-to-viewport mapping (e.g. conversion of  $-1 \dots 1$  to *width* and *height*)

## Z-Buffer Implementation Notes

- Because  $z$ -buffer also calculates the  $z$  per polygon per scan line, we can take advantage of the same incremental  $z$  calculation optimization used by the scan-line algorithm
- But take note that  $z$ -buffer uses significantly more memory than the other algorithms — it needs a buffer of the same width/height as the viewport!
  - Actual memory used would be  $width \times height \times sizeof( real )$
  - These days,  $sizeof( real )$  is around 4 to 8 bytes — so note how a depth buffer is *double or greater* than the size of the frame buffer itself
  - Thus, a typical, modern-day, 3D rendering environment (such as the one provided by OpenGL), immediately requires a minimum of *thrice* the desired display resolution: 2 swappable buffers for animation, and a third buffer for depth
  - See why it's easy to outgrow video memory now?

## HSR Algorithm Comparisons

- Published by Sutherland, Sproull, and Schumaker in *ACM Computing Surveys*, March 1974
- Yes, 1974

<i>Algorithm</i>	<i>Number of Faces/Polygons</i>			
	<i>100</i>	<i>2500</i>	<i>60000</i>	
<i>Depth Sort</i>	1	10	507	<i>T</i>
<i>Scan-Line</i>	5	21	100	<i>I</i>
<i>Warnock</i>	11	64	307	<i>M</i>
<i>z-Buffer</i>	54	54	54	<i>E</i>

- Depth sort is quickest for a small number polygons, but slows down in proportion to the number of polygons to process
- Scan-line and Warnock are also proportional to the number of polygons, but don't degrade as much as depth sort
- $z$ -buffer is virtually independent of the number of polygons; more polygons means smaller regions to test (because we are constrained by the frame buffer's size), so the total number of calculations tends to even out

## OpenGL Uses Z-Buffer

- At this point, it is probably quite clear that with today's technology,  $z$ -buffer is the way to go for HSR, and thus it is OpenGL's HSR algorithm
- This revelation should now explain code snippets such as:

```
glutInitDisplayMode( ... | ... | GLUT_DEPTH);  
// --- Initializes depth buffer.
```

```
glClear(... | ... | GL_DEPTH_BUFFER_BIT);  
// --- Sets depth buffer to 1.0.
```

```
glEnable(GL_DEPTH_TEST);  
// --- Activates depth buffer HSR.
```