Visible Surface/Line Determination

(Hidden surface/line elimination)

Goal: Given a set of 3D objects and a viewing specification, determine which lines or surfaces of the objects are visible.

Two approaches to solution: image space
object space

Image space approach ← first written for raster devices

for each pixel in image {
1) determine the object closest to the viewer that is pierced by the projector thru the pixel
2) draw pixel in the appropriate color
}

Implemented in screen coordinate system to the precision of display.

Brute-fore effort $\propto np$

n: #objects, p: # pixels

Object space approach ← first written for vector graphics system

for each object in world {
1) determine those parts of the object whose view is unobstructed by other parts of it or any other object.
2) draw those parts in the appropriate color
}

Performed at precision with which each object is defined. If size of finished image changes, only redo (2)

Effort $\propto n^2$

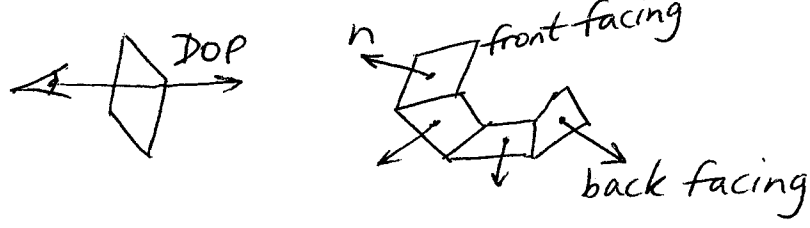
Algorithms often combine object- and image-space approaches.
To reduce complexity, we often use coherency to reduce the number of tests.

Back-face Culling

For objects approximated by solid polyhedron, we can avoid drawing back-facing polygons (those that point away from eye).

↑ Dir of proj (DOP)
OR ctr of proj (COP)

Assume: surface normals point out of polyhedron, and interior is not exposed by front clipping plane.



$DOP \cdot n > 0 \Rightarrow$ back-facing polygon
$DOP \cdot n = 0 \Rightarrow$ polygon is being viewed on edge
$DOP \cdot n < 0 \Rightarrow$ front-facing polygon

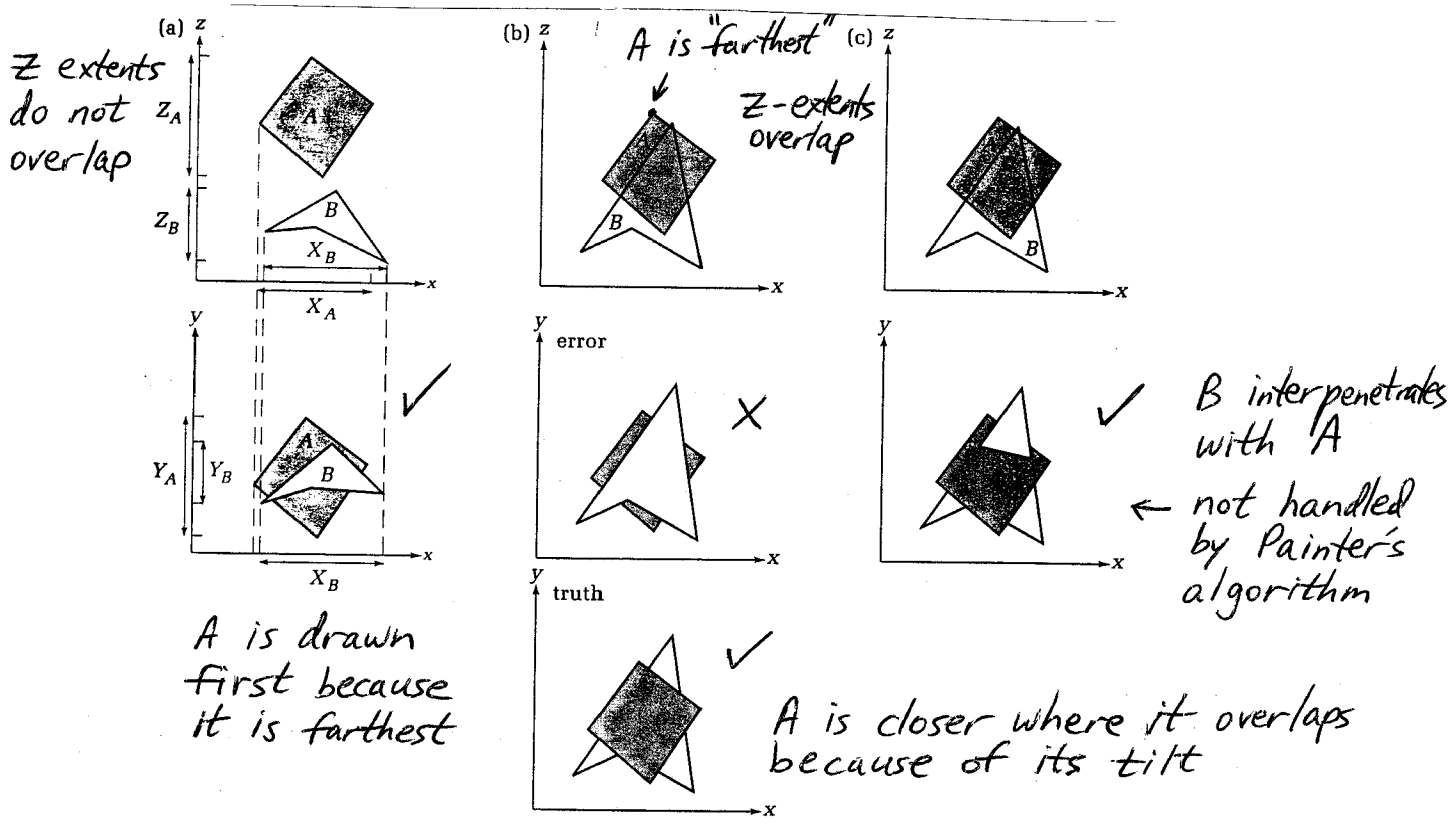
Do not draw edges of back-facing polygons. This typically halves the number of polygons that need consideration during HSR
If scene consists of one convex polyhedron (cube), back-face culling eliminates all hidden surfaces.

Painter's Algorithm

Sort faces by their farthest value
Each face is painted into frame buffer from farthest to closest; its color paints over whatever was drawn there before.

Note: sorting is important in all hidden surface algorithms.

Usually produces bad results because objects overlap in z.



Problem with Painter's algorithm:
it is wrong to paint entire faces over one another based on a single criterion such as maximum depth. This doesn't permit for one face to interpenetrate another.

Depth Buffer (z-buffer) ← Catmull, 1974

Use a z-buffer Z to store the depth (distance from viewer) of each pixel in frame buffer F.

F — image
Z — pixel depths

← F and Z have identical dimensions

```
for(y=0; y<YMAX; y++){
    for(x=0; x<XMAX; x++) {
        writePixel(x, y, BACKGROUND_VALUE);
        writeZ(x, y, ZMAX);
                      ↑ largest z value
    }
}
```

↑
Initialize F and Z

```
for each polygon {
    for each pixel in polygon's projection {
        pz = polygon's z-value at (x,y);
        if (pz < readZ(x,y)) {
            writePixel(x, y, polygon's color at (x,y));
            writeZ(x,y, pz);
        }
    }
}
```

Note: no presorting necessary
no object-object comparisons

Each polygon may be scan converted in arbitrary order. We exploit planar polygons to compute z:

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

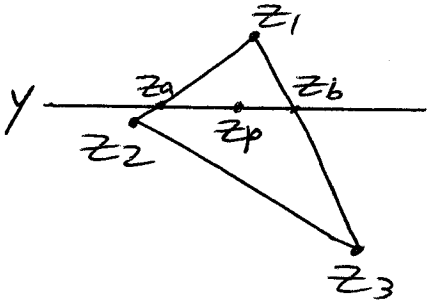
$$z = \frac{-D - Ax - By}{C}$$

Let $z_1 = z(x,y)$, then at $(x+\Delta x, y)$

$$z = z_1 - \frac{A}{C}(\Delta x)$$

Let $\Delta x = 1$ (one pixel over):

$$z(x+1, y) = z(x, y) - \frac{A}{C}$$



Interpolate z along polygon edges and scanlines

$$z_a = z_1 + (z_2 - z_1)\left(\frac{y - y_1}{y_2 - y_1}\right)$$

$$z_b = z_1 + (z_3 - z_1)\left(\frac{y - y_1}{y_3 - y_1}\right)$$

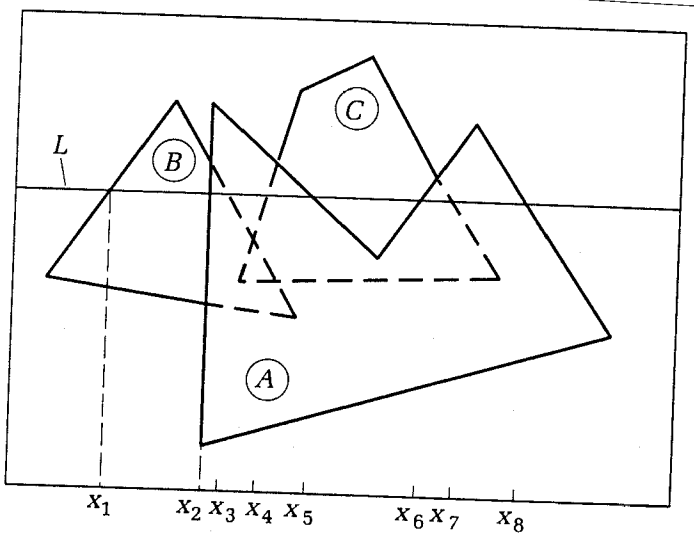
$$z_p = z_a + (z_b - z_a)\left(\frac{x_p - x_a}{x_b - x_a}\right)$$

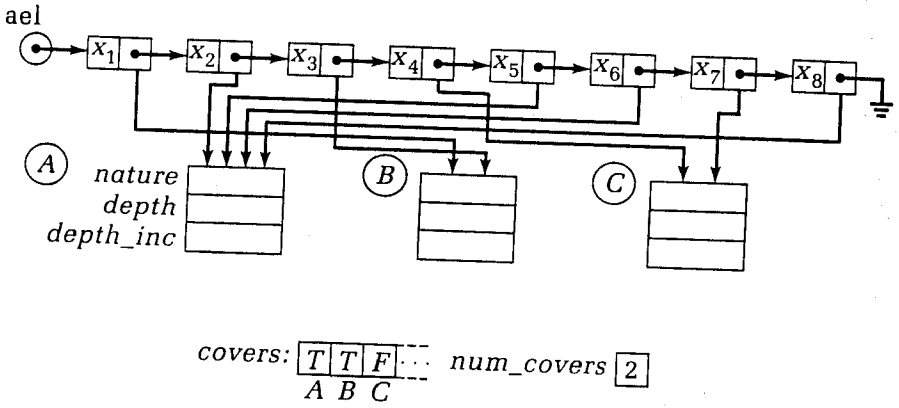
<u>Advantages:</u> z-buffer can be used for any object (not just polygons).
No explicit intersection algorithms needed
If memory is limited, one scanline z-buffer is possible.

<u>Disadvantages:</u> image-precision solution that is subject to aliasing (undersampling), especially at shared edges. Requires a lot of memory.

Scanline Algorithms

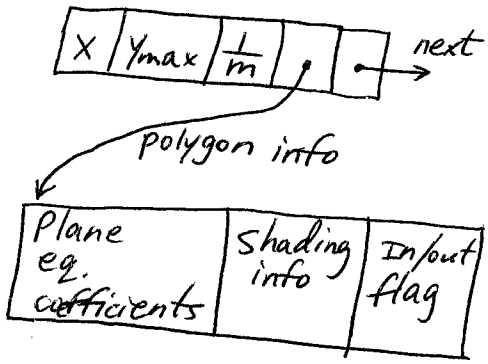
Extension of the polygon scan conversion algorithm for filling polygons.
Exploits scanline and edge coherence.
Introduce polygon table to contain info on: plane equation coefficients, color information for polygon, and an in-out flag.





covers: T T F ... num_covers 2
A B C

An edge structure may look like:



Area-Subdivision Algorithms: Warnock's Alg

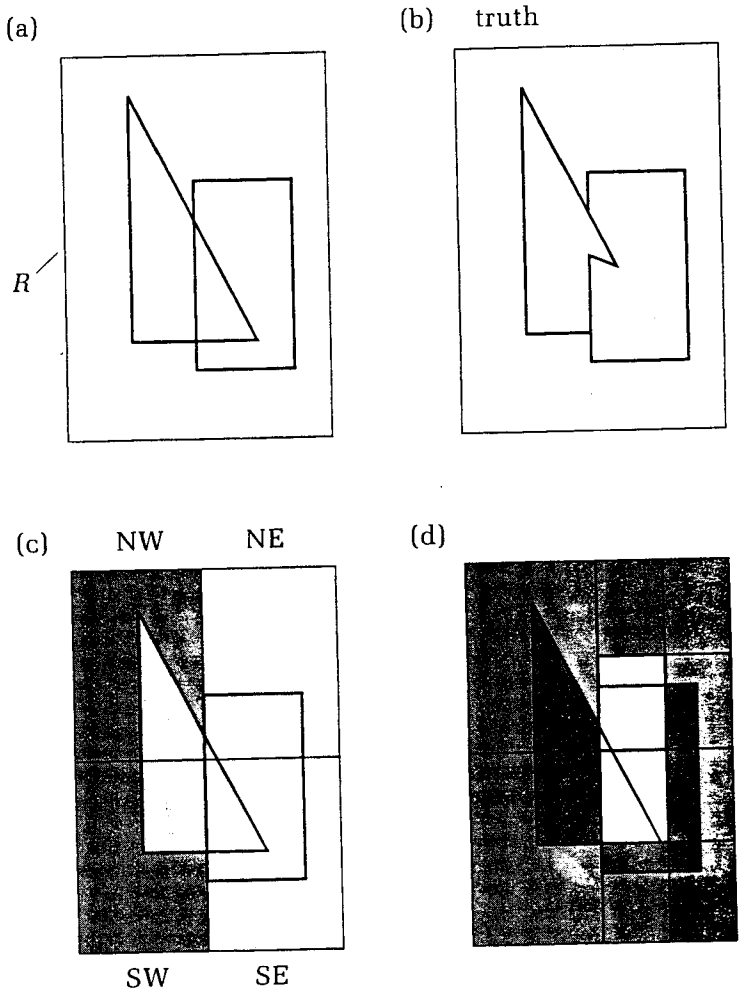
Scanline coherence → many neighboring pixels along a scanline share the same property of being covered or not covered by a face.

Area coherence → many pixels that are neighbors in x or y share the same property. (Symmetry between the x and y dirs).

Divide-and-conquer

Strategy: partition picture into subregions and test each subregion for visible surfaces. When a subregion is "simple" enough to be drawn in its entirety without further depth testing, the subdivision stops. Else subdivide further (until pixel size).

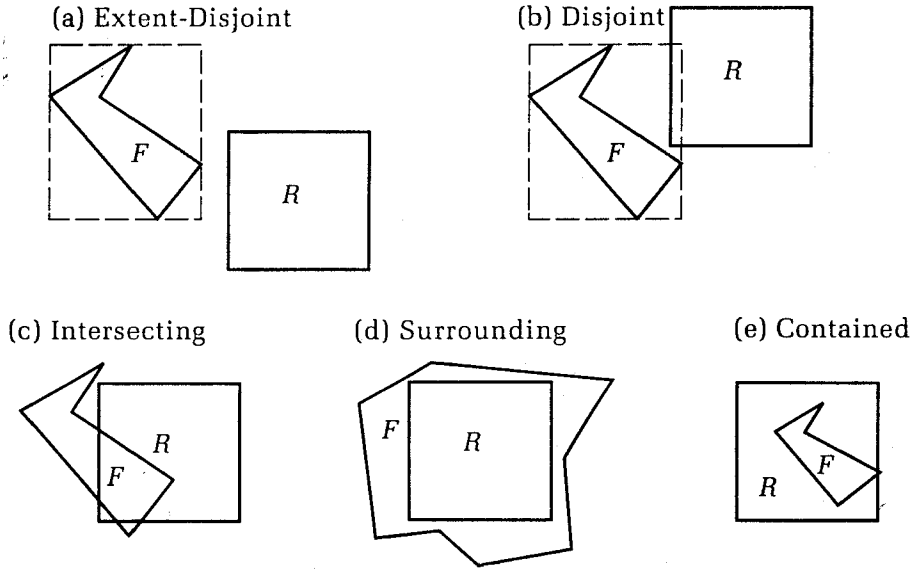
Pseudocode for Warnock's Algorithm




```
drawRegion(R)
 regionP R;
 {
     int size, easy;
     rectP NW, NE, SW, SE;
     faceP faces;

# pixels in R:
0, 1, or many → size = regionSize(R);
     if(size > 1) {
         easyRegion(R, &easy, faces);   ← main part of algorithm
         if(easy) drawFaces(faces);  /* region is simple */
         else {
             buildRegions(R, NW, NE, SW, SE);  /* subdivide */
             drawRegion(NW);
             drawRegion(NE);
             drawRegion(SW);
             drawRegion(SE);
         }
     } else if(size == 1) drawClosestFace(faces);
 }
```

easyRegion considers the following cases:



Def. #1: R is simple if at most one face is intersecting, surrounding, or contained

We may reduce the number of faces requiring testing by modifying argument list to easyRegion:

easyRegion(R, in_faces, &easy_faces)

↑ candidate faces

↑ all faces involved with R (subset of in_faces)

We may reduce subdivision by choosing a more complex definition for "simple"

Def. #2: R is simple as long as one face dominates (surrounds R + is everywhere closer) or at most one face intersects or is contained in R

↓ test depth at 4 corners of R