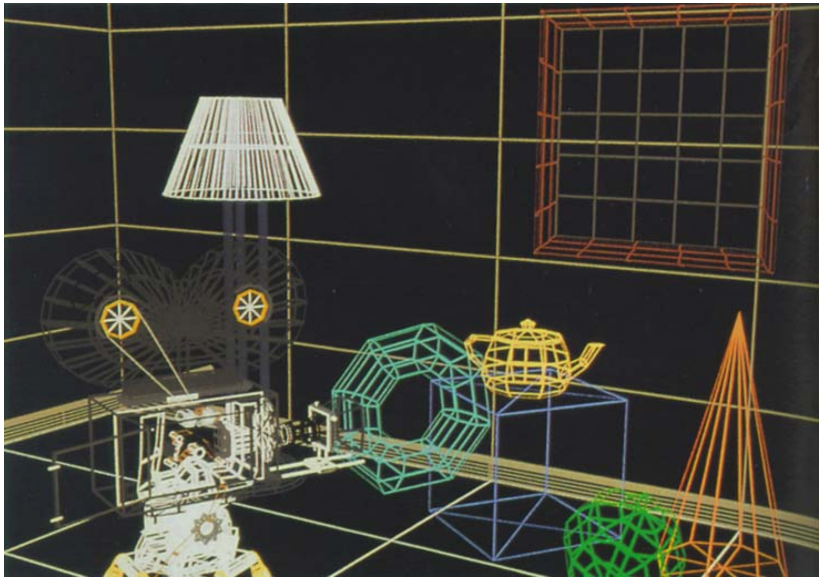
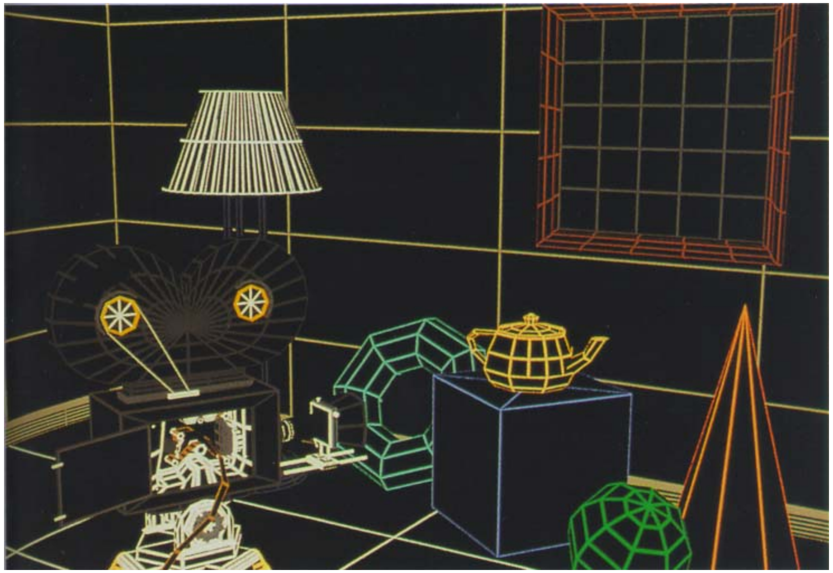
Visible Surface Determination

Foley & Van Dam, Chapter 15



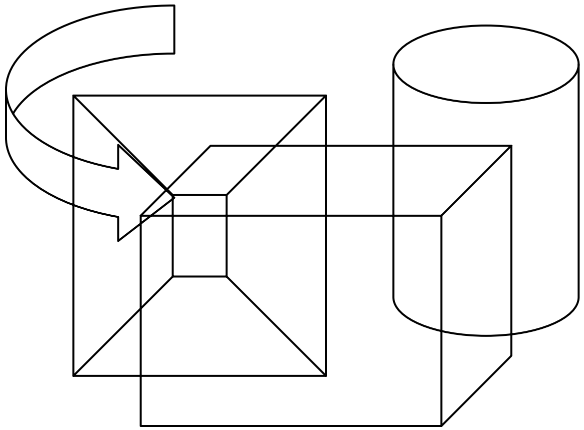


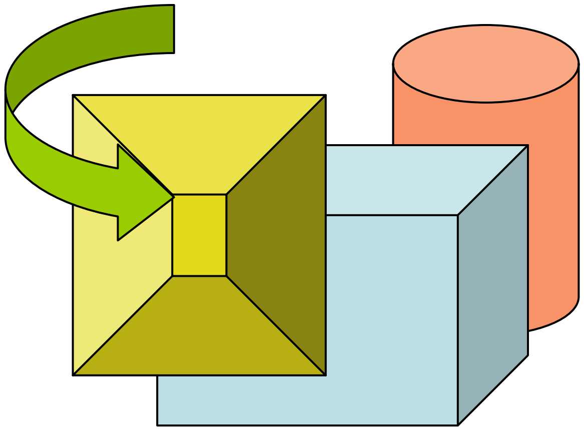
Visible Surface Determination

- Description of the problem
- Image vs. pixel precision algorithms
- Horizon line algorithm for a single valued function of two variables
- Back face detection
- Quantitative invisibility
- Z-buffer algorithm

Visible Surface Determination

- Given a set of 3D objects and a viewing specification, determine which lines or surfaces should be visible
- A surface might be occluded by other objects or by the same object (self occlusion)
- Two main classes of algorithms:
  - Image-precision: determine what is visible at each pixel
  - Object-precision: determine which parts of each object are visible





Object vs. Image Precision

- Assume a **p** pixels image and an **n** objects world:

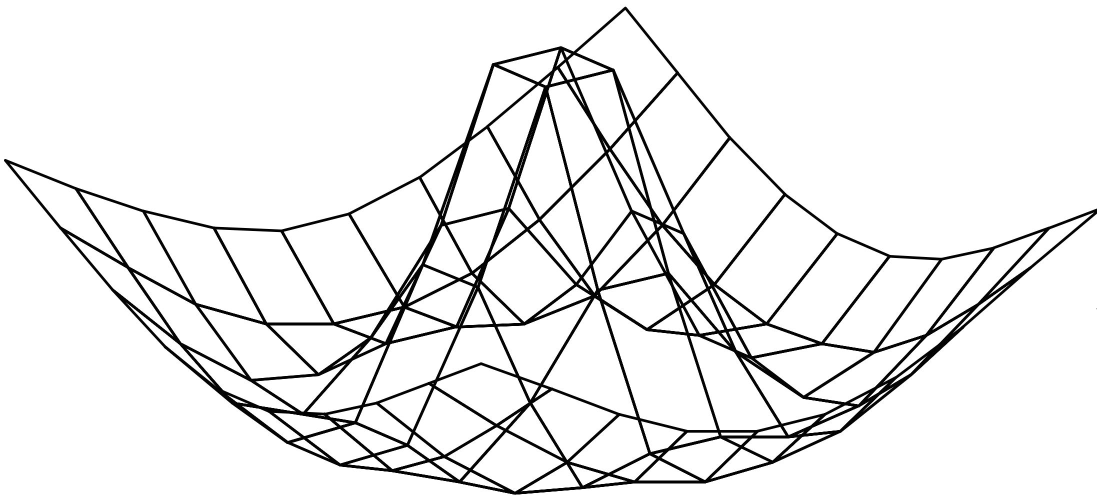
```
// Image or Pixel Precision – O(p) operations
for(each pixel) {
    determine the object closest to the viewer that is
       pierced by the projector through the pixel;
    draw the pixel in the appropriate color;
}

// Object Precision – O(n²) operations
for(each object) {
    determine those parts of the object whose view is
      unobstructed by other parts of it or any other object;
    draw those parts of appropriate color;
}
```

Operations used in object precision algorithms are typically more complex than operations used in pixel precision methods

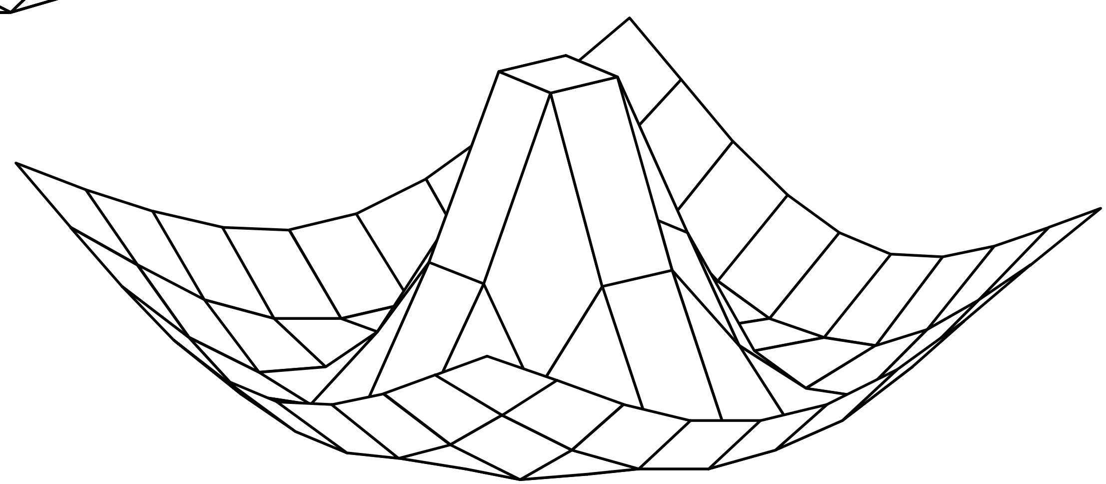
Coherence

- Most methods for visible surface determination take advantage of coherence features in the surface:
  - Object coherence
  - Face coherence
  - Edge coherence
  - Scan-line coherence
  - Depth coherence
  - Frame coherence

Single Valued Function of Two Variables



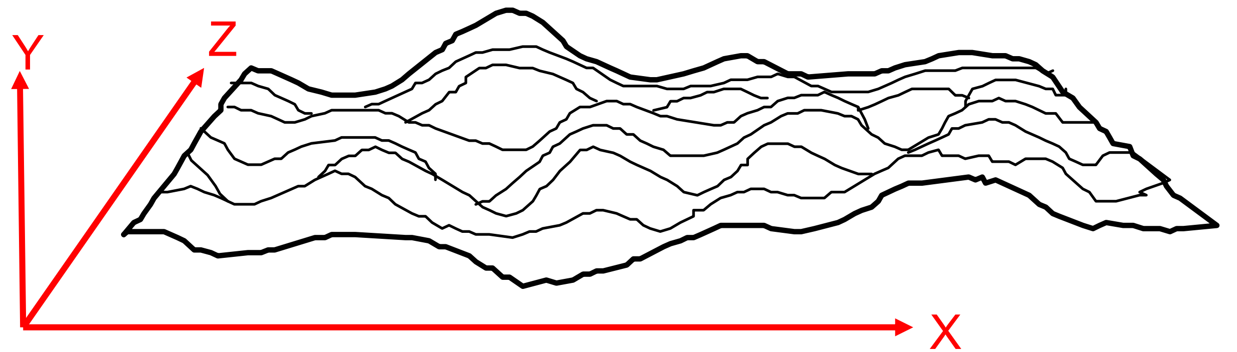
Without hidden lines removal



With hidden lines removal

Horizon Line Algorithm

- An implicit function y = f(x, z) represents as 2D array of x and z values in which each entry is a y-value
- Surface composed by polylines. Each polyline is constant in z



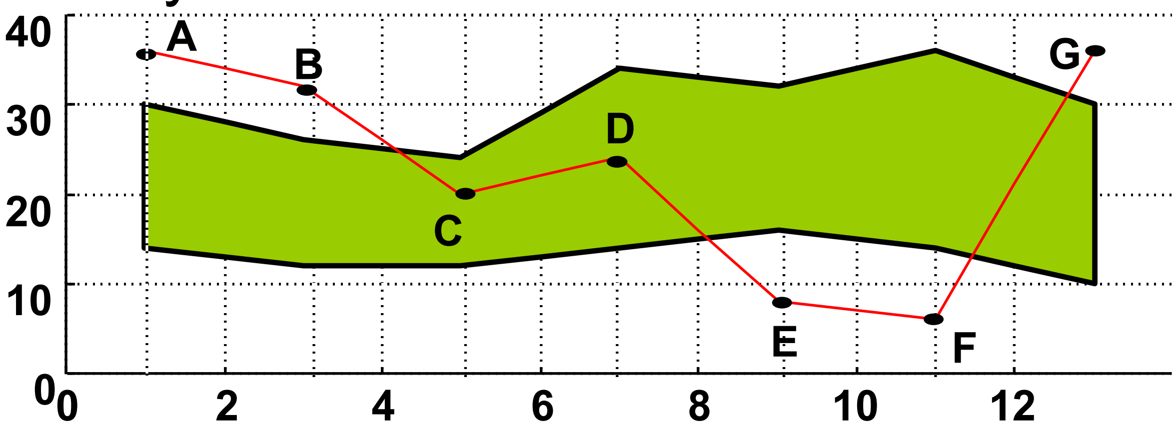
Algorithm:

– Draw polylines of constant z from front (near) to back (far)

– Draw only parts of polyline that are visible: i.e. above/below the silhouette (horizon)

Horizon Line Algorithm

- **Implementation**: Use two 1D arrays YMAX and YMIN (with 1 entry for each x). When drawing a polyline of constant z, for each x-value, test if above/below YMAX/YMIN (at x location) and update arrays



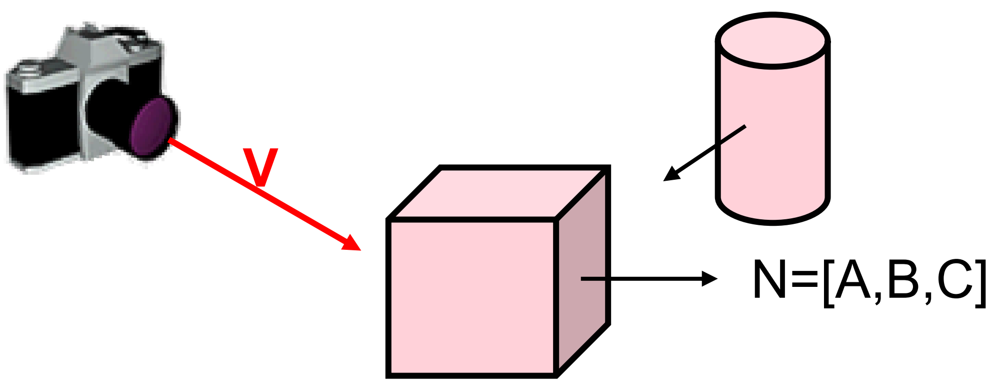
old YMAX	30	28	26	25	24	29	34	33	32	34	36	33	30
old YMIN	10	12	14	15	16	15	14	13	12	12	12	13	14
Polyline	36	34	32	26	20	22	24	16	8	7	6	21	36
	A		B		C		D		E		F		G
new YMAX	36	34	32	26	24	29	34	33	32	34	36	33	36
new YMIN	10	12	14	15	16	15	14	13	8	7	6	13	14

Horizon Line Algorithm

- **Characteristics**:
  - Applied in image space (image precision)
  - Limited to explicit functions only
  - Exploiting edge coherence
  - Applicable to free-form surfaces

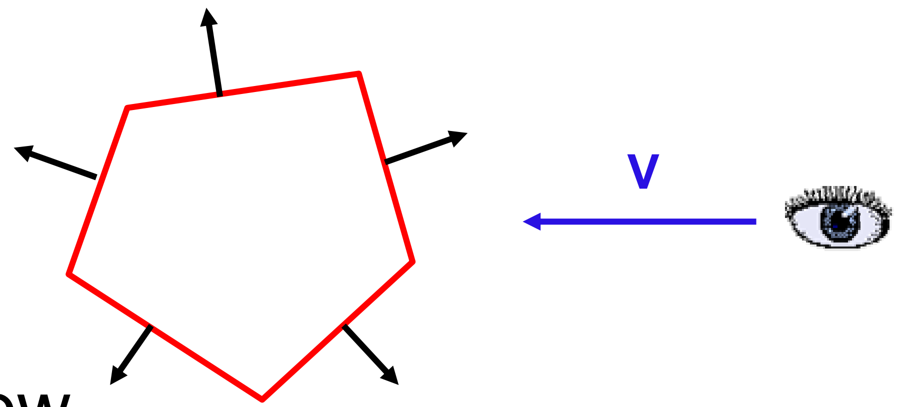
Back Face Detection

- **Observation**: In a volumetric object, we don't see the "back" faces of the object (self occlusion)
- **Reminder**:
  - Plane equation: $Ax+By+Cz+D=0$
  - $N=[A,B,C]^T$ is the plane normal
  - N points "outside"
- Back facing and front facing faces can be identified by using the sign of $V \bullet N$



Back Face Detection

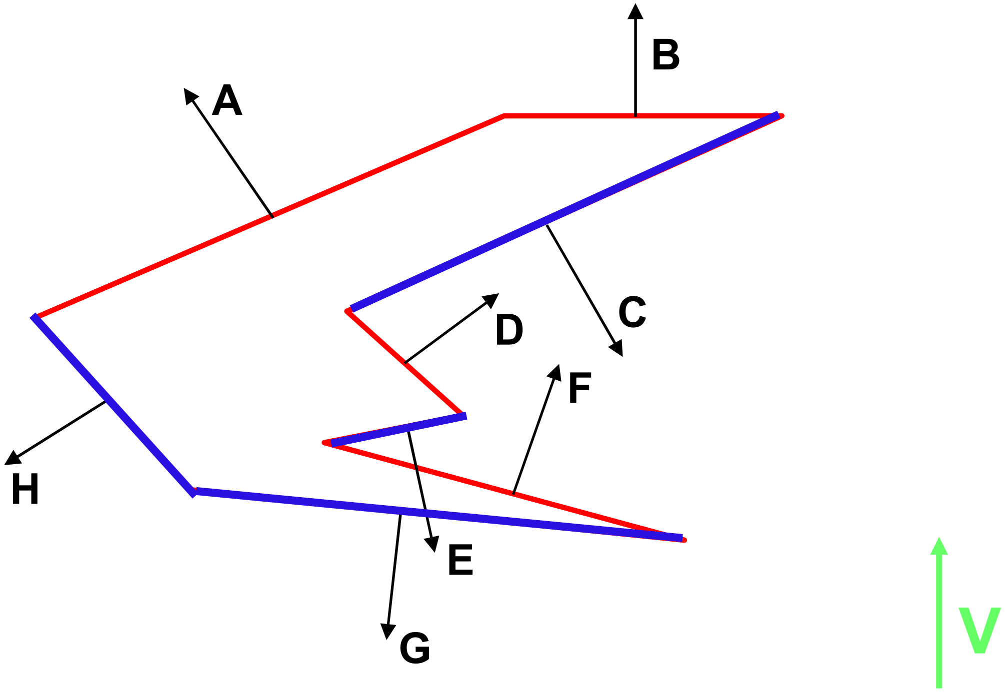
- Three possibilities:
  - V•N> 0   back face
  - V•N< 0   front face
  - V•N= 0   on line of view



- Back face detection is easily applied to convex polyhedral objects
- For convex objects, *back face detection* actually solves the *visible surfaces problem*
- In a general object, a front face can be visible, invisible, or partially visible

Back Face Detection

- Example



Back Face Polygons: A, B, D, F

Front Face Polygons: C, E, G, H

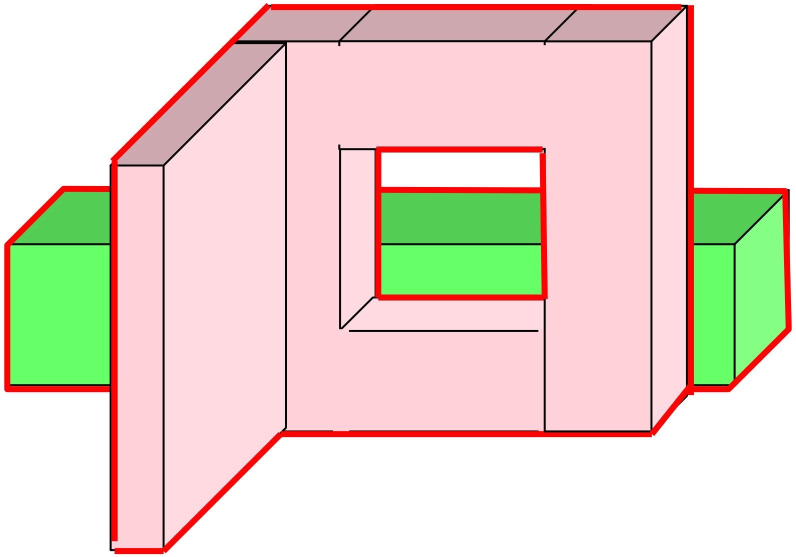
Quantitative Invisibility

- Proposed by Appel in 1967
- **Definitions**:
  - Every edge is associated to a non-negative value Q_v called quantitative invisibility
  - Q_v which corresponds to the number of times the edge is obscured
  - If $Q_v=0$ the edge is visible

Quantitative Invisibility

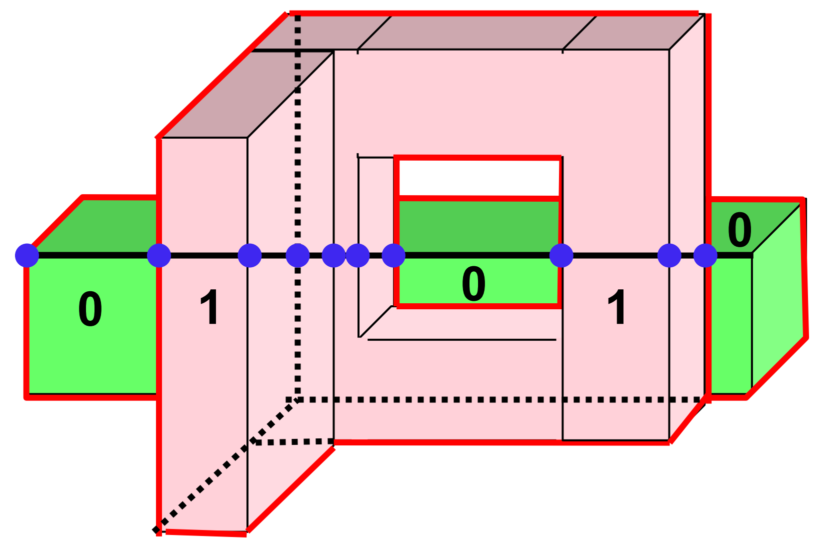
- **Definition**: An active edge is a silhouette edge, i.e an edge shared by back and front faces

- **Observations**:
  - The visibility of an edge can be changed only where it intersects another active edge in the viewing plane
  - If the edge does not intersect any active edge, its visibility is homogeneous



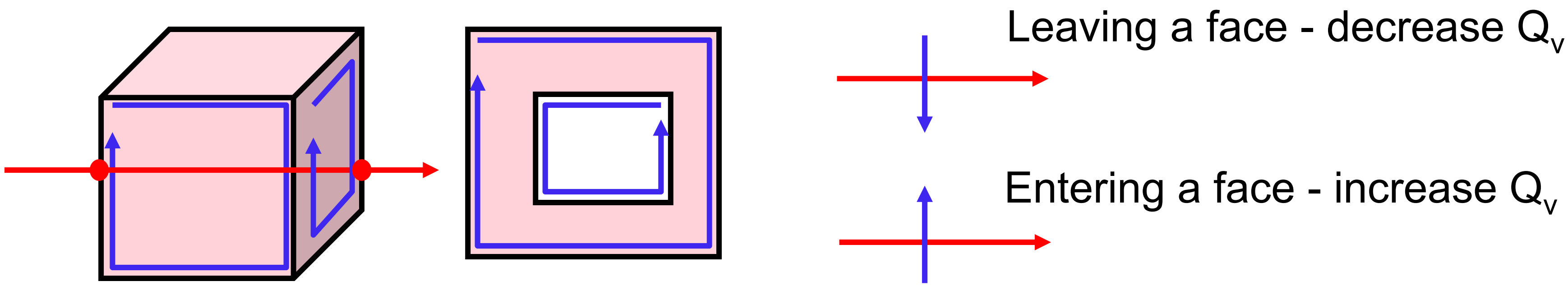
Quantitative Invisibility

- **Algorithm**:
  - Select a single point on line (seed) and test how many polygons obscure it (with a brute force algorithm)
  - Increment/decrement Q_v any time the line intersects an active edge, and the intersection is inside the view triangle
  - Propagate from the end point to a neighboring line
  - Fill the resulting polygons appropriately



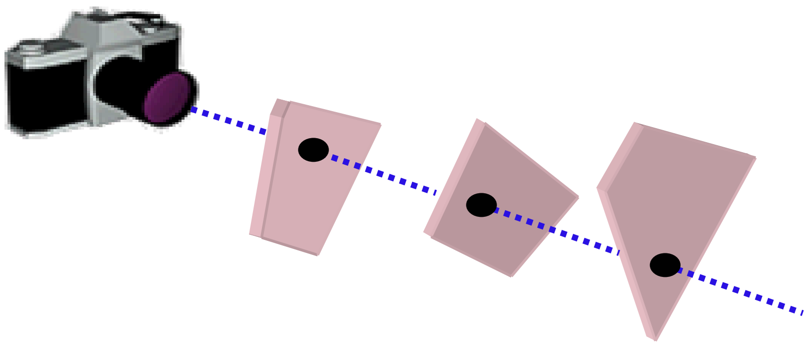
Quantitative Invisibility

- **Problem**: How do we know if the line “enters” or “leaves” an obscuring face?
- **Solution**: If edges of a polygon are described clockwise when viewing the object from “outside”, we can test the line direction with the direction of the intersecting edge describing the front face



Depth-Buffer Method (Z-Buffer)

- In addition to the frame buffer (that keeps the pixel values), keep a Z-buffer containing the depth value of each pixel
- Surfaces are scan-converted in an arbitrary order. For each pixel (x, y), the Z-value is computed as well. The pixel (x, y) is overwritten only if it is closer to the viewing plane than the pixel already written at the same location



Depth-Buffer Method (Z-Buffer)

- **Algorithm**:
  – Initialize the *z*-buffer depth and the frame-buffer I:

    depth(x,y) = MAX_Z ; $I(x, y) = I_{background}$

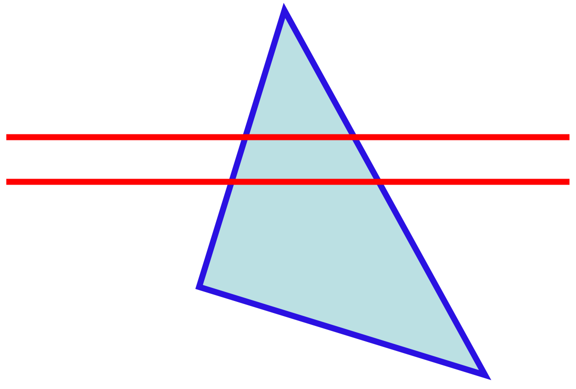
  – Calculate the depth z for each (x, y) position on any surface:
    - If z < depth(x, y), then depth(x, y) = z and $I(x, y) = I_{surf}(x,y)$

- Very simple implementation in the case of polygon surfaces.

  Uses polygon scan line conversion, and exploits face coherence and scan-line coherence :
  - $z = -(Ax+By+D)/C$
  - Along scan lines

    $z' = -(A(x+1)+By+D)/C = z - A/C$
  - Between successive scan lines:

    $z' = -(Ax+B(y+1)+D)/C = z - B/C$



Depth-Buffer Method (Z-Buffer)

- **Example**:

M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M

+

5	5	5	5	5	5	5
5	5	5	5	5	5	
5	5	5	5	5		
5	5	5	5			
5	5	5				
5	5					
5						

=

5	5	5	5	5	5	5	M
5	5	5	5	5	5	M	M
5	5	5	5	5	M	M	M
5	5	5	5	M	M	M	M
5	5	5	M	M	M	M	M
5	5	M	M	M	M	M	M
5	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M

5	5	5	5	5	5	5	M
5	5	5	5	5	5	M	M
5	5	5	5	5	M	M	M
5	5	5	5	M	M	M	M
5	5	5	M	M	M	M	M
5	5	M	M	M	M	M	M
5	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M

+

7					
6	7				
5	6	7			
4	5	6	7		
3	4	5	6	7	
2	3	4	5	6	7

=

5	5	5	5	5	5	5	M
5	5	5	5	5	5	M	M
5	5	5	5	5	M	M	M
5	5	5	5	M	M	M	M
5	5	5	M	M	M	M	M
4	5	6	7	M	M	M	M
3	4	5	6	7	M	M	M
2	3	4	5	6	7	M	M

Depth-Buffer Method (Z-Buffer)

- Implemented in the image space
- Very common in hardware due its simplicity (SGI)
- 32 bits per pixel for *Z* is common

- **Advantages**:
  - Simple and easy to implement
  - Buffer may be saved with image for re-processing
- **Disadvantages**:
  - Requires a lot of memory
  - Finite depth precision can cause problems
  - Spends time while rendering polygons that are not visible
  - Requires re-calculations when changing the scale