Z-buffer and Rasterization

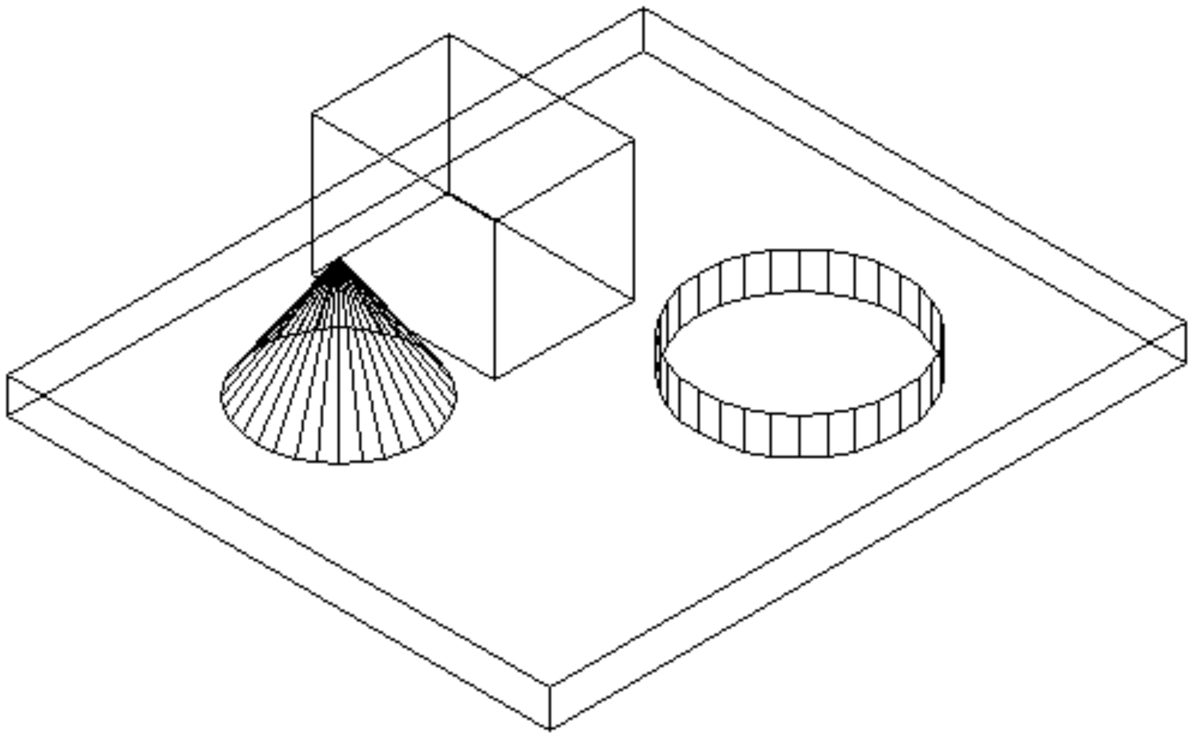
Jian Huang

Visibility Determination

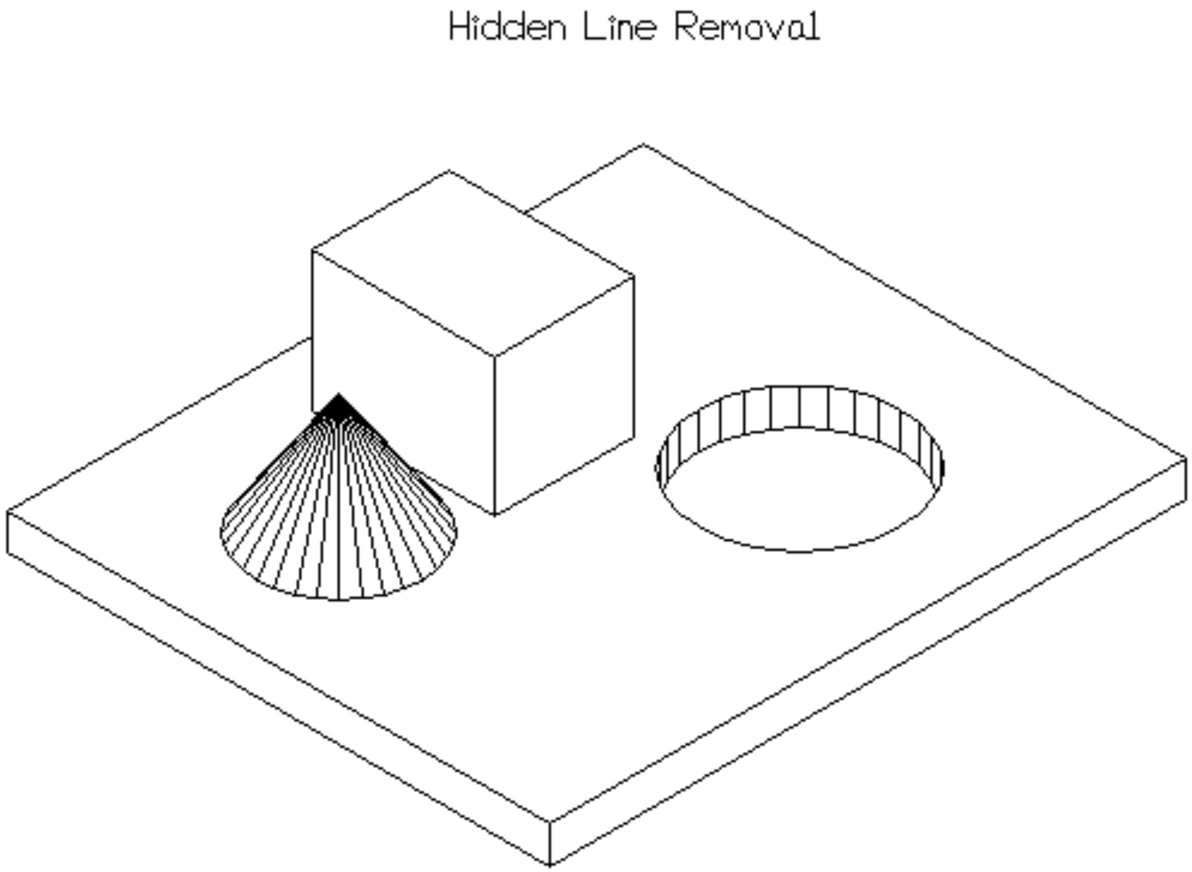
- AKA, hidden surface elimination

Hidden Lines

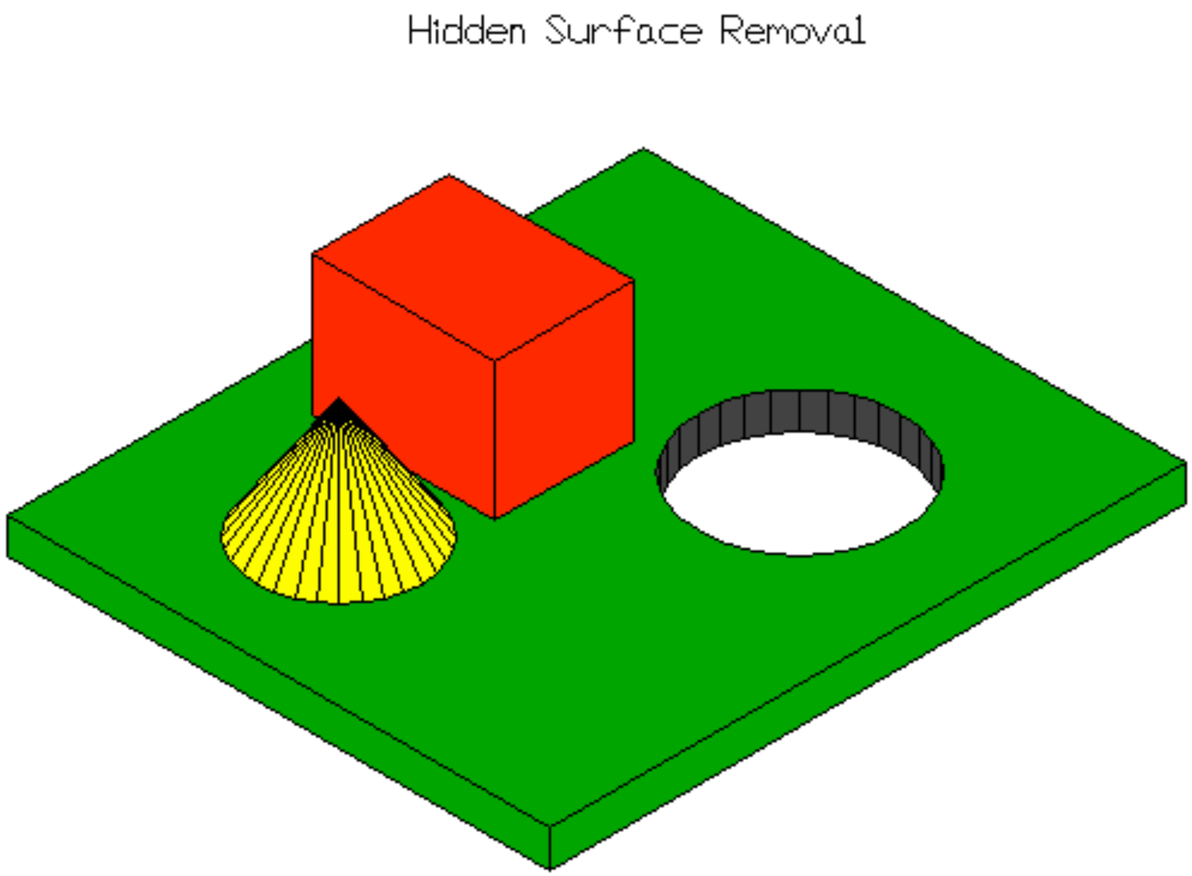
Wireframe



Hidden Lines Removed



Hidden Surfaces Removed

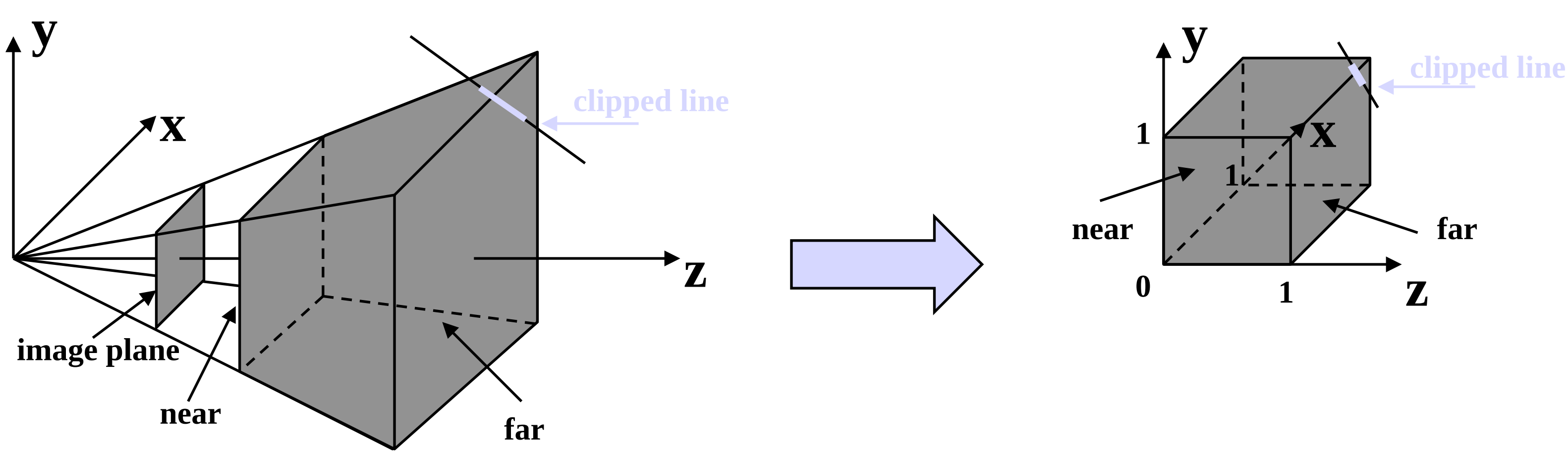


Various Algorithms

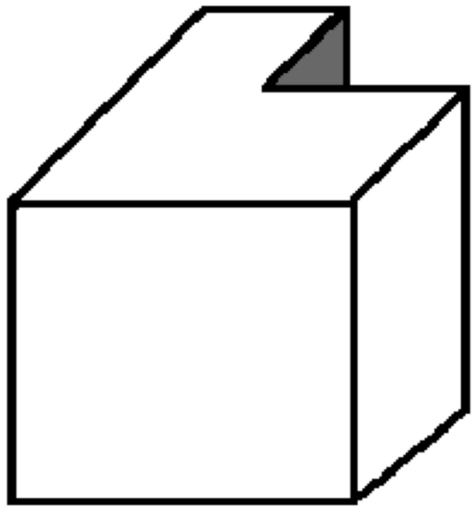
- Backface Culling
- Hidden Object Removal: Painters Algorithm
- Z-buffer
- Spanning Scanline
- Warnock
- Atherton-Weiler
- List Priority, NNA
- BSP Tree
- Taxonomy

Where Are We ?

- Canonical view volume (3D image space)
- Clipping done
- division by w
- z > 0



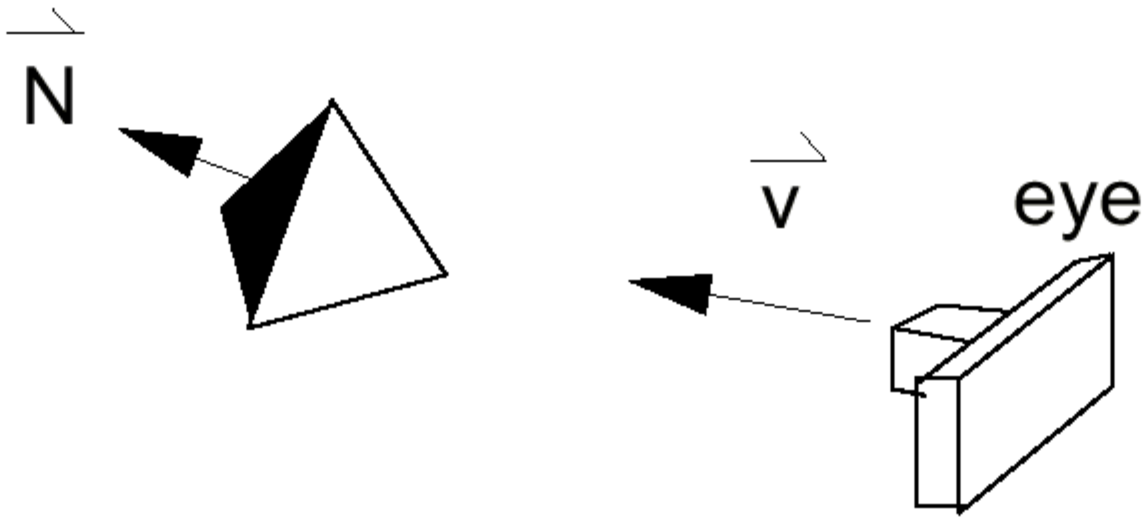
Back-face Culling



- Problems ?
- Conservative algorithms
- Real job of visibility never solved

Back-face Culling

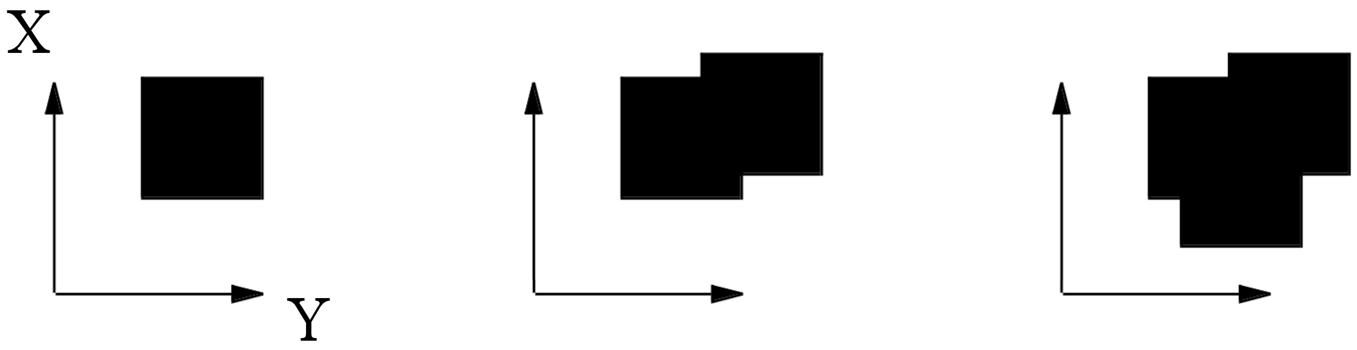
- If a surface's normal is pointing to the same direction as our eye direction, then this is a back face
- The test is quite simple: if N * V > 0 then we reject the surface



Painters Algorithm

- Sort objects in depth order
- Draw all from Back-to-Front (far-to-near)
- Is it so simple?

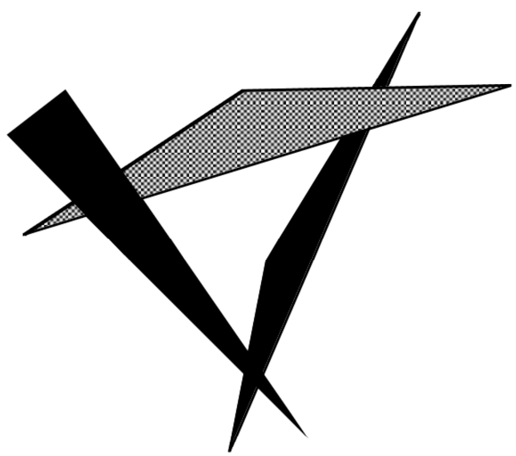
 at $z = 22$,  at $z = 18$,  at $z = 10$,

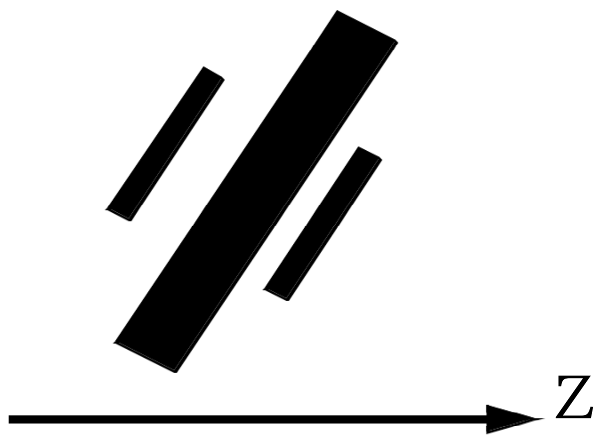


1 2 3

3D Cycles

- How do we deal with cycles?
- Deal with intersections
- How do we sort objects that overlap in Z?





Form of the Input

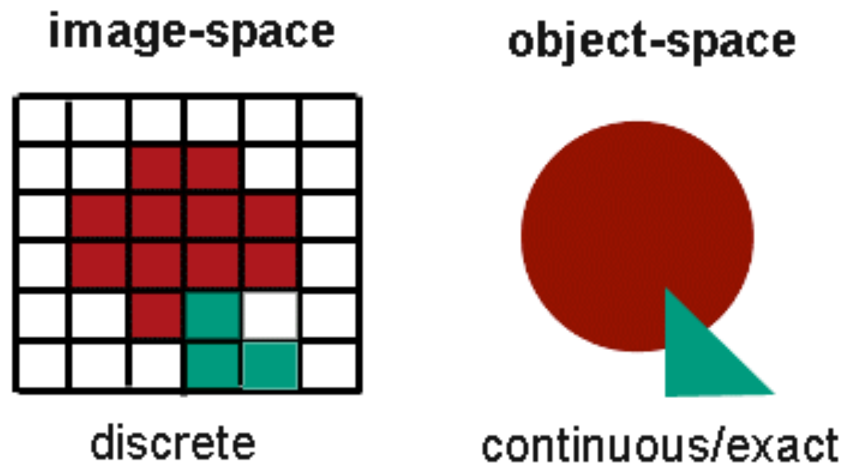
Object types: what kind of objects does it handle?

- convex vs. non-convex

- polygons vs. everything else - smooth curves, non-continuous surfaces, volumetric data

Form of the output

Precision: image/object space?



- Object Space
  - Geometry in, geometry out
  - Independent of image resolution
  - Followed by scan conversion
- Image Space
  - Geometry in, image out
  - Visibility only at pixels

Object Space Algorithms

- Volume testing – Weiler-Atherton, etc.
  - input: convex polygons + infinite eye pt
  - output: visible portions of wireframe edges

Image-space algorithms

- Traditional Scan Conversion and Z-buffering
- Hierarchical Scan Conversion and Z-buffering
  - input: any plane-sweepable/plane-boundable objects
  - preprocessing: none
  - output: a discrete image of the exact visible set

Conservative Visibility Algorithms

- Viewport clipping
- Back-face culling
- Warnock's screen-space subdivision

Z-buffer

- Z-buffer is a 2D array that stores a depth value for each pixel.

- ***InitScreen***:

  for i := 0 **to** N **do**

  for j := 1 **to** N **do**

  Screen[i][j] := *BACKGROUND_COLOR*; Zbuffer[i][j] := ∞;

- ***DrawZpixel*** (x, y, z, color)

  if (z <= Zbuffer[x][y]) **then**

  Screen[x][y] := color; Zbuffer[x][y] := z;

Z-buffer: Scanline

I. **for** *each polygon* **do**

for *each pixel* (x,y) *in the polygon's projection* **do**

z := -(D+A*x+B*y)/C;

DrawZpixel(x, y, z, polygon's color);

II. **for** *each scan-line y* **do**

for *each "in range" polygon projection* **do**

for each pair (x_1, x_2) of X-intersections **do**

for x := x_1 to x_2 **do**

z := -(D+A*x+B*y)/C;

DrawZpixel(x, y, z, polygon's color);

If we know $z_{x,y}$ at (x,y) than: $z_{x+1,y} = z_{x,y} - A/C$

Incremental Scanline

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

$$z = \frac{-(Ax + By + D)}{C}, C \neq 0$$

On a scan line Y = j, a constant

Thus depth of pixel at $(x_1 = x + \Delta x, j)$

$$z_1 - z = \frac{-(Ax_1 + Bj + D)}{C} + \frac{-(Ax + Bj + D)}{C}$$

$$z_1 - z = \frac{A(x - x_1)}{C}$$

$$z_1 = z - (\frac{A}{C})\Delta x$$, since Δx = 1,

$$z_1 = z - \frac{A}{C}$$

Incremental Scanline (contd.)

- All that was about increment for pixels on each scanline.
- How about across scanlines for a given pixel ?
- Assumption: next scanline is within polygon

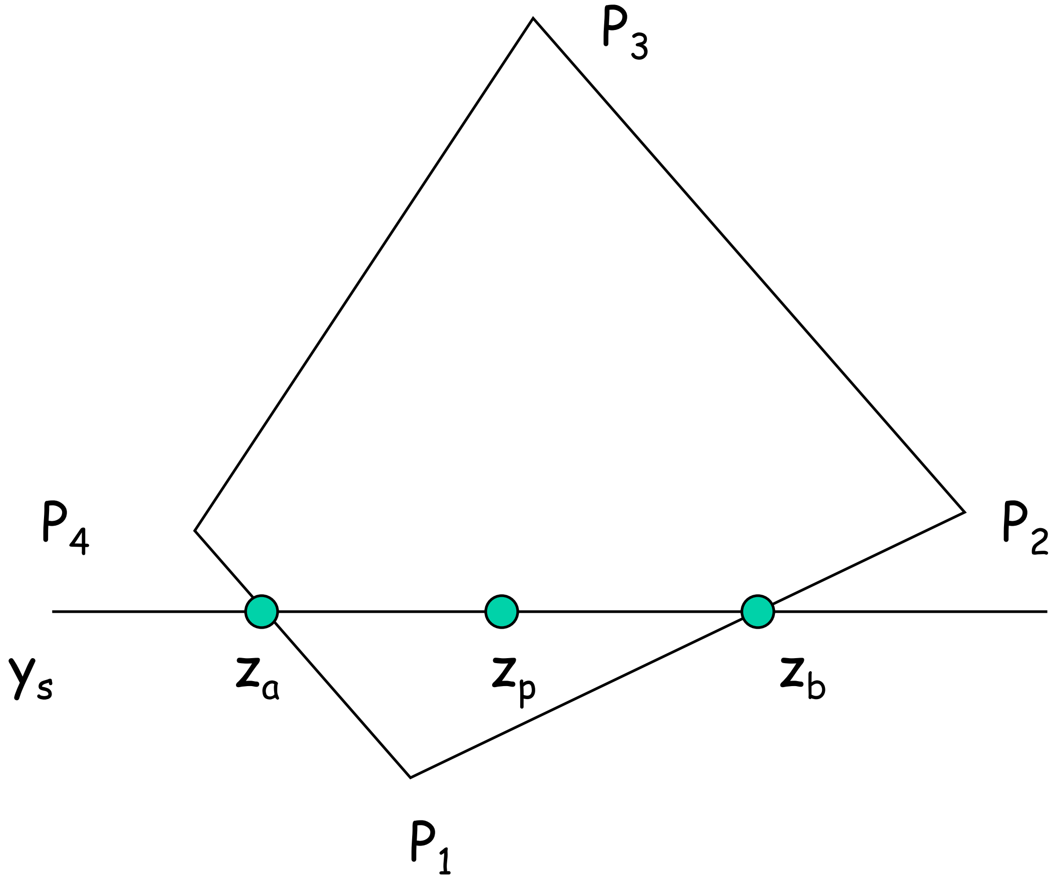
$$z_1 - z = \frac{-(Ax + By_1 + D)}{C} + \frac{(Ax + By + D)}{C}$$

$$z_1 - z = \frac{A(y - y_1)}{C}$$

$$z_1 = z - (\frac{B}{C})\Delta y$$, since Δy = 1,

$$z_1 = z - \frac{B}{C}$$

Non-Planar Polygons



$$z_a = z_1 + (z_4 - z_1)\frac{(y_1 - y_s)}{(y_1 - y_4)}$$

$$z_b = z_1 + (z_2 - z_1)\frac{(y_1 - y_s)}{(y_1 - y_2)}$$

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

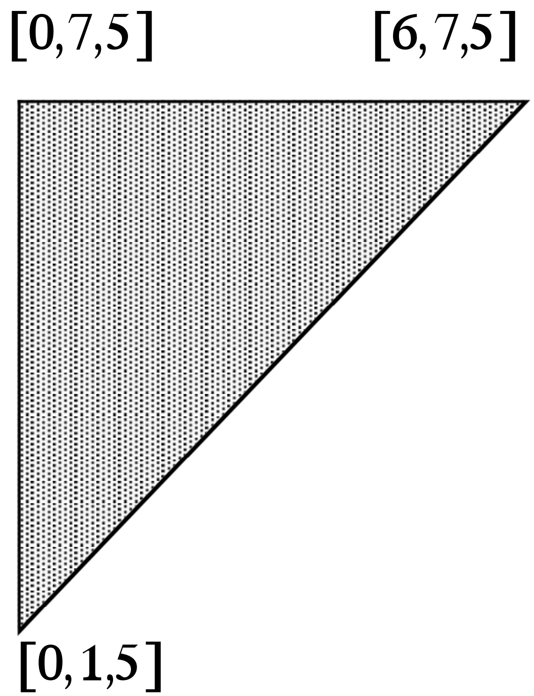
Bilinear Interpolation of Depth Values

Z-buffer - Example

∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞

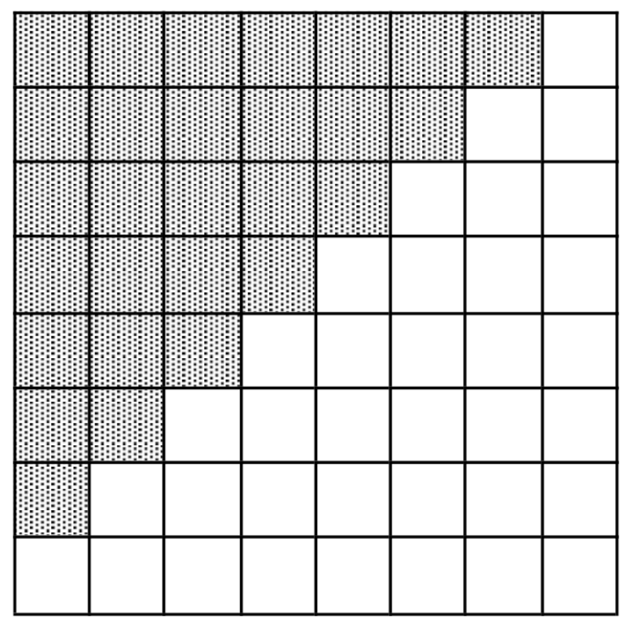
Z-buffer

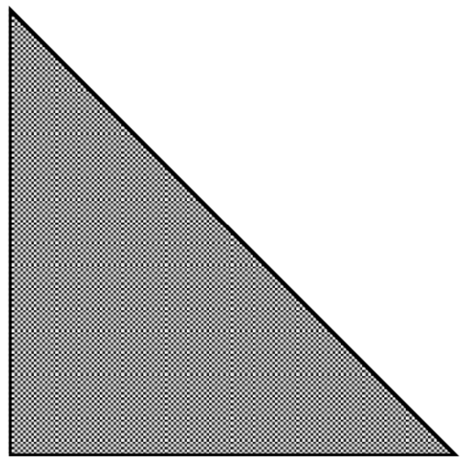
Screen



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						

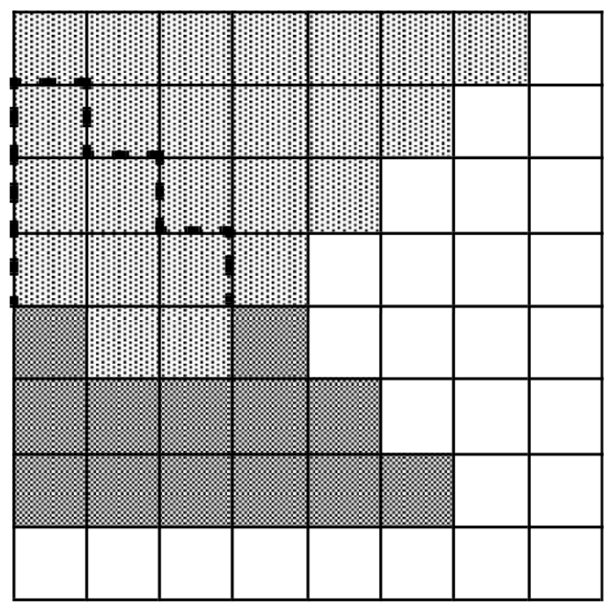
5	5	5	5	5	5	5	∞
5	5	5	5	5	5	∞	∞
5	5	5	5	5	∞	∞	∞
5	5	5	5	∞	∞	∞	∞
5	5	5	∞	∞	∞	∞	∞
5	5	∞	∞	∞	∞	∞	∞
5	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞





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



Non Trivial Example ?

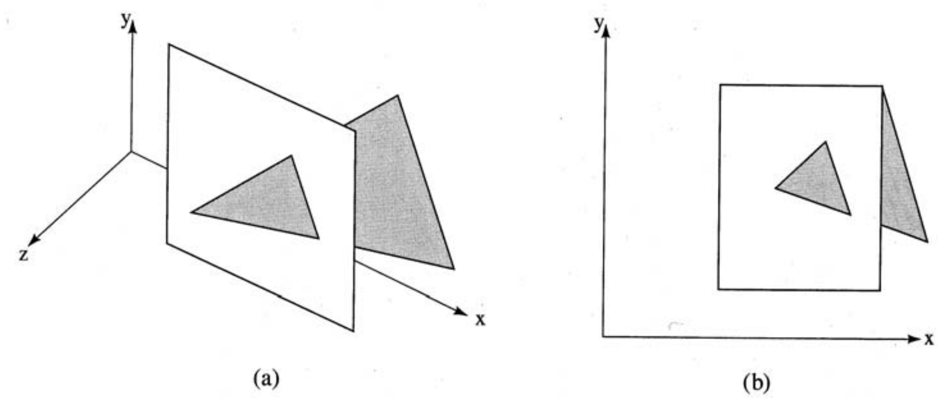


Figure 4–57 Penetrating triangle. (a) Three-dimensional view; (b) two-dimensional projection.

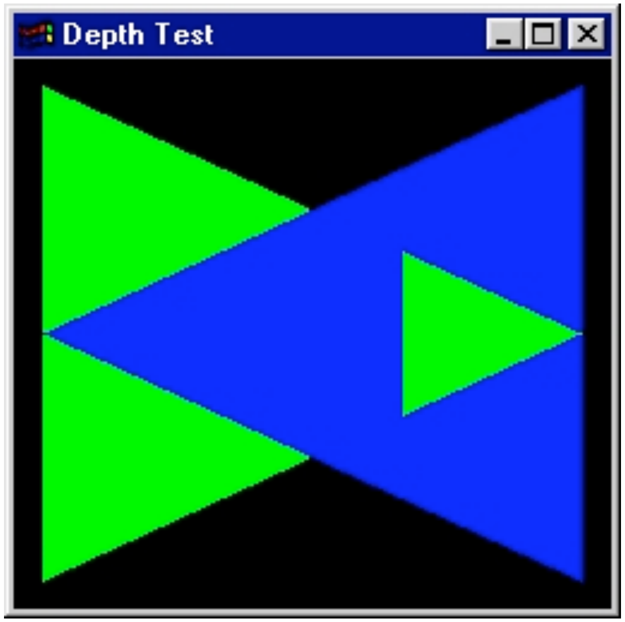
Rectangle: P1(10,5,10), P2(10,25,10), P3(25,25,10), P4(25,5,10)

Triangle: P5(15,15,15), P6(25,25,5), P7(30,10,5)

Frame Buffer: Background 0, Rectangle 1, Triangle 2

Z-buffer: 32x32x4 bit planes

Example



Z-Buffer Advantages

- Simple and easy to implement
- Amenable to scan-line algorithms
- Can easily resolve visibility cycles

Z-Buffer Disadvantages

- Does not do transparency easily
- Aliasing occurs! Since not all depth questions can be resolved
- Anti-aliasing solutions non-trivial
- Shadows are not easy
- Higher order illumination is hard in general

Scanline Rasterization

- Polygon scan-conversion:
- Intersect scanline with polygon edges and fill between pairs of intersections

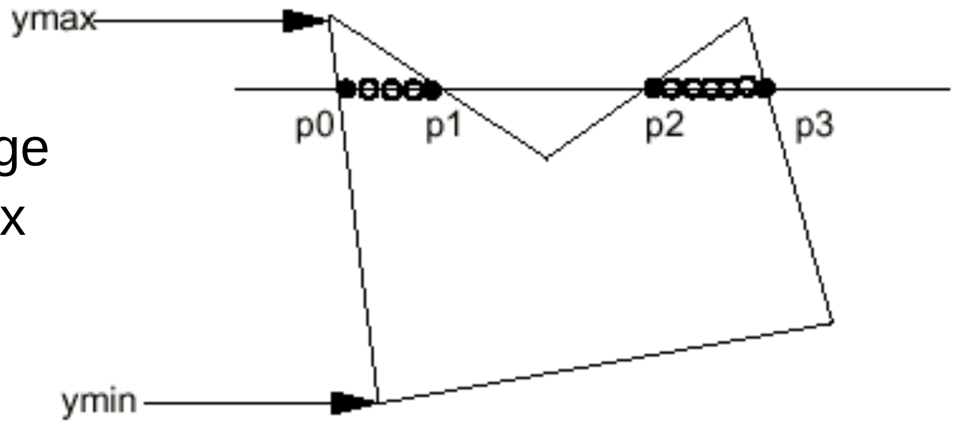
For y = ymin to ymax

1) intersect scanline y with each edge

2) sort interesections by increasing x

[p0,p1,p2,p3]

3) fill pairwise (p0 > p1, p2> p3, ...)

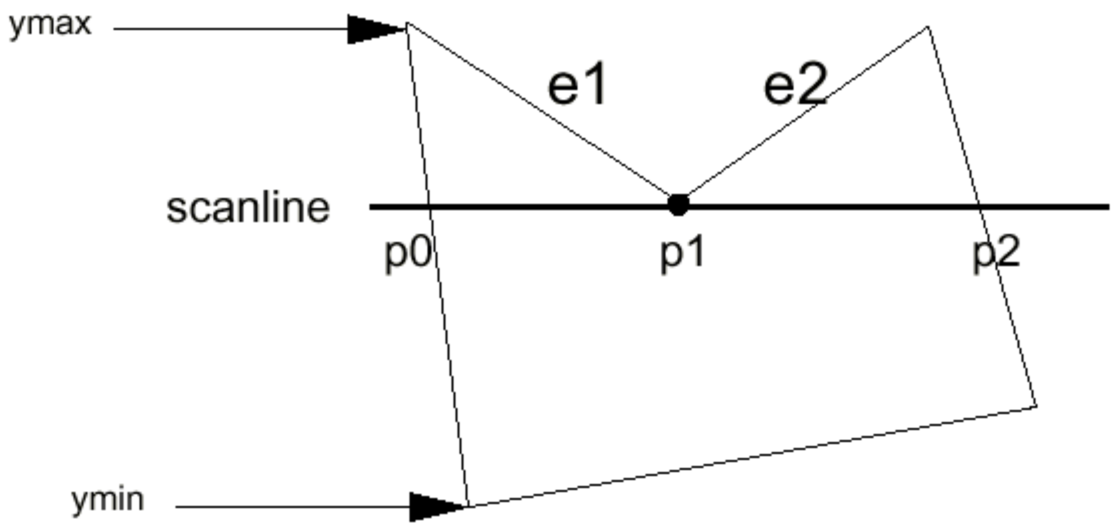


Scanline Rasterization Special Handling

- Make sure we only fill the interior pixels
  - Define interior: For a given pair of intersection points (Xi, Y), (Xj, Y)
  - Fill ceiling(Xi) to floor(Xj)
  - important when we have polygons adjacent to each other
- Intersection has an integer X coordinate
  - if Xi is integer, we define it to be interior
  - if Xj is integer, we define it to be exterior
  - (so don’t fill)

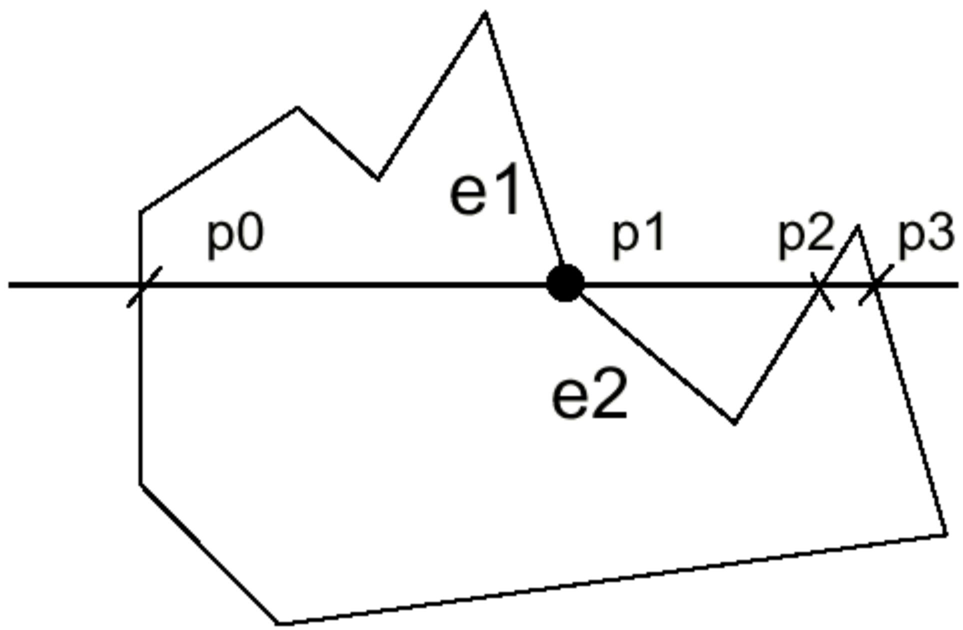
Scanline Rasterization Special Handling

- Intersection is an edge end point, say: (p0, p1, p2) ??
- (p0,p1,p1,p2), so we can still fill pairwise
- In fact, if we compute the intersection of the scanline with edge e1 and e2 separately, we will get the intersection point p1 twice. Keep both of the p1.



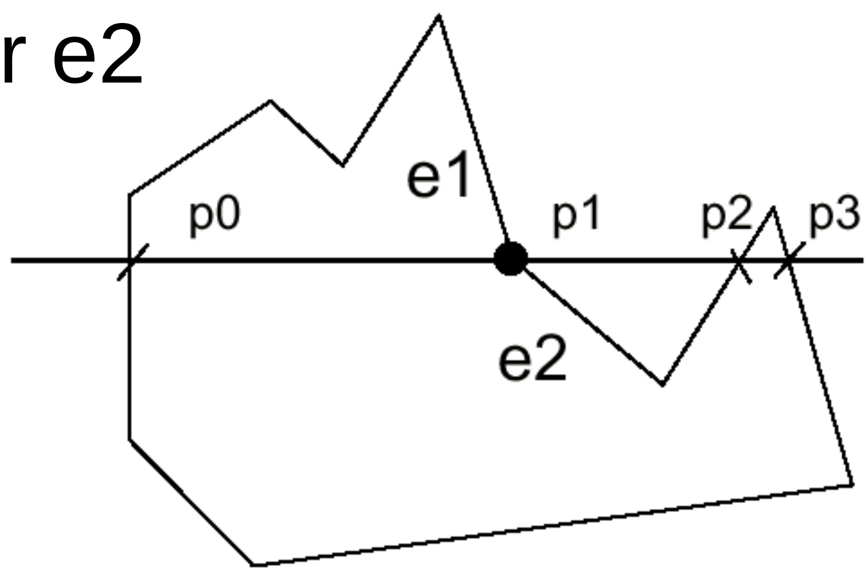
Scanline Rasterization Special Handling

- But what about this case: still (p0,p1,p1,p2)



Rule

- Rule:
  – If the intersection is the ymin of the edge's endpoint, count it. Otherwise, don't.
- Don't count p1 for e2



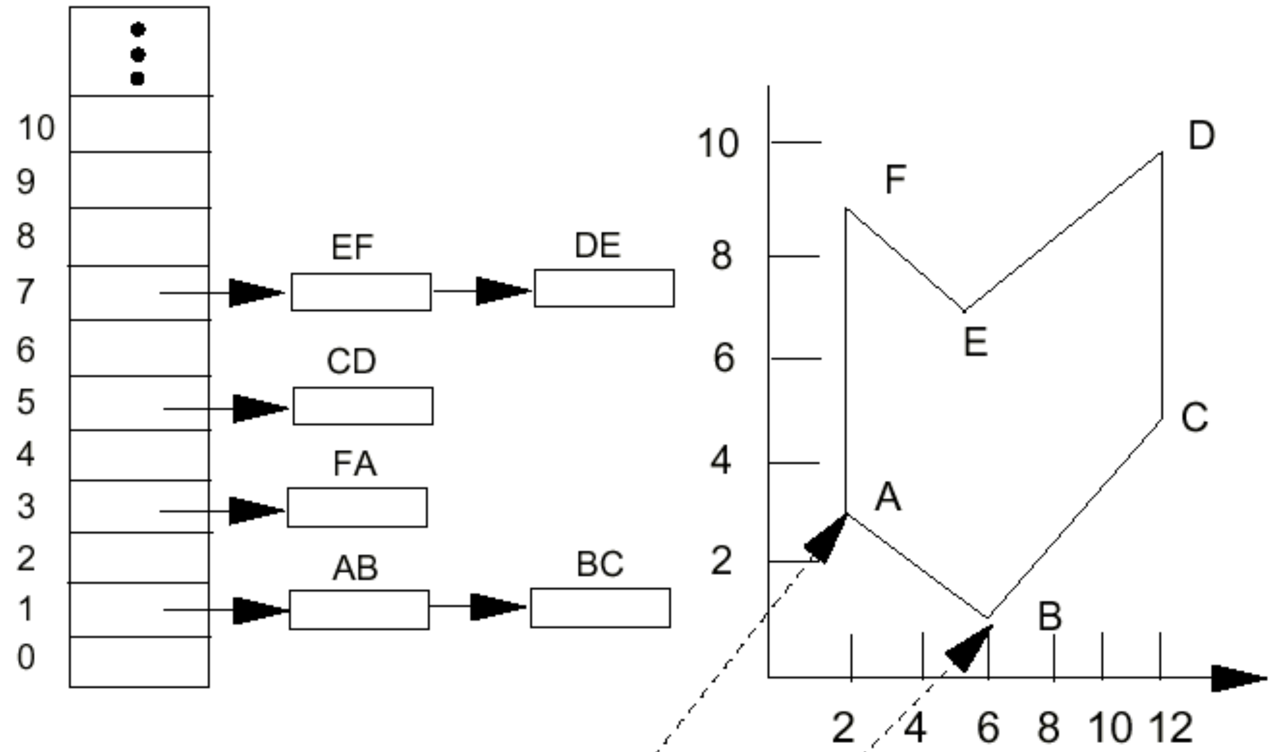
Performance Improvement

- The goal is to compute the intersections more efficiently. Brute force: intersect all the edges with each scanline
  - find the ymin and ymax of each edge and intersect the edge only when it crosses the scanline
  - only calculate the intersection of the edge with the first scan line it intersects
  - calculate dx/dy
  - for each additional scanline, calculate the new intersection as $x = x + dx/dy$

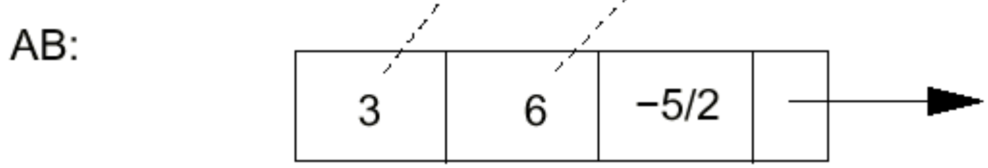
Data Structure

- Edge table:
  - all edges sorted by their ymin coordinates.
  - keep a separate bucket for each scanline
  - within each bucket, edges are sorted by increasing x of the ymin endpoint

Edge Table



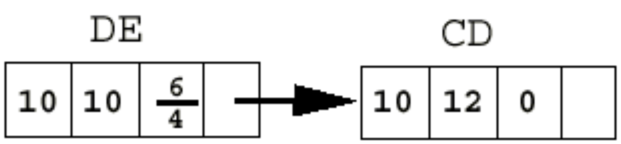
– Edge structure: ymax, xmin, dx/dy, next

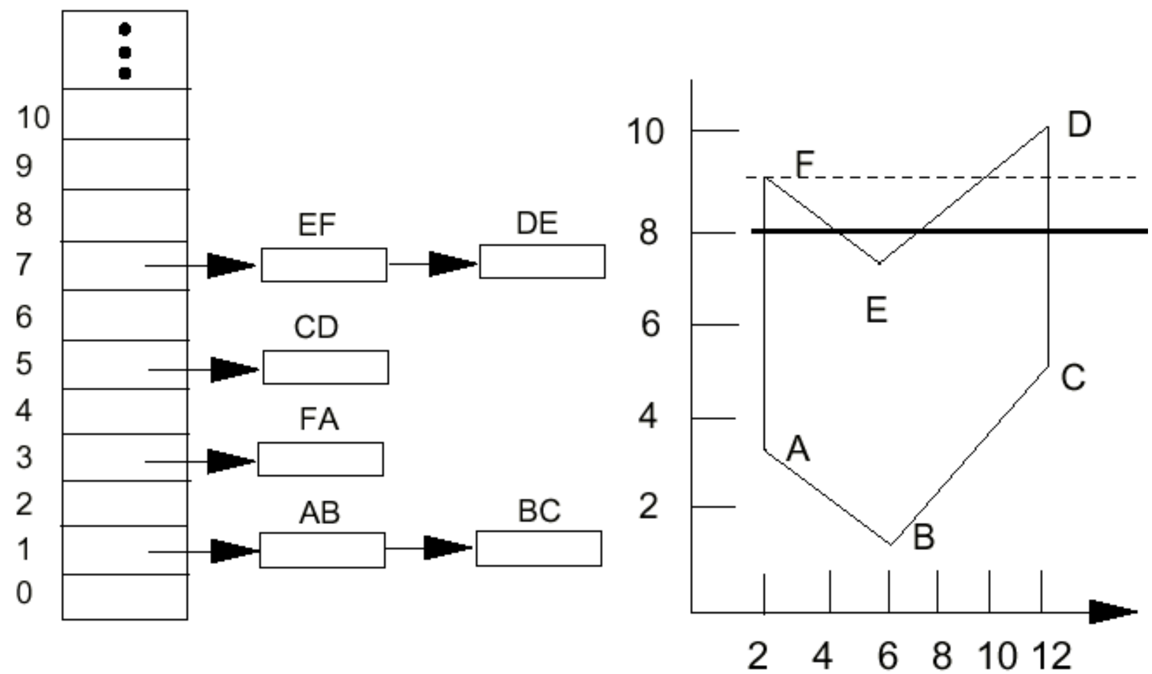


Active Edge Table (AET)

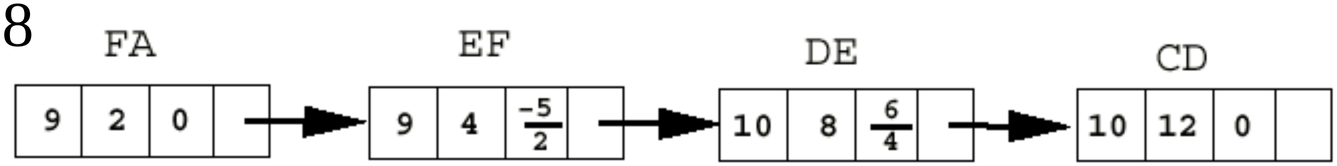
- A list of edges active for current scanline, sorted in increasing x

y = 9





y = 8



Polygon Scan-conversion Algorithm

```
Construct the Edge Table (ET);
Active Edge Table (AET) = null;
for y = Ymin to Ymax
    Merge-sort ET[y] into AET by x value
    Fill between pairs of x in AET
    for each edge in AET
        if edge.ymax = y
            remove edge from AET
        else
            edge.x = edge.x + dx/dy
    sort AET by x value
end scan_fill
```