Computer Graphics

- Scan Conversion -

Marcus Magnor

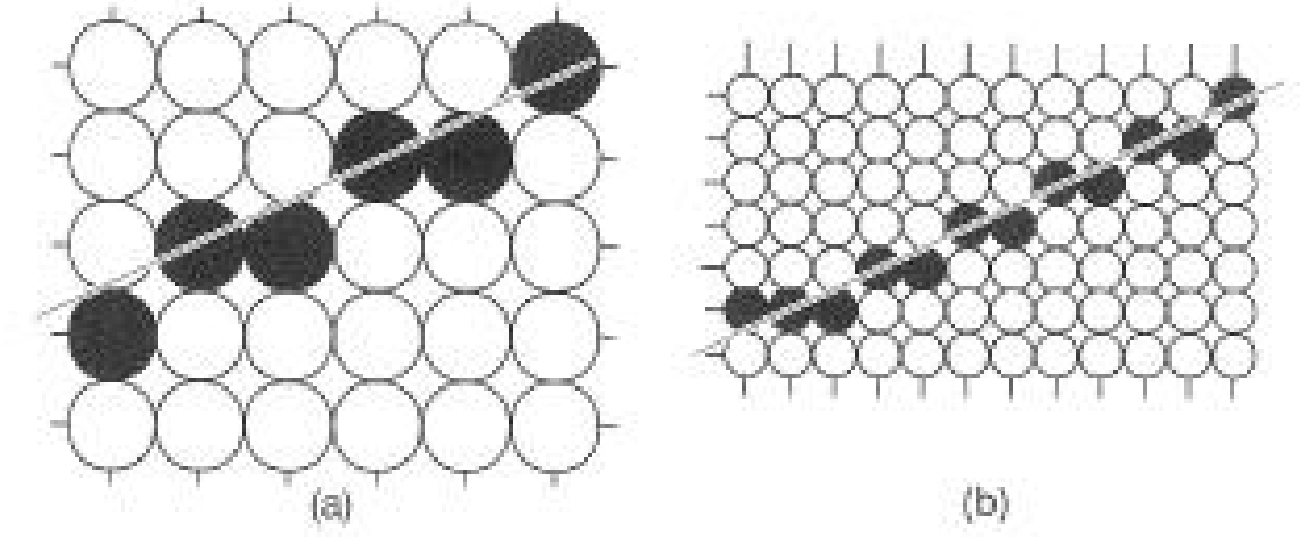
Philipp Slusallek

Overview

- **So far:**
  - Clipping
  - Rasterization
- **Today:**
  - Antialiased lines
  - Scan conversion
    - Edge coherence
    - Span coherence
  - Interpolation
- **Next time:**
  - Cg

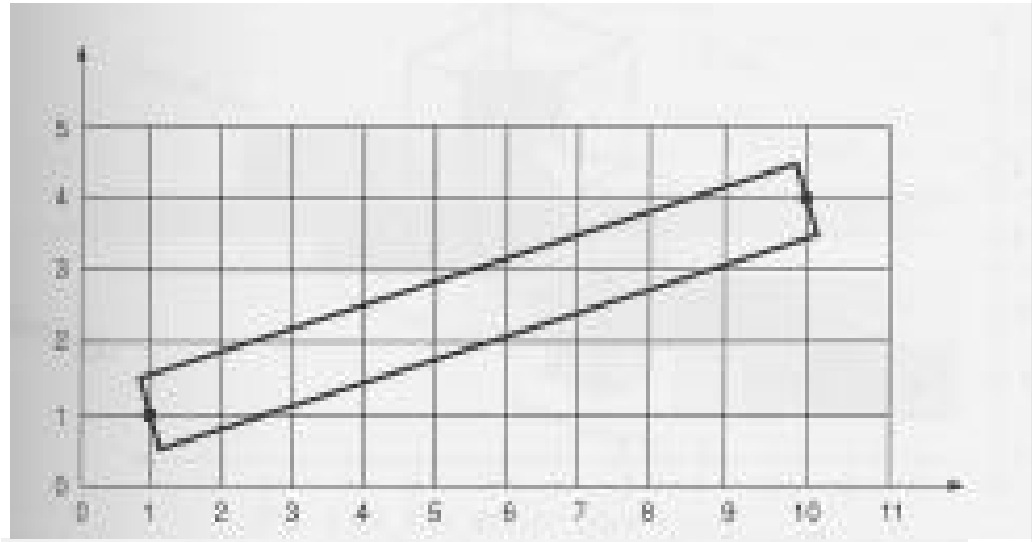
Antialiased Line Drawing

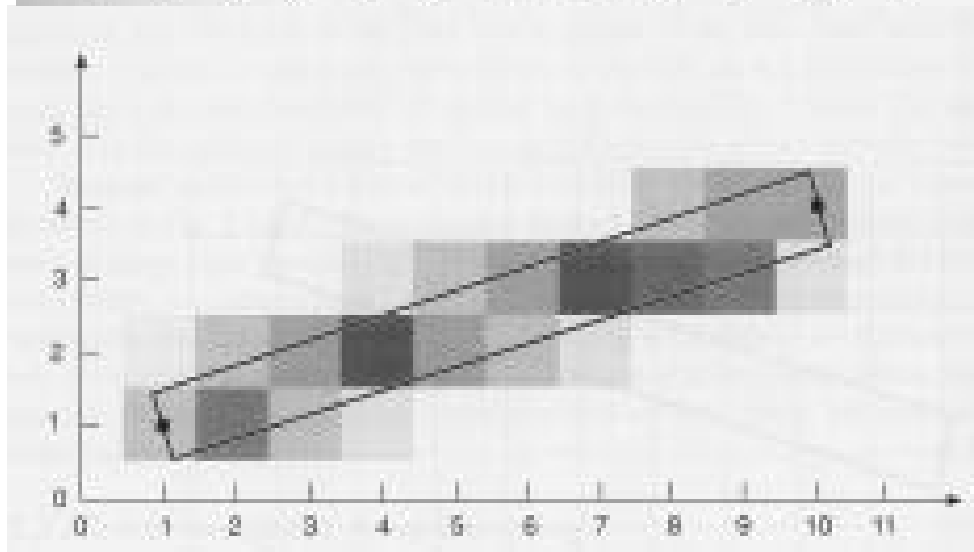
- **Aliasing effects**
  - Moire patterns
  - Staircase, jaggies
- **Trivial solution: Increasing resolution**
  - 4x memory, bandwidth, rendering time
  - Reduces aliasing, doesn’t eliminate



Unweighted Area Sampling

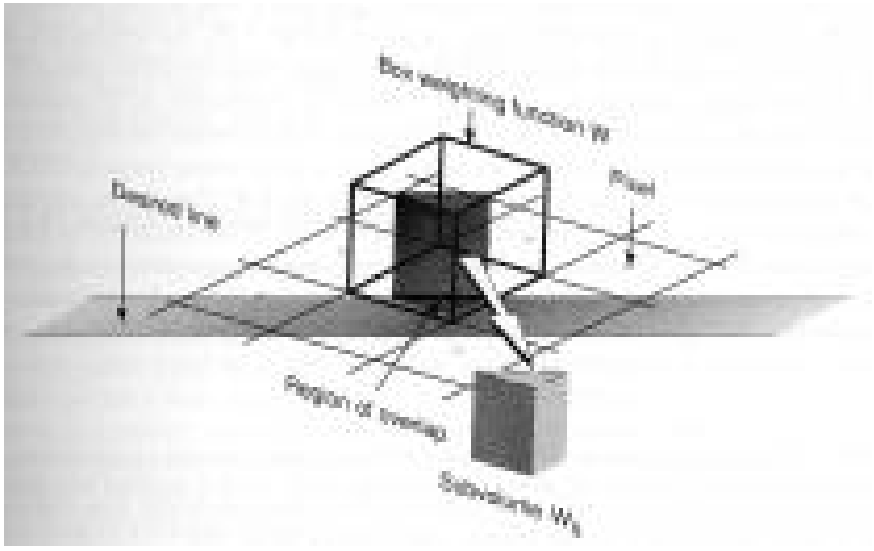
- **Line has finite area**
  - Draw rectangle
- **Intensity distribution**
  - According to percentage covered
  - Only intersected pixels affected
- **Unweighted sampling**
  - Equal areas contribute equally
  - Entire pixel area of equal weight
  - distance pixel center - line no criterion

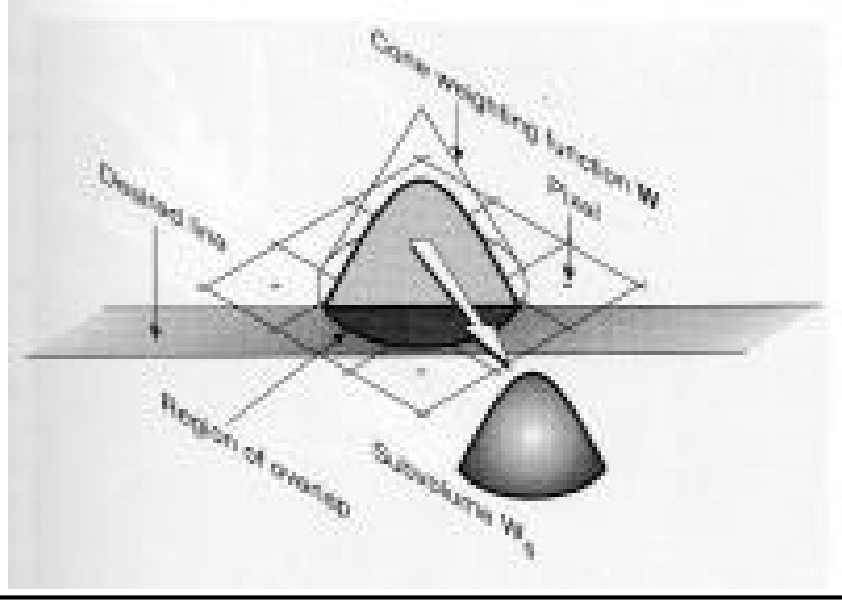




Weighted Area Sampling

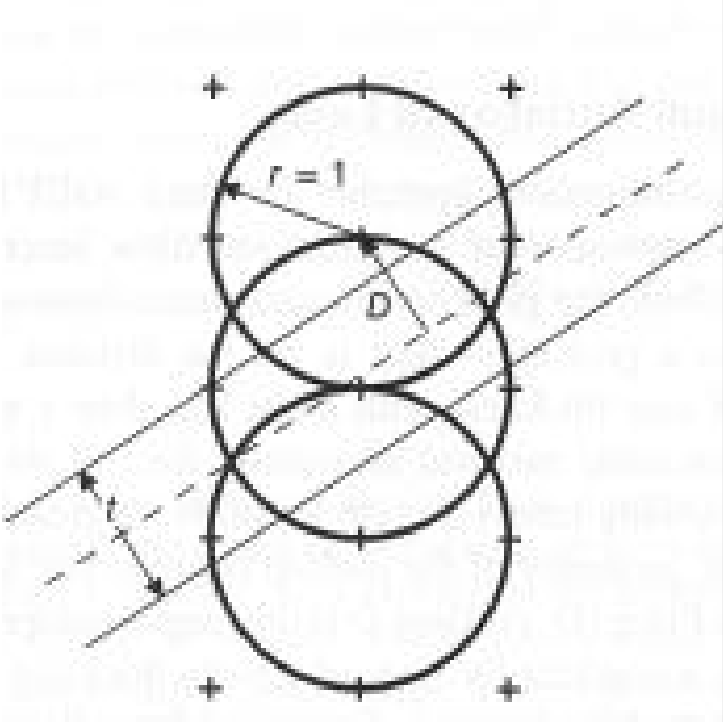
- **Equal areas contribute unequally**
  - Area close to pixel center has greater influence
- **Weighting function**
  - Centered on each pixel
  - Integral = 1
  - Larger than pixel diameter
  - Weighted intersection area with line rectangle: intensity
- **Unweighted Sampling**
  - Box filter
- **Weighted Sampling**
  - Cone filter





Cone-filtered Line Drawing

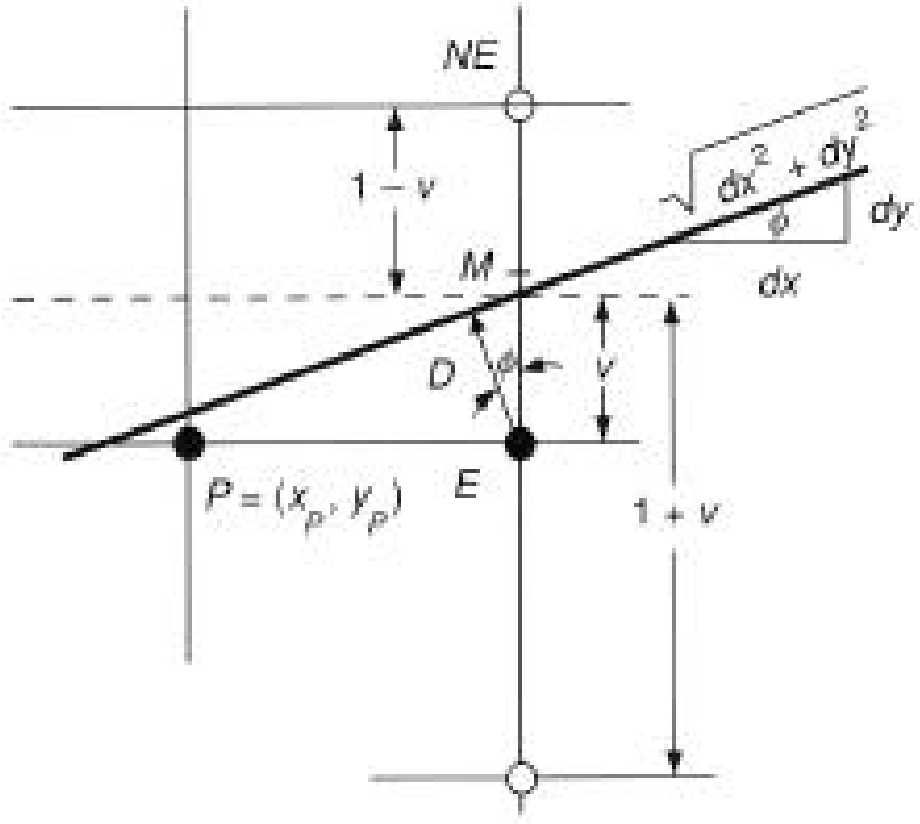
- **Gupta-Sproull algorithm**
- **Need to know smallest distance from line to pixel center *D***
  - Weighted intersected line area: table lookup $W(D,t)$
    - t: line thickness
- **Bresenham**
  - Decision variable d to find mid-line pixel
  - Intensity for mid-pixel and both vertically neighboring pixels ?
    - Distance pixels' center – line D



Antialiased Bresenham Lines

- **Distance to line center**

$$D = v\cos f = \frac{vdx}{\sqrt{dx^2 + dy^2}}$$

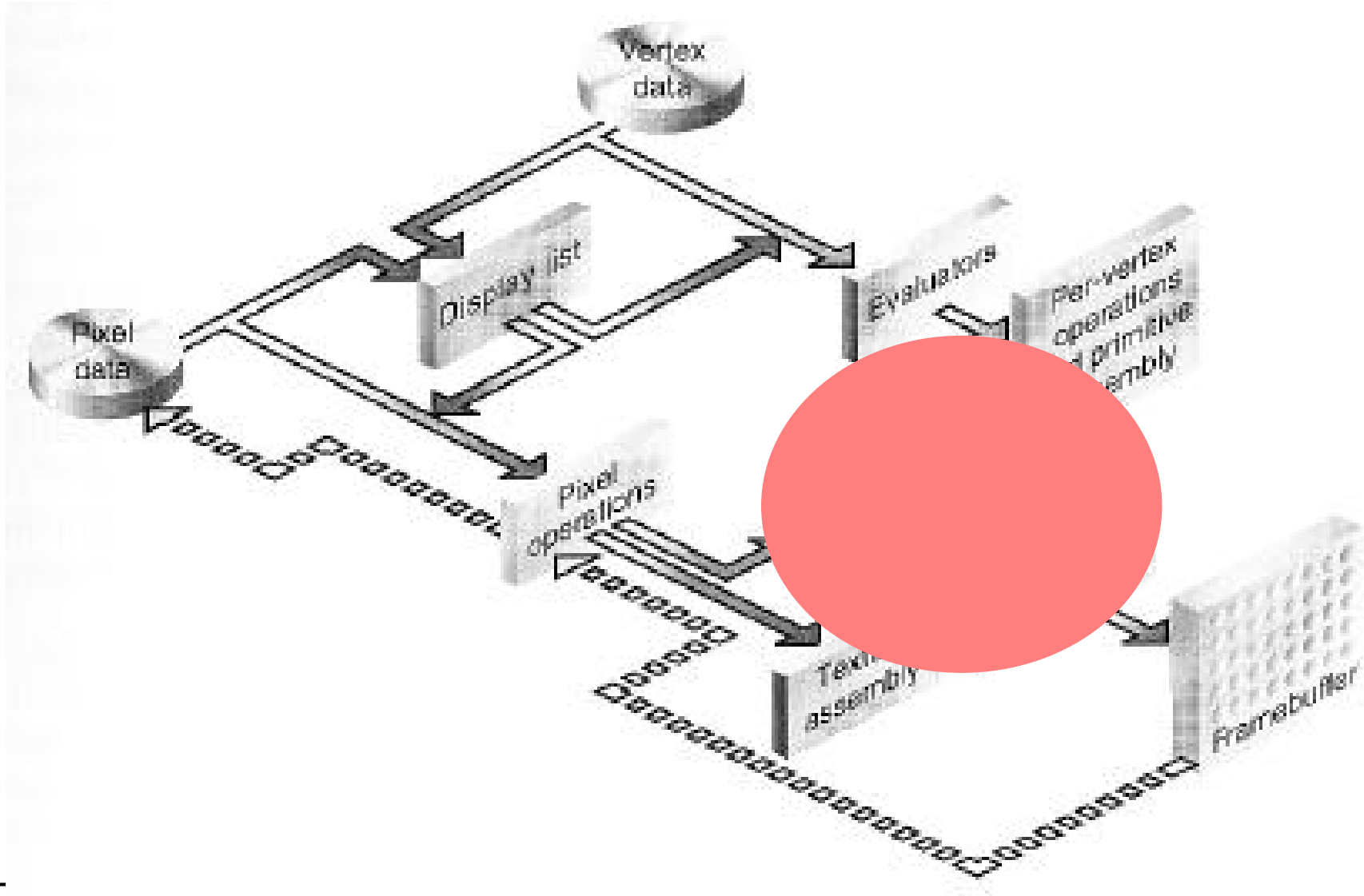


- **In incremental form**

$$D_{closest} = \frac{d \pm 1}{2\sqrt{\Delta x^2 + \Delta y^2}}$$

$$D_{\pm 1} = \frac{2 \pm (d \pm 1)}{2\sqrt{\Delta x^2 + \Delta y^2}}$$

We are here ...

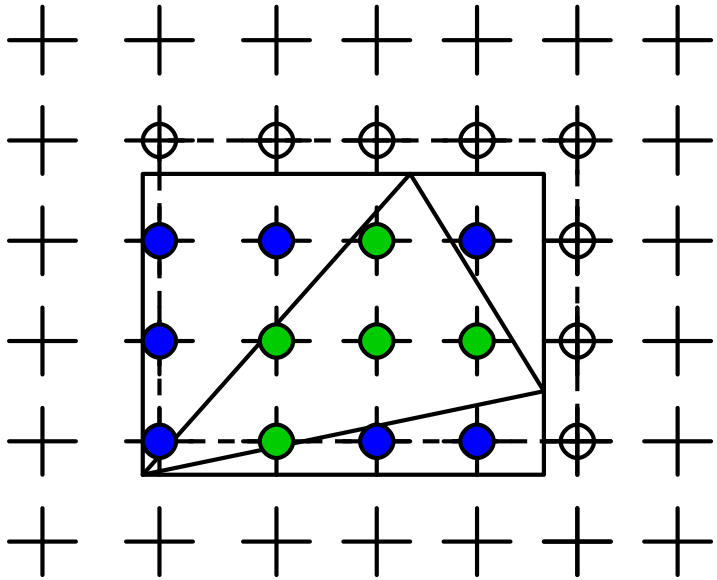

Vertex data
display list
Evaluators
Per-vertex operations & primitive assembly
Pixel data
Pixel operations
Texture assembly
Rasterization
Framebuffer

Scene-to-Screen Conversion

- **Scene composed of primitives**
- **Rendering:**
  - Determine 2D projection coordinates of each primitive
    - Per-vertex transformations
  - Find screen pixels covered by primitive
    - Clipping
    - rasterization
  - Handle occlusions between primitives
    - Hidden surface removal
  - Calculate pixel colors from visible primitives
    - shading

Triangle Filling

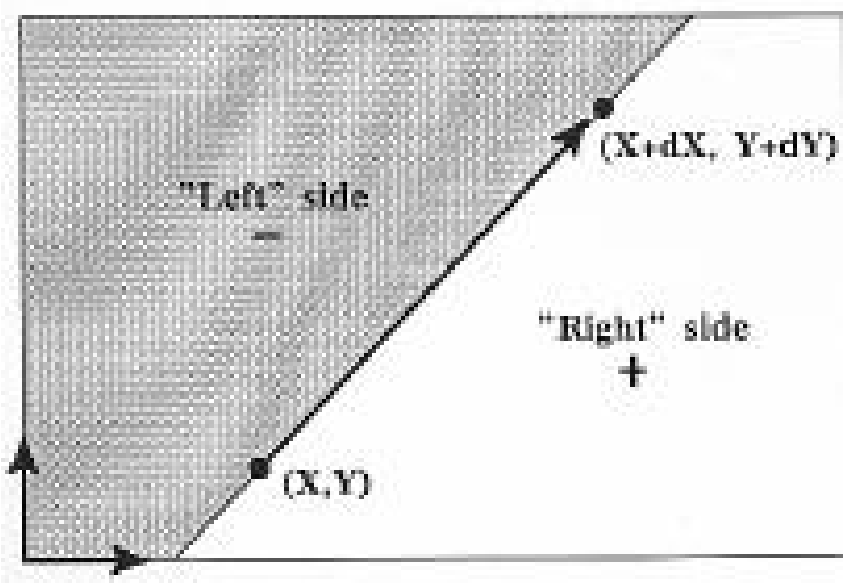
- **Brute-Force algorithm**



```
Raster3_box(vertex v[3])
{
  int x, y;
  bbox b;
  bound3(v, &b);
  for (y= b.ymin; y < b.ymax; y++)
    for (x= b.xmin; x < b.xmax; x++)
      if (inside(v, x, y))
       fragment(x,y);
}
```

Inside-Outside Test for Triangles

- **Approach**
  - Implicit edge functions to describe the triangle $F_i(x,y)= ax+by+c$
  - Point inside triangle, if every $F_i(x,y) <= 0$
  - Incremental evaluation of the linear function F by adding a or b

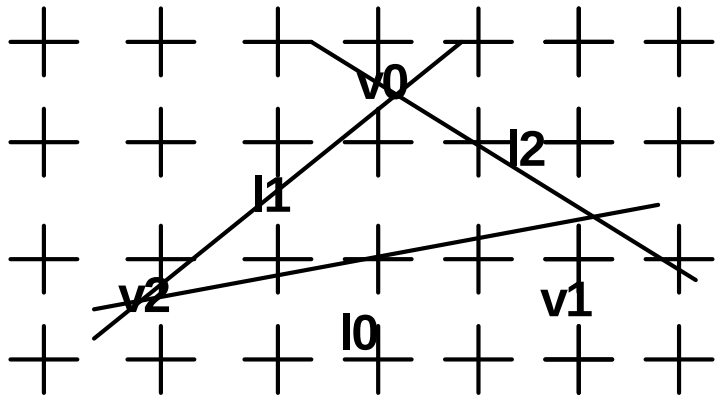


Incremental Rasterization Process

```
Raster3_incr(vertex v[3])
{
  edge l0, l1, l2;
  value d0, d1, d2;
  bbox b;
  bound3(v, &b);
  mkedge(v[0],v[1],&l2);
  mkedge(v[1],v[2],&l0);
  mkedge(v[2],v[0],&l1);

  d0 = l0.a * b.xmin + l0.b * b.ymin + l0.c;
  d1 = l1.a * b.xmin + l1.b * b.ymin + l1.c;
  d2 = l2.a * b.xmin + l2.b * b.ymin + l2.c;

  for( y=b.ymin; y<b.ymax, y++ ) {
    for( x=b.xmin; x<b.xmax, x++ ) {
      if( d0<=0 && d1<=0 && d2<=0 ) fragment(x,y);
      d0 += l0.a; d1 += l1.a; d2 += l2.a;
    }
    d0 += l0.a * (b.xmin - b.xmax) + l0.b; . . . }
}
```



Coherence

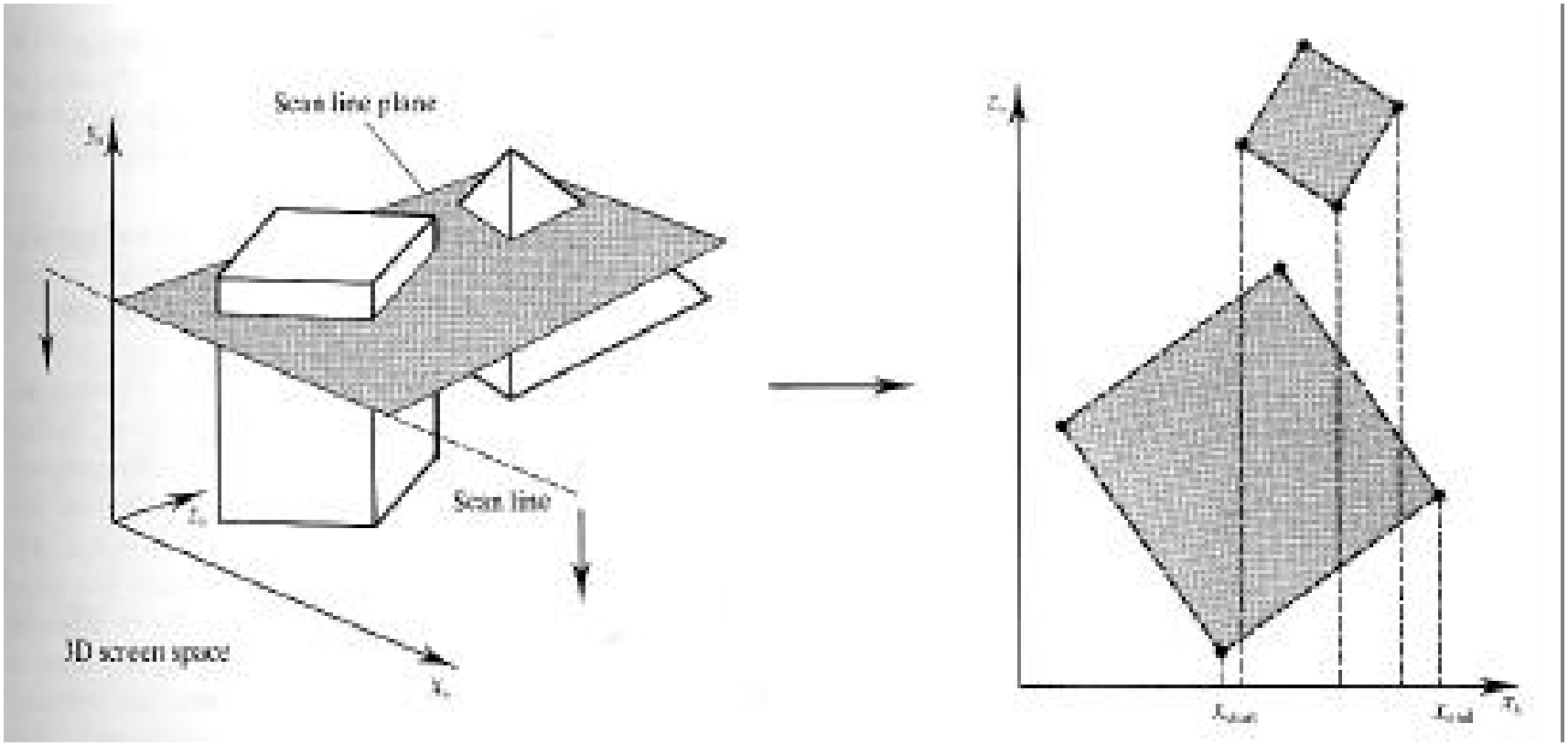
- ***Adjacent pixels generally exhibit the same properties***

- If a given pixel is ***inside*** a polygon, then ***immediately adjacent*** pixels are also likely to be ***inside*** the polygon. The converse is also true.

- We say that the ***visibility*** of adjacent pixels differs only if an edge (or boundary) of the polygon passes between them - a relatively un-common event.

$\Rightarrow$ **Exploit pixel coherence**
for efficient, fast polygon rendering

Rasterization



- **3D screen space**
- **Scan line: horizontal line of screen**
- **Span: extension of polygonal edge on scan line**
  - From x_{start} to x_{end}

Pixel-Level Processes

- **Rasterization, hidden surface removal, shading**
  - Carried out in inner loop of renderer
    - Regard one polygon at a time
    - Regard one scan line at a time
    - Span: intersection of scan line with polygon
    - Conversion into run of consecutive pixels: exploits coherence
- **Two-dimensional linear interpolation processes**

  For each scan line, find span
  - Limits / x-coordinates: **rasterization**
    - Interpolated from 2D edge vertex positions in screen space

  For each span's limits, find
  - Scene depth: **hidden surface removal**
    - Interpolated from edge vertices' z-coordinate in screen space
  - Color/normal: **shading**
    - Interpolated from edge vertices' color/normal
  - Interpolate depth/color between span limits

Pixel-Level Processes

- **Innermost rendering loop**

```
for each polygon
{
   perform geometric transformations into screen space
   for each scan line within the polygon
   {
       find span by interpolation of edge vertex
       coordinates
       find span limits' depth & color/normal
       rasterize span
       for each pixel within the span
       {
              interpolate depth & color/normal from span
              limits
              perform hidden surface removal
              shade pixel
       }
   }
}
```

Rasterization

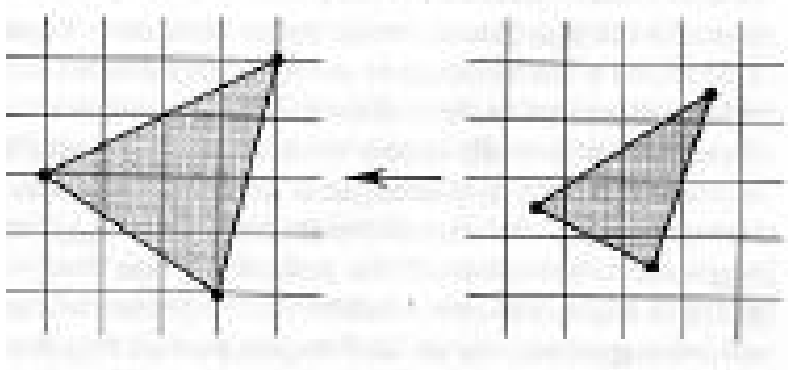
- **Definition**
  - Given a primitive, specify which pixels on a raster display are covered by this primitive
  - Extension: specify fraction of partially covered pixels
    - Antialiasing
- **Questions**
  - Where exactly does a span start/end ?
  - Where within the pixel should the sample point be ?
  - Round screen coordinates to integer ?
    Store with fractional precision ?
- **Problems**
  - Holes between polygons
  - Overlapping polygons (problems with transparency)
  - Discontinuities in textured surfaces
  - Inaccuracies during anti-aliasing

Errors in Rasterization

Screen coordinates

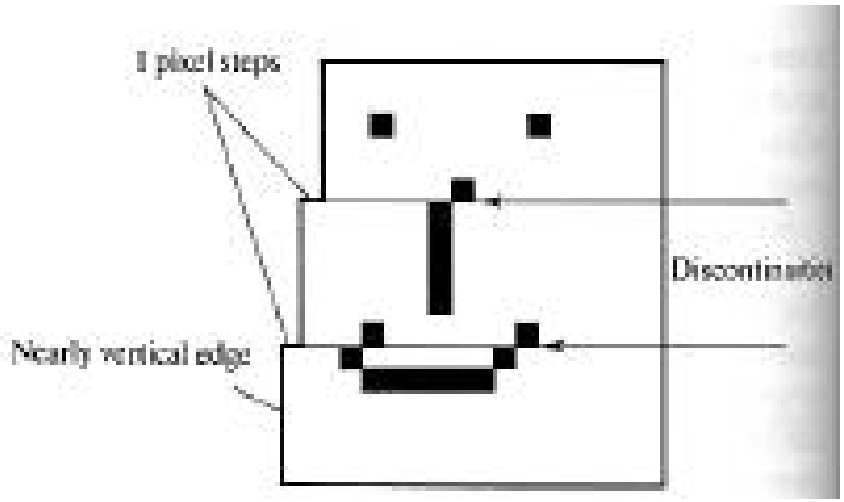
- **Integer rasterization**
  - Distorts shape
  - Correct supersampling not possible
  - Animation: “wobble”
  - Nearly vertical edge: step discontinuity
    - Shift in texture mapping

  ⇒ High precision necessary



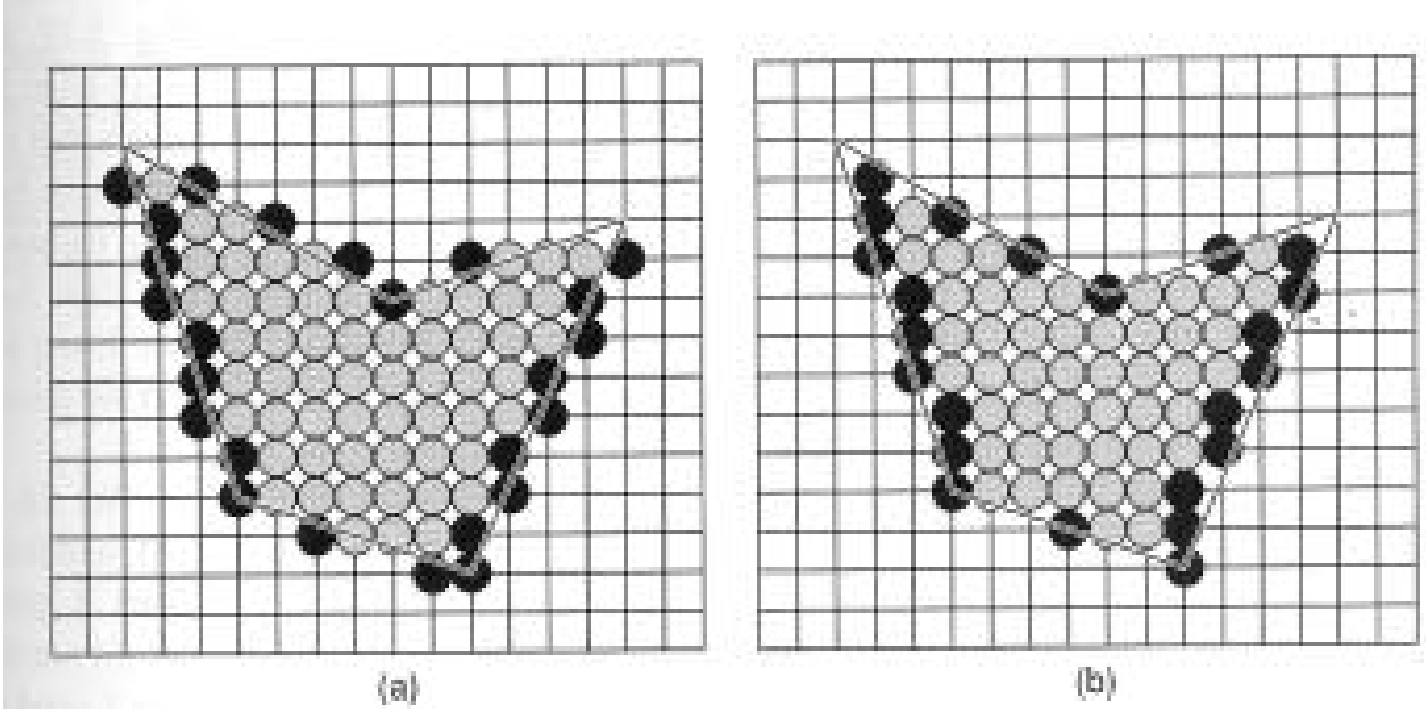
Sample point position

- **Pixel center**
  - Unnecessary fractional-pel offset
- **Pixel corner**
  - Maximal ½-pixel displacement
  - Doesn’t matter as long as consistent

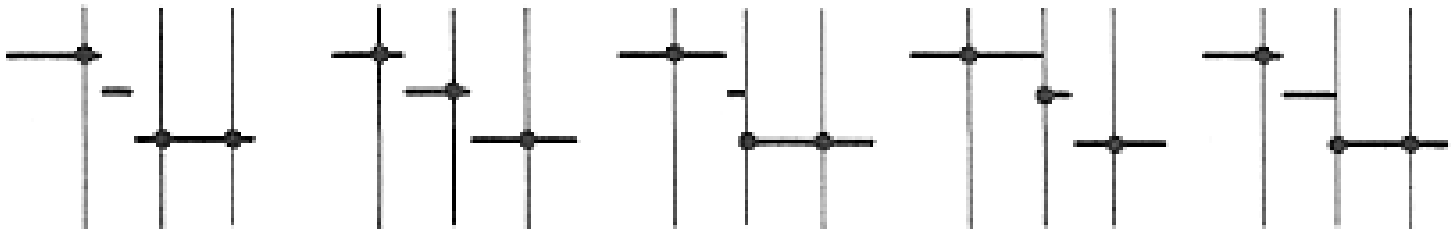


Polygon Edges

- **Bresenham: Closest pixels along edge lines**
  - Inside or outside polygon
  - Overdrawing from neighboring polygons, flickering
- **Combine with knowledge about per-scanline span**
  - Inside-outside: Odd-parity rule



Span Boundary Rounding



- **Real numbers**
  - Round x_{start} up
  - Round x_{end} down
  - If fractional part of x_{end} is 0, subtract 1
- **Integer arithmetic (4-bit example)**
  - Round x_{start} up to next multiple of 16
  - Round x_{end} down to next multiple of 16
  - If x_{end} is multiple of 16, subtract 16 from it
- **No holes between spans**
- **No overlap of span**
- **Generated pixels are always within span bounds**

Span Interpolation

- **Find x_{start} and x_{end} of span**
  - Linear interpolation between polygon edge endpoints
    - Floating point accuracy
  - Modified Bresenham algorithm
    - Integer arithmetic
- **Determine depth for each pixel**
  - Find depth at span boundaries
    - Linear interpolation between polygon edge endpoints (z-coordinates of vertex positions in 3D screen space)
  - Linear interpolation between span boundaries
- **Determine color for each pixel**
  - Gouraud: Find color at span boundaries
  - Phong: Find normal direction at span boundaries
    - Linear interpolation between polygon edge endpoints
  - Linear interpolation between span boundaries

Efficient Scan Conversion

- **In which order to draw polygons/scanlines ?**
  - One polygon after the other
    - Frequent scanline jumps
  - One scanline after the other
    - considers pixels incrementally
    - Exploits pixel coherence
  - Acceleration data structures
    - E.g., triangle strips
- **Brute force: intersect all the edges with each scanline**
  - Find *ymin* and *ymax* of each edge
  - Intersect edge only when it crosses the scanline
  - 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 $x = x + dx/dy$

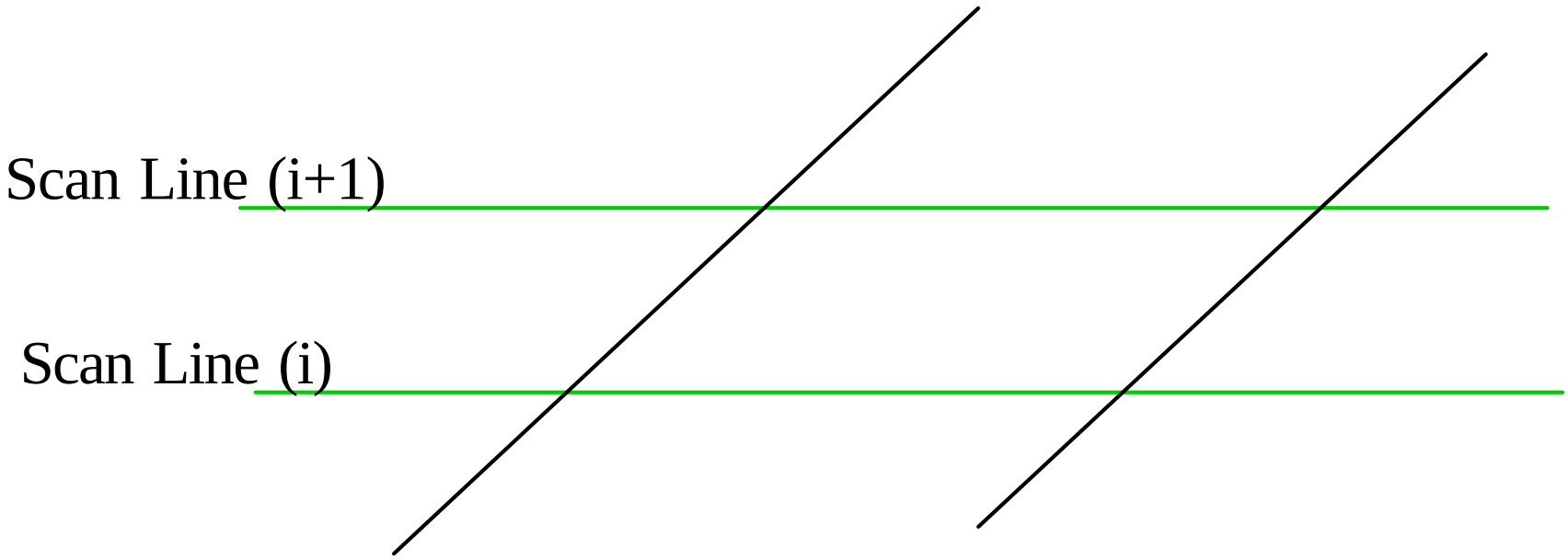
Edge Coherence Property

"**Many of the edges intersected by scan line (i) will also be intersected by scan line (i + 1)**."

Using the ***edge coherence*** property we can save time in computing the intersection of an edge with scan line **(i+1)** if we know:

- **the edge's intersection with scan line (i)**
- **the slope of the line segment m**

Moving from Scan-Line to Scan-Line



$$X_{(i+1)} = X_{(i)} + (1/m)$$

where: $m = dy / dx$

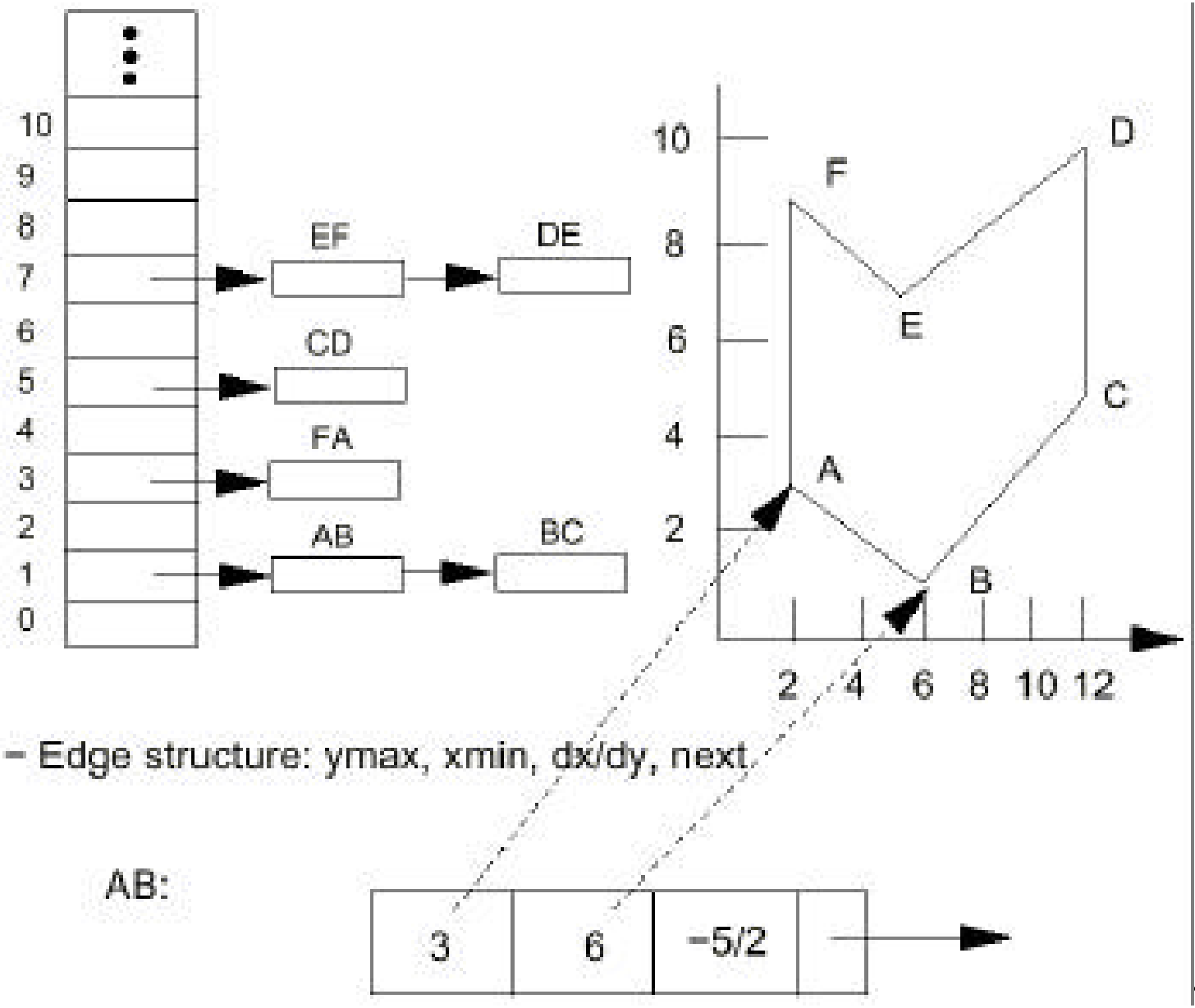
$dy = 1 \Rightarrow dx = 1/m$

Edge Table

- **The Edge Table consists of a series of entries**
- **Each entry is a linked list**

- **All edges are 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**
- **A scan line will have a non-empty linked list entry ONLY if it corresponds to the lower y-coordinate of a line segment**

Edge Table



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

AB:

Entry in the Linked List

- **An *entry* in the linked list contains (if an entry is needed):**
  - The ***larger y-coordinate*** of the ***edge*** (i.e., the ***maximum scan line***)
  - The ***x-coordinate*** of the ***lower*** (***bottom***) end point ($\mathbf{X_{min}}$) (i.e., the **X** value for $\mathbf{Y_{min}}$)
  - The x increment used in stepping from one scan line to the next i.e. 1/m
  - If necessary, a pointer to another entry in the linked list of this scan line
- **Incremental algorithm**
  - Utilization of coherence
    - along the edges
    - on scanlines
    - „sweepline-algorithm“

Use of Active Edge List

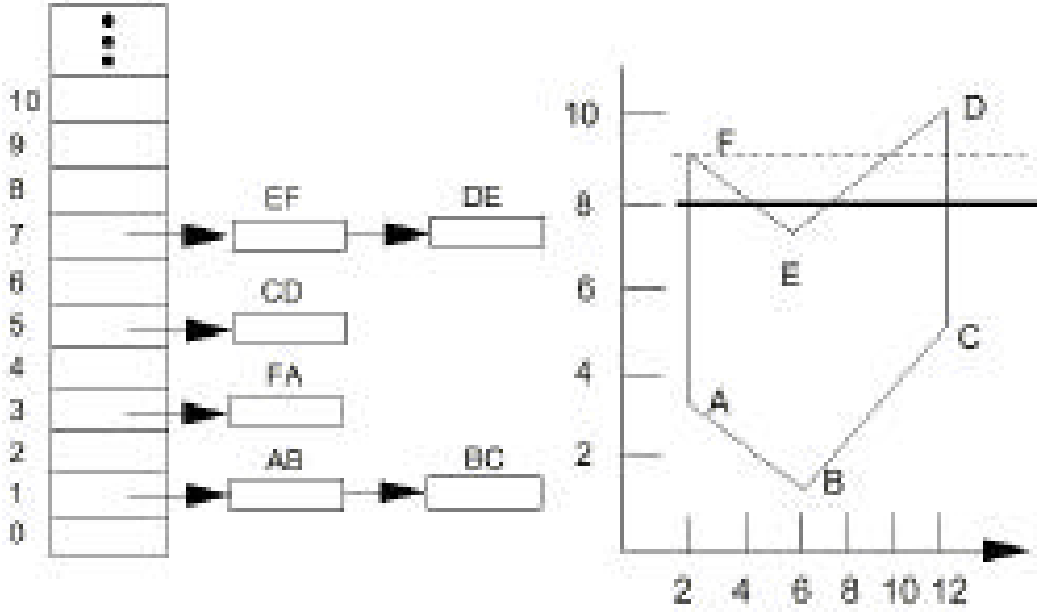
- Having created an **edge table**, we can scan (line by line) using only those edges revelant for that scan line.
- These are held in an ***active edge list*** which is created and maintained from the **edge table**.
- Moving from scan line to scan line, we calculate new x intersections using the equation:

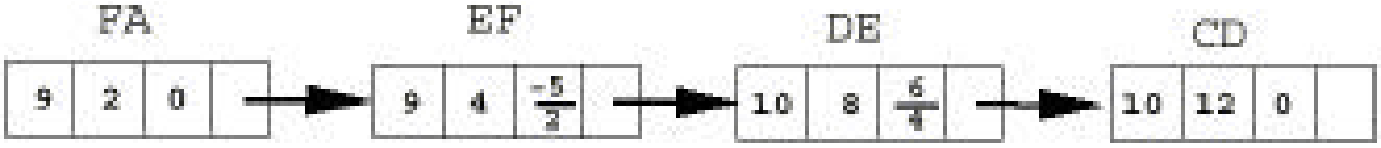
$$x_{(i+1)} = x_{(i)} + 1/m$$

- Any new edges intersected by this next scan line are introduced (from the **edge table**) into the **active edge list** and edges not intersected by this next scan line are removed.
- This may involve sorting.

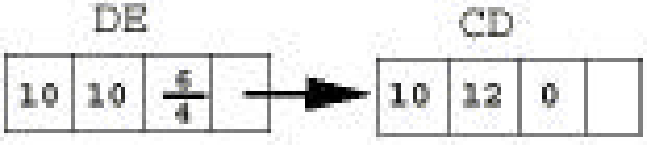
Active Edge List

- **A list of edges active for current scanline, sorted in increasing *x***
- ***Active edge list at***
  - *Y=8*





  - *Y=9*



Polygon Scan-Conversion Algorithm

Construct the Edge Table (ET);

Active Edge Table (AET) = null;

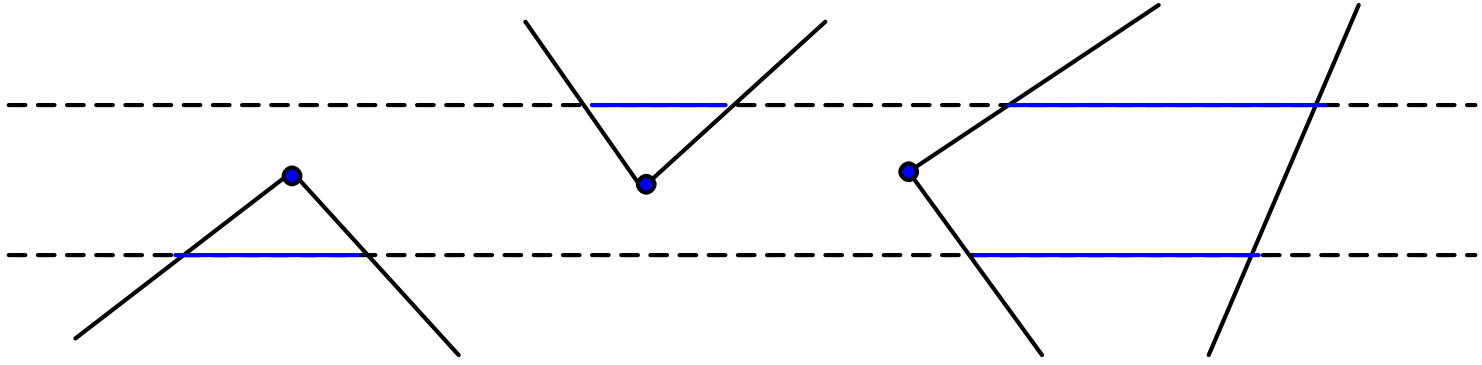
for y = Ymin to Ymax

```
    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

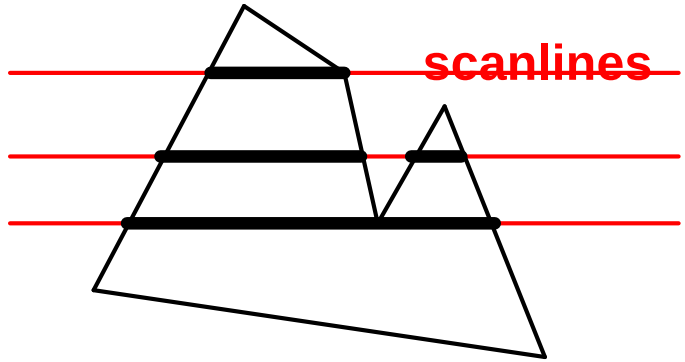
Scanline Algorithm

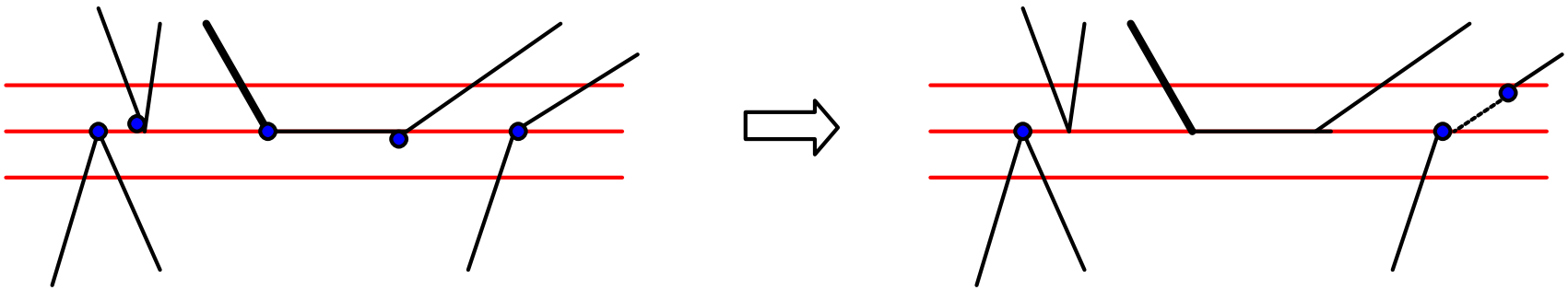
- **For each scan line**
  - Update the Active-Edge-Table
    - Linked-list of entries
      - Link to edge-entries,
      - x, horizontal increment of depth, color, etc
    - Remove edges if their ymax is reached
    - Insert new edges (from Edge-Table)
  - Sorting
    - Incremental update of x
    - Sorting by X-coordinate of the intersection point with scanline
  - Filling the gap between pairs of entries



Polygon Scan-Conversion

- **Special cases**
  - Edge along a scanline
    - *(x+e, y+e)*, shadow test:
      - draw the bottom edge
      - skip the top edge
  - Vertex on a scanline
    - If edges sharing the vertex are located on the **same side** of the scanline – properly handled
    - If edges sharing the vertex are located on **opposite sides** of the scanline – one edge (top) is shortened: y_{min}/y_{max} rule





Wrap-Up

- **Per-pixel processes**
  - ⇒ Exploit pixel coherence
  - – Rasterization
  - – Shading
- **Scanline conversion**
  - – Spans
  - – Linear interpolation
  - – Floating point accuracy
  - – Correct rounding
- **Edge table**
- **Active edge list**