

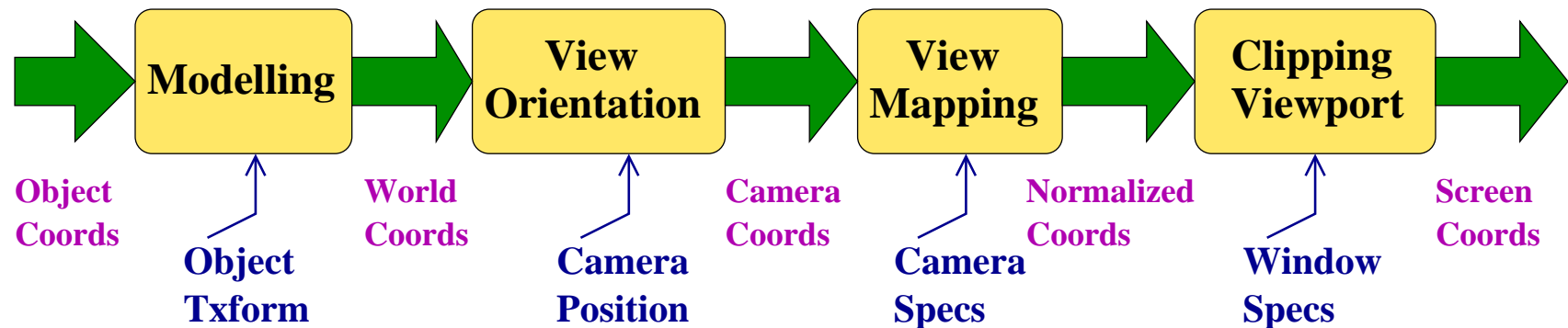
CS3500
Computer Graphics
Module: Scan Conversion

P. J. Narayanan

Spring 2009

Graphics in Practice: Summary

- Basic primitives: Points, Lines, Triangles/Polygons.
- Each constructed fundamentally from points.
- Points can be specified in different coordinate systems.
The pipeline of operations on a point is:



Scan Conversion or Rasterization

- Primitives are defined using points, which have been mapped to the screen coordinates.
- In vector graphics, connect the points using a pen directly.
- In Raster Graphics, we create a discretized image of the whole screen onto the **frame buffer** first. The image is scanned automatically onto the display periodically.
- This step is called **Scan Conversion or Rasterization**.

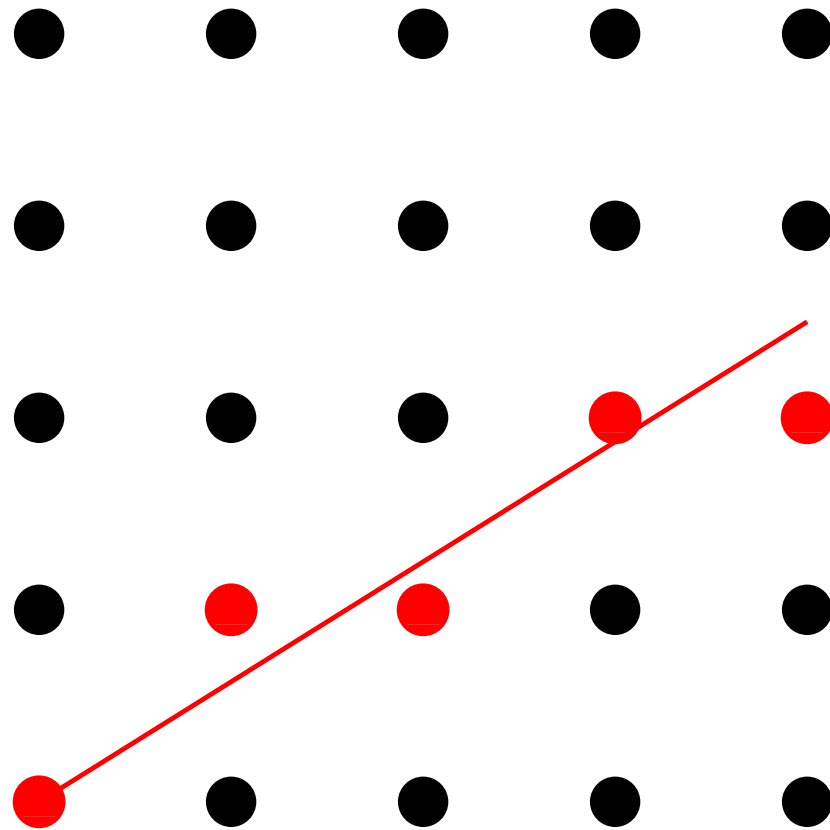
Scan Converting a Point

- The 3D point has been transformed to its screen coordinates (u, v) .
- Round the coordinates to frame buffer array indices (i, j) .
- Current colour is defined/known. Frame buffer array is initialized to the background colour.
- Perform: `frameBuf[i, j] ← currentColour`
- The function `WritePixel(x, y, colour)` does the above.
- If *PointSize* > 1 , assign the colour to a number of points in the neighbourhood!

Scan Converting a Line

- Identify the grid-points that lie on the line and colour them.
- Problem: Given two end-points on the grid, find the pixels on the line connecting them.
- Incremental algorithm or Digital Differential Analyzer (DDA) algorithm.
- Mid-Point Algorithm

Line on an Integer Grid



Incremental Algorithm

Function DrawLine(x_1, y_1, x_2, y_2 , colour)

$\Delta x \leftarrow x_2 - x_1, \Delta y \leftarrow y_2 - y_1, \text{slope} \leftarrow \Delta y / \Delta x$

$x \leftarrow x_1, y \leftarrow y_1$

While ($x < x_2$)

 WritePixel ($x, \text{round}(y)$, colour)

$x \leftarrow x + 1, y \leftarrow y + \text{slope}$

EndWhile

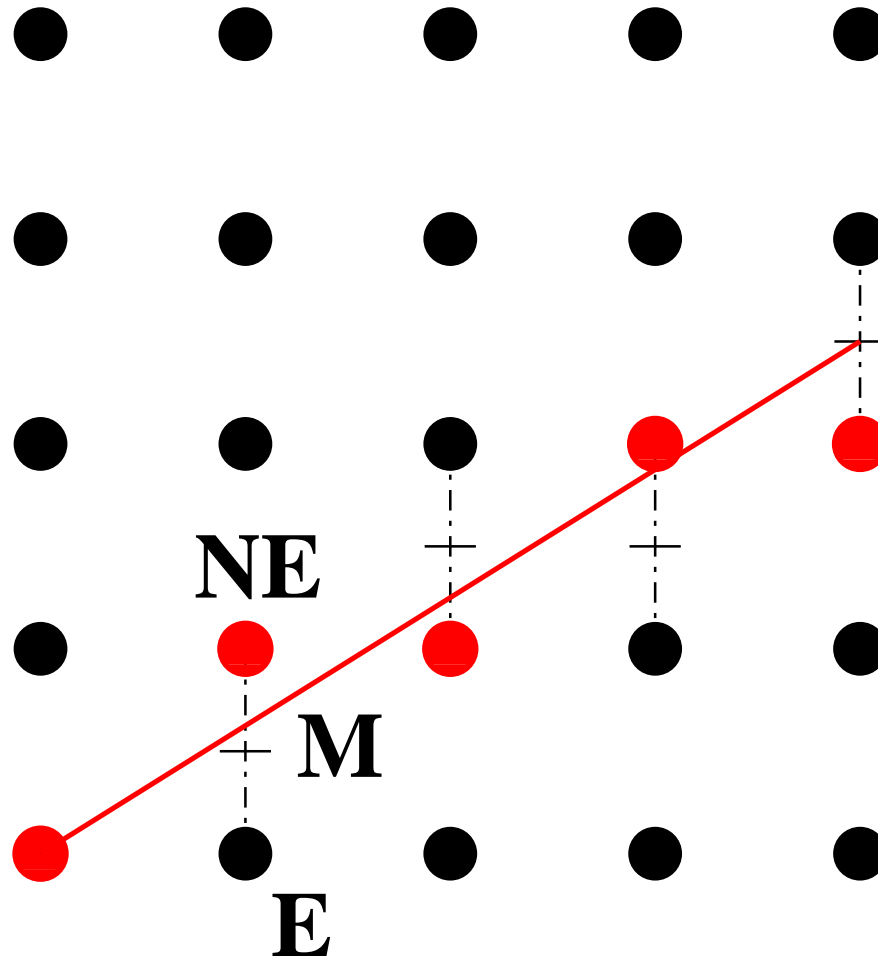
WritePixel (x_2, y_2 , colour)

EndFunction

Points to Consider

- If $\text{abs}(\text{slope}) > 1$, step through y values, adding inverse slopes to x at each step.
- Simple algorithm, easy to implement.
- **Need floating point calculations (add, round), which are expensive.**
- Can we do with integer arithmetic only?
Yes: **Bresenham's Algorithm**
- We will study a simplified version of it called the **Mid-Point Line Algorithm.**

Two Options at Each Step!



Mid-Point Line Algorithm

- Line equation: $ax + by + c = 0$, $a > 0$.
Let $0 < \text{slope} = \Delta y / \Delta x = -a/b < 1.0$
- $F(x, y) = ax + by + c > 0$ for below the line, < 0 for above.
- **NE** if $d = F(\mathbf{M}) > 0$; **E** if $d < 0$; else any!
- $d_{\mathbf{E}} = F(M_{\mathbf{E}}) = d + a$, $d_{\mathbf{NE}} = d + a + b$.
- Therefore, $\Delta_{\mathbf{E}} = a$, $\Delta_{\mathbf{NE}} = a + b$.
- Initial value: $d_0 = F(x_1 + 1, y_1 + \frac{1}{2}) = a + b / 2$
- Similar analysis for other slopes. Eight cases in total.

Pseudocode

```
Function DrawLine ( $l, m, i, j$ , colour)
   $a \leftarrow j - m, b \leftarrow (l - i), x \leftarrow l, y \leftarrow m$ 
   $d \leftarrow 2a + b, \Delta_E \leftarrow 2a, \Delta_{NE} \leftarrow 2(a + b)$ 
  While ( $x < i$ )
    WritePixel( $x, y$ , colour)
    if ( $d < 0$ ) // East
       $d \leftarrow d + \Delta_E, x \leftarrow x + 1$ 
    else // North-East
       $d \leftarrow d + \Delta_{NE}, x \leftarrow x + 1, y \leftarrow y + 1$ 
  EndWhile
  WritePixel( $i, j$ , colour)
EndFunction
```

Example: (10, 10) to (20, 17)

$$F(x, y) = 7x - 10y + 30, \quad a = 7, \quad b = -10$$

$$d_0 = 2 * 7 - 10 = 4, \quad \Delta_{\mathbf{E}} = 2 * 7 = 14, \quad \Delta_{\mathbf{NE}} = -6$$

$$d > 0 : \mathbf{NE} (11, 11), \quad d = 4 + -6 = -2$$

$$d < 0 : \mathbf{E} (12, 11), \quad d = -2 + 14 = 12$$

$$d > 0 : \mathbf{NE} (13, 12), \quad d = 12 + -6 = 6$$

$$d > 0 : \mathbf{NE} (14, 13), \quad d = 6 + -6 = 0$$

$$d = 0 : \mathbf{E} (15, 13), \quad d = 0 + 14 = 14$$

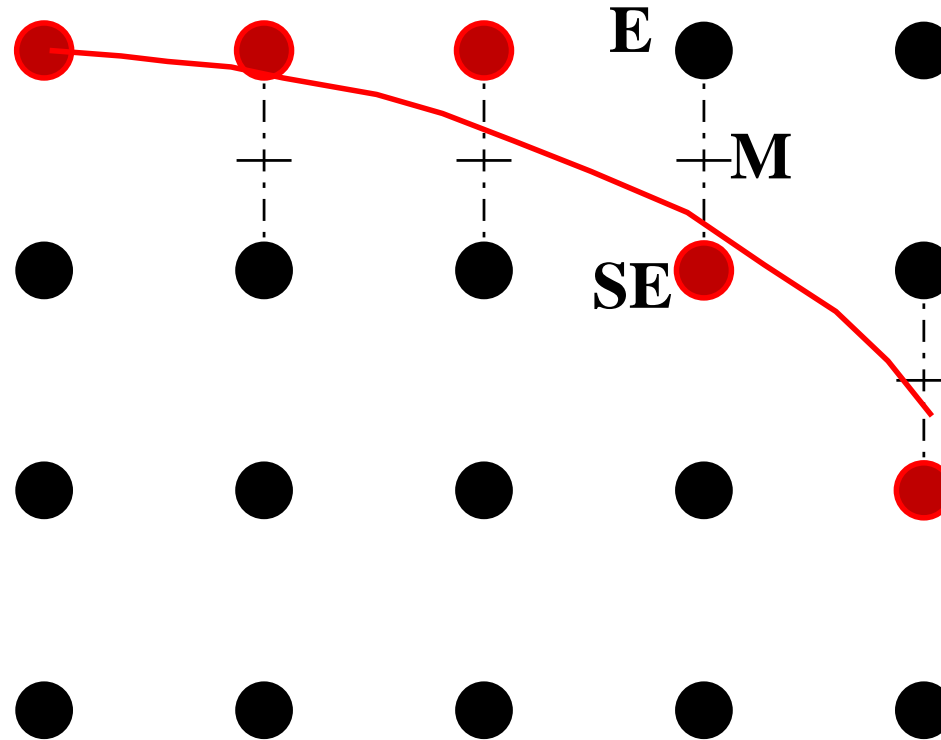
$$d > 0 : \mathbf{NE} (16, 14), \quad d = 14 + -6 = 8$$

Later, **NE** (17, 15), **NE** (18, 16), **E** (19, 16), **NE** (20, 17).

Scan Converting Circles

- Need to consider only with centre at origin: $x^2 + y^2 = r^2$.
- For arbitrary centre, add (x_c, y_c) to each point.
- 8-way symmetry: Only an eighth of the circle need to be scan converted!
- If (x, y) on circle, $(\pm x, \pm y)$, $(\pm y, \pm x)$ are also on the circle!
- Easy way: $y = \sqrt{r^2 - x^2}$, but floating point calculations!

Back to Two Points??



- Choice between E and SE neighbours between the vertical and the 45 degree lines.

Mid-Point Circle Algorithm

- Circle equation: $x^2 + y^2 - r^2 = 0$
- $F(x, y) = x^2 + y^2 - r^2 < 0$ for inside circle, > 0 for outside.
- **SE** if $d = F(\mathbf{M}) > 0$; **E** if $d < 0$; else any!
- $d_{\mathbf{E}} = F(M_{\mathbf{E}}) = d + 2x + 3$, $d_{\mathbf{SE}} = d + 2(x - y) + 5$.
- Therefore, $\Delta_{\mathbf{E}} = 2x + 3$, $\Delta_{\mathbf{SE}} = 2(x - y) + 5$.
- Initial value: $d_0 = F(1, r - \frac{1}{2}) = \frac{5}{4} - r$

Pseudocode

Function DrawCircle (r , colour)

$x \leftarrow 0$, $y \leftarrow r$, $d \leftarrow 1 - r$

CirclePoints (x , y , colour)

While ($x < y$)

 if ($d < 0$) // East

$d \leftarrow d + 2 * x + 3$, $x \leftarrow x + 1$

 else // South-East

$d \leftarrow d + 2 * (x - y) + 5$, $x \leftarrow x + 1$, $y \leftarrow y - 1$

 CirclePoints (x , y , colour)

EndWhile

EndFunction

Eliminate Multiplication?

- **Current selection is E:** What are the new Δ 's?

$$\Delta'_E = 2(x + 1) + 3 = \Delta_E + 2$$

$$\Delta'_{SE} = 2(x + 1 - y) + 5 = \Delta_{SE} + 2$$

- **Current selection is SE:** What are the new Δ 's?

$$\Delta'_E = 2(x + 1) + 3 = \Delta_E + 2$$

$$\Delta'_{SE} = 2(x + 1 - (y - 1)) + 5 = \Delta_{SE} + 4$$

- **if ($d < 0$) // East**

$$d \leftarrow d + \Delta_E, \Delta_E += 2, \Delta_{SE} += 2, x++$$

else // South-East

$$d \leftarrow d + \Delta_{SE}, \Delta_E += 2, \Delta_{SE} += 4, x++, y = y - 1$$

Patterned Line

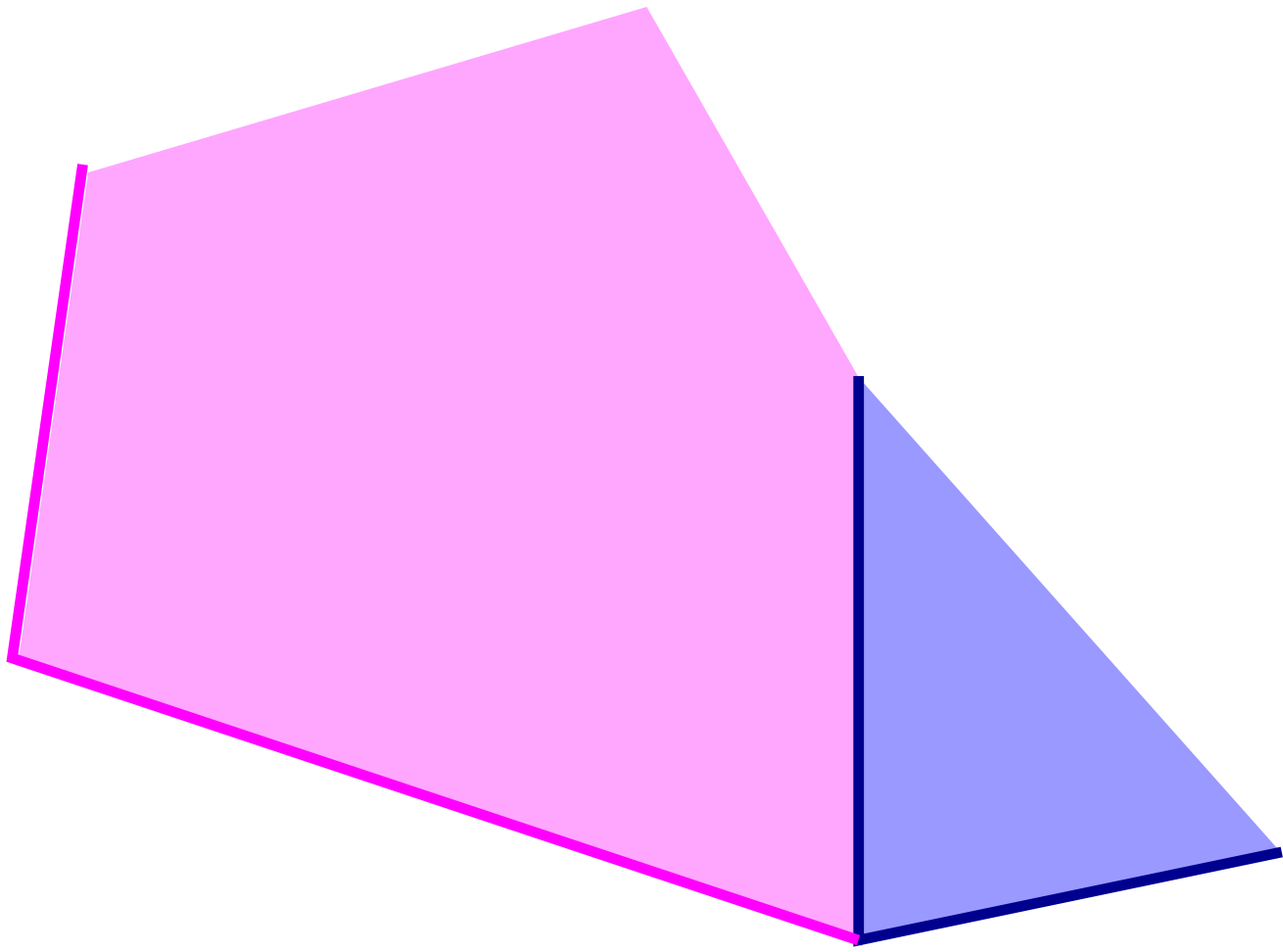
- Represent the pattern as an array of booleans/bits, say, 16 pixels long.
- Fill first half with 1 and rest with 0 for dashed lines.
- Perform WritePixel(x, y) only if pattern bit is a 1.

if (pattern[i]) WritePixel(x, y)

where **i** is an index variable starting with 0 giving the ordinal number (modulo 16) of the pixel from starting point.

Shared Points/Edges

- It is common to have points common between two lines and edges between two polygons.
- They will be scan converted **twice**. Not efficient. Sometimes harmful.
- Solution: Treat the intervals closed on the left and open on the right. $[x_m, x_M)$ & $[y_m, y_M)$
- Thus, edges of polygons on the **top** and **right** boundaries are not drawn.



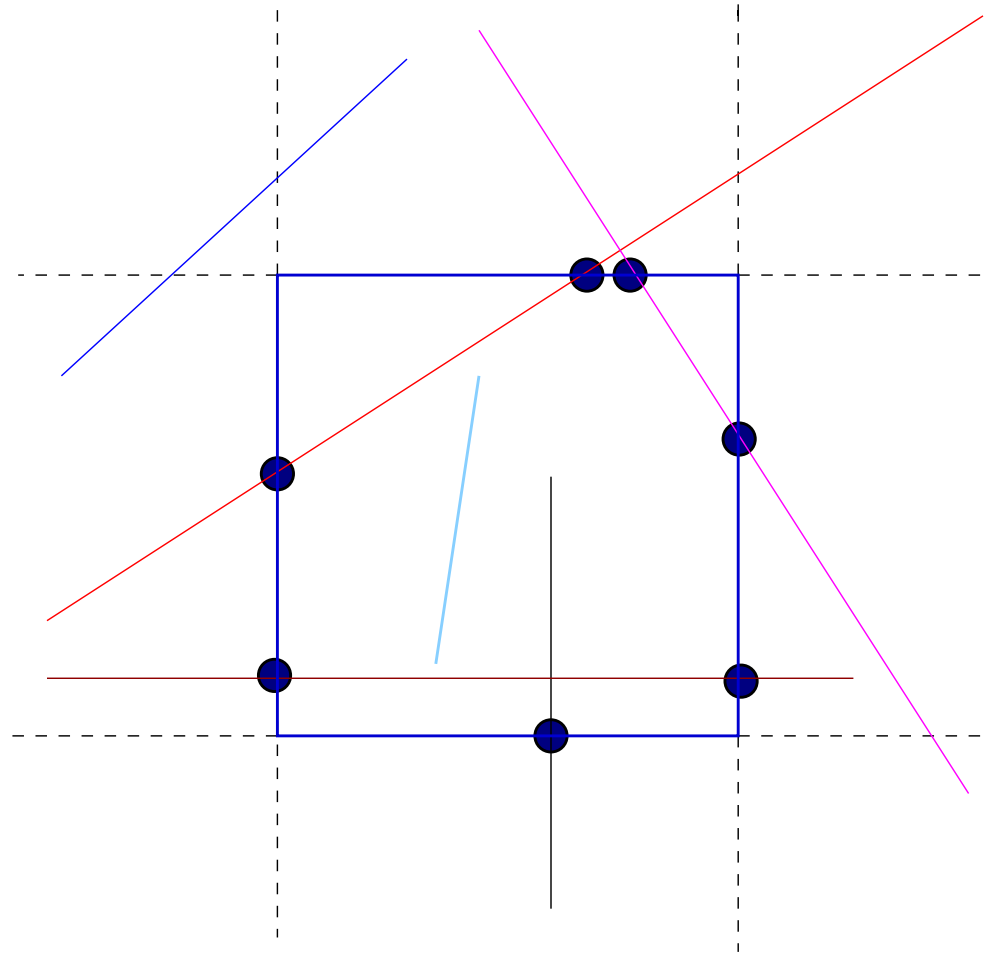
Clipping

- Often, many points map to outside the range in the normalized 2D space.
- Think of the FB as an infinite canvas, of which a small rectangular portion is sent to the screen.
- Let's get greedy: draw only the portion that is visible. That is, **clip** the primitives to a *clip-rectangle*.
- **Scissoring**: Doing scan-conversion and clipping together.

Clipping Points

- Clip rectangle: (x_m, y_m) to (x_M, y_M) .
- For (x, y) : $x_m \leq x \leq x_M, \quad y_m \leq y \leq y_M$
- Can use this to clip any primitives: **Scan convert normally.**
Check above condition before writing the pixel.
- Simple, but perhaps we do more work than necessary.
- Analytically clip to the rectangle, then scan convert.

Clipping Lines



Intersecting Line Segments

- Infinite line equation: $ax + by + c = 0$. Not good for line segments!
- $P = P_1 + \mathbf{t} (P_2 - P_1)$, $0 \leq t \leq 1$.
- Represent sides of clip-rectangles and lines for clipping this way, with two parameters \mathbf{t} and \mathbf{s} . Solve for s, t . Both should be within $[0, 1]$.

Cohen-Sutherland Algorithm

- Identify line segments that can be accepted trivially.
- Identify line segments that can be rejected trivially.
- For the rest, identify the segment that falls within the clip-rectangle.
- For ease of this, assign **outcodes** to each of the 9 regions.

Region Outcodes

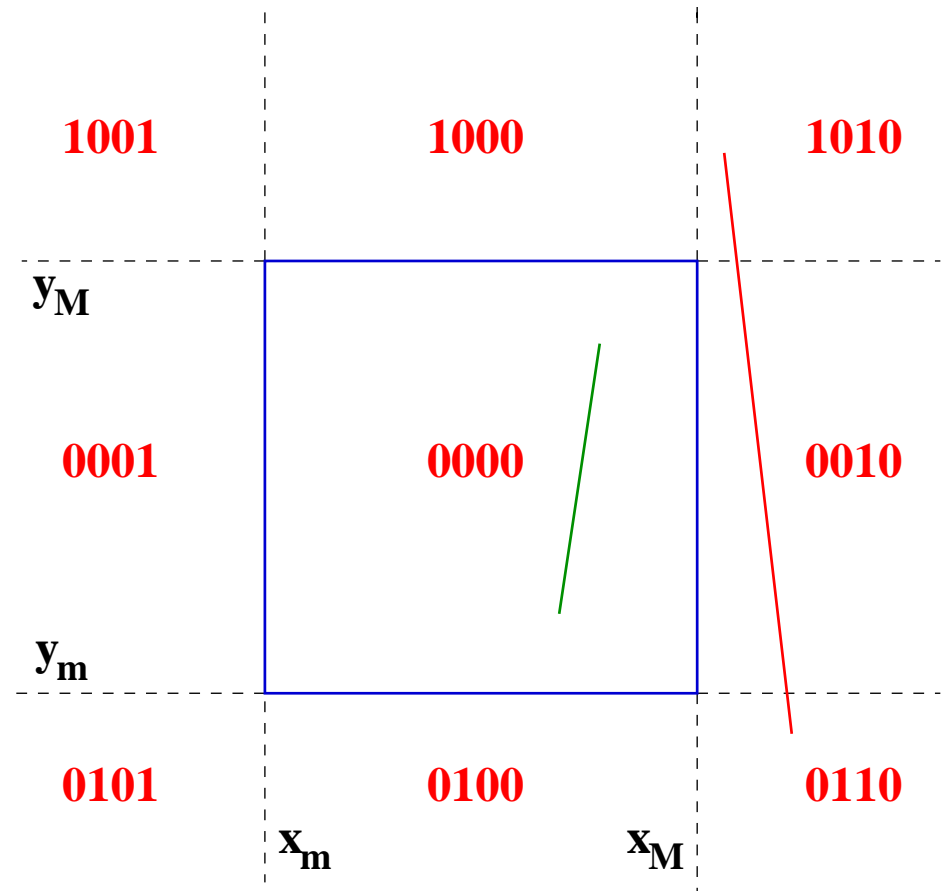
Bits from left to right:

$$y > y_M$$

$$y < y_m$$

$$x > x_M$$

$$x < x_m$$



Overall Algorithm

- Accept: $\text{code1} \mid \text{code0} == 0$
- Reject: $\text{code1} \& \text{code0} != 0$
- Else, identify one of the boundaries crossed by the line segment and clip it to the inside.
- Do it in some order, say, TOP, RIGHT, BOTTOM, LEFT.
- We also have: $\text{TOP} = 1000$, $\text{BOTTOM} = 0100$, $\text{LEFT} = 0001$, $\text{RIGHT} = 0010$

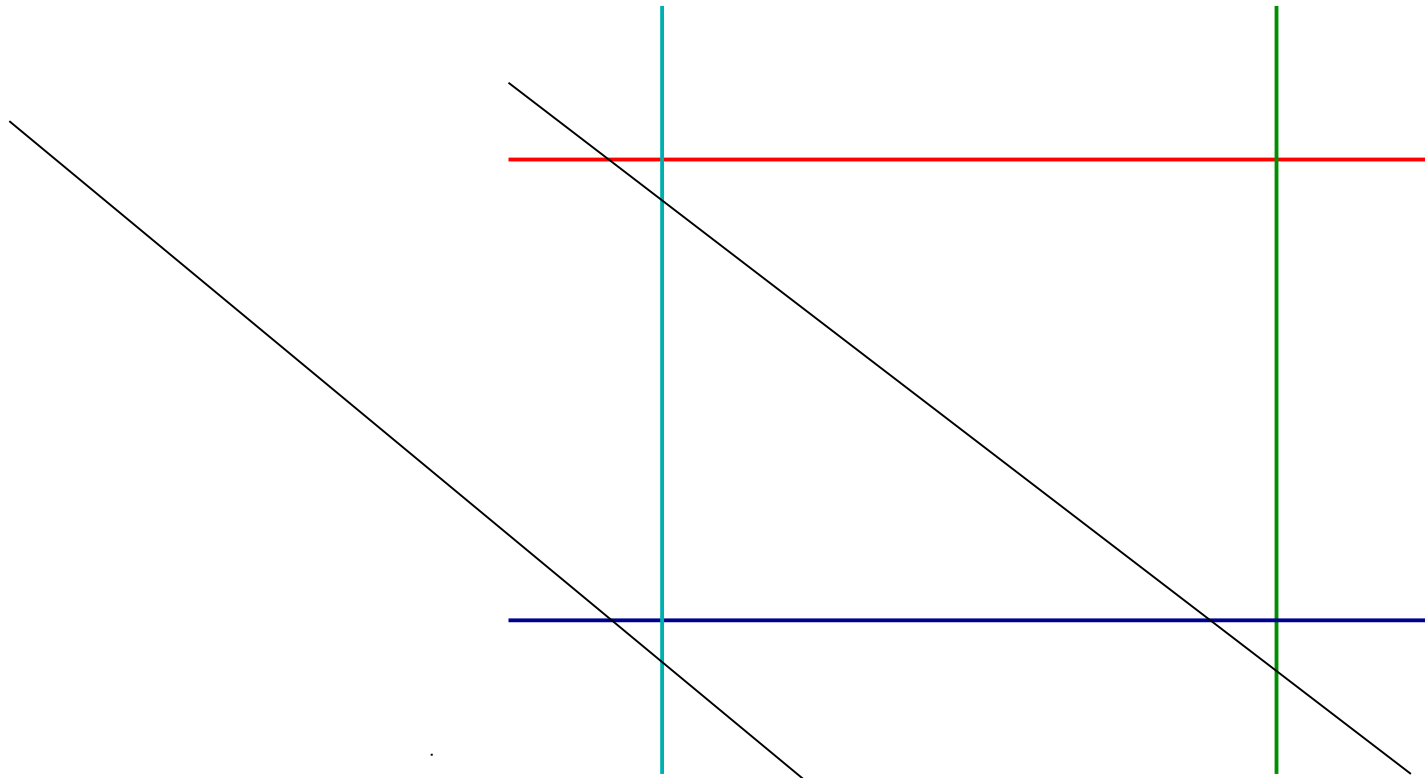
Intersecting with Right/Top

```
if (code & RIGHT) // Intersects right boundary
    // Adjust right boundary to the intersection with  $x_M$ 
     $y \leftarrow y_0 + (y_1 - y_0) * (x_M - x_0) / (x_1 - x_0)$ 
     $x \leftarrow x_M$ 
    ComputeCode( $x, y$ )

if (code & TOP) // Intersects top boundary
    // Adjust top boundary to the intersection with  $y_M$ 
     $x \leftarrow x_0 + (x_1 - x_0) * (y_M - y_0) / (y_1 - y_0)$ 
     $y \leftarrow y_M$ 
    ComputeCode( $x, y$ )
```

Whole Algorithm

```
0 code0 ← ComputeCode(x0, y0), code1 ← ...
1 if (! (code1 | code0)) Accept and Return
2 if (code1 & code0) Reject and Return
3 code ← code1 ? code1 : code0
4 if (code & TOP) Intersect with  $y_M$  line.
5 elsif (code & RIGHT) Intersect with  $x_M$  line.
6 elsif (code & BOTTOM) Intersect with  $y_m$  line.
7 elsif (code & LEFT) Intersect with  $x_m$  line.
8 if (code == code1) Replace EndPoint1.
9 else Replace EndPoint0.
10 Goto step 1.
```



4 to accept and 3 to reject.

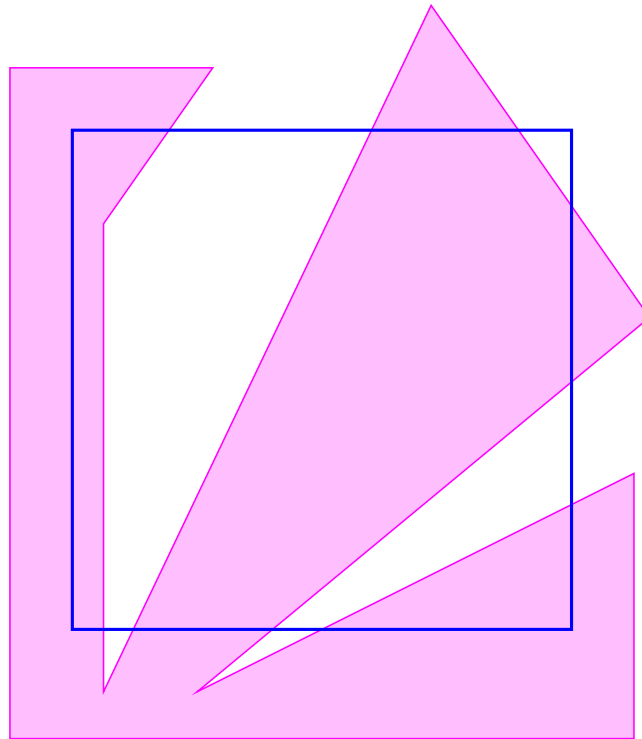
Discussion

- Simple logical operations to check intersections etc.
- Not efficient, as external intersections are not eliminated.
- In the worst case, 3 intersections may be computed and then the line segment could be rejected.
- 4 intersections may be computed before accepting a line segment.

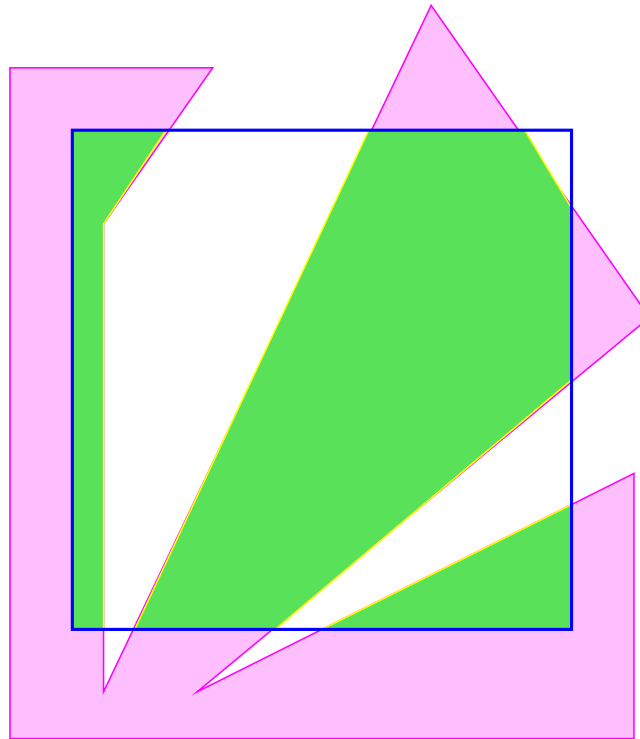
Clipping Polygons

- Restrict drawing/filling of a polygon to the inside of the clip rectangle.
- A convex polygon remains convex after clipping.
- A concave polygon can be clipped to multiple polygons.
- Can perform by intersecting to the four clip edges in turn.

An Example



An Example



Sutherland-Hodgman Algorithm

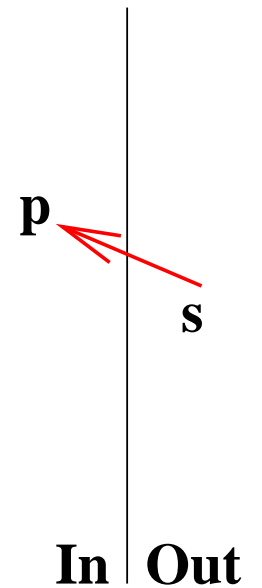
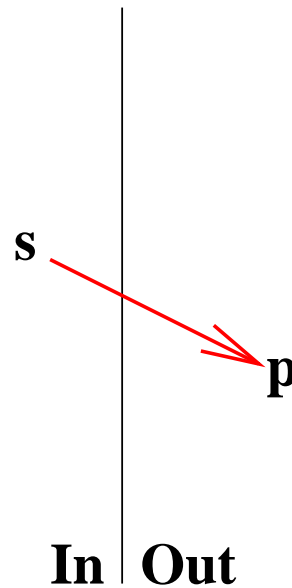
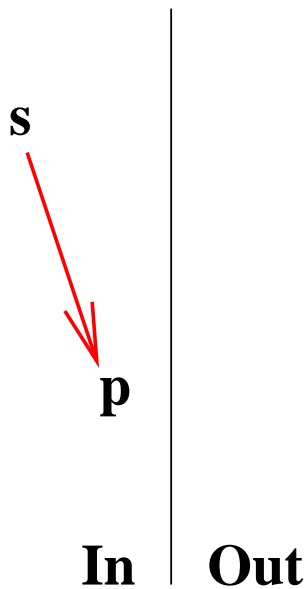
- *Input:* A list of vertices v_1, v_2, \dots, v_n . Implied edges from v_i to v_{i+1} and from v_n to v_1 .
- *Output:* Another list of vertices giving the clipped polygon.
- *Method:* Clip the entire polygon to the infinite line for each clip edge in turn.
- Four passes, the output of each is a partially clipped polygon used as input to the next.
- Post-processing to eliminate degenerate edges.

Algorithm Detail

- Process edges one by one and clip it to a line.
- Start with the edge $E(v_n, v_1)$.
- Compare the current edge $E(v_{i-1}, v_i)$ with the current clip line. Clip it to lie within the clip rectangle.
- Repeat for the next edge $E(v_i, v_{i+1})$. Till all edges are processed.
- When processing $E(v_{i-1}, v_i)$, treat v_{i-1} as the **in** vertex and v_i as the **out** vertex.

At Each Step ...

- **in** vertex: **s** (already handled). **out** vertex: **p**. Four cases:



Function SuthHodg()

```
p ← last(inVertexList) // Copy, not remove
while (notEmpty(inVertexList))
  s ← p, p ← removeNext(inVertexList)
  if (inside(p, clipBoundary))
    if (inside(s, clipBoundary))
      addToList(p, outVertexList) // Case 1
    else i ← intersect(s, p, clipBoundary) // Case 4
      addToList(i, outVertexList), addToList(p, outVertexList)
  elseif (inside(s, clipBoundary)) // Case 2
    addToList(intersect(s, p, clipBoundary), outVertexList)
```

Complete Algorithm

- Invoke `SuthHodg()` 4 times for each clip edge as `clipBoundary`.
- The `outVertexList` after one run becomes the `inVertexList` for the next.
- Uses list data structures to implement polygons.
- Function `inside()` determines if a point is in the **inside** of the clip-boundary. We can define it as “being on the left when looking from first vertex to the second”.
- Can be extended to clip to any convex polygonal region!

Filled Rectangles

- Write to all pixels within the rectangle.

Function FilledRectangle ($x_m, x_M, y_m, y_M, \text{colour}$)

 for $x_m \leq x \leq x_M$ do

 for $y_m \leq y \leq y_M$ do

 WritePixel (x, y, colour)

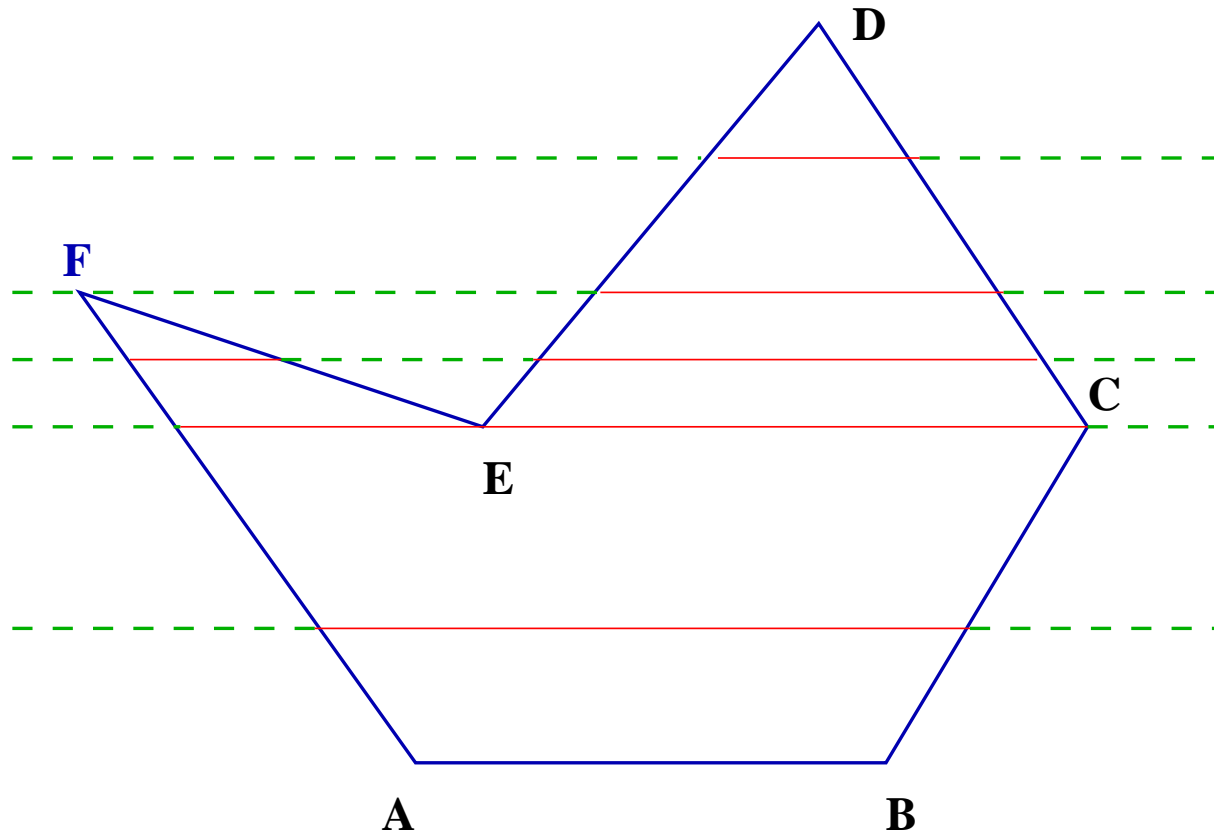
EndFunction

- How about non-upright rectangles? General polygons?

Filled Polygons

- For each scan line, identify **spans** of the polygon interior. Strictly interior points only.
- For each scan line, the **parity** determines if we are inside or outside the polygon. Odd is inside, Even is outside.
- Trick: End-points count towards parity enumeration only if it is a **y_{\min}** point.
- Span extrema points and other information can be computed during scan conversion. This information is stored in a suitable data structure for the polygon.

Parity Checking



Edge Coherence

- If scan line y intersects with an edge E , it is likely that $y + 1$ also does. (Unless intersection is the y_{\max} vertex.)
- When moving from y to $y + 1$, the X -coordinate goes from x to $x + 1/m$. $1/m = (x_2 - x_1)/(y_2 - y_1) = \Delta x / \Delta y$
- Store the integer part of x , the numerator (Δx) and the denominator (Δy) of the fraction separately.
- For next scan line, add Δx to numerator. If sum goes $> \Delta y$, increment integer portion, subtract Δy from numerator.

Scan Converting Filled Polygons

- Find intersections of each scan line with polygon edges.
- Sort them in increasing X -coordinates.
- Use parity to find interior spans and fill them.
- Most information can be computed during scan conversion.
A list of intersecting polygons stored for each scan line.
- Use edge coherence for the computation otherwise.

Special Concerns

- Fill only strictly interior pixels: Fractions rounded up when even parity, rounded down when odd.
- Intersections at integer pixels: Treat interval closed on left, open on right.
- Intersections at vertices: Count only y_m vertex for parity.
- Horizontal edges: Do not count as y_m !

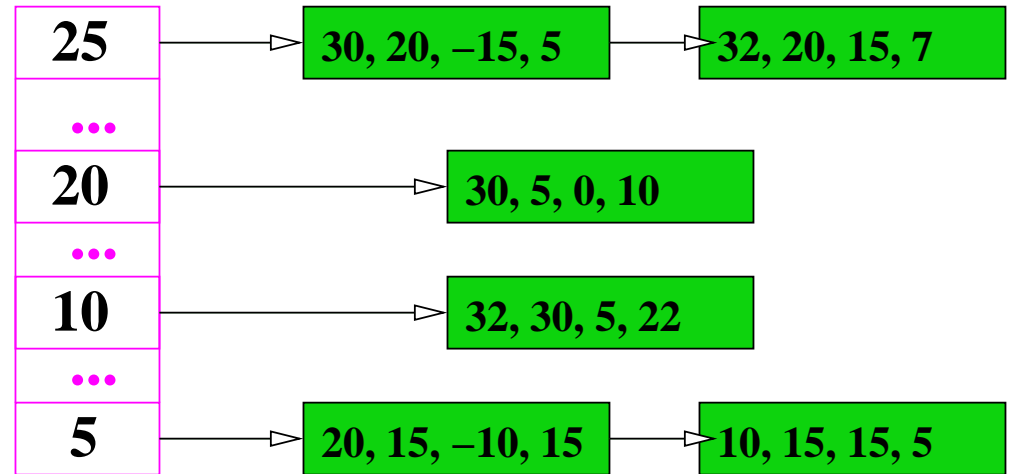
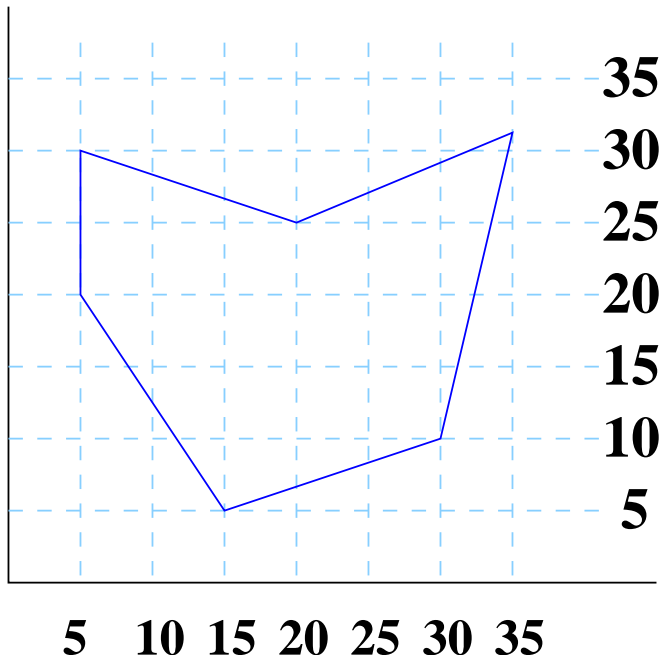
Filled Polygon Scan Conversion

- Perform all of it together. Each scan line should not be intersected with each polygon edge!
- Edges are known when polygon vertices are mapped to screen coordinates.
- Build up an edge table while that is done.
- Scan conversion is performed in the order of scan lines. Edge coherence can be used; an active edge table can keep track of which edges matter for the current scan line.

Edge Table for a Polygon

- Construct a bucket-sorted table of edges, sorted into buckets of y_m for the edge.
- Each bucket y contains a list of edges with $y = y_m$, in the increasing order of x coordinate of the lower end point.
- Each edge is represented by its y_M for the edge, x of the lower (that is y_m) point, and the slope as a rational number.
- This is the basis for constructing the Active Edge Table to compute the spans.

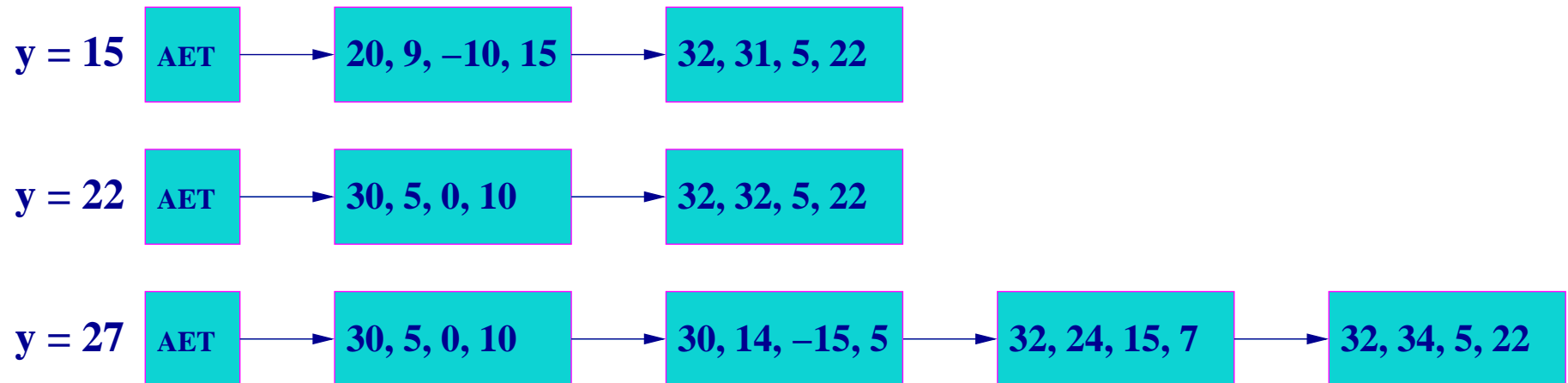
Polygon and Edge Table



Active Edge Tables

- Start with the lowest y value and an empty AET.
- Insert edges from bucket y of ET to AET.
(They have $y = y_m$ and are sorted on x .)
- Remove edges from AET where y is the y_M point.
- Between pairs of AET entries lie spans. Fill them.
- Compute next point on edge using coherence. (Increment y by 1 and numerator by Δx , etc. Or vice versa)
- Continue above 4 steps till ET and AET are empty.

Active Edge Table: Snapshots



Pattern Filling

- A rectangular bit-map with the desired pattern can be used to fill the interior with a pattern.
- If $pattern(i \bmod M, j \bmod N)$, draw pixel, else ignore.
- i, j are row, col indices. Lower left corner at 0 and 0.
- M, N are the pattern height and width.

Scan Conversion: Summary

- Filling the frame buffer given 2D primitives.
- Convert an analytical description of the basic primitives into pixels on an integer grid in the frame buffer.
- Lines, Polygons, Circles, etc. Filled and unfilled primitives.
- Efficient algorithms required since scan conversion is done repeatedly.
- 2D Scan Conversion is all, even for 3D graphics.

Scan Conversion: Summary

- High level primitives (point, line, polygon) map to window coordinates using transformations.
- Creating the display image on the Frame Buffer is important. Needs to be done efficiently.
- Clipping before filling FB to eliminate futile effort.
- After clipping, line remains line, polygons can become polygons of greater number of sides, etc.
- General polygon algorithm for clipping and scan conversion are necessary.