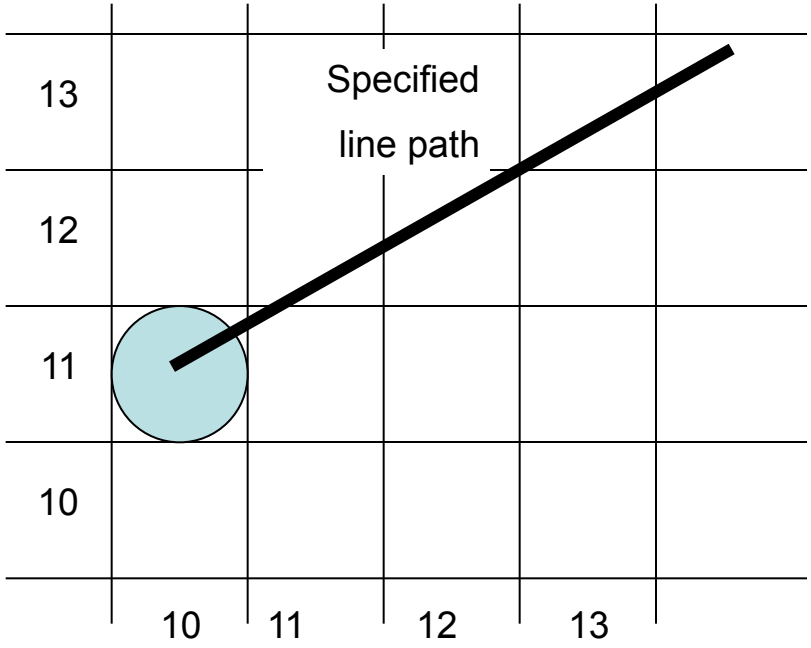
Bresenham's line algorithm

- Accurate and efficient
- Uses only incremental integer calculations

The method is described for a line segment with a positive slope less than one

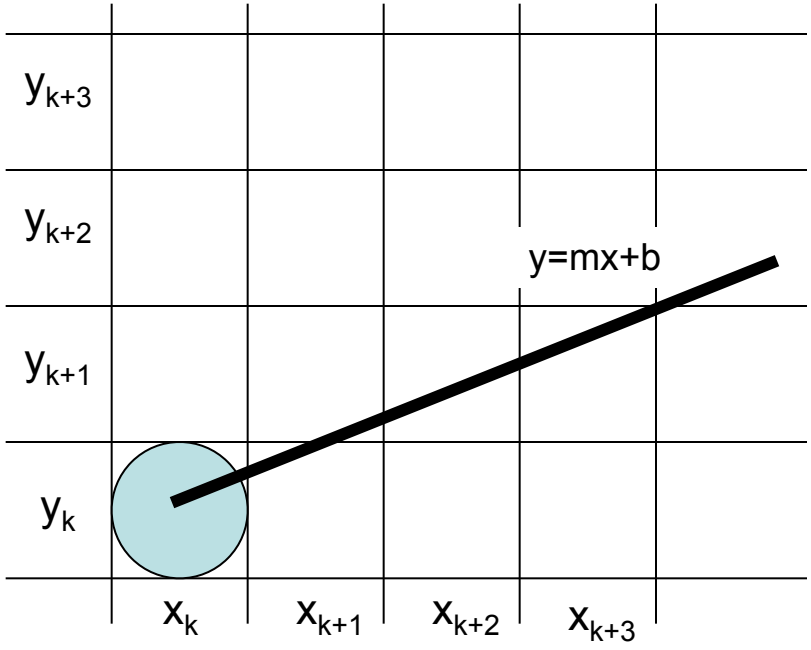
The method generalizes to line segments of other slopes by considering the symmetry between the various octants and quadrants of the xy plane

Bresenham's line algorithm



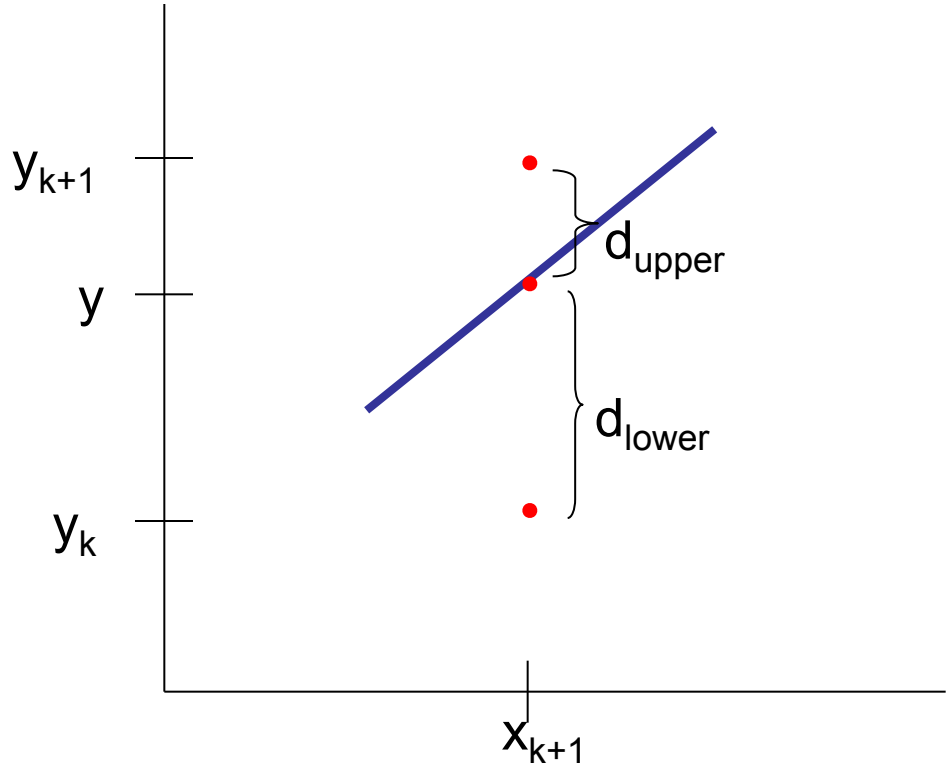
- Decide what is the next pixel position
  - (11,11) or (11,12)

Illustrating Bresenham's Approach



- For the pixel position $x_{k+1}=x_k+1$, which one we should choose:
- (x_{k+1},y_k) or (x_{k+1}, y_{k+1})

Bresenham's Approach



- $y=m(x_k + 1)+b$
- $d_{lower}=y-y_k$

$=m(x_k + 1)+b-y_k$

- $d_{upper}=(y_k+1)-y$

$= y_k+1 -m(x_k + 1)-b$

- $d_{lower}- d_{upper}= 2m(x_k + 1)+2y_k+2b-1$
- Rearrange it to have integer calculations:

  $m=\Delta y/\Delta x$

  Decision parameter: $p_k= \Delta x(d_{lower}- d_{upper})=2\Delta y.x_k - 2\Delta x. y_k + c$

The Decision Parameter

Decision parameter: $p_k = \Delta x(d_{lower} - d_{upper}) = 2\Delta y.x_k - 2\Delta x.\ y_k + c$

- p_k has the same sign with $d_{lower} - d_{upper}$ since $\Delta x > 0$.
- c is constant and has the value $c = 2\Delta y + \Delta x(2b-1)$
  - c is independent of the pixel positions and is eliminated from decision parameter p_k.

- If $d_{lower} < d_{upper}$ then p_k is negative.
  - Plot the lower pixel (East)
- Otherwise
  - Plot the upper pixel (North East)

Succesive decision parameter

- At step k+1

  $p_{k+1} = 2\Delta y.x_{k+1} - 2\Delta x.\ y_{k+1} + c$

- Subtracting two subsequent decision parameters yields:

  $p_{k+1} - p_k = 2\Delta y.(x_{k+1} - x_k) - 2\Delta x.\ (y_{k+1} - y_k)$

- $x_{k+1} = x_k + 1$ so

  $p_{k+1} = p_k + 2\Delta y - 2\Delta x.\ (y_{k+1} - y_k)$

  - $y_{k+1} - y_k$ is either 0 or 1 depending on the sign of p_k
- First parameter p_0
  - $p_0 = 2\,\Delta y - \Delta x$

Bresenham's Line-Drawing Algorithm for I m I < **1**

1. Input the twoline endpoints and store the left endpoint in (x_0, y_0).

2. Load (x_0, y_0) into the frame buffer; that is, plot the first point.

3. Calculate constants Δx, Δy, $2\Delta y$, and $2\Delta y - 2\Delta x$, and obtain the starting value for the decision parameter as

   $$p_0 = 2\,\Delta y - \Delta x$$

4. At each x_k along the line, starting at k = 0, perform the following test:

   If $p_k < 0$, the next point to plot is (x_{k+1}, y_k) and

   $$p_{k+1} = p_k + 2\Delta y$$

   Otherwise, the next point to plot is (x_{k+1}, y_{k+1}) and

   $$p_{k+1} = p_k + 2\Delta y - 2\Delta x$$

5. Repeat step 4 Δx -1 times.

Trivial Situations: Do not need Bresenham

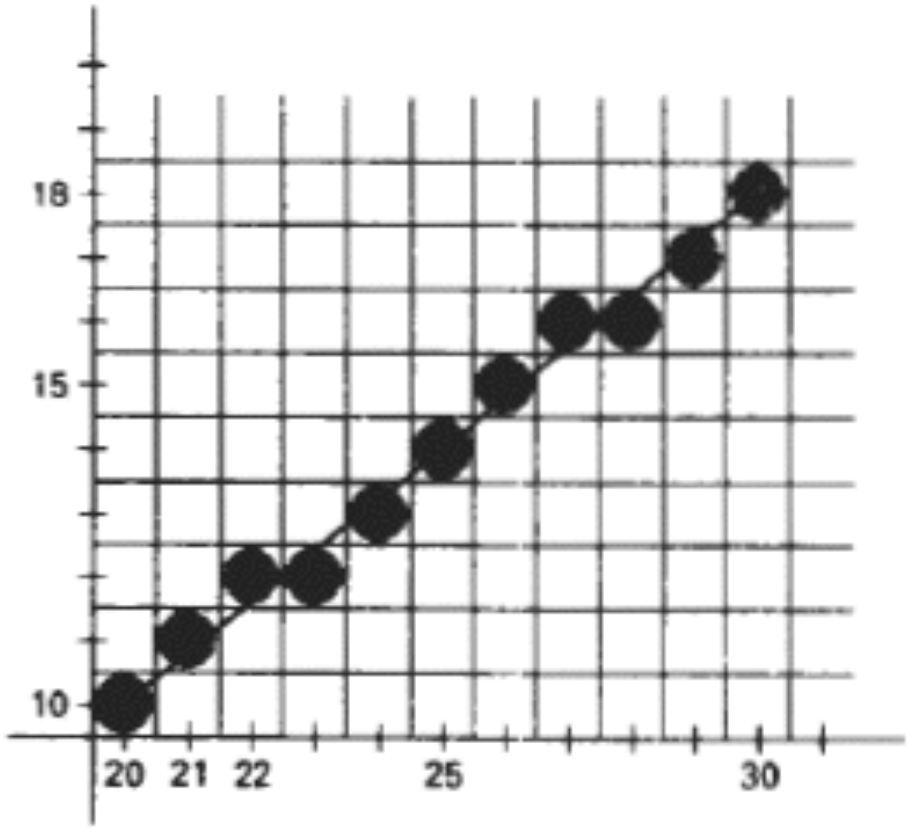
- $m = 0 \quad \Rightarrow \quad$ horizontal line
- $m = \pm 1 \quad \Rightarrow \quad$ line $y = \pm x$
- $m = \infty \quad \Rightarrow \quad$ vertical line

Example

- Draw the line with endpoints (20,10) and (30, 18).
  - **Δx=30-20=10, Δy=*18-10*=8,**
  - $p_0 = 2\Delta y - \Delta x$=16-10=6
  - ***2Δy=16,*** and ***2Δy*** - **2Δx=-4**
- Plot the initial position at (20,10), then

k	p_k	(x_{k+1}, y_{k+1})	k	p_k	(x_{k+1}, y_{k+1})
0	6	(21, 11)	5	6	(26, 15)
1	2	(22, 12)	6	2	(27, 16)
2	−2	(23, 12)	7	−2	(28, 16)
3	14	(24, 13)	8	14	(29, 17)
4	10	(25, 14)	9	10	(30, 18)

k	p_k	(x_{k+1}, y_{k+1})	k	p_k	(x_{k+1}, y_{k+1})
0	6	(21, 11)	5	6	(26, 15)
1	2	(22, 12)	6	2	(27, 16)
2	−2	(23, 12)	7	−2	(28, 16)
3	14	(24, 13)	8	14	(29, 17)
4	10	(25, 14)	9	10	(30, 18)



Example

- Line end points: $(x_0, y_0) = (5,8); \quad (x_1, y_1) = (9,11)$
- Deltas:

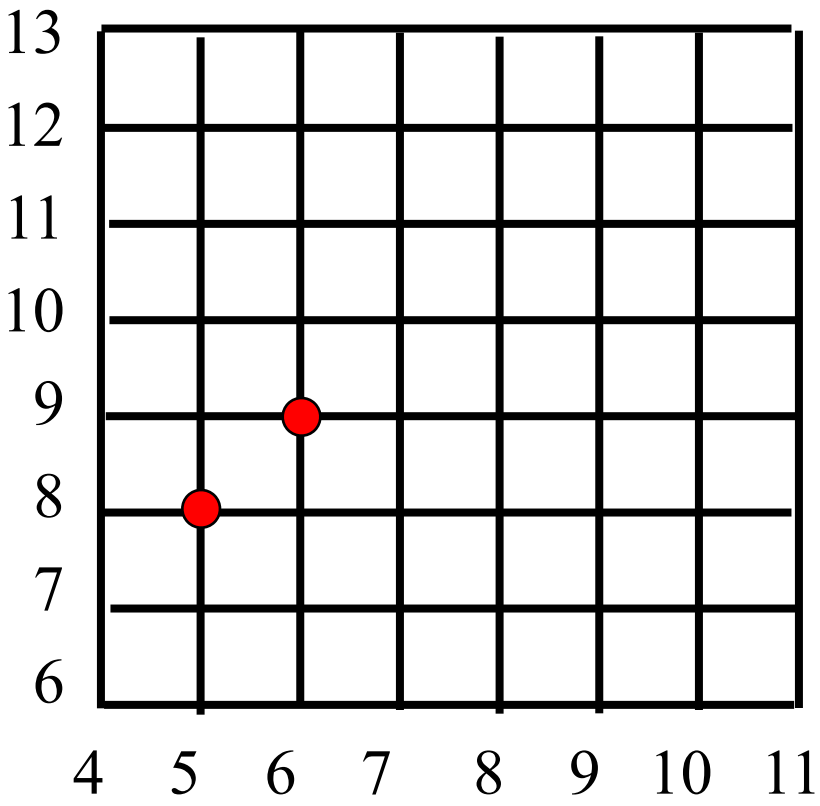
$$dx = 4; dy = 3$$

initially $p(5,8) = 2(dy)\text{-}(dx)$

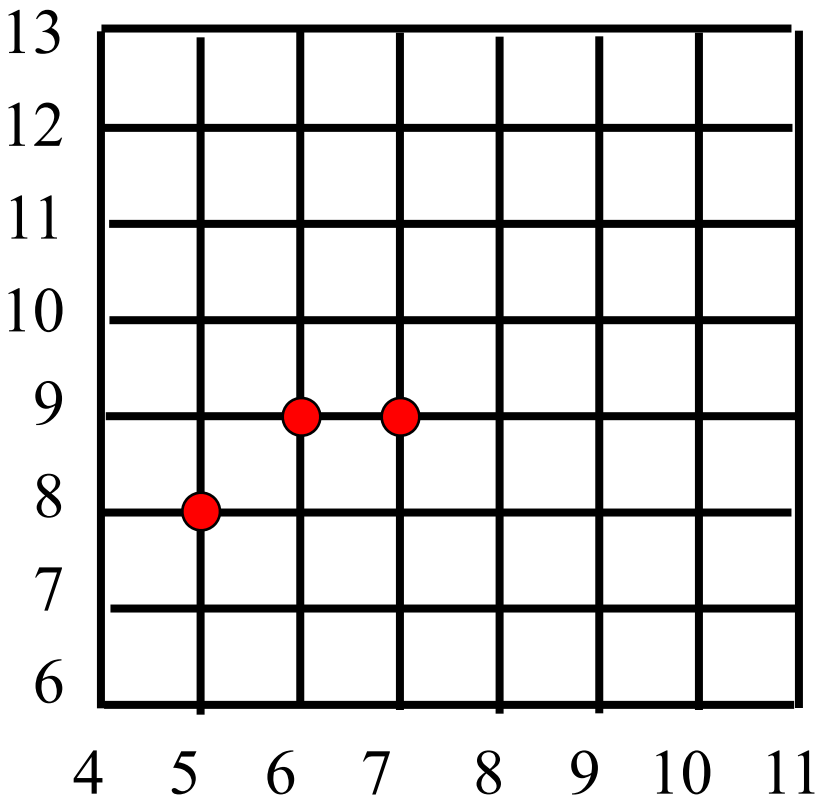
$$= 6 - 4 = 2 > 0$$

$$p = 2 \quad \Rightarrow \quad NE$$

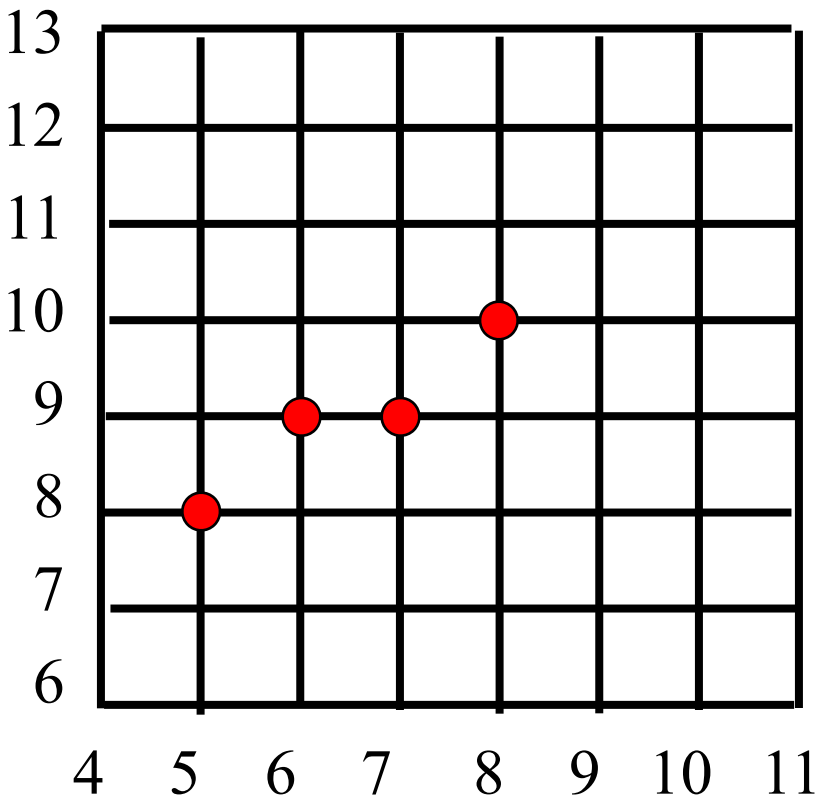
Graph



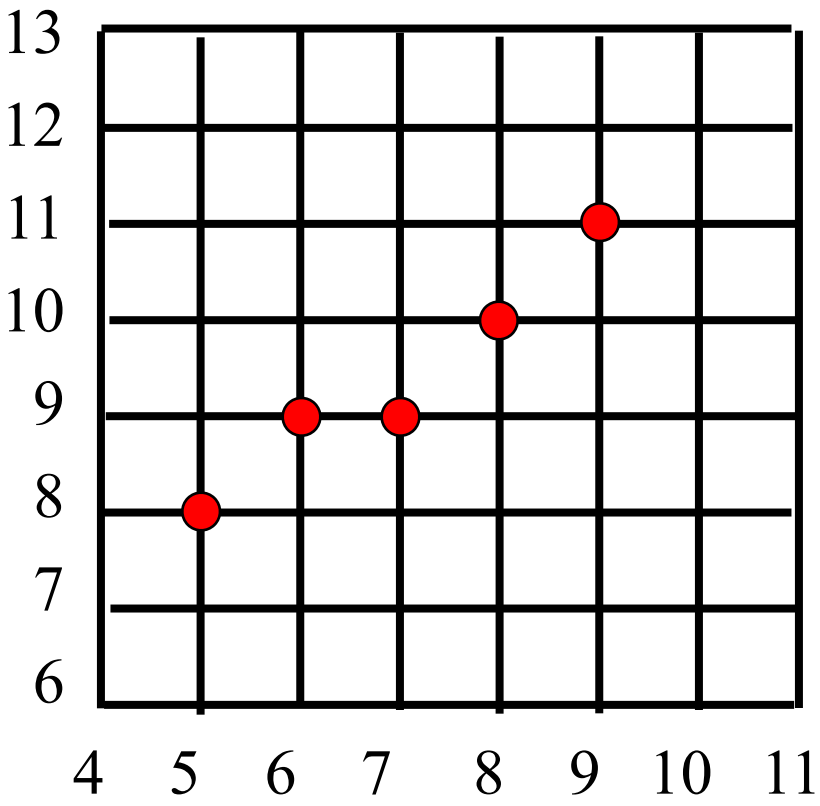
Continue the process...



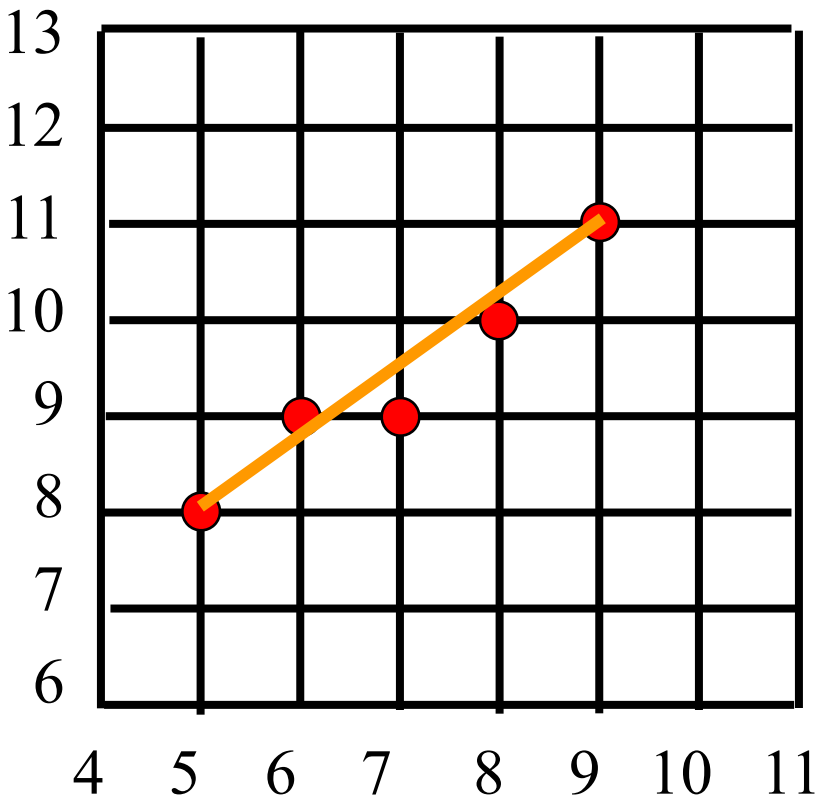
Graph



Graph



Graph

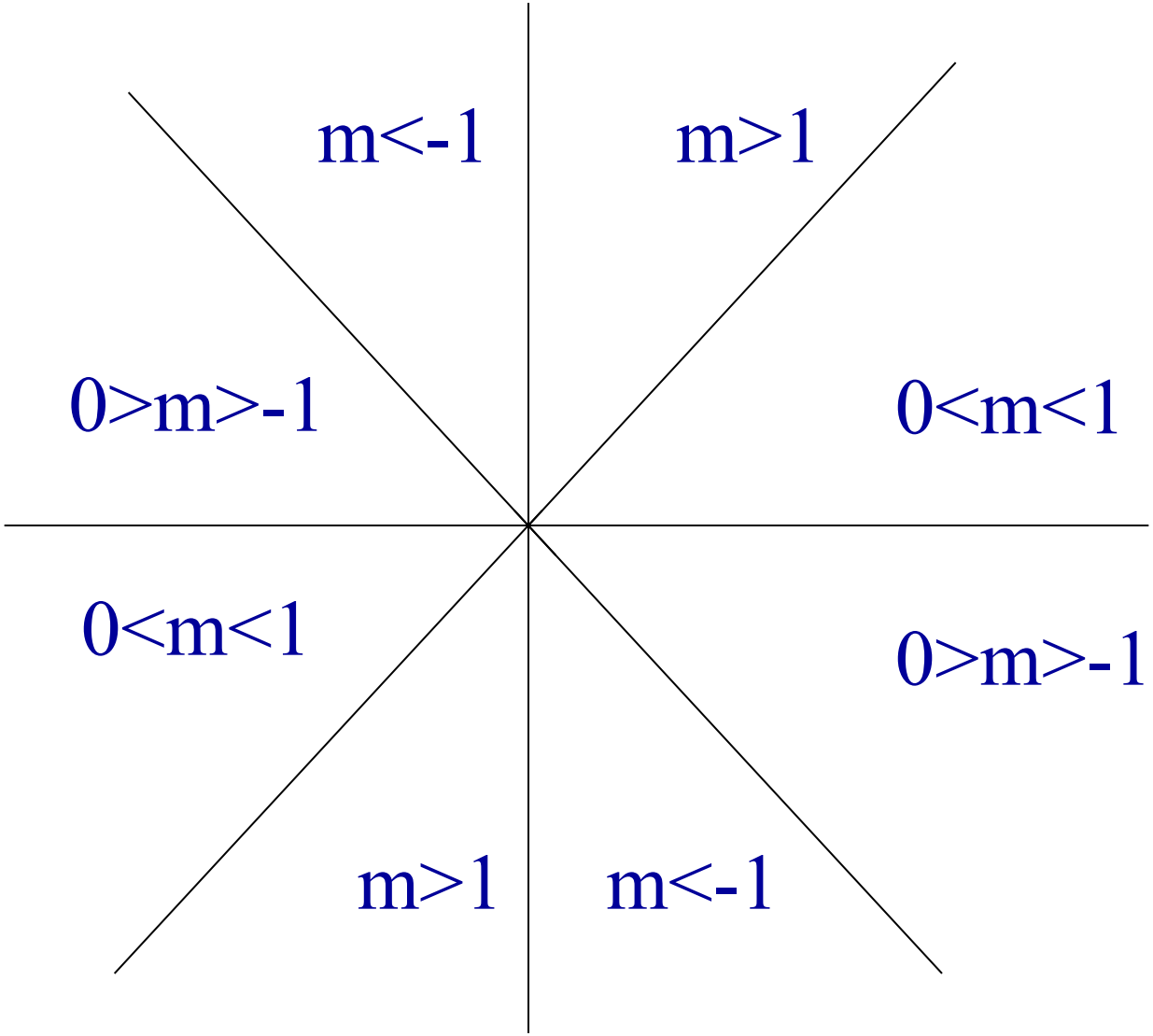


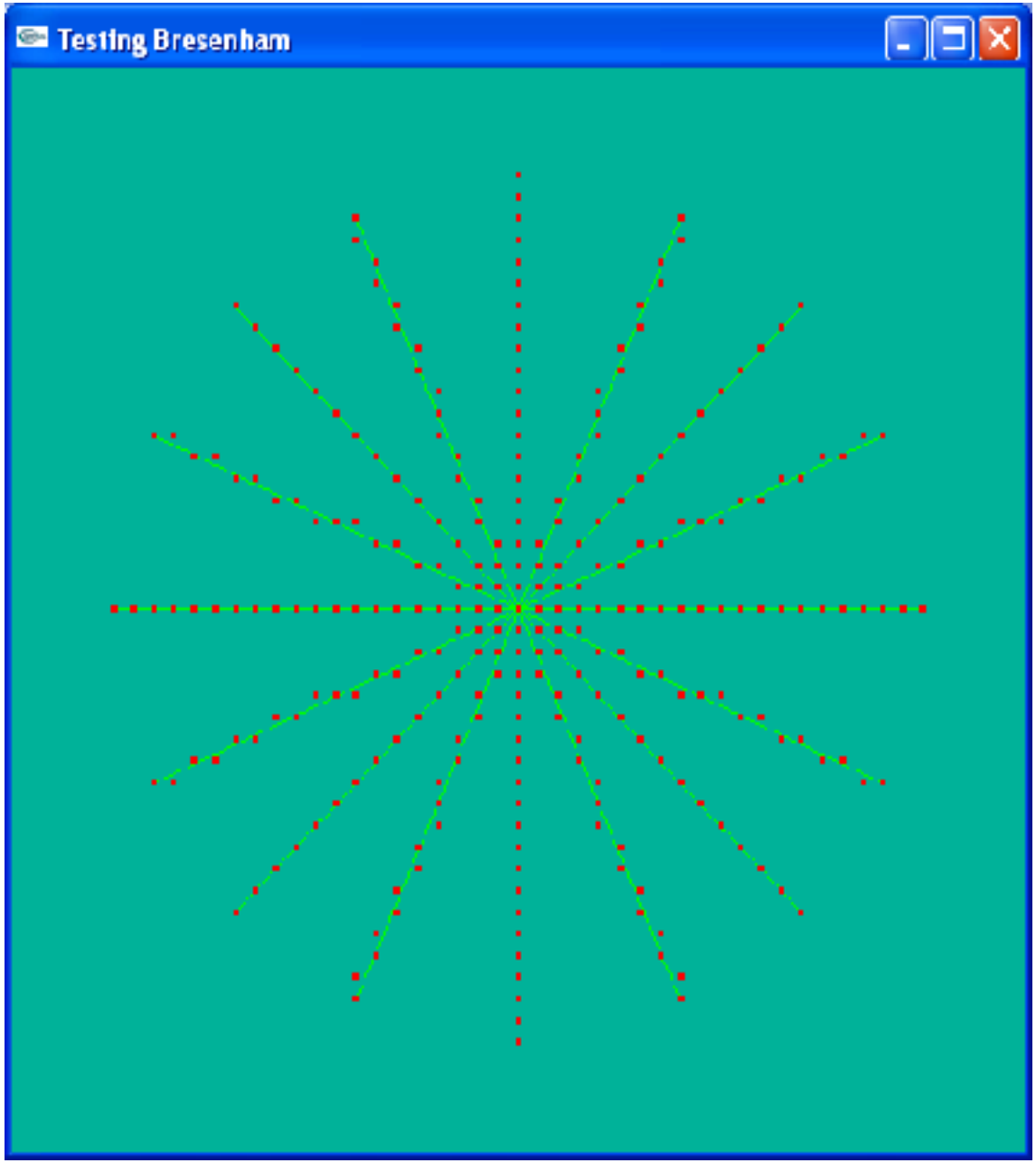
```
/*  Bresenham line-drawing procedure for |m| < 1.0.  */
   void lineBres (int x0, int y0, int xEnd, int yEnd)
   {
      int dx = fabs (xEnd - x0),  dy = fabs(yEnd - y0);
      int x, y, p = 2 * dy - dx;
      int twoDy = 2 * dy,  twoDyMinusDx = 2 * (dy - dx);

      /* Determine which endpoint to use as start position.  */
      if (x0 > xEnd) {
         x = xEnd;    y = yEnd;   xEnd = x0;
      }
      else {
         x = x0;    y = y0;
      }
      setPixel (x, y);

      while (x < xEnd) {
         x++;
         if (p < 0)
            p += twoDy;
         else {
            y++;
            p += twoDyMinusDx;
         }
         setPixel (x, y);
      }
   }
```

Line-drawing algorithm should work in every octant, and special cases





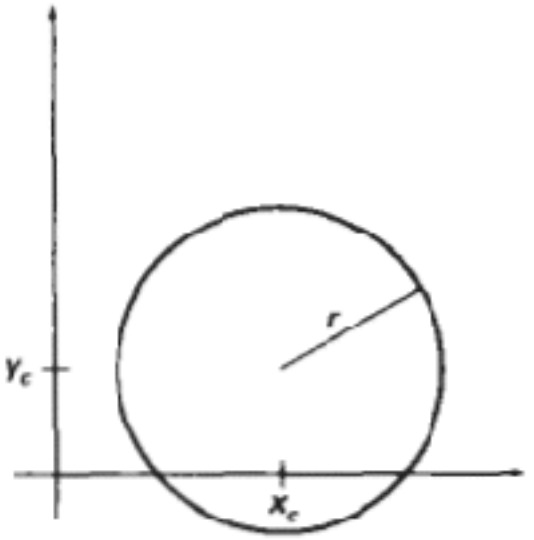
Simulating the Bresenham algorithm in drawing 8 radii on a circle of radius 20

Horizontal, vertical and ±45° radii handled as special cases

Scan Converting Circles

$$(x - x_c)^2 + (y - y_c)^2 = R^2$$

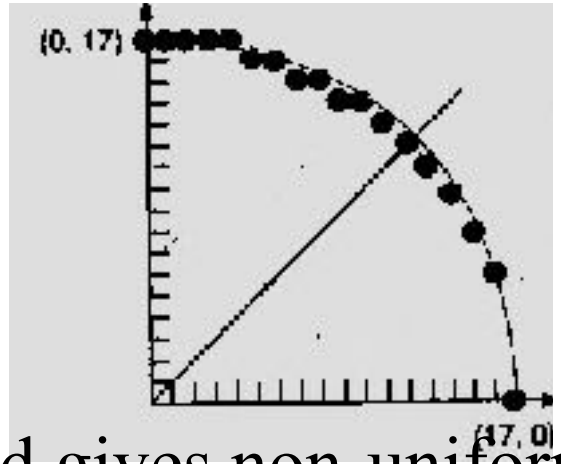
$$y = y_c \pm \sqrt{R^2 - (x - x_c)^2}$$



- Explicit: y = f(x)

$$y = \pm\sqrt{R^2 - x^2}$$

We could draw a quarter circle by incrementing x from 0 to R in unit steps and solving for +y for each step.



Method needs lots of computation, and gives non-uniform pixel spacing

Scan Converting Circles

- Parametric:

$$x = R\cos\theta$$

$$y = R\sin\theta$$

Draw quarter circle by stepping through the angle from 0 to 90
-avoids large gaps but still unsatisfactory
-How to set angular increment
Computationally expensive trigonometric calculations

Scan Converting Circles

- Implicit: $f(x,y) = x^2+y^2-R^2$

If $f(x,y) = 0$ then it is on the circle.
$f(x,y) > 0$ then it is outside the circle.
$f(x,y) < 0$ then it is inside the circle.

Try to adapt the Bresenham midpoint approach

Again, exploit symmetries

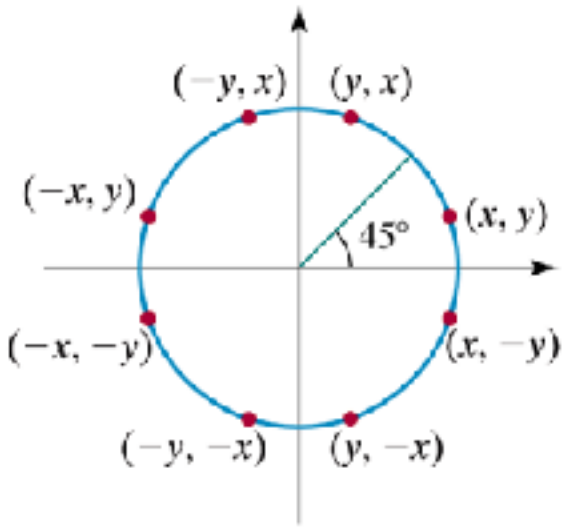


Figure 3-18

Symmetry of a circle. Calculation of a circle point (x, y) in one octant yields the circle points shown for the other seven octants.

Generalising the Bresenham midpoint approach

- Set up decision parameters for finding the closest pixel to the cırcumference at each sampling step
  - Avoid square root calculations by considering the squares of the pixel separation distances
- Use direct comparison without squaring.

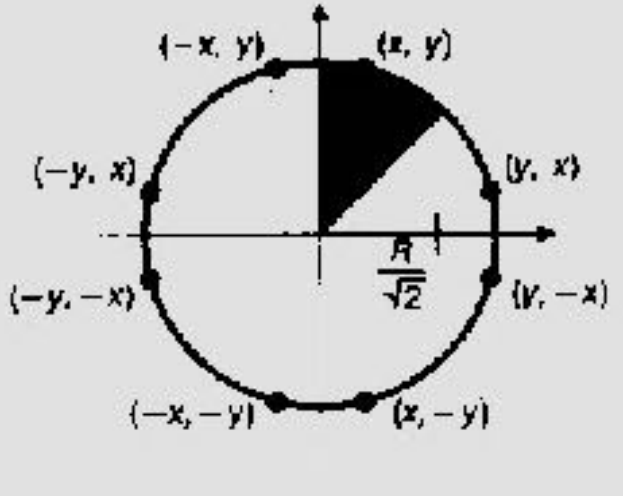
  Adapt the midpoint test idea: test the halfway position between pixels to determine if this midpoint is inside or outside the curve

  This gives the **midpoint algorithm for circles**

  Can be adapted to other curves: conic sections

Eight-way Symmetry

- only one octant's calculation needed




```
void CirclePoints (int x, int y, int value)
{
    WritePixel (x, y, value);
    WritePixel (y, x, value);
    WritePixel (y, −x, value);
    WritePixel (x, −y, value);
    WritePixel (−x, −y, value);
    WritePixel (−y, −x, value);
    WritePixel (−y, x, value);
    WritePixel (−x, y, value);
}   /* CirclePoints */
```

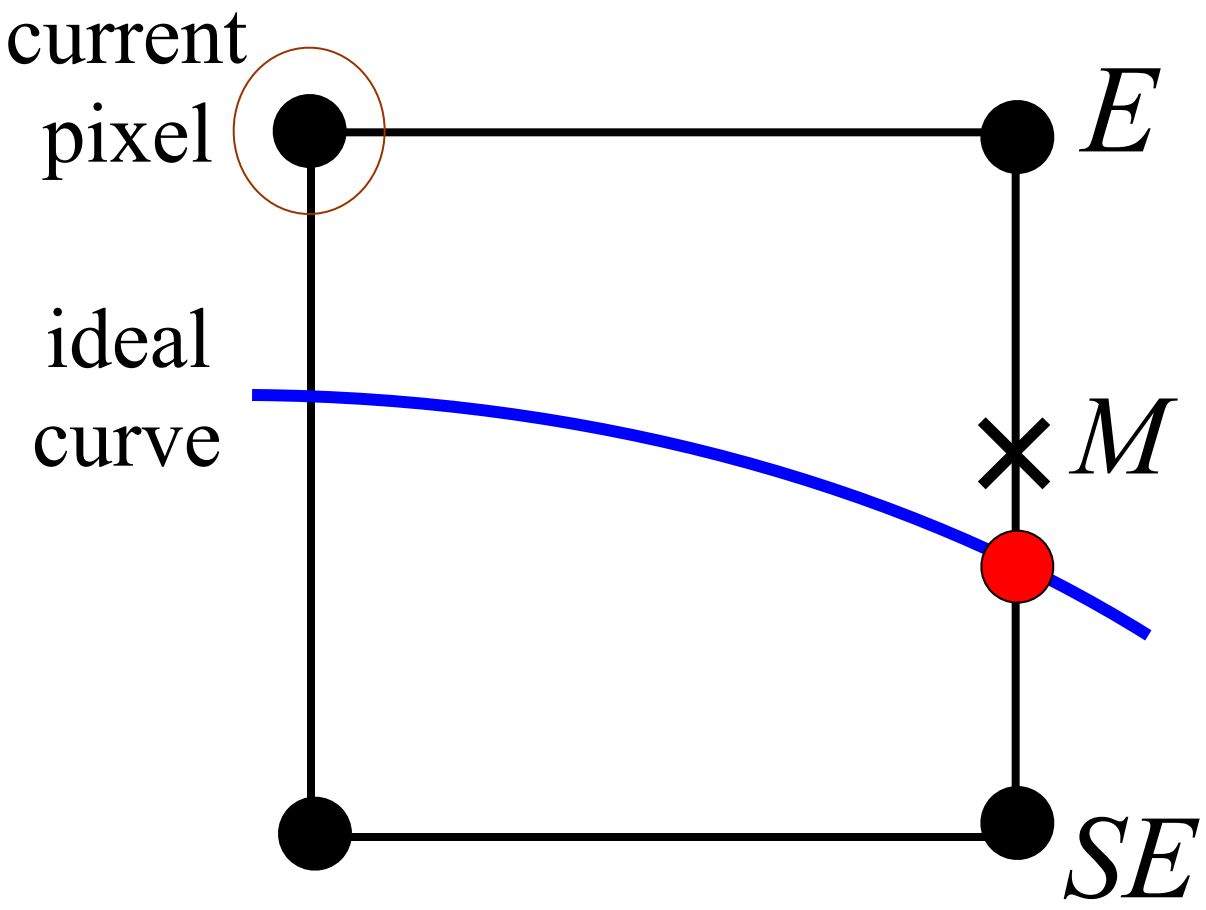
The 2nd Octant is a good arc to draw

- It is a well-defined function in this domain
  - single-valued
  - no vertical tangents: $|slope| \leq 1$
- Lends itself to the midpoint approach
  - only need consider E or SE
- Implicit formulation $F(x,y) = x^2 + y^2 - r^2$
  - For (x,y) on the circle, $F(x,y) = 0$
  - $F(x,y) > 0 \Rightarrow (x,y)$ *Outside*
  - $F(x,y) < 0 \Rightarrow (x,y)$ *Inside*

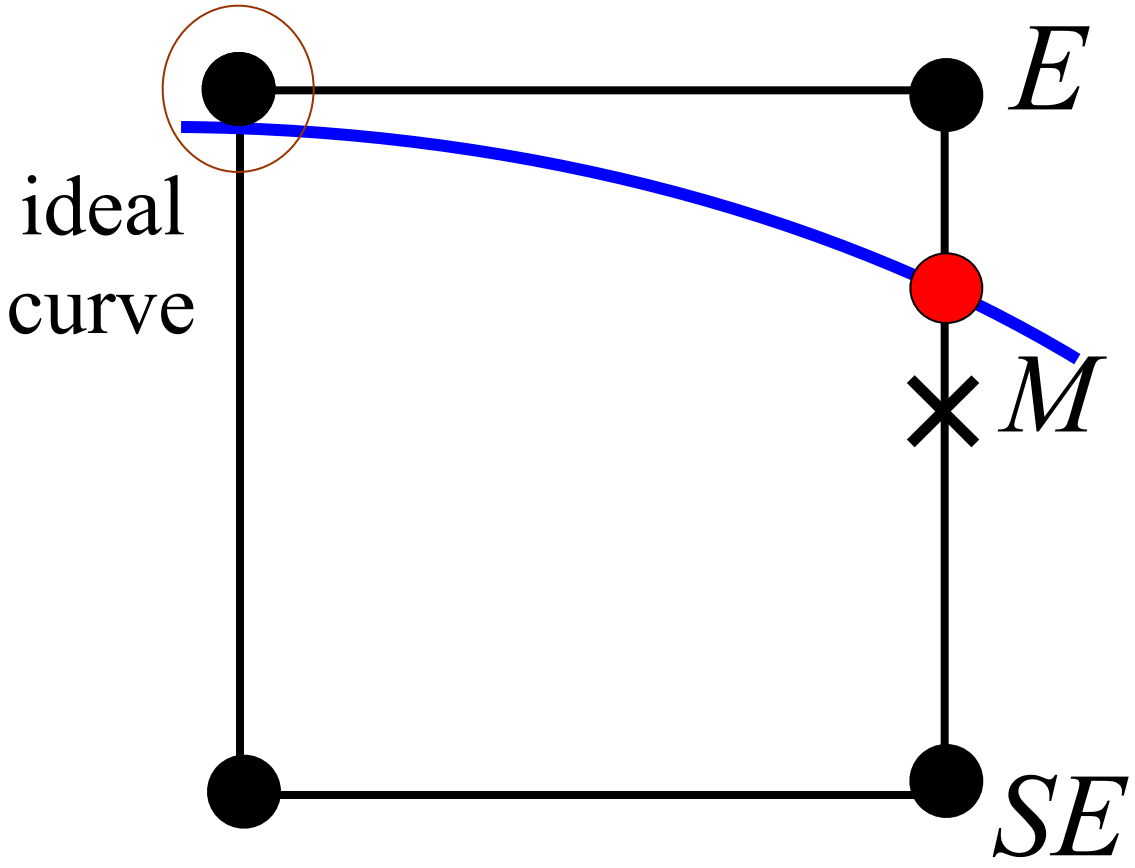
Choose *E* or *SE*

- Decision variable d is $x^2 + y^2 - r^2$
- Then d = $F(M) \geq 0 \Rightarrow$ SE
- Or d = $F(M) < 0 \Rightarrow E$

$F(M) \geq 0 \Rightarrow SE$



$F(M) < 0 \Rightarrow E$



Decision Variable *p*

As in the Bresenham line algorithm we use a decision variable to direct the selection of pixels.

Use the implicit form of the circle equation

$$p = F(M) = x^2 + y^2 - r^2$$

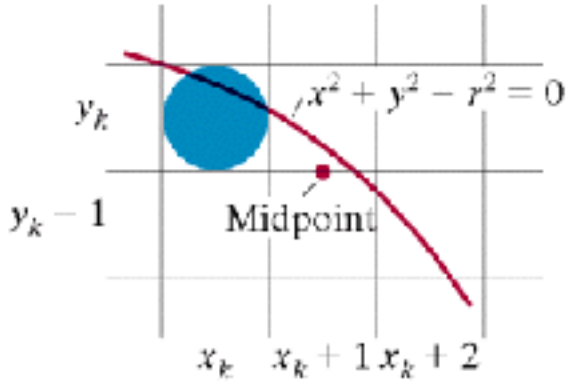


Figure 3-19

Midpoint between candidate pixels at sampling position $x_k + 1$ along a circular path.

Midpoint coordinates are $(x_k + 1, y_k - \frac{1}{2})$

Assuming we have just plotted point at (x_k,y_k) we determine whether move E or SE by evaluating the circle function at the midpoint between the two candidate pixel positions

$$p_k = F_{circ}(x_k + 1, y_k - \tfrac{1}{2})$$

$$= (x_k + 1)^2 + (y_k - \tfrac{1}{2})^2 - r^2$$

p_k is the decision variable

if $p_k < 0$ the midpoint is inside the circle

Thus the pixel above the midpoint is closer to the ideal circle, and we select pixel on scan line y_k. i.e. Go E

If $pk > 0$ the midpoint is outside the circle. Thus the pixel below the midpoint is closer to the ideal circle, and we select pixel on scan line yk-1. i.e. Go SE

Calculate successive decision parameter values p by incremental calculations.

$$p_{k+1} = F_{circ}(x_{k+1} + 1, y_{k+1} - \tfrac{1}{2})$$

$$= [(x_k + 1) + 1]^2 + (y_{k+1} - \tfrac{1}{2})^2 - r^2$$

recursive definition for successive decision parameter values *p*

$$p_{k+1} = F_{circ}(x_{k+1} + 1, y_{k+1} - \tfrac{1}{2})$$

$$= [(x_k + 1) + 1]^2 + (y_{k+1} - \tfrac{1}{2})^2 - r^2$$

$$p_{k+1} = p_k + 2(x_k + 1) + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) + 1$$

Where $y_{k+1} = y_k$ if $p<0$ (move E)

$y_{k+1} = y_k - 1$ if $p>0$ (move SE)

y_{k+1} and x_{k+1} can also be defined recursively

Initialisation

$x_0 = 0, \quad y_0 = r$

Initial decision variable found by evaluating circle function at first midpoint test position

$$p_0 = F_{circ}(1, r - 1/2)$$

$$= 1 + (r - 1/2)^2 - r^2$$

$$= \frac{5}{4} - r$$

For integer radius r p_0 can be rounded to $p_0 = 1-r$ since all increments are integer.

Midpoint Circle Algorithm

1. Input radius r and circle center (x_c, y_c), and obtain the first point on the circumference of a circle centered on the origin as

$$(x_0, y_0) = (0, r)$$

2. Calculate the initial value of the decision parameter as

$$p_0 = \frac{5}{4} - r$$

3. At each x_k position, starting at $k = 0$, perform the following test: If $p_k < 0$, the next point along the circle centered on $(0, 0)$ is (x_{k+1}, y_k) and

$$p_{k+1} = p_k + 2x_{k+1} + 1$$

   Otherwise, the next point along the circle is $(x_k + 1, y_k - 1)$ and

$$p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$$

   where $2x_{k+1} = 2x_k + 2$ and $2y_{k+1} = 2y_k - 2$.
4. Determine symmetry points in the other seven octants.
5. Move each calculated pixel position (x, y) onto the circular path centered on (x_c, y_c) and plot the coordinate values:

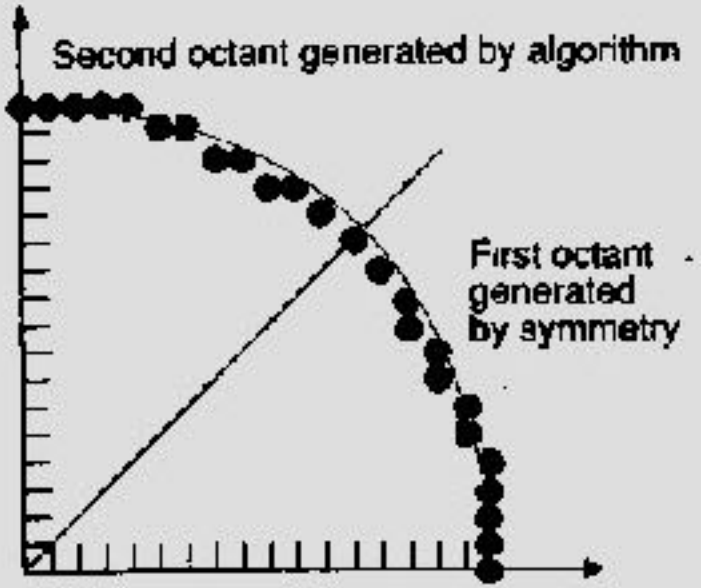
$$x = x + x_c, \qquad y = y + y_c$$

6. Repeat steps 3 through 5 until $x \geq y$.

Midpoint Circle Algorithm (cont.)

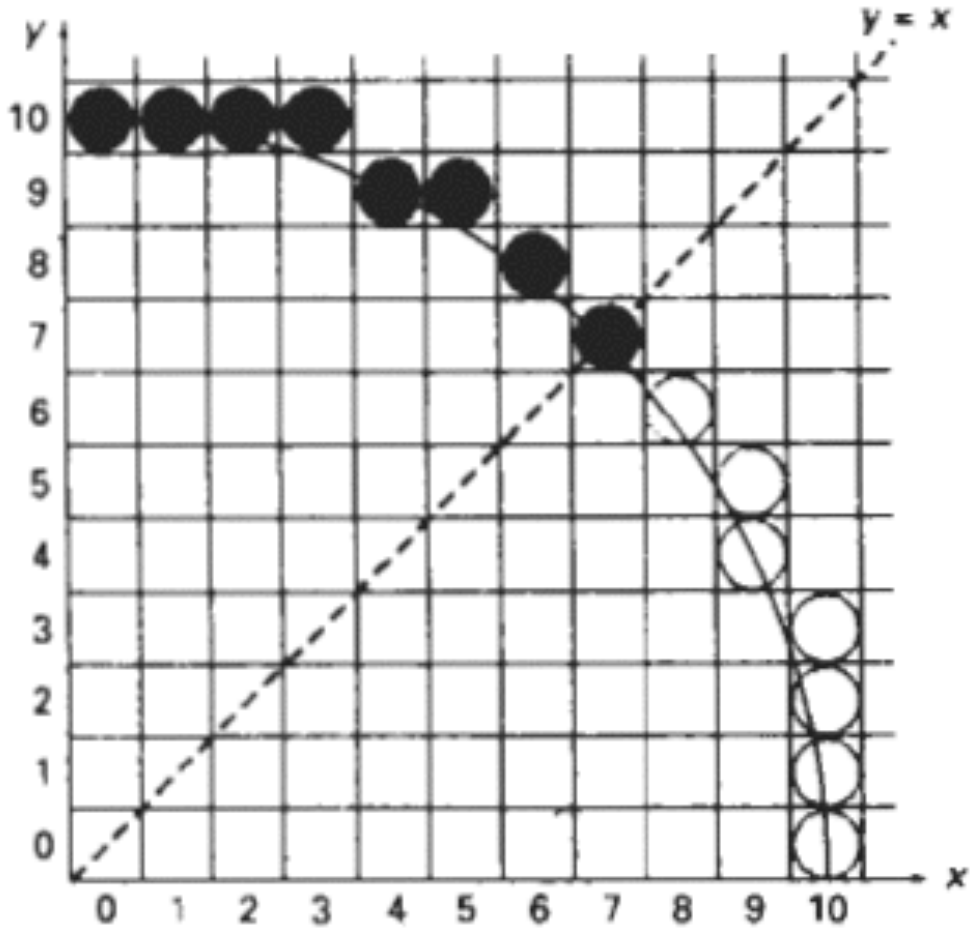
```
void MidpointCircle (int radius, int value)
/* Assumes center of circle is at origin. Integer arithmetic only */
{
    int x = 0;
    int y = radius;
    int d = 1 - radius;
    CirclePoints (x, y, value);

    while (y > x) {
        if (d < 0)              /* Select E */
            d += 2 * x + 3;
        else {                  /* Select SE */
            d += 2 * (x - y) + 5;
            y--;
        }
        x++;
        CirclePoints (x, y, value);
    } /* while */
} /* MidpointCircle */
```



Example

- r=10



Another method:

Approximate it using a polyline.

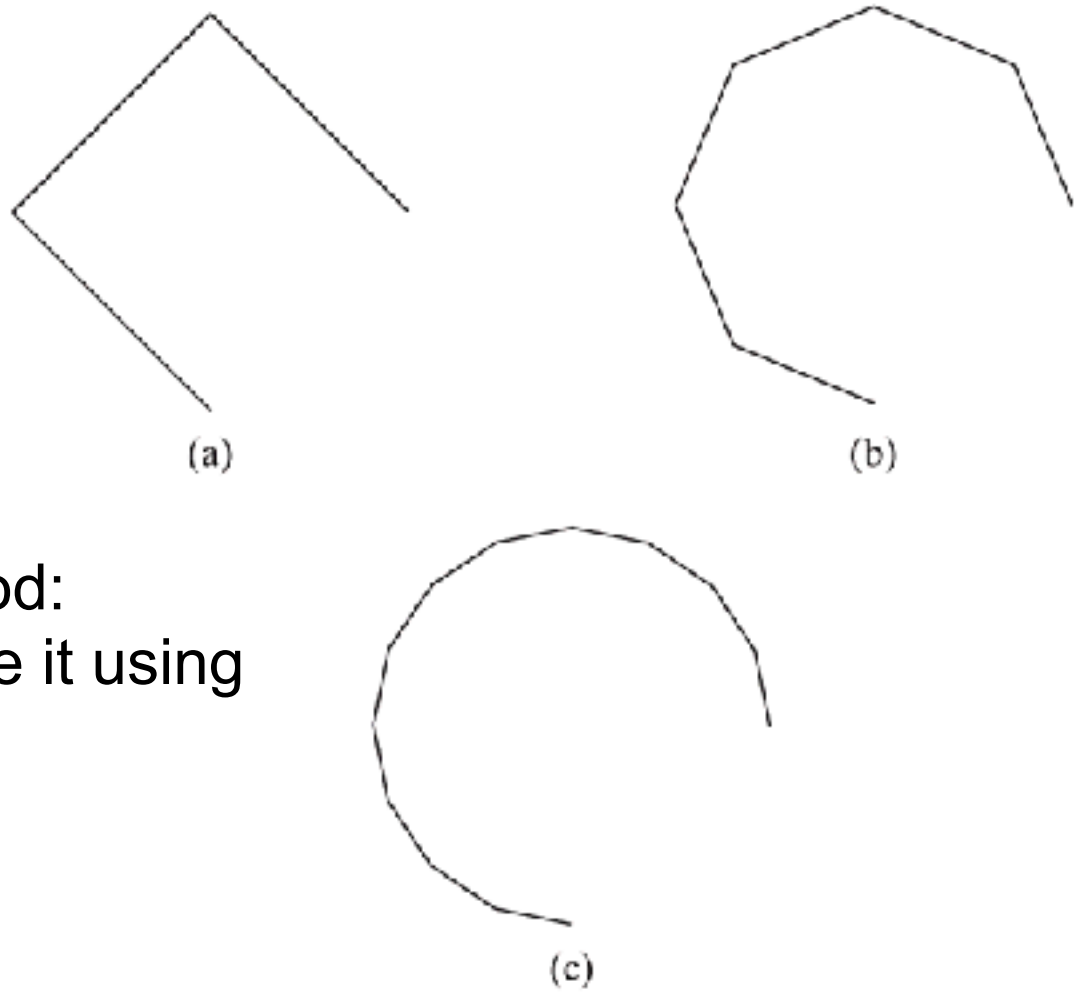


Figure 3-15

A circular arc approximated with (a) three straight-line segments, (b) six line segments, and (c) twelve line segments.

Fill area : an area that is filled with solid colour or pattern

Polygon Fill Areas

- Most library routines require that a fill area be specified as a polygon
  - OpenGL only allows ***convex*** polygons
- Non-polygon (curved) objects can be approximated by polygons
  - Surface tessellation, polygon mesh, triangular mesh

Polygon types

- Simple polygons are either ***convex*** or ***concave***:
  - Convex polygon: All interior angles < 180°, or any line segment combining two points in the interior is also in the interior

convex polygon

concave polygon

can be split into a number of convex polygons

Identifying a concave polygon

- has at least one interior angle >180 degrees
- extensions of some edges will intersect other edges

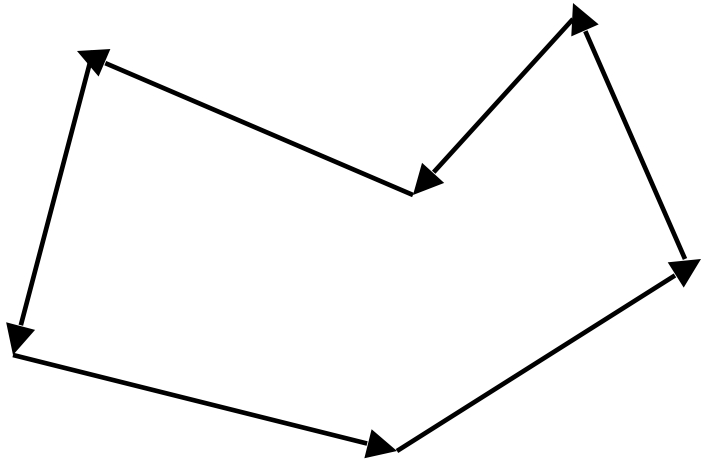
One test:

Express each polygon edge as a vector, with a consistent orientation.

Can then calculate cross-products of adjacent edges

Identifying a concave polygon

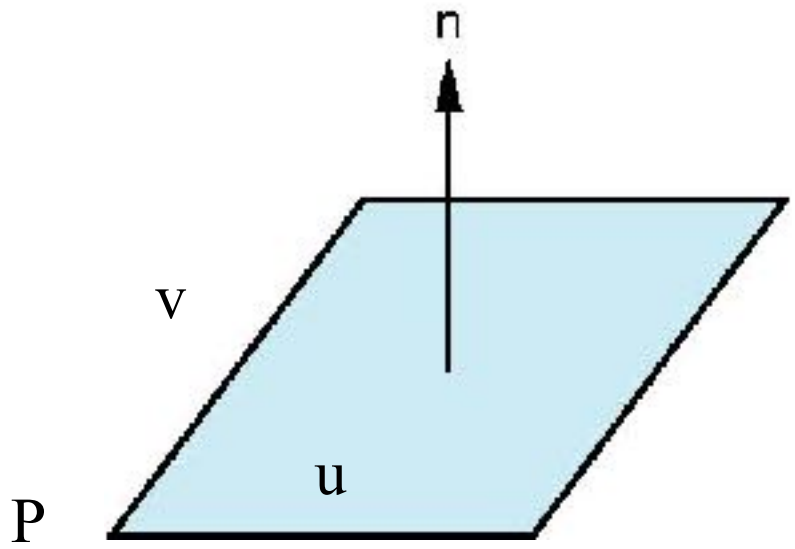
- When polygon edges are oriented with an anti-clockwise sense
  - cross product at convex vertex has positive sign
  - concave vertex gives negative sign



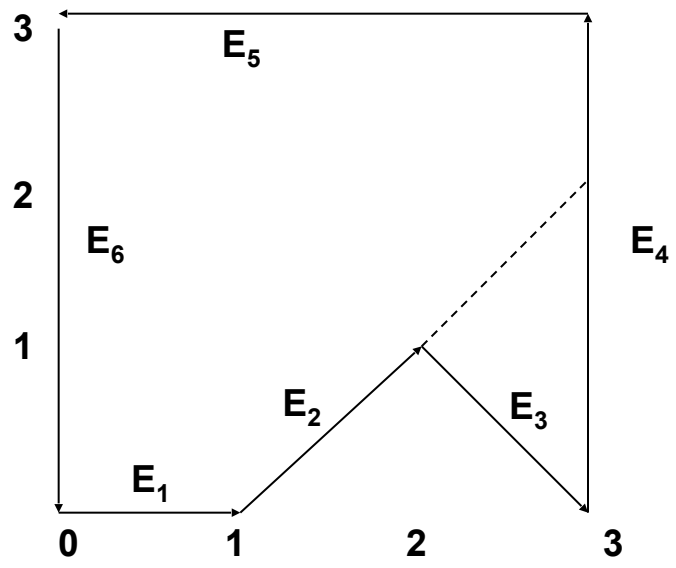
Normals

- Every plane has a vector **n** normal (perpendicular, orthogonal) to it
- **n** = **u** x **v** (vector cross product)

$$\overline{u} \times \overline{v} = \begin{pmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{pmatrix}$$



Vector method for splitting concave polygons

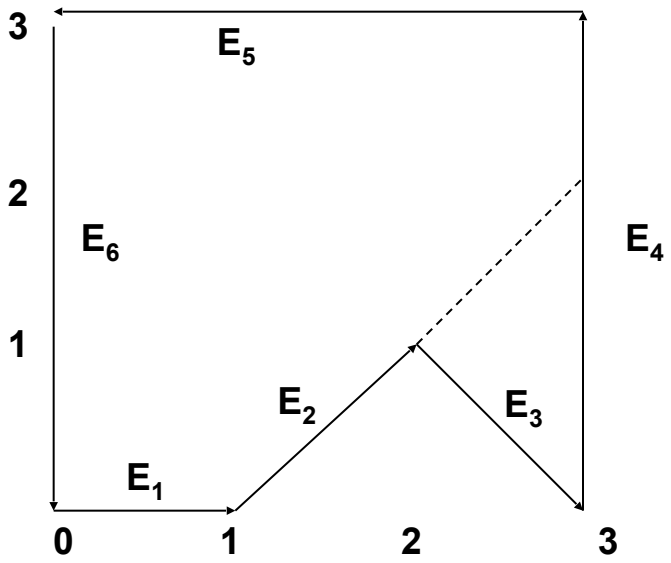


- E_1=(1,0,0) E_2=(1,1,0)
- E_3=(1,-1,0) E_4=(0,3,0)
- E_5=(-3,0,0) E_6=(0,-3,0)

- All z components have 0 value.
- Cross product of two vectors $E_j x E_k$ is perpendicular to them with z component

$$E_{jx}E_{ky}-E_{kx}E_{jy}$$

Example continued

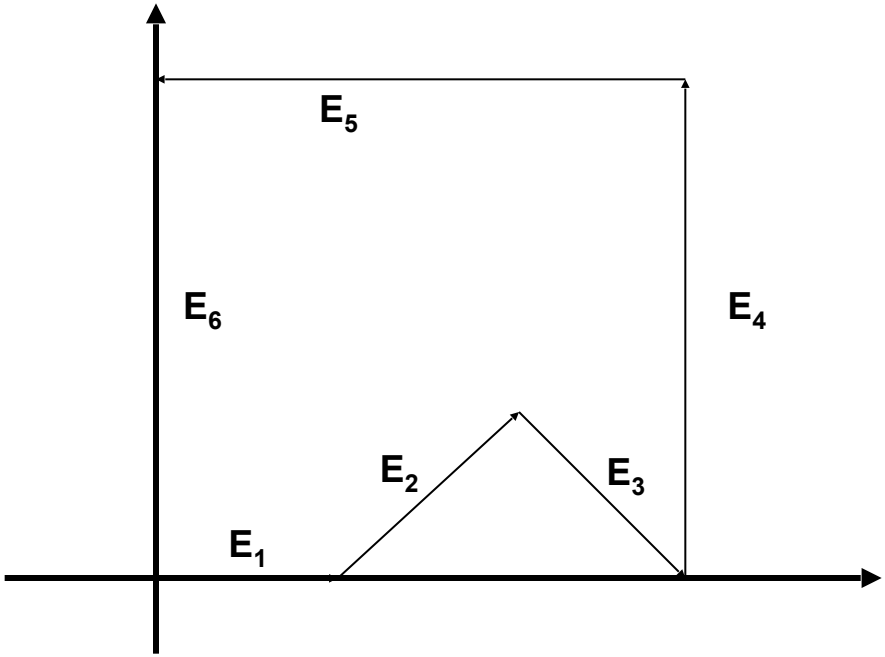


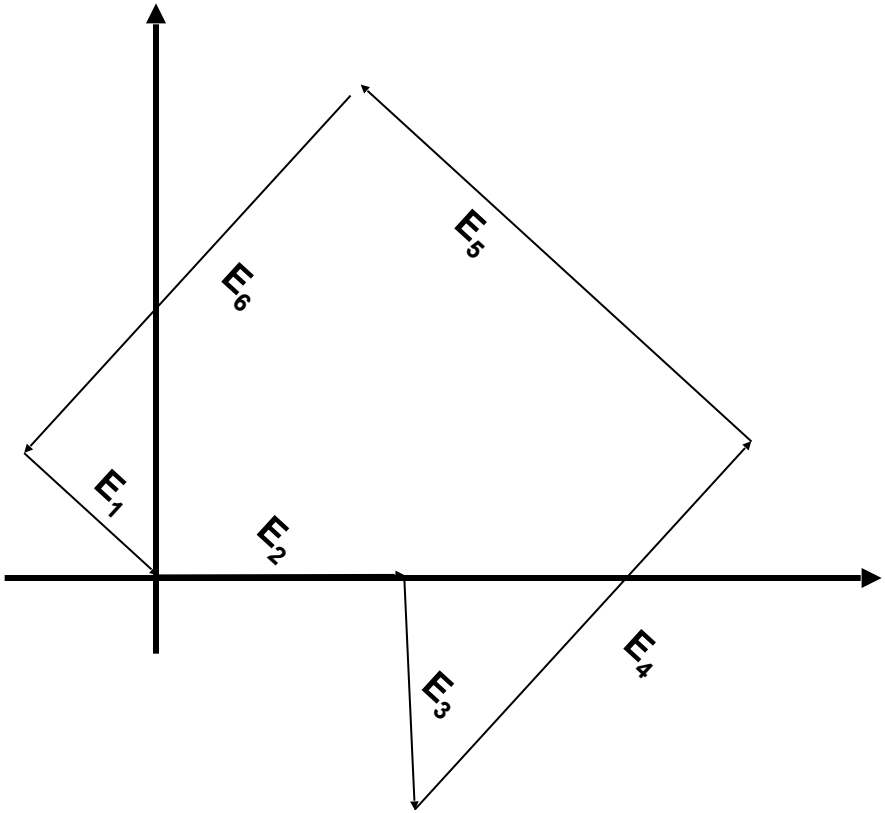
- $E_1 x E_2 = (0,0,1)$
  $E_2 x E_3 = (0,0,-2)$ $E_3 x E_4 =$ …
  $E_4 x E_5 =$ …
- $E_5 x E_6 =$ …
  $E_6 x E_1 =$ …

- Since $E_2 x E_3$ has negative sign, split the polygon along the line of vector E_2

Rotational method

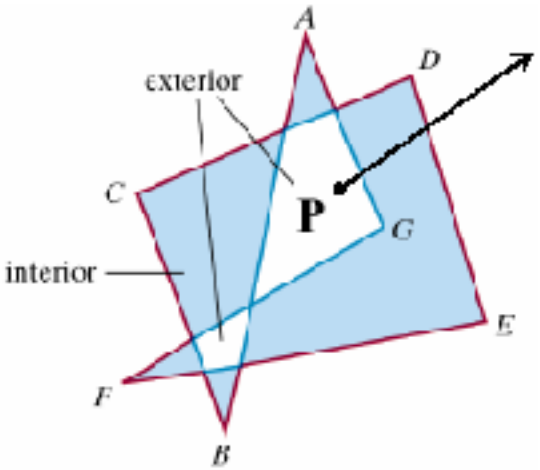
- Rotate the polygon so that each vertex in turn is at coordinate origin.
- If following vertex is below the x axis, polygon is concave.
- Split the polygon by x axis.


E5
E6
E4
E2
E3
E1


E6
E5
E1
E2
E3
E4

Inside-Outside Tests

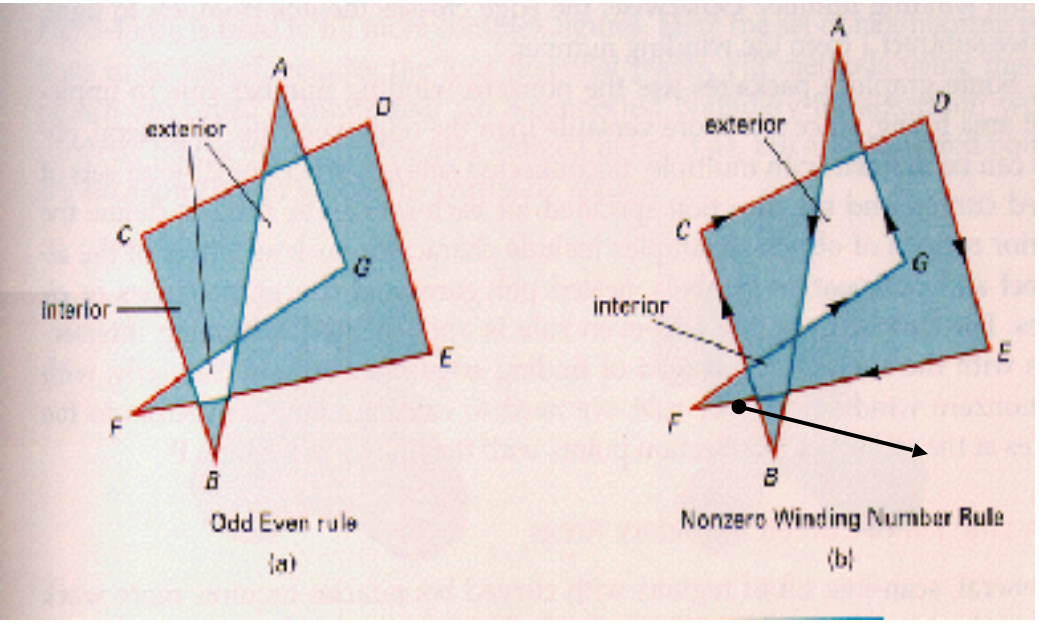
- Identifying the interior of a polygon (simple or complex) is important to identify the region to be filled
- Odd-even rule: To determine whether point **P** is inside or not. Draw a line starting from P to a distant position. Count the number of edges that cross this line. If the count is **odd** then the point is **inside**, otherwise it is outside.



Odd-Even Rule

Inside-Outside? nonzero winding-number rule

A winding number is an attribute of a point with respect to a polygon that tells us how many times the polygon encloses (or wraps around) the point. It is an integer, greater than or equal to 0. Regions of winding number 0 (unenclosed) are obviously outside the polygon, and regions of winding number 1 (simply enclosed) are obviously inside the polygon.



Initially 0

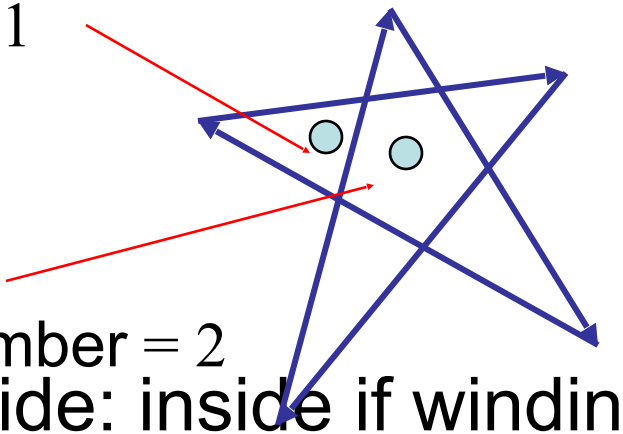
+1: edge crossing the line from right to left

-1: left to right

Winding Number

- Count clockwise encirclements of point

winding number = 1



winding number = 2

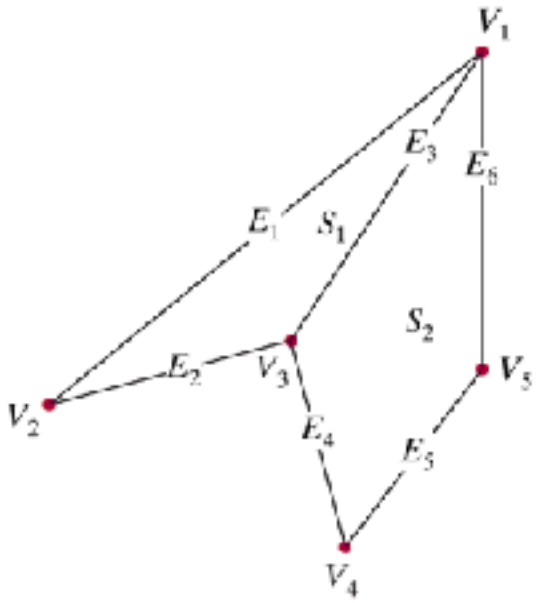
- Alternate definition of inside: inside if winding number ≠ 0

Polygon tables

- store descriptions of polygon geometry and topology, and surface parameters: colour, transparency, light-reflection
- organise in 2 groups
  - geometric data
  - attribute data

Polygon Tables:

Geometric data



VERTEX TABLE	
V_1:	x_1, y_1, z_1
V_2:	x_2, y_2, z_2
V_3:	x_3, y_3, z_3
V_4:	x_4, y_4, z_4
V_5:	x_5, y_5, z_5

EDGE TABLE	
E_1:	V_1, V_2
E_2:	V_2, V_3
E_3:	V_3, V_1
E_4:	V_3, V_4
E_5:	V_4, V_5
E_6:	V_5, V_1

SURFACE-FACET TABLE	
S_1:	E_1, E_2, E_3
S_2:	E_3, E_4, E_5, E_6

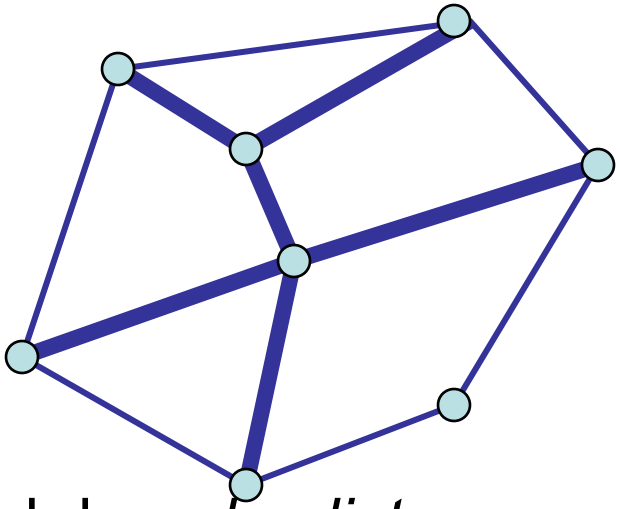
Figure 3-50

Geometric data-table representation for two adjacent polygon surface facets, formed with six edges and five vertices.

- Data can be used for consistency checking
- Additional geometric data stored: slopes, bounding boxes

Shared Edges

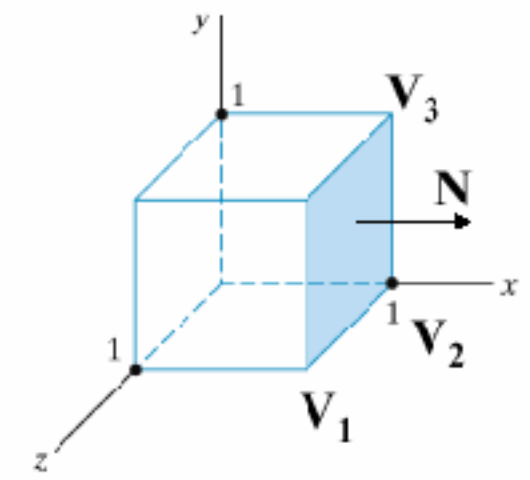
- Vertex lists will draw filled polygons correctly but if we draw the polygon by its edges, shared edges are drawn twice



- Can store mesh by *edge list*

Front and Back Face of a Polygon

- The normal vector points in a direction from the back face of the polygon to the front face
- Normal vector is the cross product of the two edges of the polygon in counter-clockwise direction



$$\mathbf{N} = (\mathbf{V}_2 - \mathbf{V}_1) \times (\mathbf{V}_3 - \mathbf{V}_2)$$

Inward and Outward Facing Polygons

- The order $\{v_1, v_6, v_7\}$ and $\{v_6, v_7, v_1\}$ are equivalent in that the same polygon will be rendered by OpenGL but the order $\{v_1, v_7, v_6\}$ is different
- The first two describe *outwardly facing* polygons
- Use the *right-hand rule* = counter-clockwise encirclement of outward-pointing normal
- OpenGL can treat inward and outward facing polygons differently