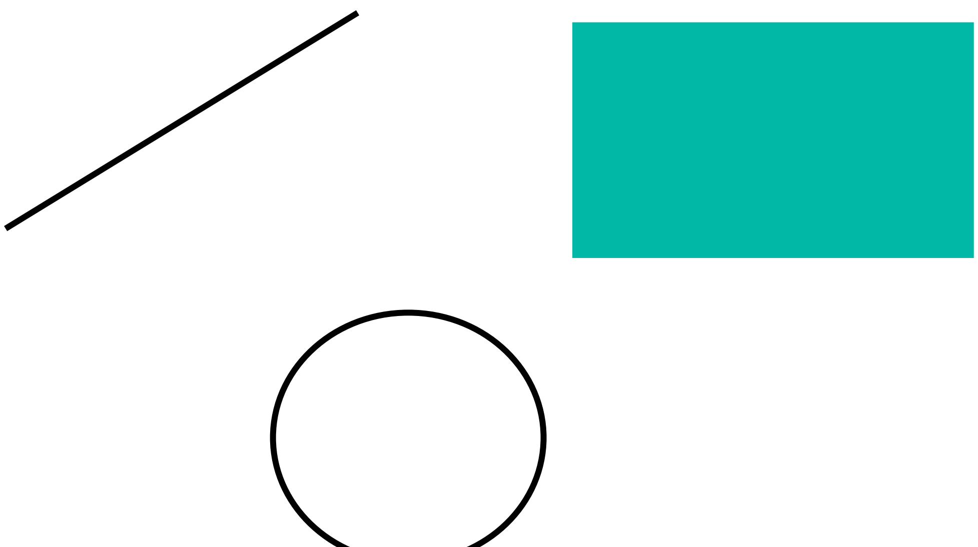
# Primitives

*Representations for Lines and Curves*



# Representations for lines and Curves

## *Line (in 2D)*

- Explicit

$$y = \frac{dy}{dx}(x - x_0) + y_0$$

- Implicit

$$F(x, y) = (x - x_0)dy - (y - y_0)dx$$

$$\begin{array}{llll} \text{if} & F(x,y) = 0 & \text{then} & (x,y) \text{ is on line} \\ & F(x,y) > 0 & & (x,y) \text{ is below line} \\ & F(x,y) < 0 & & (x,y) \text{ is above line} \end{array}$$

- Parametric

$$x(t) = x_0 + t(x_1 - x_0)$$
$$y(t) = y_0 + t(y_1 - y_0)$$
$$t \in [0, 1]$$

$$P(t) = P_0 + t(P_1 - P_0), \text{ or}$$
$$P(t) = (1 - t)P_0 + tP_1$$

# Circle

- Explicit

$$y = \pm\sqrt{r^2 - x^2}, \quad |\, x \,| \leq r$$

- Implicit

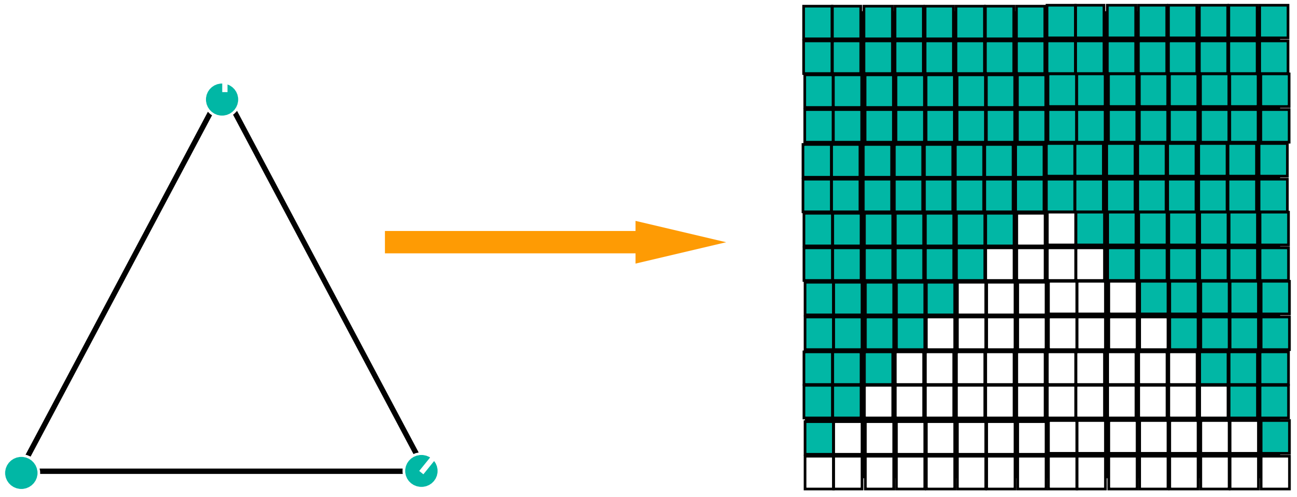
$$x^2 + y^2 = r^2$$
$$F(x, y) = x^2 + y^2 - r^2$$

$$\begin{array}{llll} \text{if} & F(x,y) = 0 & \text{then} & (x,y) \text{ is on circle} \\ & F(x,y) > 0 & & (x,y) \text{ is outside} \\ & F(x,y) < 0 & & (x,y) \text{ is inside} \end{array}$$

- Parametric

$$x(\theta) = r\cos(\theta)$$
$$y(\theta) = r\sin(\theta)$$
$$\theta \in [0, 2\pi]$$

# Rasterization



# Line rasterization

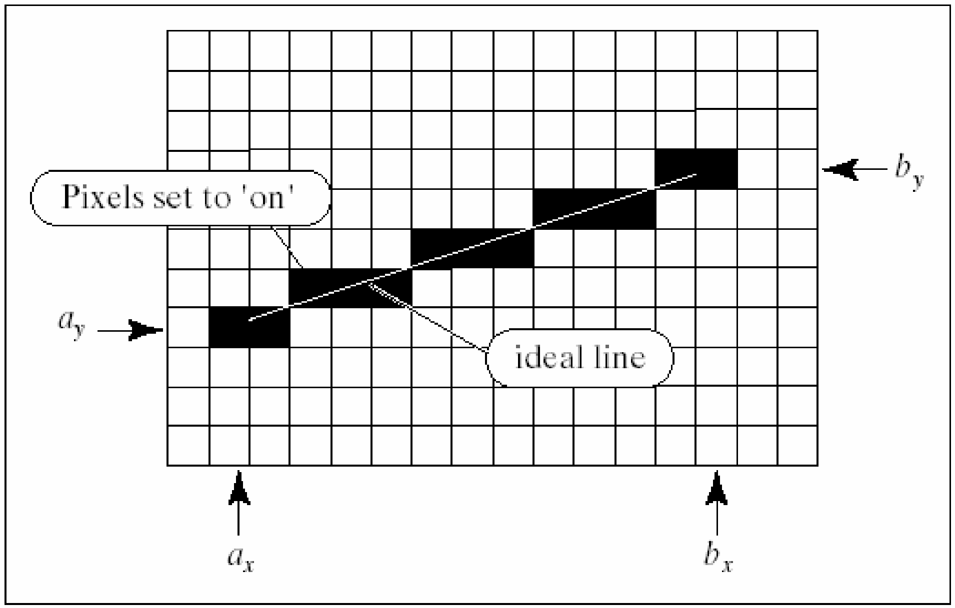



**FIGURE 10.23** Drawing a straight-line-segment.

*from*: **Computer Graphics Using OpenGL, 2e**, by F. S. Hill

# Line rasterization

## *Desired properties*

- Straight
- Pass through end points
- Smooth
- Independent of end point order
- Uniform brightess
- Brightness independent of slope
- Efficient

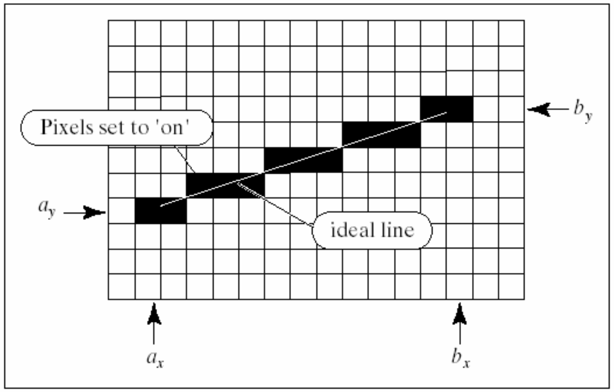



**FIGURE 10.23** Drawing a straight-line-segment.

# Straightforward Implementation

***Line between two points***

```
DrawLine(int x1,int y1,int x2,int y2)
    {
        float y;
        int x;

        for (x=x1; x<=x2; x++) {
                y = y1 +  (x-x1)*(y2-y1)/(x2-x1)
                SetPixel(x, Round(y) );

        }
    }
```

# Better Implementation

***How can we improve this algorithm?***

```
DrawLine(int x1,int y1,int x2,int y2)
  {
      float y;
      int x;

      for (x=x1; x<=x2; x++) {
            y = y1 +  (x-x1)*(y2-y1)/(x2-x1)
            SetPixel(x, Round(y) );

      }
  }
```

# Better Implementation

```
DrawLine(int x1,int y1,int x2,int y2)
  {
      float y,m;
      int x;
      dx = x2-x1 ;
      dy = y2-y1 ;
      m = dy/ (float) dx ;
      for (x=x1; x<=x2; x++) {
            y = y1 + m*(x-x1) ;
            SetPixel(x, Round(y) );
      }
  }
```

# Even Better Implementation

```
DrawLine(int x1,int y1,int x2,int y2)
  {
       float y,m;
       int x;
       dx = x2-x1 ;
       dy = y2-y1 ;
       m = dy/ (float) dx ;
       y = y1 + 0.5 ;

       for (x=x1; x<=x2; x++) {
              SetPixel(x, Floor(y) );
              y = y + m ;
       }
  }
```

# Midpoint algorithm (Bresenham)

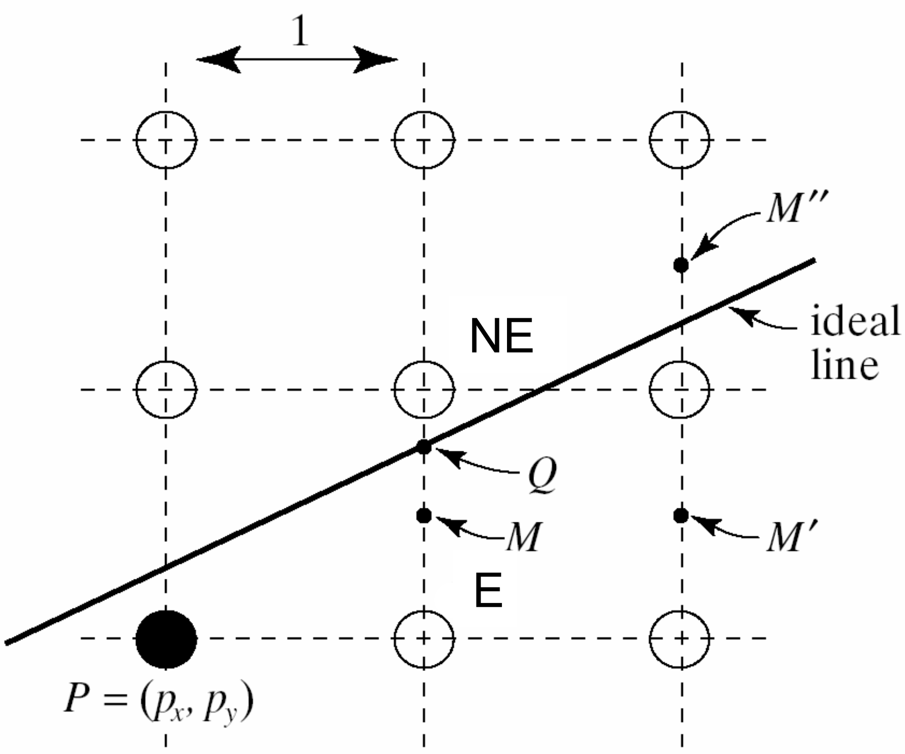
***Line in the first quadrant ( 0<slope < 45 deg)***

***Implicit function:***

***F(x,y) = xdy - ydx  + c,***

***dx,dy > 0 and dy/dx <= 1.0 ;***

- Current choice P = (x,y).
- How do we chose next of P, P'= (x+1,y') ?



# Midpoint algorithm (Bresenham)

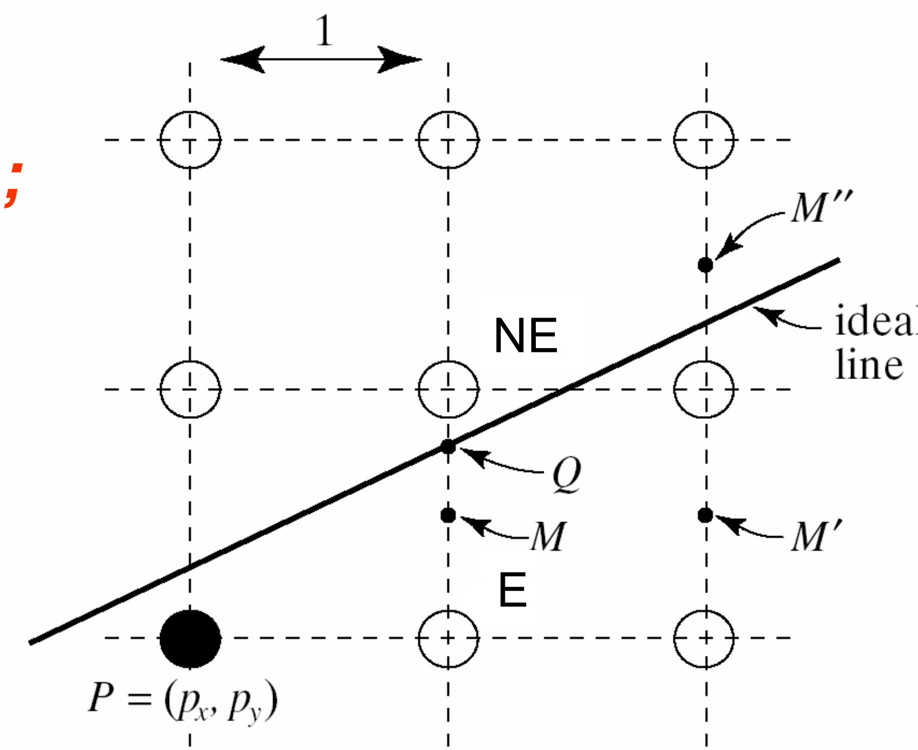
***Line in the first quadrant ( 0<slope < 45 deg)***

***Implicit function:***

***F(x,y) = xdy - ydx  + c,***

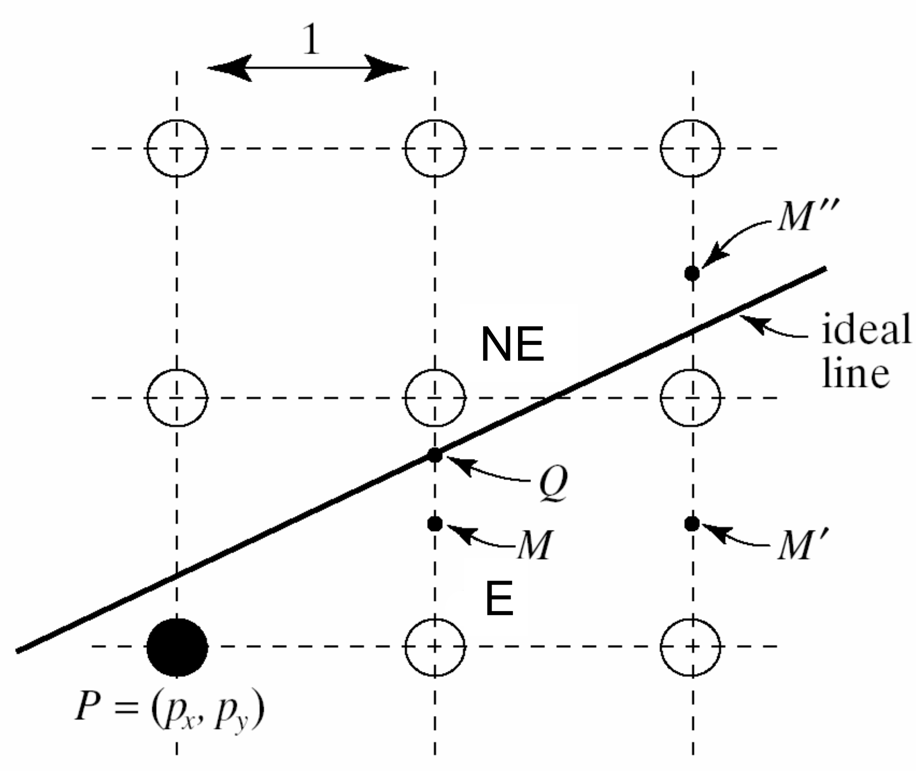
***dx,dy > 0 and dy/dx <= 1.0 ;***

- Current choice P = (x,y).
- How do we chose next of P,
  P'= (x+1,y') ?
  If( F(M) = F(x+1,y+0.5) < 0)
  
        M above line so E
  
  else
  
        M below line so NE



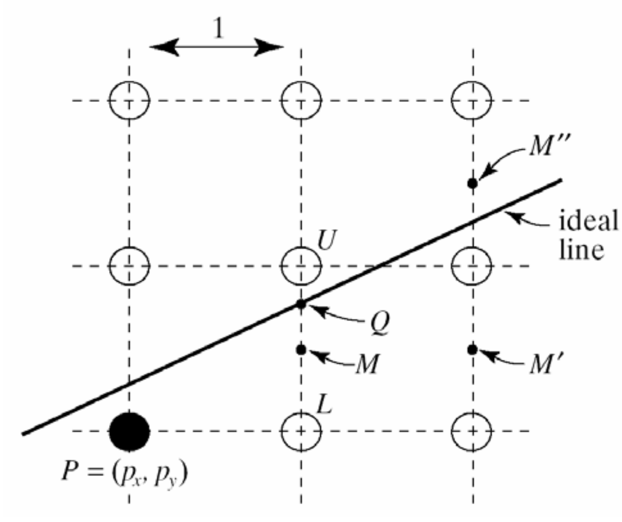
# Midpoint algorithm (Bresenham)

```
DrawLine(int x1, int y1, int x2, int y2, int color)
   {
        int x,y,dx,dy;

        y = Round(y1) ;
        for (x=x1; x<=x2; x++) {
                SetPixel(x, y );
                if (F(x+1,y+0.5)>0) {
                        y = y + 1 ;
                }

        }
   }
```



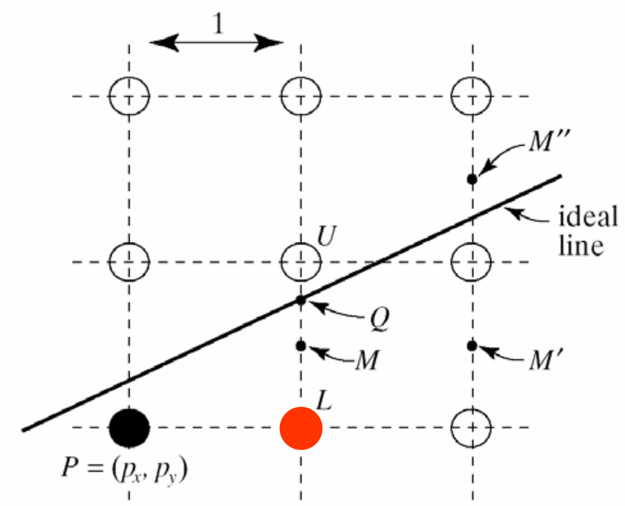
# Can we compute F in a smart way?

- We are at pixel (x,y) we evaluate F at M = (x+1,y+0.5) and E=(x+1,y) or NE=(x+1,y+1) accordingly.
  (Reminder: F(x,y) = xdy - ydx + c)



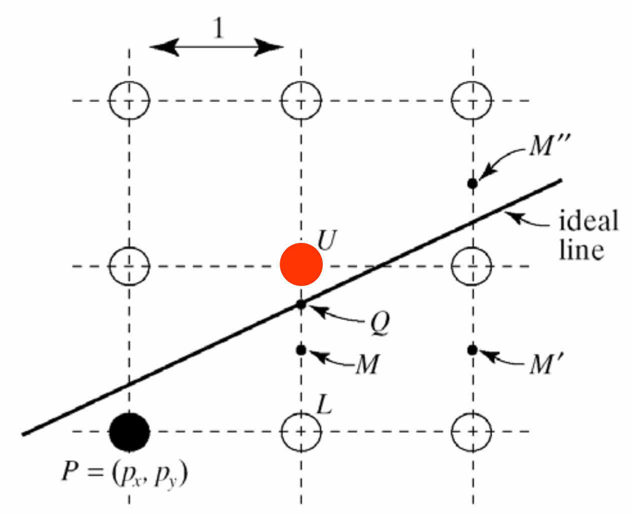
# Can we compute F in a smart way?

- We are at pixel (x,y) we evaluate F at M = (x+1,y+0.5) and E=(x+1,y) or NE=(x+1,y+1) accordingly.
  (Reminder: F(x,y) = xdy - ydx + c)
- If we chose E for x+1 the next criteria will be at M':
  F(x+2,y+0.5) = (x+1)dy +dy - (y+0.5)*dx +c→
  F(x+2,y+0.5) = F(x+1,y+0.5) + dy →
  $F_E = F + dy = F + dF_E$



# Can we compute F in a smart way?

- We are at pixel (x,y) we evaluate F at M = (x+1,y+0.5) and E=(x+1,y) or NE=(x+1,y+1) accordingly.
  (Reminder: F(x,y) = xdy - ydx + c)
- If we chose E for x+1 the next criteria will be at M':
  F(x+2,y+0.5) = (x+1)dy +dy - (y+0.5)*dx +c→
  F(x+2,y+0.5) = F(x+1,y+0.5) + dy →
  $F_E$ = F + dy
- If we chose NE then the next criteria will be at M'':
  F(x+2,y+1+0.5) =
  F(x+1,y+0.5) + dy – dx →
  $F_{NE}$ = F + dy - dx



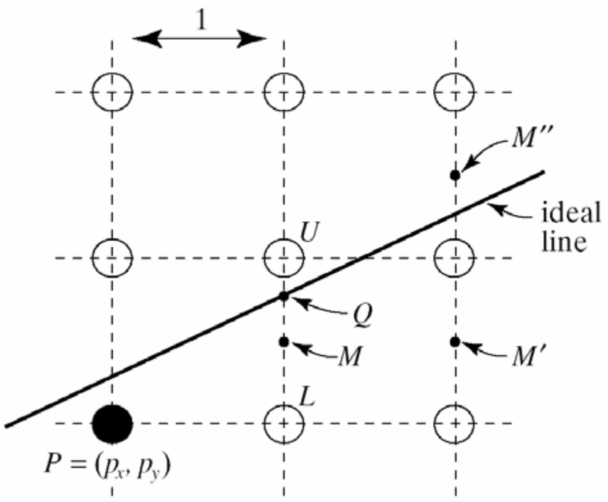
# Can we compute F in a smart way?

- We are at pixel (x,y) we evaluate F at M = (x+1,y+0.5) and E=(x+1,y) or NE=(x+1,y+1) accordingly.
  (Reminder: F(x,y) = xdy - ydx + c)
- If we chose E for x+1 the next criteria will be at M':

$F_E = F + dy$

- If we chose NE then the next criteria will be at M'':

$F_{NE} = F + dy - dx$



# Criterion update

## *Update*

$F_E = F + dy = F + dF_E$

$F_{NE} = F + dy - dx = F + dF_{NE}$

## *Starting value?*

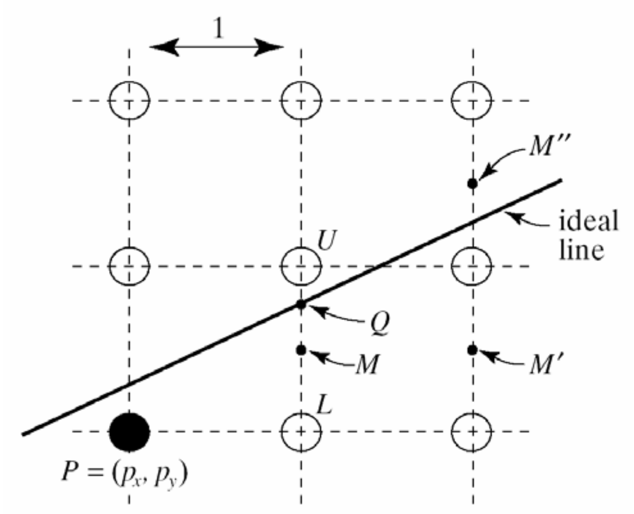
Assume line starts at pixel (x0,y0)

$F_{start} = F(x0+1, y0+0.5) \rightarrow$

$F_{start} = (x0+1)dy - (y0+0.5)dx + c$

$c = y0dx - x0dy$

$F_{start} = dy - 0.5dx$



# Criterion update (Integer version)

***Update***

$F_{start} = dy - 0.5dx$

$F_E = F + dy = F + dF_E$

$F_{NE} = F + dy - dx = F + dF_{NE}$

***Everything is integer except $F_{start}$.***

Multiply by 2 → $F_{start} = 2dy - dx$

$dF_E = 2dy$

$dF_{NE} = 2(dy-dx)$

# Midpoint algorithm

```
DrawLine(int x1, int y1, int x2, int y2, int color)
    {
        int x,y,dx,dy,dE, dNE;
        dx = x2-x1 ;
        dy = y2-y1 ;
        d = 2*dy-dx ; // initialize d
        dE = 2*dy ;
        dNE = 2*(dy-dx) ;
        y = y1 ;
        for (x=x1; x<=x2; x++) {
                SetPixel(x, y, color );
                if (d>0) {      // chose NE
                        d = d + dNE ;
                        y = y + 1 ;
                } else {        // chose E
                        d = d + dE ;
                }
        }
    }
```

# Incremental algorithms for polynomials

$$
\begin{aligned}
F(x) &= a_n x^n + a_{n-1} x^{n-1} \ldots + a_1 x + a_0, a_n \neq 0 \\
F(x+d) &= a_n (x+d)^n + a_{n-1} (x+d)^{n-1} \ldots + a_1 (x+d) + a_0 = \\
&= a_n (x+d)^n + P^{n-1}(x) \\
&= a_n \sum_{k=0}^{n} \begin{pmatrix} n \\ k \end{pmatrix} x^{n-k} d^k + P^{n-1}(x) \\
&= a_n \sum_{k} \left( \frac{n}{k!(n-k)!} \right) x^{n-k} d^k + P^{n-1}(x) \\
&= a_n x^n + \sum_{k=1}^{n} \left( \frac{n}{k!(n-k)!} \right) x^{n-k} d^k + P^{n-1}(x) \\
&= a_n x^n + R^{n-1}(x) + P^{n-1}(x)
\end{aligned}
$$

# N-order differences

$$\begin{aligned} F(x) &= a_n x^n + a_{n-1}x^{n-1} \ldots + a_1 x + a_0, a_n \neq 0 \\ F(x+d) &= a_n x^n + R^{n-1}(x) + P^{n-1}(x) \end{aligned}$$

First order

$$\Delta F = F(x+d) - F(x) = R^{n-1}(x) + P^{n-1}(x) = G_1^{n-1}(x)$$

N-order

$$\begin{aligned} \Delta^2 F(x) &= \Delta F(x+d) - \Delta F(x) = G_2^{n-2}(x) \\ &\vdots \\ \Delta^n F(x) &= \Delta^{n-1} F(x+d) - \Delta^{n-1} F(x) = G_n^0 = c \end{aligned}$$

# N-order difference update

$$
\begin{aligned}
F(x) &= a_n x^n + a_{n-1}x^{n-1} \ldots + a_1 x + a_0, a_n \neq 0 \\
F(x+d) &= a_n x^n + R^{n-1}(x) + P^{n-1}(x)
\end{aligned}
$$

$$
\begin{aligned}
F(x+d) &= F(x) + \Delta F(x) \\
\Delta F(x+d) &= \Delta F(x) + \Delta^2 F(x) \\
&\vdots \\
\Delta^{n-1} F(x+d) &= \Delta^{n-1} F(x) + \Delta^n F(x) \\
\Delta^n F(x+d) &= c
\end{aligned}
$$

We need $n$ initial conditions to initialize the differences.

# Example: y = x^2

$$
\begin{aligned}
y(x+d) &= x^2 + 2xd + d^2 = y(x) + 2xd + d^2 \\
\rightarrow \quad & y(x+d) = y(x) + \Delta y(x) \\
& where \quad \Delta y(x) = 2xd + d^2 \\
\Delta y(x+d) &= 2(x+d)d + d^2 = \Delta y(x) + 2d^2 \\
\rightarrow \quad & \Delta y(x+d) = \Delta y(x) + \Delta^2 y(x) \\
& where \quad \Delta^2 y(x) = 2d^2
\end{aligned}
$$

# The incremental algorithm to compute y = x^2

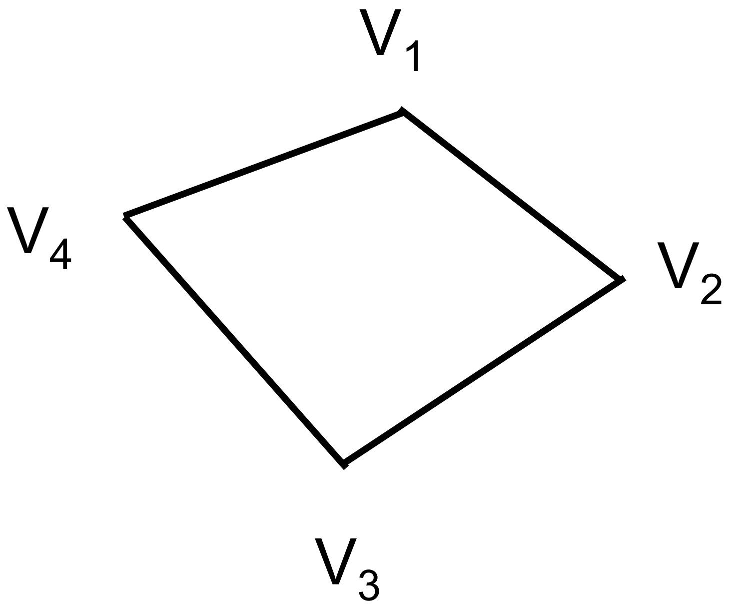
```
computePar(int d)
  {
      float y = 0 ;
      int x = 0 ;
      DY = d^2 ; // at x = 0
      DDY = 2*d^2 ;
      for( x = 0 ; x < X_MAX ; x++ ) {
            printf("d, %f\", x,y) ;
            y = y + DY ;
            DY = DY + DDY ;
  }
```

# Polygon

***Collection of points connected with lines***

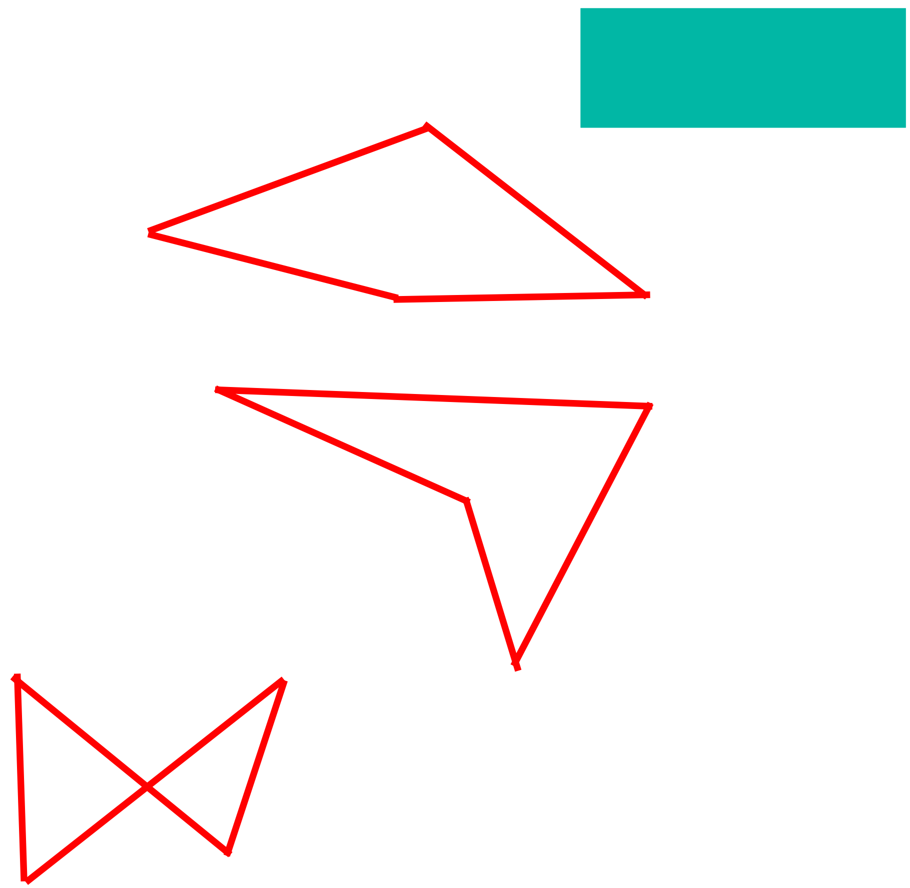
- Vertices: v1,v2,v3,v4
- Edges:

  $e_1 = v_1v_2$
  $e_2 = v_2v_3$
  $e_3 = v_3v_4$
  $e_4 = v_4v_1$



# Polygons

- Open / closed
- Planar / non-planar
- Filled / wireframe
- Convex / concave
- Simple / non-simple



# Triangles

***The most common primitive***

- Convex
- Planar
- Simple

# Background

## *Plane equations*

Implicit

$F(x,y,z) = Ax + By + Cz + D = \mathbf{N} \bullet P + D$

Points on Plane $F(x,y,z) = 0$

Parametric

$Plane(s,t) = P_0 + s(P_1 - P_0) + t(P_2 - P_0)$

$P_0, P_1, P_2$ not colinear

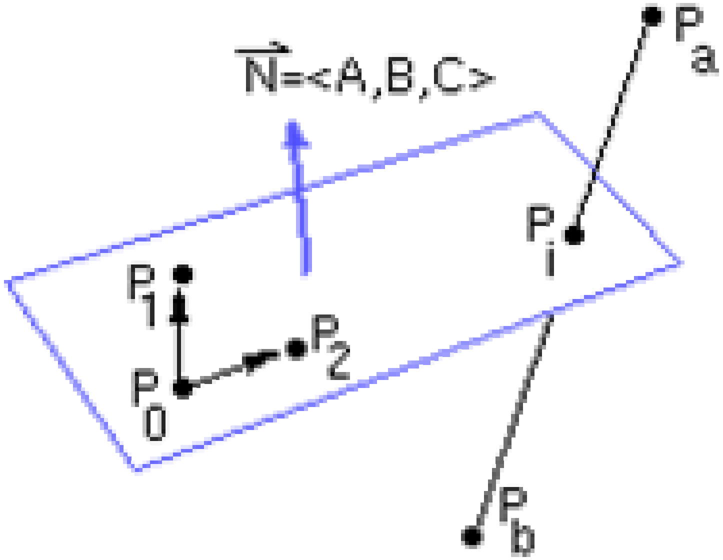
or

$Plane(s,t) = (1 - s - t)P_0 + sP_1 + tP_2$

$Plane(s,t) = P_0 + sV_1 + tV_2$ where $V_1, V_2$ basis vectors

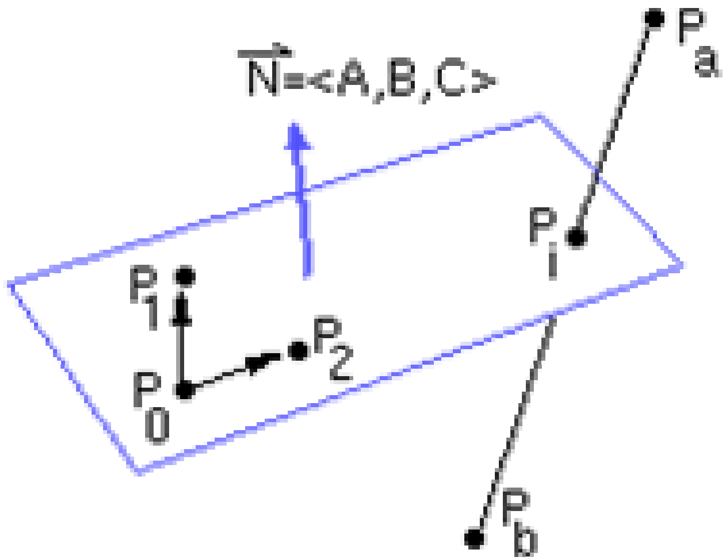
Explicit

$z = -(A/C)x - (B/C)y - D/C, \; C \neq 0$



# Point normal form

## *Plane equation*




$$F(x,y,z) = Ax + By + Cz + D = \mathbf{N} \bullet P + D$$

Points on Plane $F(x,y,z) = 0$

Observation : Let's take an arbitrary vector u that lies on the plane which can be defined by two points e.g. P1, P2 on the plane.

$$\mathbf{u} = P2 - P1$$

$$\left.\begin{array}{l} \mathbf{N} \bullet P1 + D = 0 \\ \mathbf{N} \bullet P2 + D = 0 \end{array}\right\} \Rightarrow \mathbf{N} \bullet (P2 - P1) = 0 \Rightarrow \mathbf{N} \bullet \mathbf{u} = 0 \Rightarrow \boxed{\mathbf{N} \perp \mathbf{u}}$$

# Computing point normal form from 3 Points

$$F(x,y,z) = Ax + By + Cz + D = \mathbf{N} \bullet P + D$$

Points on Plane $F(x,y,z) = 0$

First way:

$\mathbf{N} \bullet P0 + D = 0$

$\mathbf{N} \bullet P1 + D = 0$

$\mathbf{N} \bullet P2 + D = 0$

$|\mathbf{N}| = 1$ (arbitrary choice)

Second way:

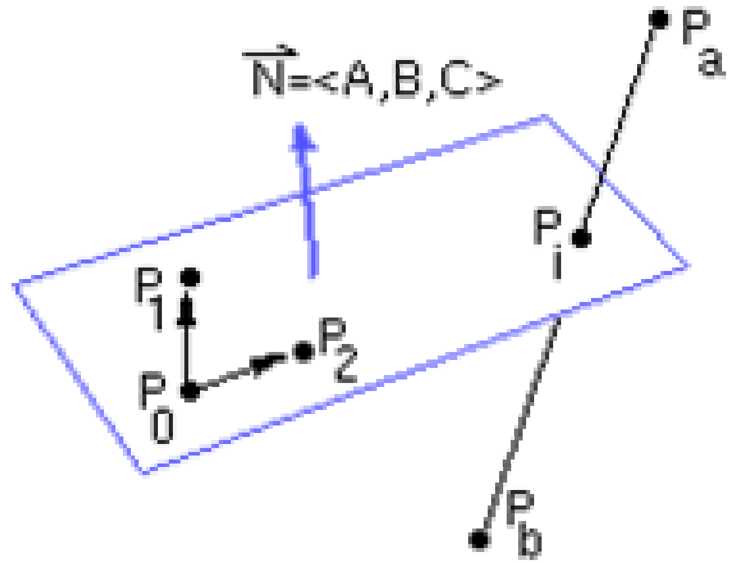
$\mathbf{N}$ is normal to $F$

Let's find a normal vector:

$\mathbf{N} = (P1 - P0) \times (P2 - P0)$

Compute $D$:

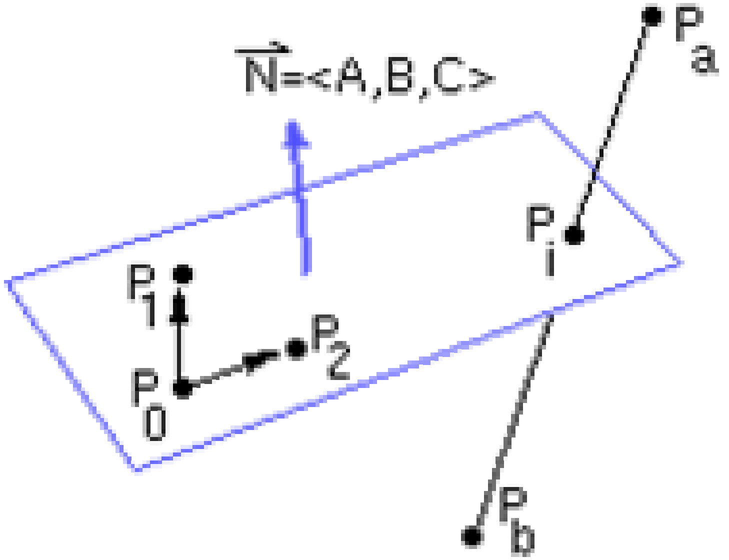
$D = -\mathbf{N} \bullet P0$



# Intersection of lines and planes

***Plane: Pl(P) = N*P +D = 0***

***Line: Pa+t(Pb-Pa), t in R***




$$\vec{N} \bullet (P_a + t\,(P_b - P_a)) + D = 0$$

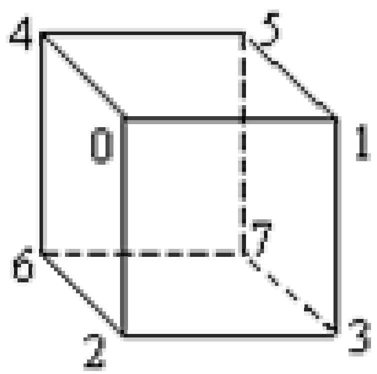
$$t = \frac{-D - \vec{N} \bullet P_a}{\vec{N} \bullet P_b - \vec{N} \bullet P_a} = \frac{-F(P_a)}{F(P_b) - F(P_a)}$$

Substitute t in L(t)

# Polygonal models/ data structures

[Hill: *p. 287-291. Foley & van Dam: p. 471-477* ]

## *Indexed face set*




faces

| # | vertex list |
|---|---|
| 0 | 0,2,3,1 |
| 1 | 1,3,7,5 |
| 2 | 5,7,6,4 |
| 3 | 4,6,2,0 |
| 4 | 4,0,1,5 |
| 5 | 2,6,7,3 |

vertex list

| # | x,y,z |
|---|---|
| 0 | 0,1,1 |
| 1 | 1,1,1 |
| 2 | 0,0,1 |
| 3 | 1,0,1 |
| 4 | 0,1,0 |
| 5 | 1,1,0 |
| 6 | 0,0,0 |
| 7 | 1,0,0 |

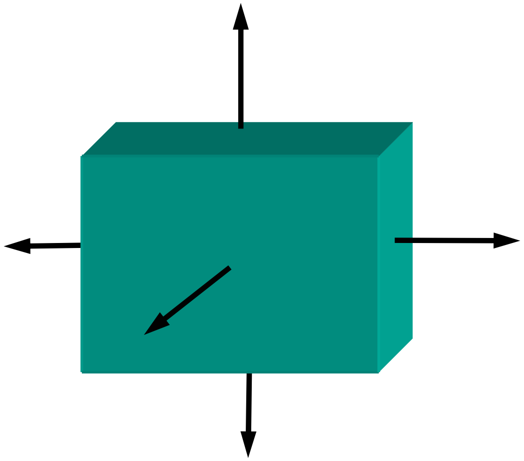
# Polygon attributes

***Per vertex or per face***

- Normal
- Color

***Per vertex***

- Texture coordinates



# Computing the normal of a polygon

One way:

$N = (V_{n-1} - V_0) x (V_1 - V_0)$

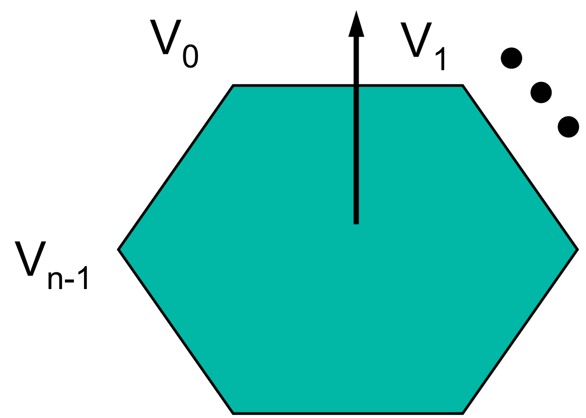
Newell's method (page 292)

$$N_x = \sum_{i=0}^{n-1} (y_i - y_{next(i)})(z_i + z_{next(i)})$$

$$N_y = \sum_{i=0}^{n-1} (z_i - z_{next(i)})(x_i + x_{next(i)})$$

$$N_z = \sum_{i=0}^{n-1} (x_i - x_{next(i)})(y_i + y_{next(i)})$$

$$where \quad next(j) = (j+1) \ mod \ n$$



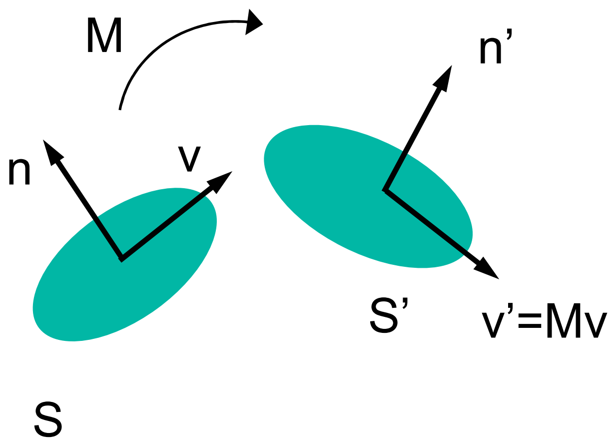
# Transforming Normals

***Normal vectors are transformed along with vertices and polygons.***

- How do you transform a normal ?
- What about unit magnitude ?

# Deriving transformation of normal

$$\mathbf{n} = (n_x, n_y, n_z, 0)^T \quad normal \quad to \quad S$$
$$\mathbf{v} = (v_x, v_y, v_z, 0)^T \quad tangent \quad to \quad S$$
$$S' = MS, \quad what \quad is \quad \mathbf{n}'?$$




$$\mathbf{n} \cdot \mathbf{v} = \mathbf{n}^T \mathbf{v} = 0$$
$$\mathbf{n}^T \mathbf{v} = 0 \rightarrow \mathbf{n}^T I \mathbf{v} = 0 \rightarrow \mathbf{n}(M^{-1}M)\mathbf{v} = 0$$
$$\rightarrow (\mathbf{n}^T M^{-1})(M\mathbf{v}) = 0 \rightarrow (M^{-T}\mathbf{n})^T(M\mathbf{v}) = 0$$
$$\rightarrow (M^{-T}\mathbf{n}) \cdot (M\mathbf{v}) = 0$$

# Normalization

***glEnable(GL_NORMALIZE) ;***

- Transformation includes scale or shear
- We provide non unit normals

# Polygons in OpenGL

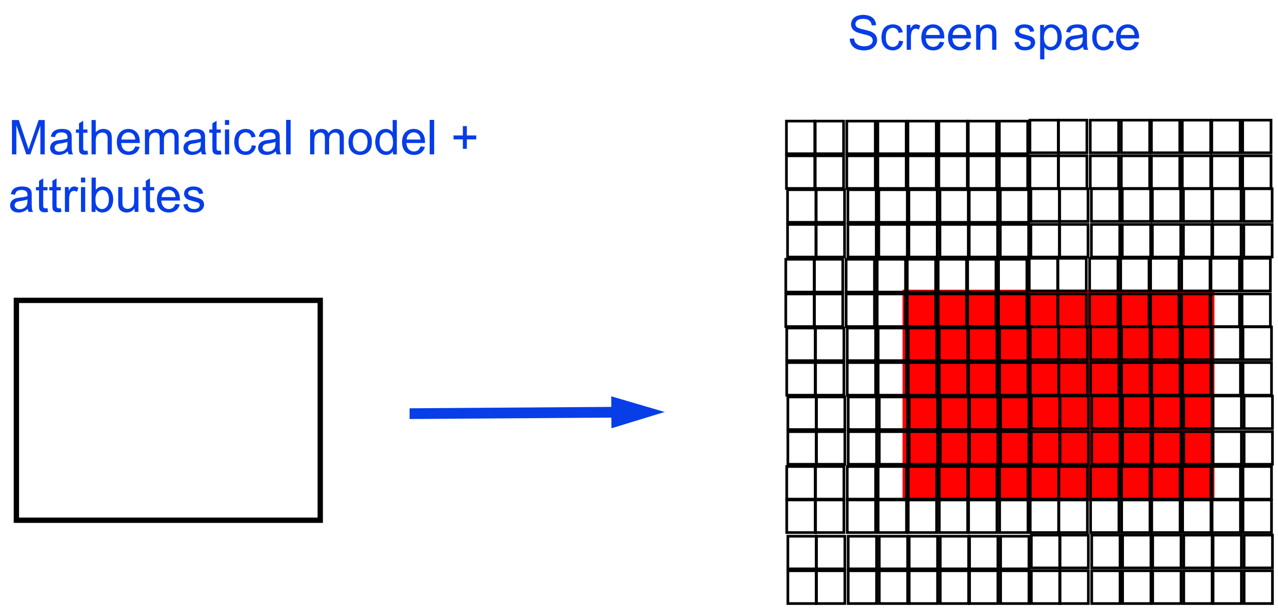
```
glPolygonMode(GL_FRONT,GL_FILL) ;

glPolygonMode(GL_BACK,GL_LINE) ;

glBegin(GL_POLYGON)

glNormal3f(v1,v2,v3) ;

glVertex3f(x1,y1,z1) ;

…

glNormal3f(v1n,v2n,v3n) ;

glVertex3f(xn,yn,zn) ;

glEnd()  ;
```

# Polygon Rasterization

*[Hill: 570-576. Foley & vanDam: p. 92-99]*

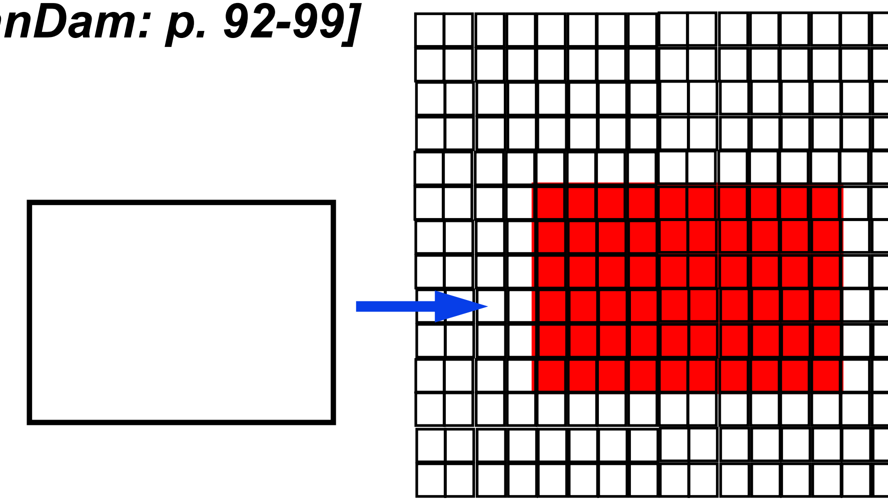
## *Scan conversion*

shade pixels lying within a closed polygon efficiently.



# Polygon Rasterization

*[Hill: 570-576. Foley & vanDam: p. 92-99]*

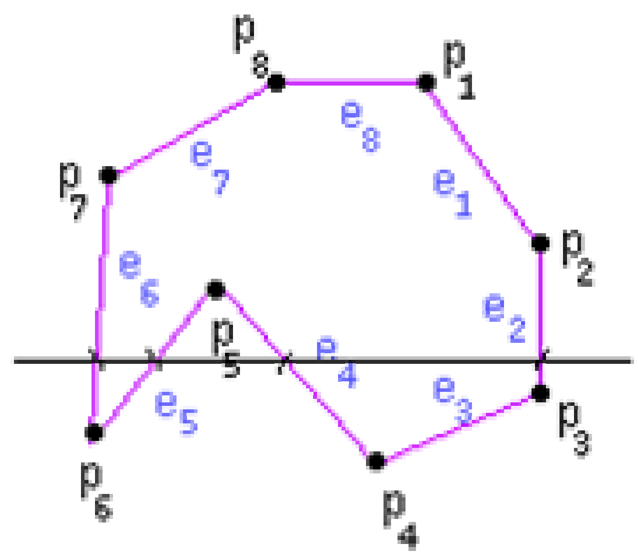


## *Scan conversion*

shade pixels lying within a closed polygon efficiently.

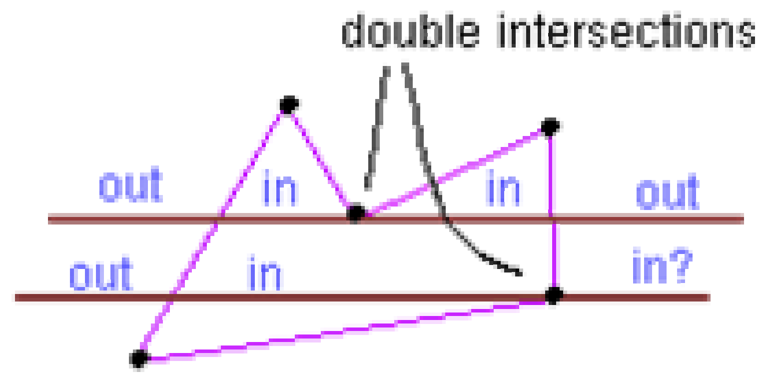
## *Algorithm*

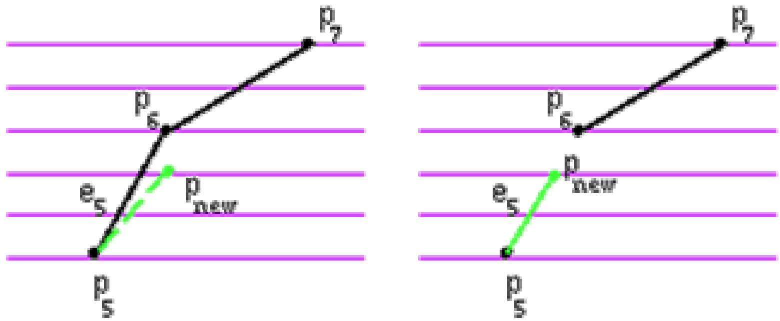
- intersect each scanline with all edges
- sort intersections in x
- calculate parity of intersections to determine in/out
- fill the 'in' pixels



# Special cases

- Horizontal edges can be excluded
- Vertices lying on scanlines
  - *Change in sign of $y_i$-$y_{i+1}$: count twice*
  - *No change: shorten edge by one scanline*



# Efficiency?

***Many intersection tests can be eliminated by taking advantage of coherence between adjacent scanlines.***

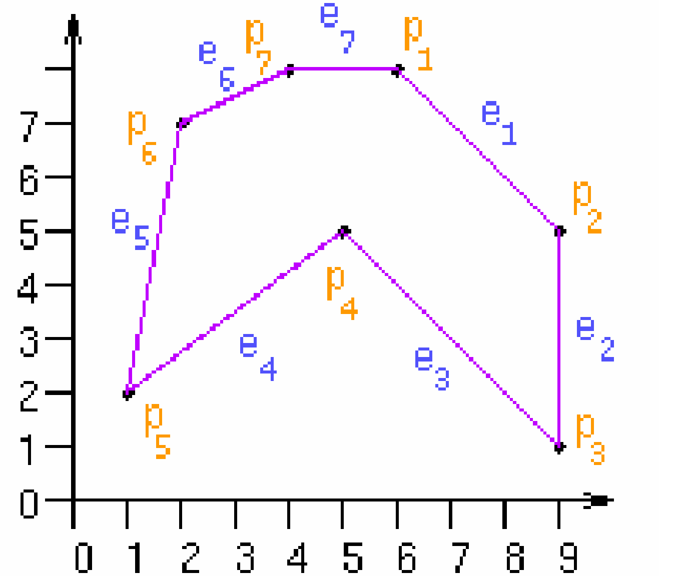
- Edges that intersect scanline y are likely to intersect y+1
- x changes predictably from scanline y to y+1

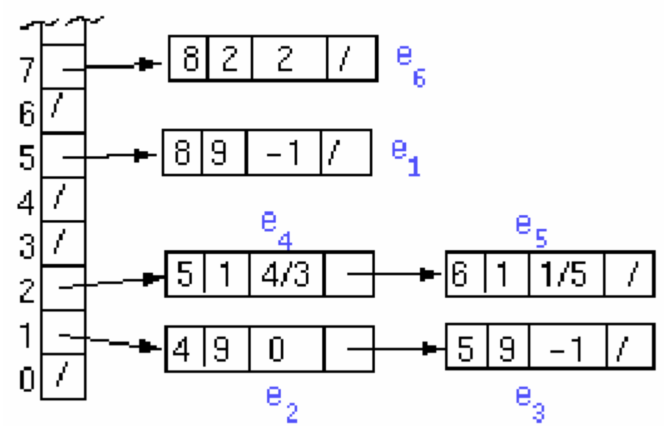
$y = mx+a \rightarrow x = 1/m(y+a) \rightarrow x(y+1) = x(y) + 1/m$

# Data structure 1:  Edge table

***Building edge table***

- Traverse edges
- Eliminate horizontal edges
- If not local extremum, shorten upper vertex
- Add edge to linked-list for the scanline corresponding to the lower vertex, storing:
  - *y_upper: last scanline to consider*
  - *x_lower: starting x coordinate for edge*
  - *1/m:      for incrementing x; compute before shortening*



# Data structure 2: Active Edge List (AEL)

- The AEL is a linked list of active edges on the current scanline, y.
- Each active edge has the following information:
  - *y_upper: last scanline to consider*
  - *x: edge's intersection with current y)*
  - *1/m: for incrementing x*

The active edges are kept sorted by x.

# Scan conversion algorithm

```
for each scanline
    add edges in edge table to AEL
    if AEL <> NIL
        sort AEL by x
        fill pixels between edge pairs
        delete finished edges
        update each edge's x in AEL
```

# Example

for each scanline
- add edges in edge table to AEL
- if AEL <> NIL
  - sort AEL by x
  - fill pixels between edge pairs
  - delete finished edges
  - update each edge's x in AEL

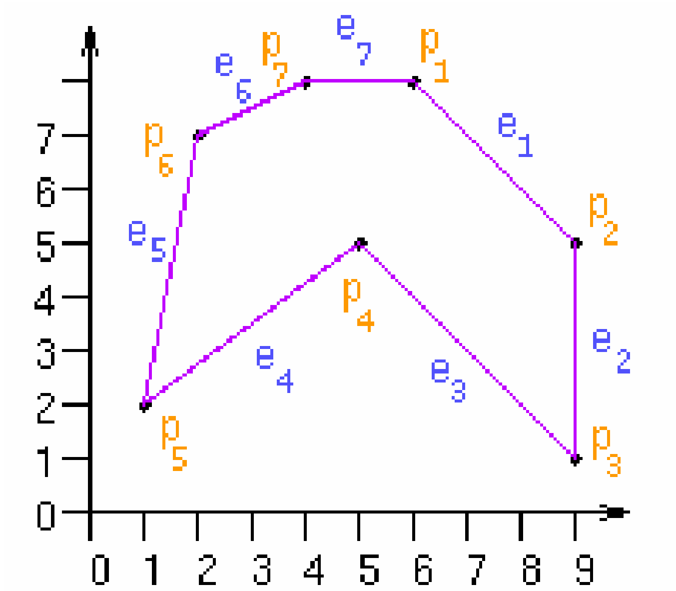
Reminder:

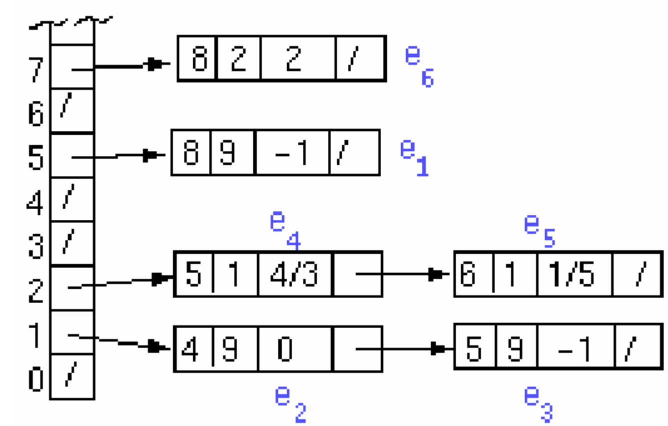
Edge table

| | | |
|---|---|---|
| y_upper | x_lower | 1/m |

AEL:

| | | |
|---|---|---|
| y_upper | x_current | 1/m |



# Special cases

***Triangles – Convex Polygons***

- Maximum two edges per scanline

***Overlaping polygons***

- priorities

***Color, patterns***

***Z for visibility***

# Interpolating information (incrementally)

## *Color, Normal, Texture coordinates*

Right edge (1,2):

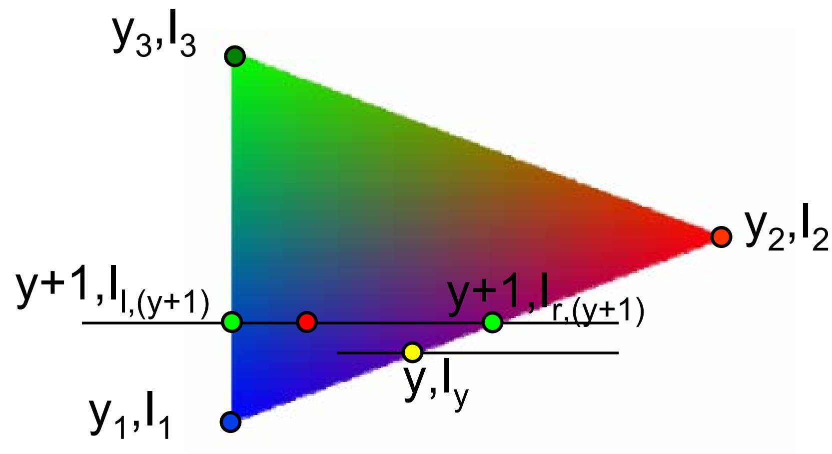
$$\frac{I_{r,(y+1)} - I_{r,y}}{(y+1) - y} = \frac{I_1 - I_2}{y_1 - y_2} \Rightarrow I_{r,(y+1)} = I_{r,y} + \frac{I_1 - I_2}{y_1 - y_2}$$

Left Edge (1,3):

$$\frac{I_{l,(y+1)} - I_{l,y}}{(y+1) - y} = \frac{I_1 - I_3}{y_1 - y_3} \Rightarrow I_{l,(y+1)} = I_{l,y} + \frac{I_1 - I_3}{y_1 - y_3}$$

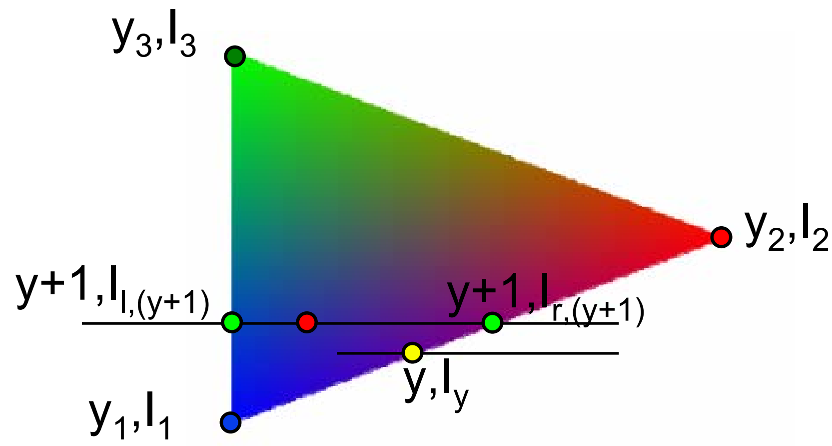
Along scanline:

$$\frac{I_{(x+1)} - I_x}{(x+1) - x} = \frac{I_r - I_l}{x_r - x_l} \Rightarrow I_{r,(y+1)} = I_{r,y} + \frac{I_r - I_l}{x_r - x_l}$$



# Interpolating information (incrementally)

## *Color, Normal, Texture coordinates*




Right edge (1,2):

$$\frac{I_{r,(y+1)} - I_{r,y}}{(y+1)-y} = \frac{I_1 - I_2}{y_1 - y_2} \Rightarrow I_{r,(y+1)} = I_{r,y} + \frac{I_1 - I_2}{y_1 - y_2}$$

Left Edge (1,3):

$$\frac{I_{l,(y+1)} - I_{l,y}}{(y+1)-y} = \frac{I_1 - I_3}{y_1 - y_3} \Rightarrow I_{l,(y+1)} = I_{l,y} + \frac{I_1 - I_3}{y_1 - y_3}$$

Along scanline:

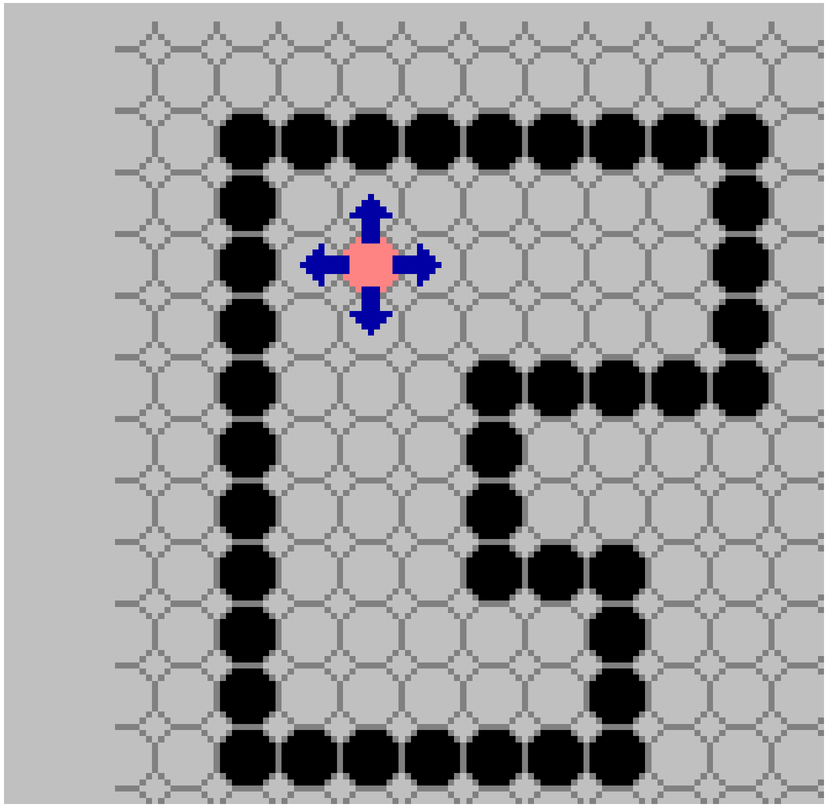
$$\frac{I_{(x+1)} - I_x}{(x+1)-x} = \frac{I_r - I_l}{x_r - x_l} \Rightarrow I_{r,(y+1)} = I_{r,y} + \frac{I_r - I_l}{x_r - x_l}$$

Constant along the line

# Pixel Region filling algorithms[Hill 561-577]

***Scan convert boundary***

***Fill in regions***



2D paint programs

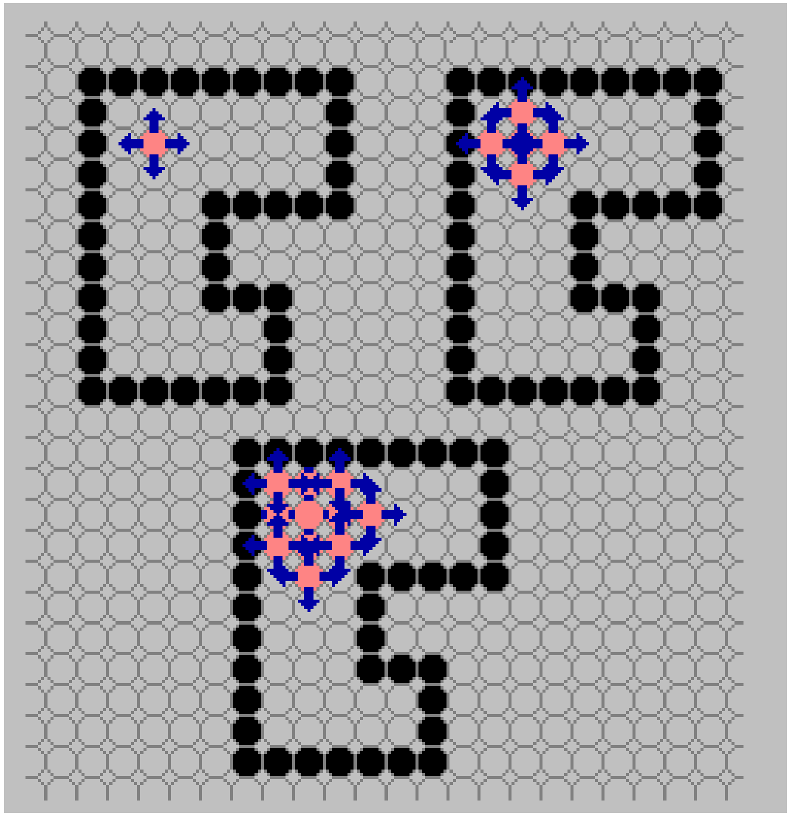
http://www.cs.unc.edu/~mcmillan/comp136/Lecture8/areaFills.html

# BoundaryFill

```
boundaryFill(int x, int y, int fill, int boundary) {
        if ((x < 0) || (x >= raster.width)) return;
        if ((y < 0) || (y >= raster.height)) return;
        int current = raster.getPixel(x, y);
        if ((current != boundary) & (current != fill)) {
            raster.setPixel(fill, x, y);
            boundaryFill(x+1, y, fill,boundary);
            boundaryFill(x, y+1, fill, boundary);
            boundaryFill(x-1, y, fill, boundary);
            boundaryFill(x, y-1, fill, boundary);
        }
}
```

# Flood Fill

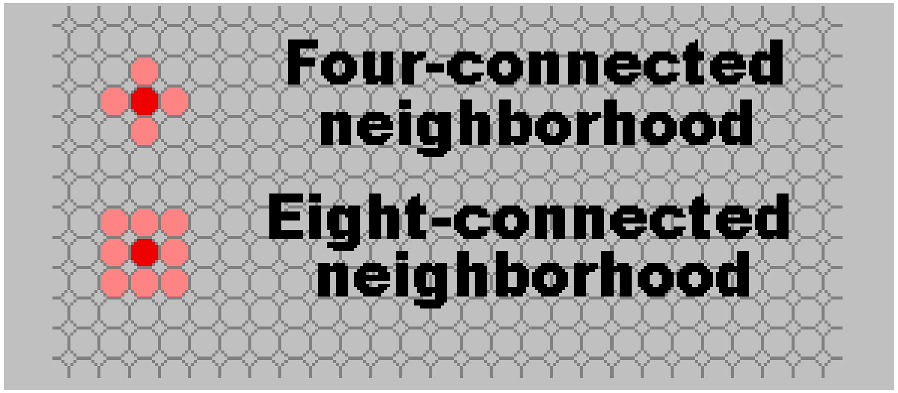
```
public void floodFill(int x, int y, int fill, int old)
{
    if ((x < 0) || (x >= raster.width)) return;
    if ((y < 0) || (y >= raster.height))  return;
    if (raster.getPixel(x, y) == old) {
        raster.setPixel(fill, x, y);
        floodFill(x+1, y, fill, old);
        floodFill(x, y+1, fill, old);
        floodFill(x-1, y, fill, old);
        floodFill(x, y-1, fill, old);
    }
}
```



# Adjacency

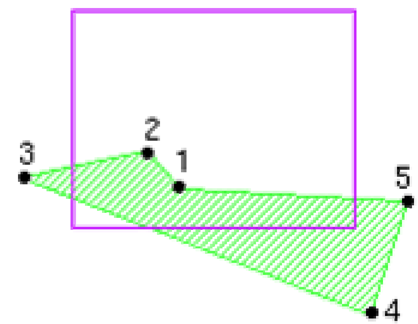
*4-connected*

*8 connected*



# Polygon clipping (2D)[Hill 181-208]

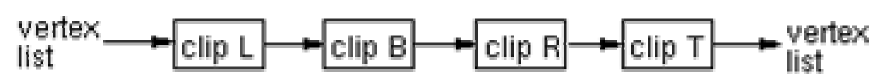
## *Sutherland-Hodgeman [Hill 202]*

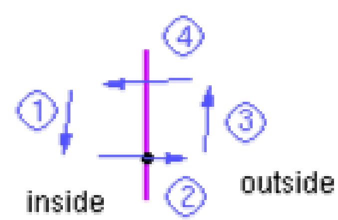



for each side of clipping window

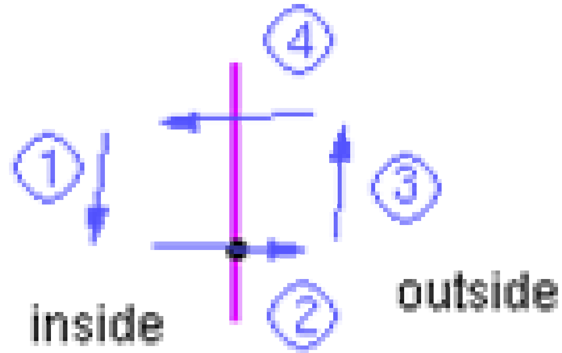
for each edge of polygon

output points based upon the following table

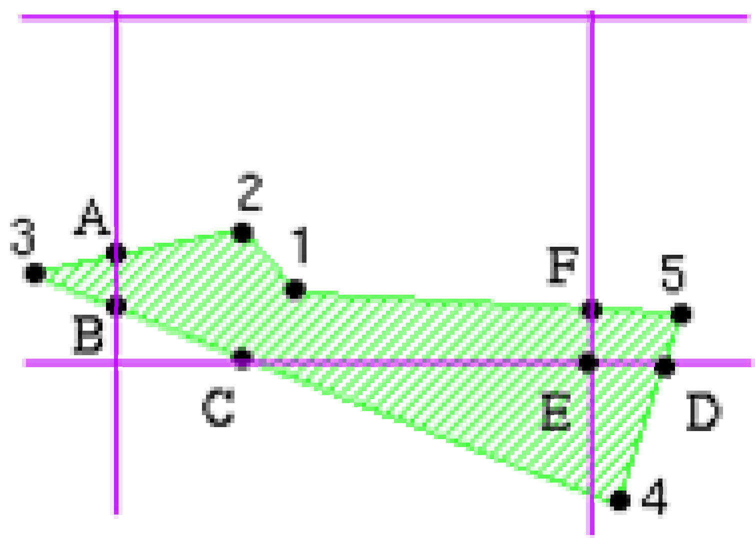



| case # | first point | second point | output point(s) |
|---|---|---|---|
| 1 | inside | inside | second point |
| 2 | inside | outside | intersection point |
| 3 | outside | outside | none |
| 4 | outside | inside | intersection point and second point |




| case # | first point | second point | output point(s) |
|---|---|---|---|
| 1 | inside | inside | second point |
| 2 | inside | outside | intersection point |
| 3 | outside | outside | none |
| 4 | outside | inside | intersection point and second point |




original: 1,2,3,4,5,1

clip L: 1,2,A,B,4,5,1

clip B: 1,2,A,B,C,D,5,1

clip R: 1,2,A,B,C,E,F,1

clip T: (same)

# Outcodes for trivial reject/accept

***[Hill 389] A vertex outcode consists of four bits: TBRL, where:***

T is set if y > top,

B is set if y < bottom,

R is set if x > right, and

L is set if x < left.

Trivial accept: all vertices are inside (all outcodes are 0000, bitwise OR)

Trivial reject: all vertices are outside with respect to any given side(bitwise AND is not 0000)