光栅图形学

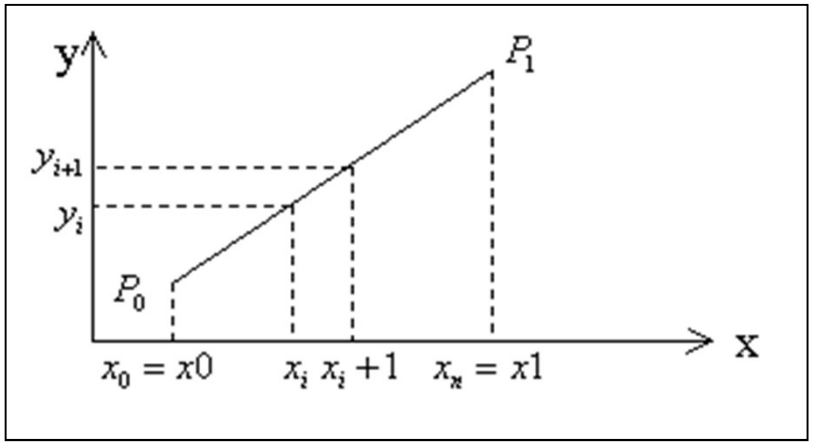
- 基本图形的扫描转换(scan conversion)
  (直线、圆弧、椭圆弧)
- 多边形的扫描转换与区域填充(area filling)
- 裁剪(clipping)
- 反走样(antialiasing)
- 投影(projection)
- 消隐(visible-surface detection / hidden-surface elimination)

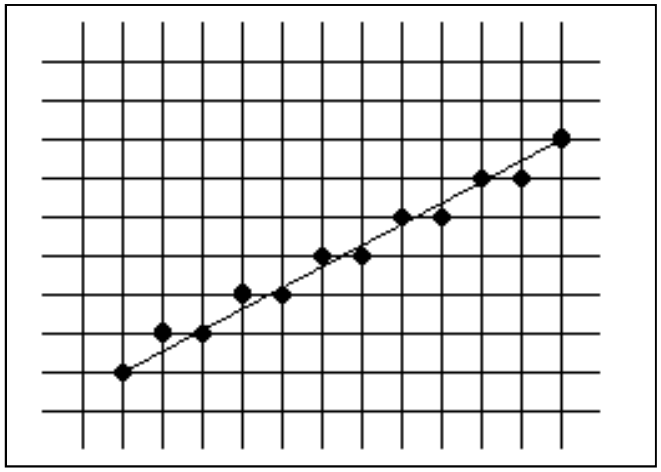
第三章

图形的扫描转换与区域填充

清华大学软件学院 张慧 2019-09-17

直线段的扫描转换





直线段的扫描转换

- 直线的扫描转换：确定最佳逼近于该直线的一组象素，并且按扫描线顺序，对这些象素进行写操作。
- 常用算法：
  - 数值微分法(DDA)
  - 中点画线法
  - **Bresenham**算法

DDA算法 (digital differential analyzer)

- 基本思想
  - 已知过端点$P_0(x_0,y_0)$, $P_1(x_1,y_1)$的直线段L：$y=kx+b$
  - 直线斜率为：$$k=\frac{y_1-y_0}{x_1-x_0}$$
  - 在x方向从x_0开始，向x_1步进，步长取1个象素，计算相应的y坐标$y=kx+b$。取象素点$(x, \text{round}(y))$作为当前点的坐标。

- 为了加速，采用增量算法的思想。
  - 计算： $y_{i+1} = kx_{i+1} + b$

    $= kx_i + b + k\Delta x$

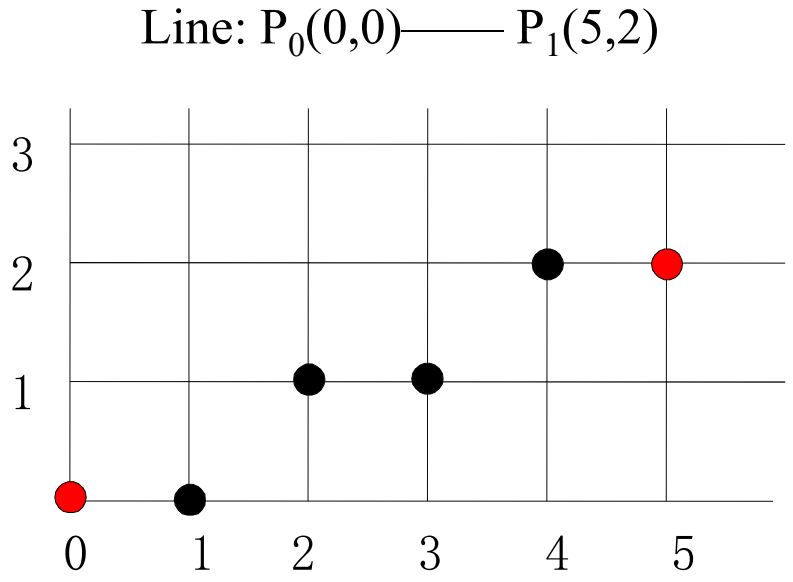
    $= y_i + k\Delta x$
  - 当 $\Delta x = 1$ 时，$y_{i+1} = y_i + k$

    即：当***x***每递增**1**，***y***递增***k***(即直线斜率)；

例：画直线段$P_0(0,0)$——$P_1(5,2)$

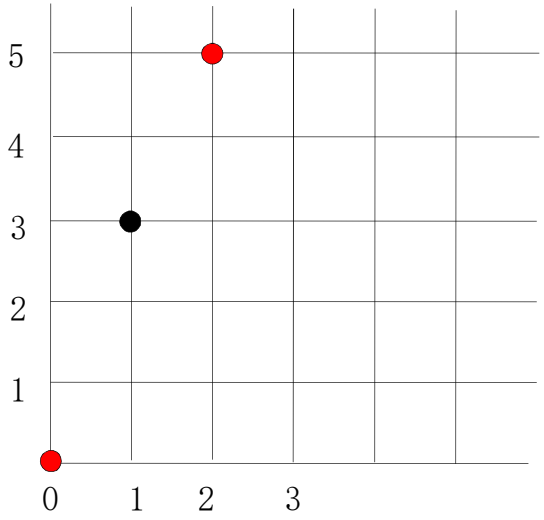
k=0.4

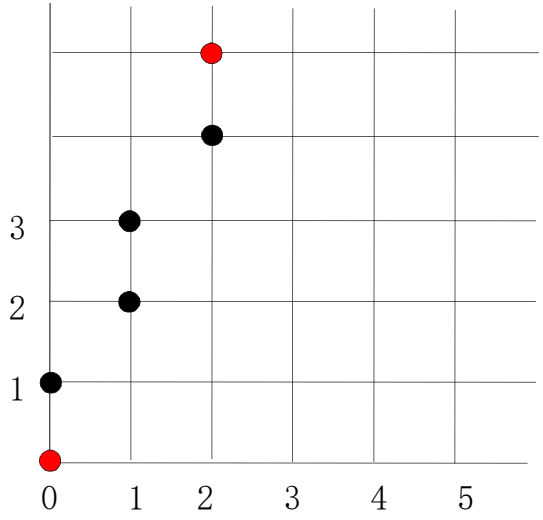
x	int(y+0.5)	y+0.5
0	0	0.5
1	0	0.5+0.4
2	1	0.9+0.4
3	1	1.3+0.4
4	2	1.7+0.4
5	2	2.1+0.4



```
void DDALine(int x0,int y0,int x1,int y1,int color)
 {
    int x;
    float dx, dy, y, k;
    dx=x1-x0; dy=y1-y0;
    k=dy/dx; y=y0+0.5;
    for (x=x0; x≤x1; x++)
    {  drawpixel (x, int(y), color);
        y=y+k;
     }
 }
```

- 问题：
  - 当 $|k|>1$时，算法是否适用？
  - 上述分析的算法仅适用于$|k| \leq 1$的情形。在这种情况下，x每增加1，y最多增加1。
  - $|k|>1$时，则应该在y方向步进，求相应的x坐标。



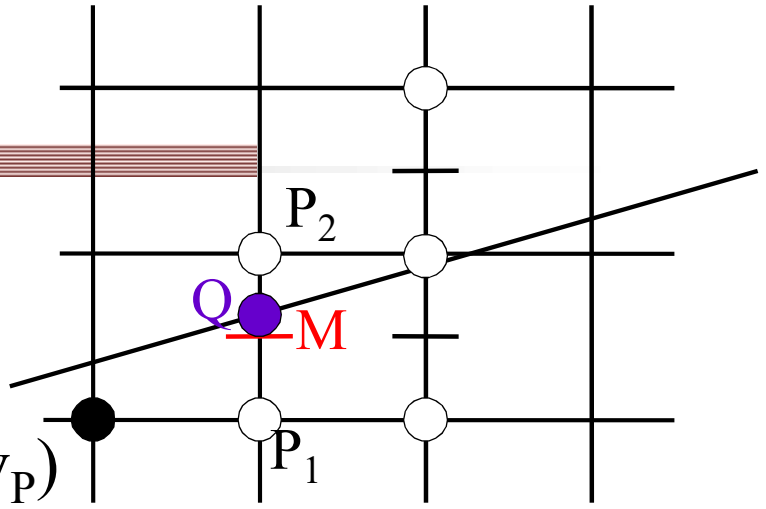


Line: $P_0(0,0)$—— $P_1(2,5)$

$|k|>1$ 示意图

- 思考
  - 采用增量思想的**DDA**算法，每计算一个象素，只需计算一个加法和一个取整，是否最优？如非最优，如何改进？
  - 目标：进一步将一个浮点数加法改为一个整数加法。
  - 新思路：可否采用直线的其它表示方式？

中点画线法



- 基本思想
  - 设当前象素点为(x_p, y_p)，下一个象素点为P_1或P_2
  - 设$M=(x_p+1, y_p+0.5)$，为P_1与P_2之中点，Q为理想直线与垂线$x=x_p+1$的交点
  - 将Q与M的y坐标进行比较。
    - 当M在Q的下方，则下一个象素点取P_2；
    - 当M在Q的上方，则下一个象素点取P_1。

- 对直线采用方程：$F(x, y)=ax+by+c=0$

其中$a=y_0-y_1$, $b=x_1-x_0$, $c=x_0y_1-x_1y_0$

- 构造判别式：

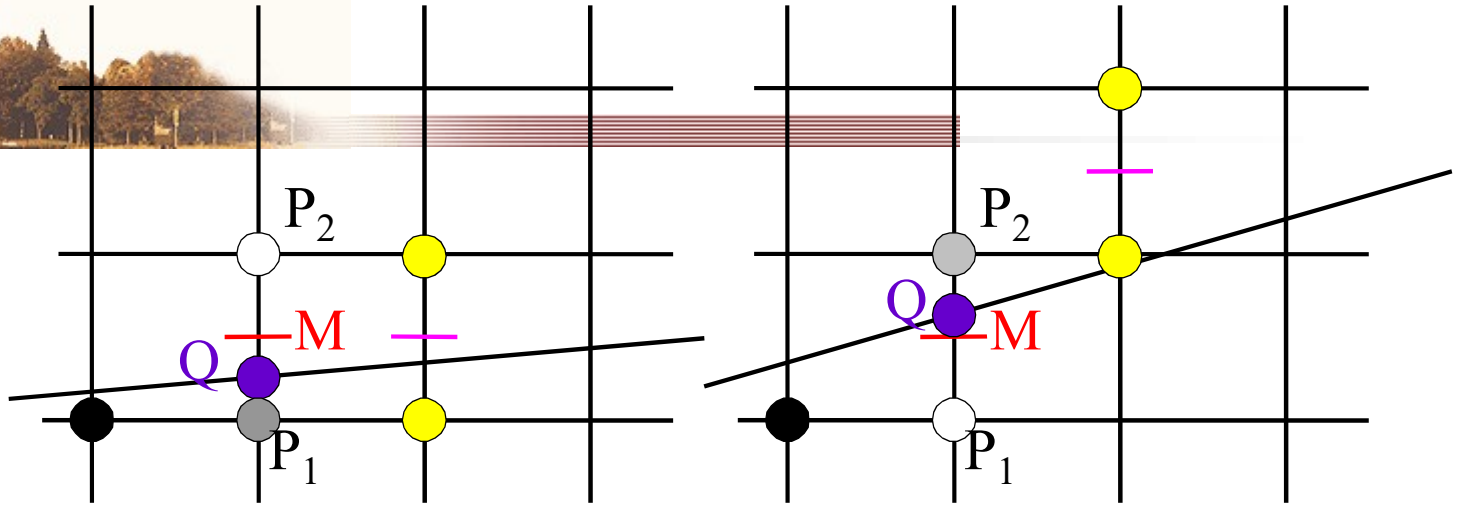
$d=F(M)=F(x_p+1, y_p+0.5)=a(x_p+1)+b(y_p+0.5)+c$

- 当$d<0$，M在线段L(Q点)下方，取右上方点P_2为下一个象素；
- 当$d>0$，M在L(Q点)上方，取正右方点P_1为下一个象素；
- 当$d=0$，选P_1或P_2均可，约定取P_1为下一个象素；

$$d=\mathrm{F}(M)=\mathrm{F}(x_p+1, y_p+0.5)=a(x_p+1)+b(y_p+0.5)+c$$

- 问题
  - 每一个象素的计算量是4个加法，2个乘法
- 是否也可以采用增量算法？
  - d是x_p, y_p的线性函数，因此可采用增量计算，提高运算效率。

- 增量



- 若当前象素处于$d \geq 0$情况，则取正右方象素P_1 (x_p+1, y_p)。要判下一个象素位置，应计算

  $d_1=F(x_p+2, y_p+0.5)=a(x_p+2)+b(y_p+0.5)+c=d+a$ 故增量为a;

- 若$d<0$时，则取右上方象素P_2 (x_p+1, y_p+1)。要判断再下一象素，则要计算

$d_2= F(x_p+2, y_p+1.5)=a(x_p+2)+b(y_p+1.5)+c=d+a+b$

故增量为$a+b$

- 至此，至少新算法可以和**DDA**算法一样好。
- 能否实现整数运算？

$$a=y_0-y_1,\ b=x_1-x_0$$

- 画线时从(x_0, y_0)开始，d的初值

$$d_0=F(x_0+1, y_0+0.5)=F(x_0, y_0)+a+0.5b=a+0.5b$$

- 可以用$2d$代替d来摆脱小数，提高效率。

令 $d_0=2a+b$, d_1的增量$\Delta d_1=2a$,

d_2的增量$\Delta d_2=2a+2b$, 可得到如下算法：

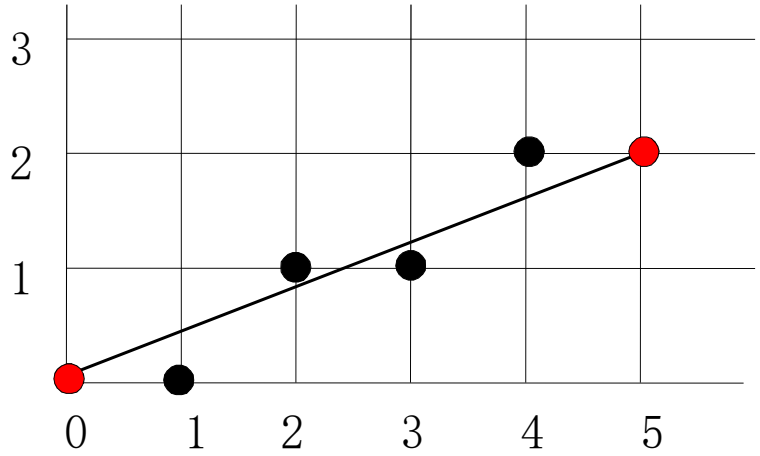
```
void Midpoint Line (int x0, int y0, int x1, int y1, int color)
 {  int a, b, d1, d2, d, x, y;
    a=y0-y1; b=x1-x0; d=2*a+b;
    d1=2*a; d2=2* (a+b);
    x=x0; y=y0;
    drawpixel(x, y, color);
    while (x<x1)
    { if (d<0)       {x++; y++; d+=d2; }
          else       {x++; d+=d1;}
      drawpixel (x, y, color);
     }
}
```

例：用中点画线法画$P_0(0,0)$——$P_1(5,2)$

$a=y_0-y_1=-2$，　$b=x_1-x_0=5$

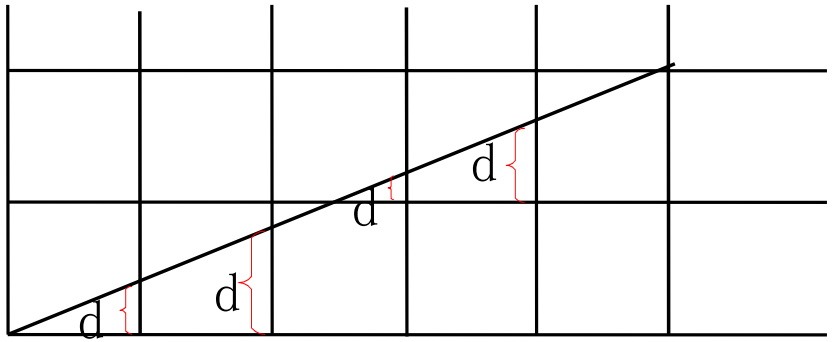
$d_0=2a+b=1$, $\Delta d_1=2a=-4$, $\Delta d_2=2(a+b)=6$

i	x_i	y_i	d
1	0	0	1
2	1	0	-3
3	2	1	3
4	3	1	-1
5	4	2	5
6	5	2	1



Bresenham算法

- 基本思想
  - 过各行各列象素中心构造一组虚拟网格线。按直线从起点到终点的顺序计算直线与各垂直网格线的交点，然后根据误差项的符号确定该列象素中与此交点最近的象素。

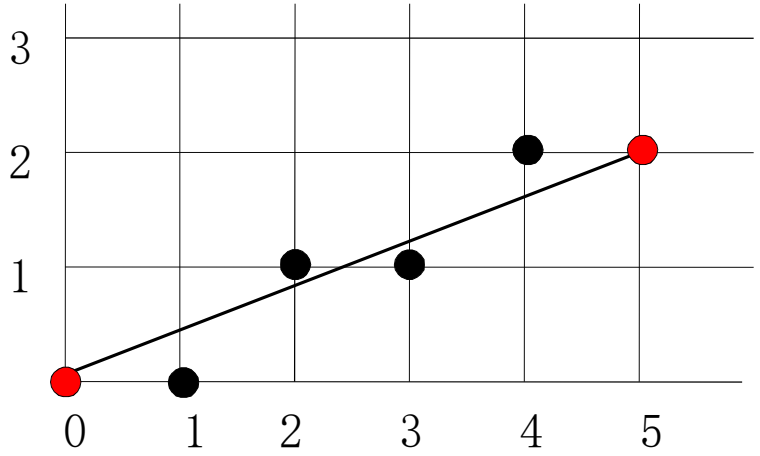


- 设直线方程为：$y_{i+1}=y_i+k(x_{i+1}-x_i)$，其中$k=dy/dx$。因为直线的起始点在象素中心，所以误差项$d$的初值$d_0=0$。
- x下标每增加1，d的值相应递增直线的斜率值k，即$d=d+k$。
  - 当$d<0.5$时，更接近于正右方象素(x_{i+1},y_i)
  - 当$d\geq0.5$时，更接近于当前象素的右上方象素(x_{i+1},y_{i+1})
- 为方便计算，令$e=d-0.5$
  - 当$e<0$时，取当前象素(x_i,y_i)的正右方象素(x_{i+1},y_i)
  - 当$e\geq0$时，取当前象素(x_i,y_i)的右上方象素(x_{i+1},y_{i+1})，e重新计算

- 例：$P_0(0, 0)$——$P_1(5,2)$

$k=dy/dx=0.4$, $e_0=-0.5$, $e_{i+1}=e_i+k$

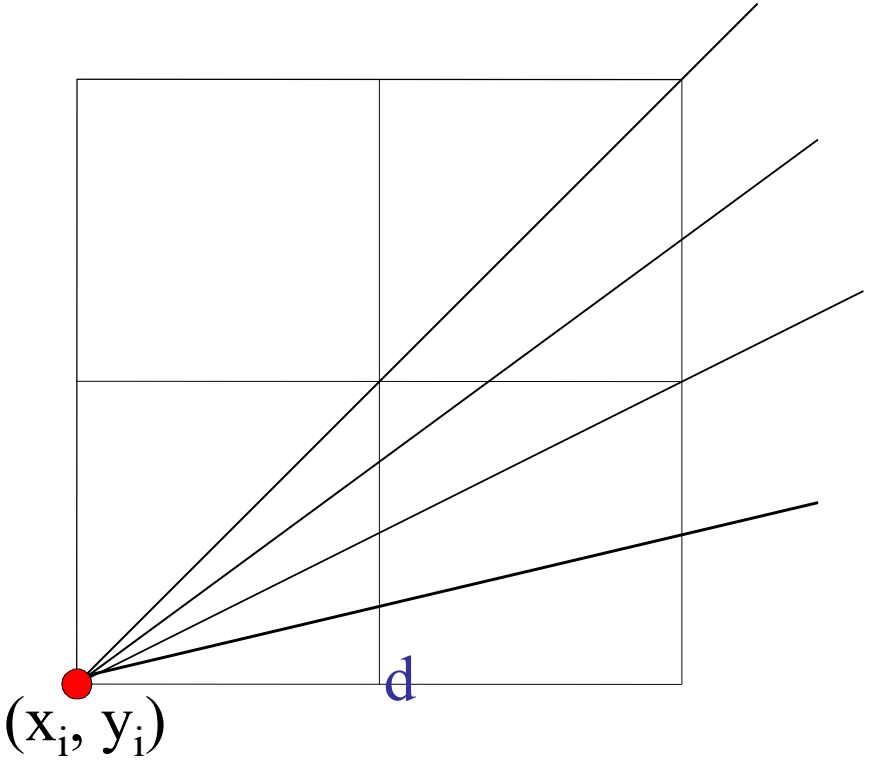
x	y		e
0	0		-0.5
1	0	←	-0.1
2	1	←	0.3
3	1	←	-0.3
4	2	←	0.1
5	2	←	-0.5



```
void Bresenhamline (int x0, int y0, int x1, int y1, int color)
{  int x, y, dx, dy;
   float k, e;
   dx = x1-x0; dy = y1- y0; k=dy/dx;
   e=-0.5; x=x0; y=y0;
   for (i=0; i<=dx; i++)
   {    drawpixel (x, y, color);
        x=x+1; e=e+k;
        if (e>=0)
        { y++; e=e-1;}
   }
 }
```

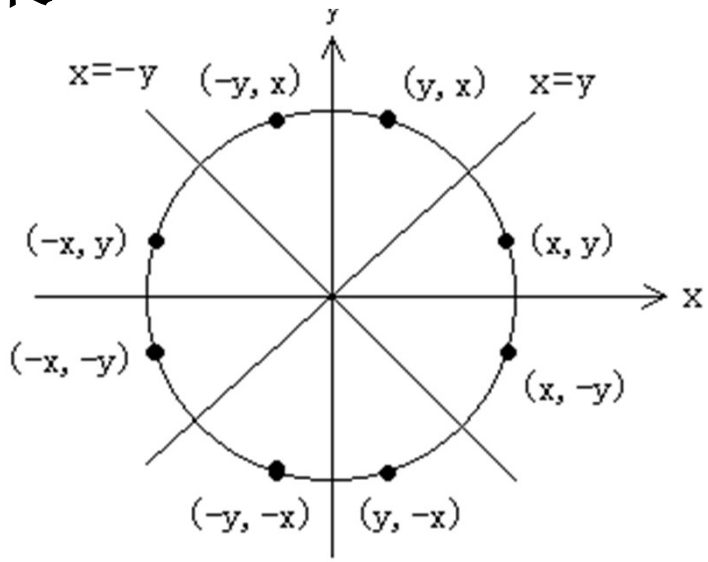
- 可以改用整数以避免除法。由于算法中只用到误差项的符号，因此可作如下替换：　　$e'=2\times e\times dx=2dy-dx$
- **Bresenham**算法中，对每个象素，仅需一个整数运算。
- 使用最广泛，可以用于其他二次曲线。
- 问题：是否能一次画接下去的两个象素，来加速算法？

- Bresenham算法加速?

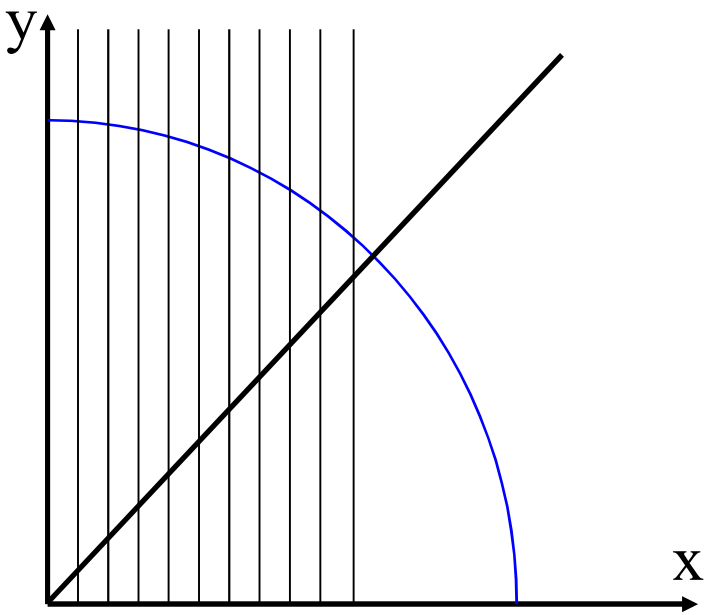


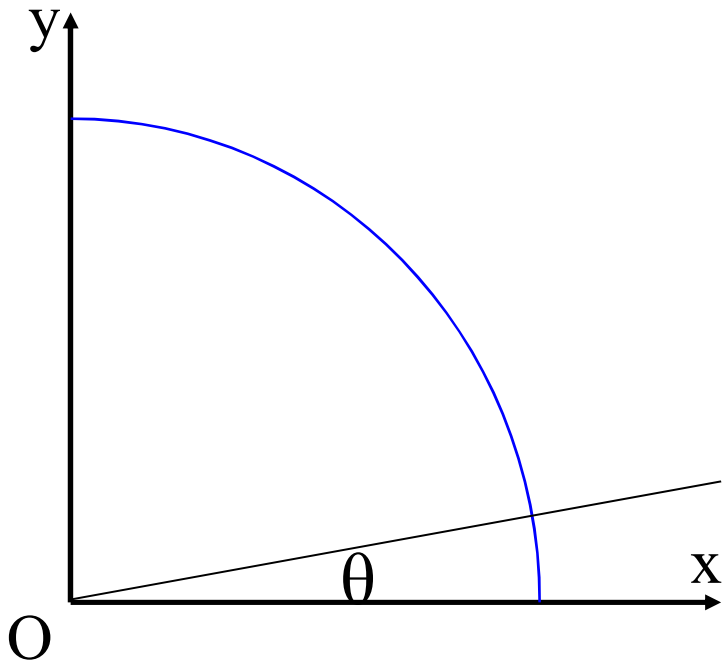
圆弧的扫描转换算法

- $x^2+y^2=R^2$
- 圆的特征：八对称性。
- 只要扫描转换八分之一圆弧，就可以求出整个圆的象素集



- **DDA?**





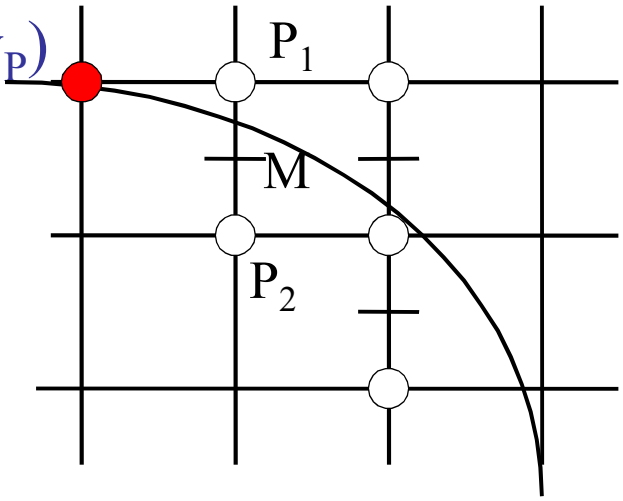
x_i

$y_i = \sqrt{R^2 - x_i^2}$

$x_i = R\cos\theta_i$

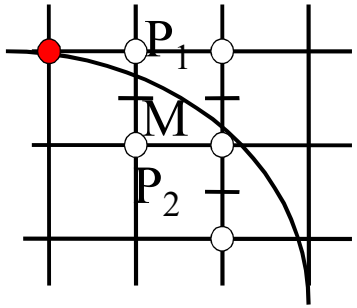
$y_i = R\sin\theta_i$

- 中点画圆法
  - 考虑中心在原点，半径为R的圆在第一象限内 $x \in [0, R/\sqrt{2}]$ 的八分之一圆弧
  - 构造判别式(圆方程)： $F(x, y)=x^2+y^2-R^2=0$
  - 已知$P(x_p, y_p)$
  - 中点 $M=(x_p+1, y_p-0.5)$



$$d=F(M)=F(x_p+1,y_p-0.5)=(x_p+1)^2+(y_p-0.5)^2-R^2$$

$$d=F(M)=F(x_p+1,y_p-0.5)=(x_p+1)^2+(y_p-0.5)^2-R^2$$



- 若 $d<0$, 则取P_1为下一象素，而且再下一象素的判别式为：

$$d'=F(x_p+2,y_p-0.5)=(x_p+2)^2+(y_p-0.5)^2-R^2=d+2x_p+3$$

- 若$d\geq 0$, 则应取P_2为下一象素，而且下一象素的判别式为：

$$d'=F(x_p+2,y_p-1.5)=(x_p+2)^2+(y_p-1.5)^2-R^2=d+2(x_p-y_p)+5$$

- 第一个象素是$(0,R)$，判别式d的初始值为：

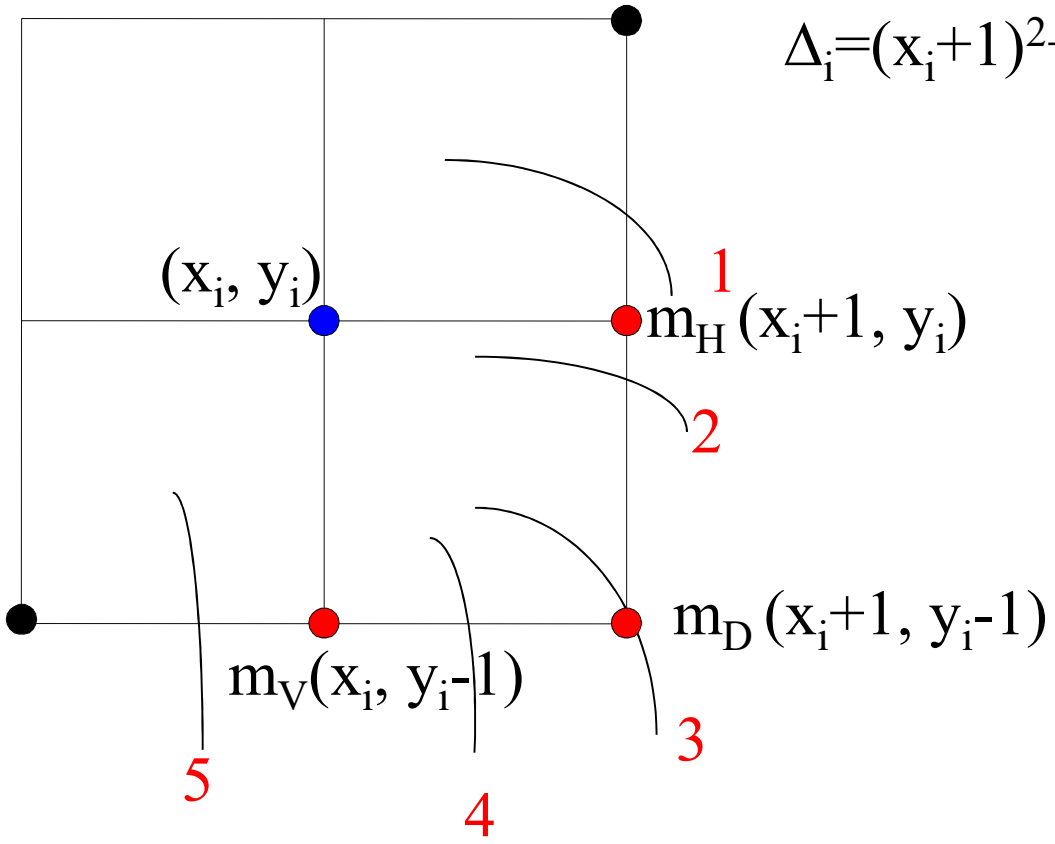
$$d_0=F(1, R-0.5)=1.25-R$$

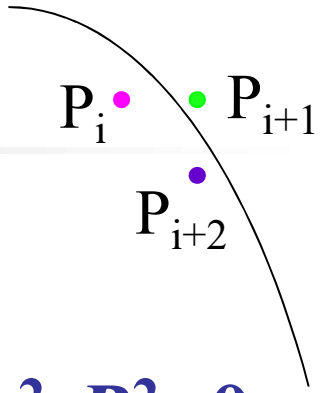
```
MidPointCircle(int r, int color)
{  int x,y;
    float d;
    x=0; y=r; d=1.25-r;
    circlepoints (x,y,color); //显示圆弧上的八个对称点
    while(x<=y)
   {    if(d<0)          d+=2*x+3;
        else    { d+=2*(x-y)+5;  y--;}
        x++;
        circlepoints (x,y,color);
   }
}
```

- Bresenham画圆法

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

$$\Delta_i=(x_i+1)^2+(y_i-1)^2-R^2$$

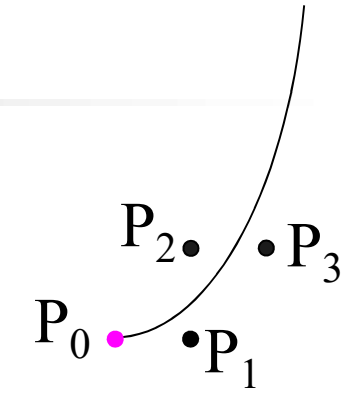




生成圆弧的正负法

- 由圆方程的隐函数$F(x, y)=x^2+y^2-R^2=0$，判断点与曲线之间关系。
- 已知$P_i(x_i, y_i)$，求P_{i+1}的原则：
  - 当$F(x_i, y_i) \le 0$，取$x_{i+1}=x_i+1, y_{i+1}=y_i$；
  - 当$F(x_i, y_i) > 0$，取$x_{i+1}=x_i, y_{i+1}=y_i-1$；
- 当$F(x_i, y_i) \le 0$时，$F(x_{i+1}, y_{i+1})= F(x_i, y_i) +2x_i+1$
- 当$F(x_i, y_i) > 0$时，$F(x_{i+1}, y_{i+1})= F(x_i, y_i) -2y_i+1$

正负法



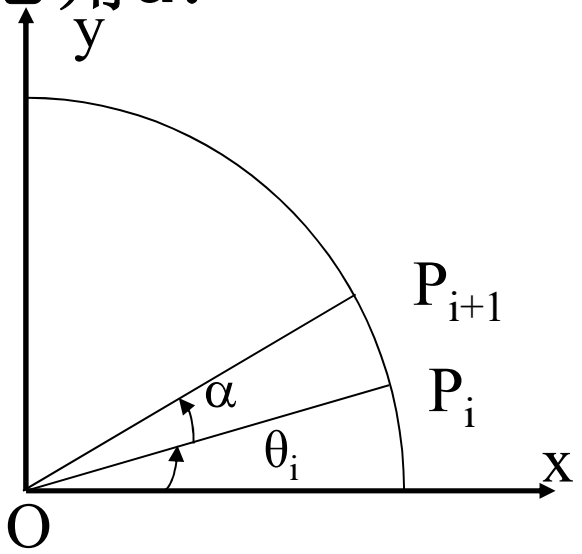
- 基本原理
  - 已知曲线方程的隐函数形式$F(x, y)=0$，该曲线将平面分成三部分G_+，G_-，G_0。
  - 曲线上的起始点P_0
  - 由P_0向某个方向前进一个步长，得到P_1，此时再沿另一方向前进一个步长，得到P_2，再按照P_2所属的区域决定下一步的前进方向……
- 优点：只需判断$F(P_i)$的符号
- 适用范围

圆的多边形逼近法

- 基本思想：当圆的内接多边形边数足够多时，可以用该多边形近似圆。
- 求圆的内接正多边形
  - 多边形每条边对应的圆心角**α**：

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} R\cos\theta_i \\ R\sin\theta_i \end{bmatrix}$$

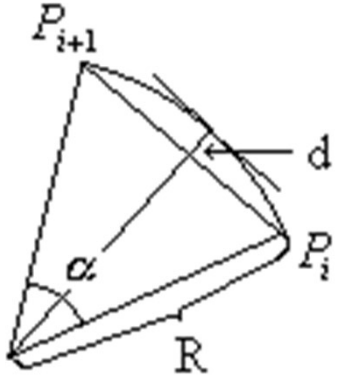


- 问题：给定最大逼近误差**delta**，确定多边形的边数***n***，***n*=2*π*/*α***。

$$R - R\cos\frac{\alpha}{2} \le \text{delta}$$

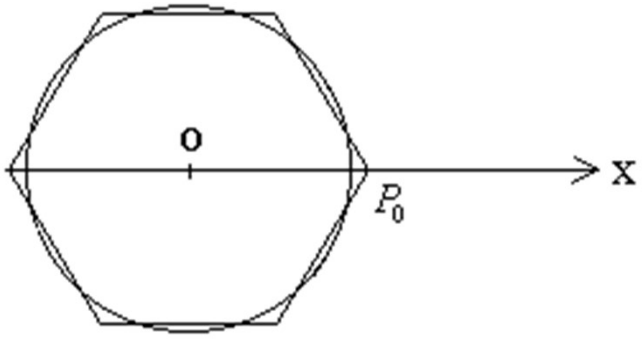
$$\Rightarrow \cos\frac{\alpha}{2} \ge \frac{R-\text{delta}}{R}$$

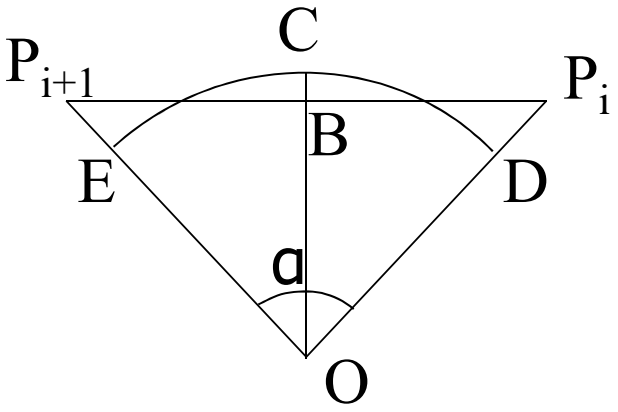
$$\Rightarrow \alpha \le 2\arccos\frac{R-\text{delta}}{R}$$



圆的等面积多边形逼近法

- 求多边形径长，从而得到多边形各顶点坐标值；





- 扇形**ODCE**面积 = 三角形OP_iP_{i+1}面积

$$|OP_i| = \sqrt{\frac{\alpha}{\sin\alpha}}R$$

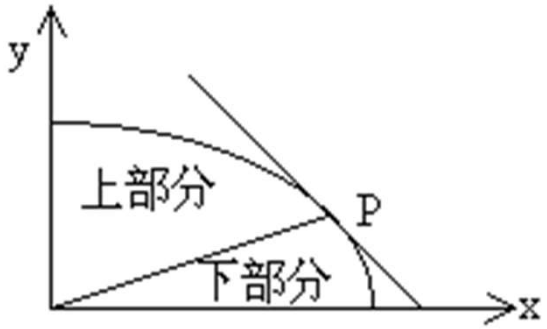
- 由逼近误差值**delta**，确定边所对应的圆心角α。

$$|BC| \le \text{delta}$$

$$\Rightarrow R - |OP_i| \cos\frac{\alpha}{2} \le \text{delta}$$

$$\Rightarrow R - R\sqrt{\frac{\alpha}{\sin\alpha}} \cos\frac{\alpha}{2} \le \text{delta}$$

椭圆的扫描转换

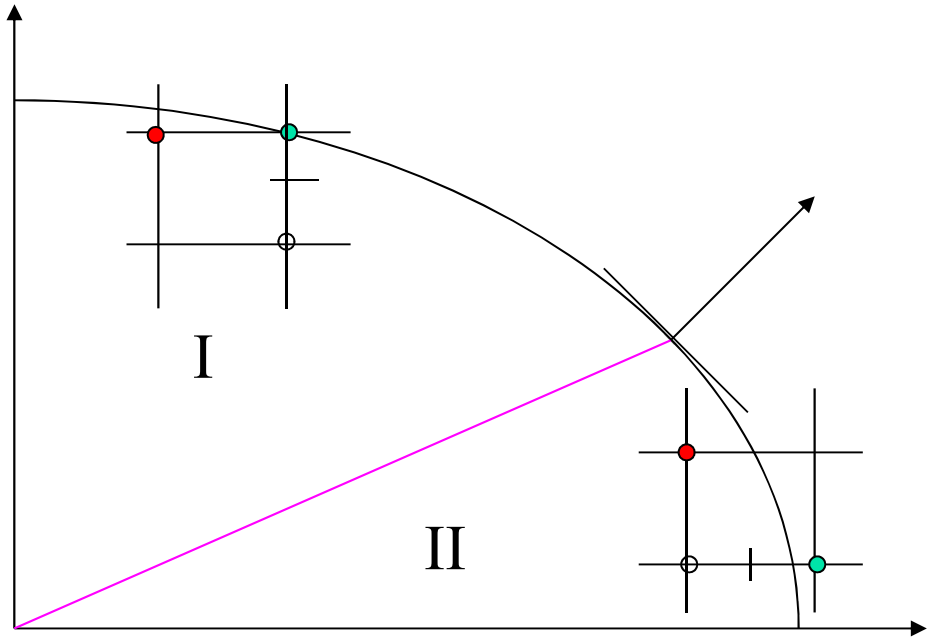


- $F(x,y)=b^2x^2+a^2y^2-a^2b^2=0$
- 由椭圆的对称性，可以只考虑第一象限椭圆弧生成：分上下两部分，以弧上切线斜率为-1的点(法向x, y两个分量相等的点)作为分界点。

椭圆的中点画法

- 与圆弧中点算法类似：确定一个象素后，接着在两个候选象素的中点计算一个判别式的值，由判别式的符号确定更近的点。
- 已知第一部分的点(x_p, y_p)
- 两个候选点的中点$(x_p+1, y_p-0.5)$

$$d_1=F(x_p+1,y_p-0.5)=b^2(x_p+1)^2+a^2(y_p-0.5)^2-a^2b^2$$


I
II

- 若$d_I<0$，中点在椭圆内，下一个象素取正右方象素，判别式更新为：

$$d_I'=F(x_p+2, y_p-0.5)=d_I+b^2(2x_p+3)$$

- 当$d_I\geq 0$，中点在椭圆外，下一个象素取右下方象素，判别式更新为：

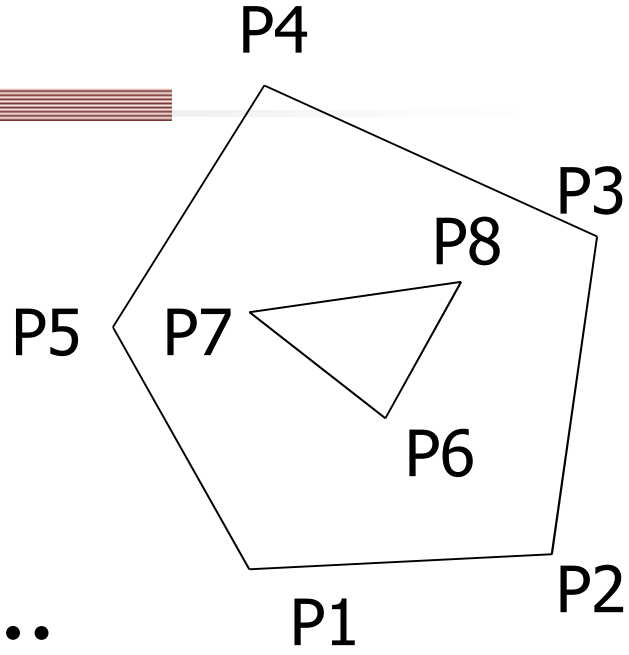
$$d_I'=F(x_p+2, y_p-1.5)=d_I+b^2(2x_p+3)+a^2(-2y_p+2)$$

- 椭圆弧起点为(0, b)，d_I判别式的初始条件：

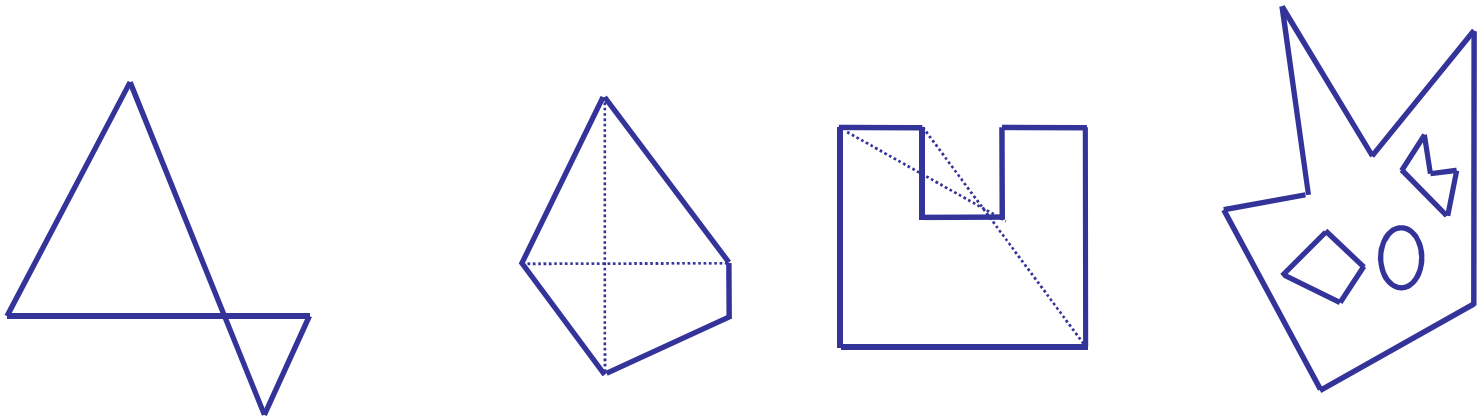
$$d_{I0}=F(1, b-0.5)=b^2+a^2(-b+0.25)$$

- 转入第二部分后，下一象素可能是正下方象素或右下方象素，此时判别式要初始化。

线框显示 ➡ 面着色

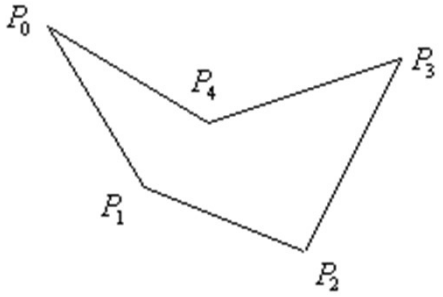


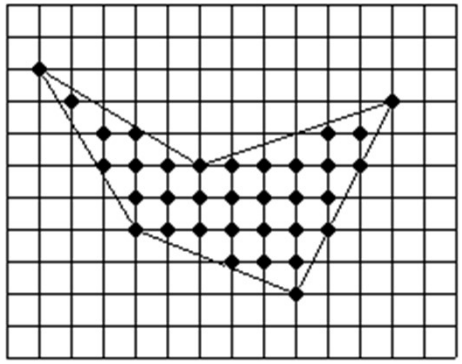
- 多边形的一些概念：
  - 凸多边形、凹多边形
  - 外环、内环、方向……
  - 内部



多边形的扫描转换

- 多边形有两种重要的表示方法：顶点表示和点阵表示。





- 多边形的扫描转换：把多边形的顶点表示转换为点阵表示。

逐点判断法

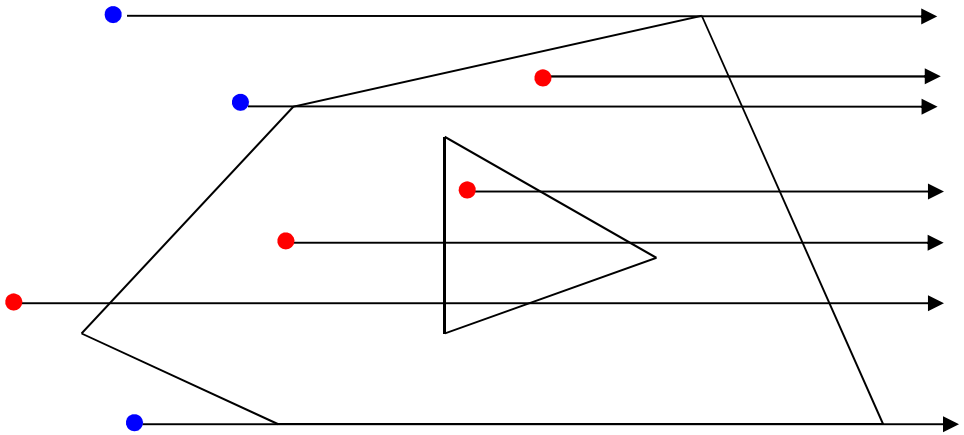
- 基本思想：逐个象素判断，确定它们是否在多边形内。

```
for(y=0; y<MaxY; y++)
  for(x=0; x<MaxX; x++)
    if inside(x, y, P)
       DrawPixel(x, y, PolygonColor);
    else DrawPixel(x, y, BackgroundColor);
```

- 点在多边形内的判别方法：
  - 射线法
  - 累计角度法

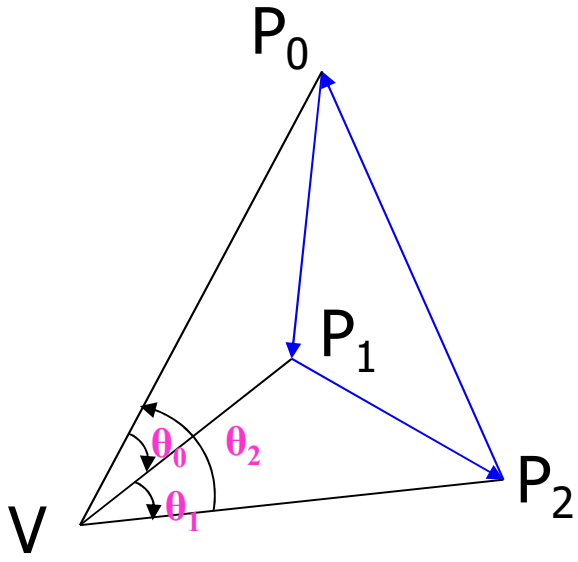
- 程序简单，但速度太慢，效率低。

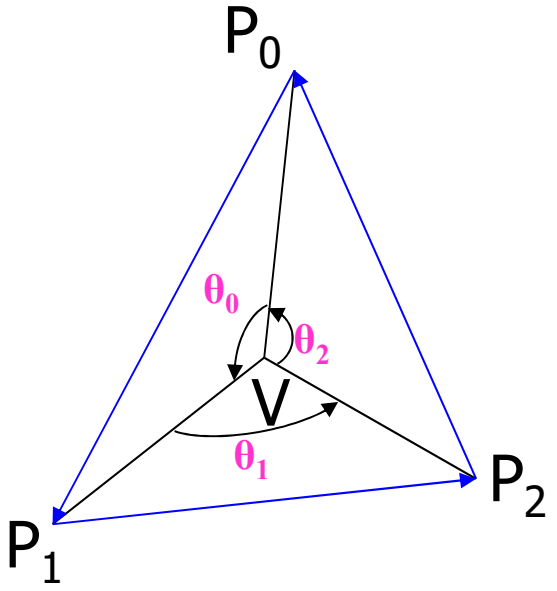
- 射线法
  - *V*点发出射线，与多边形边相交，若交点个数为偶数，则*V*点在多边形外边，反之在多边形内。
  - 交点通过多边形顶点?
    - 使射线通过边的中点(不考虑边的端点)



- 累计角度法
  - 从***V***点向多边形***P***的各顶点发出射线，形成有向角，计算有向角的和

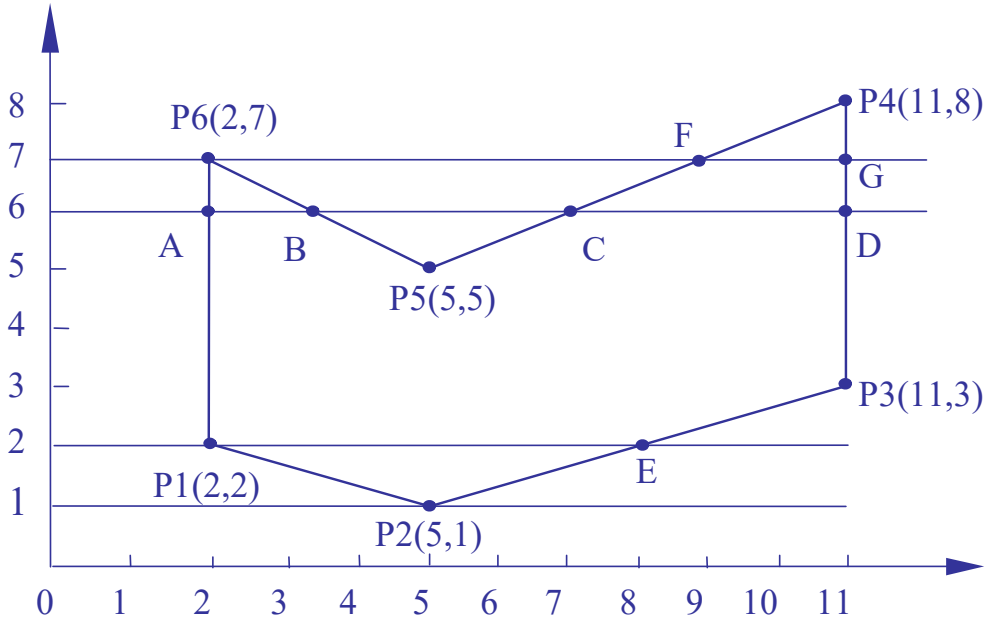
$$\sum_{i=0}^{n}\theta_i=\begin{cases}0, & V\text{位于}P\text{之外}\\ \pm 2\pi, & V\text{位于}P\text{之内}\end{cases}$$





扫描线算法

- 基本思想：按扫描线顺序，计算扫描线与多边形的相交区间，再用要求的颜色显示这些区间的象素，完成填充工作
- 对于一条扫描线填充过程的步骤：
  - 求交
  - 排序
  - 配对
  - 填色



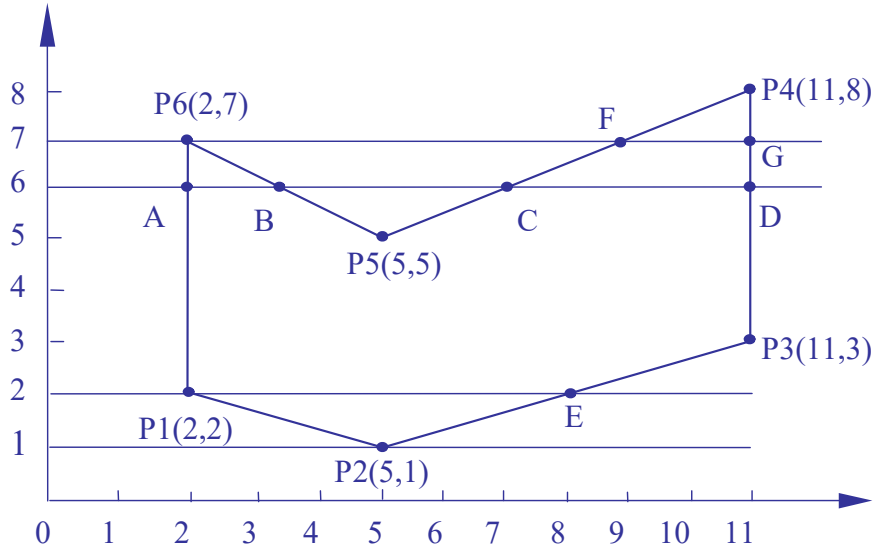
一个多边形与若干扫描线

- 扫描线与多边形的顶点或边界相交时，必须正确进行交点的取舍。

- 检查共享该顶点的两条边的另外两个端点的y值。按这两个y值中大于交点y值的个数来决定交点的个数。

- 问题：求交、排序导致算法效率低

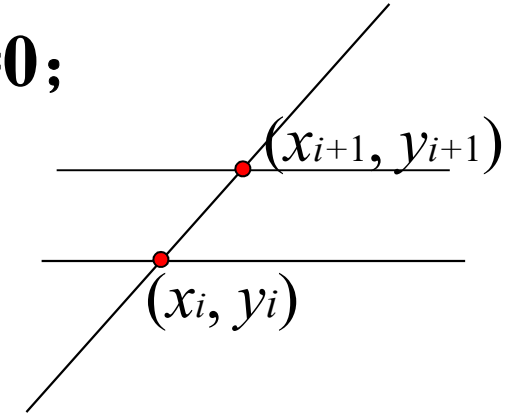
- 数据结构
  - 活性边表(AET)：把与当前扫描线相交的边称为活性边，并把它们按与扫描线交点的*x*坐标递增的顺序存放在一个链表中；
  - 边的连贯性
  - 扫描线的连贯性
  - 链表节点中存边的哪些信息？

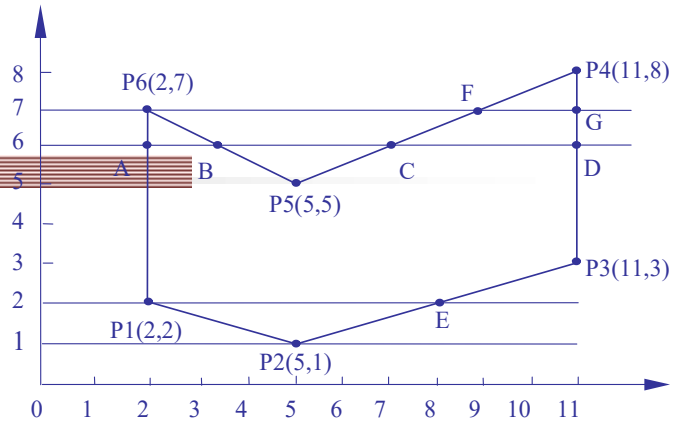


- 假定当前扫描线与多边形某一条边的交点的横坐标为x_i，则下一条扫描线与该边的交点不要重计算，只要加一个增量$\triangle x$。
- 设该边的直线方程为：$ax+by+c=0$；
- 若$y=y_i$，$x=x_i$；则当$y=y_{i+1}$时，

$$x_{i+1}=\frac{1}{a}(-b\cdot y_{i+1}-c_i)=x_i-\frac{b}{a};$$

其中 $\Delta x=-b/a$ 为常数



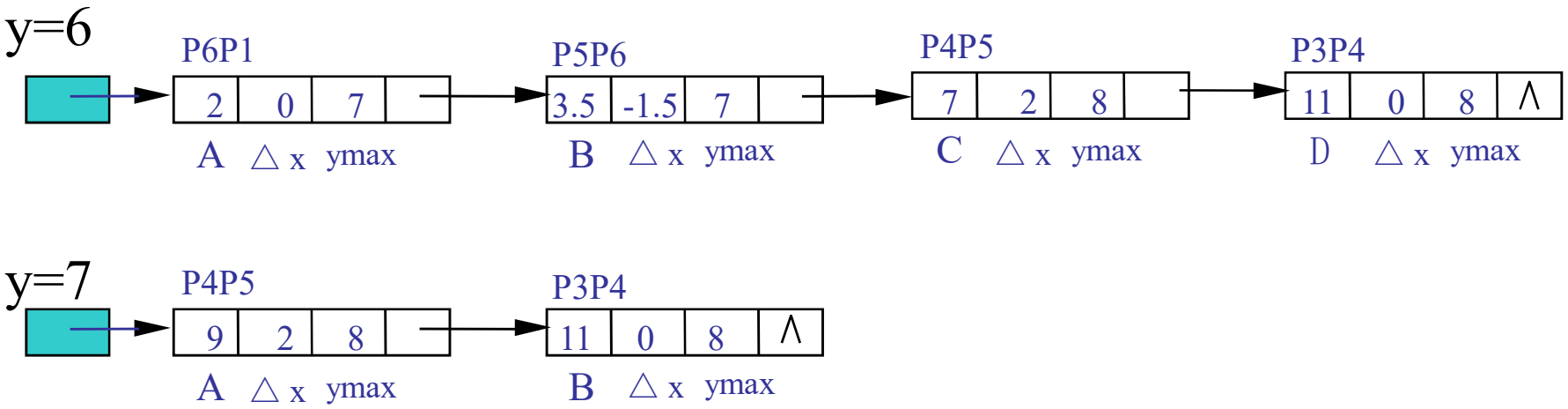


- 链表节点内容

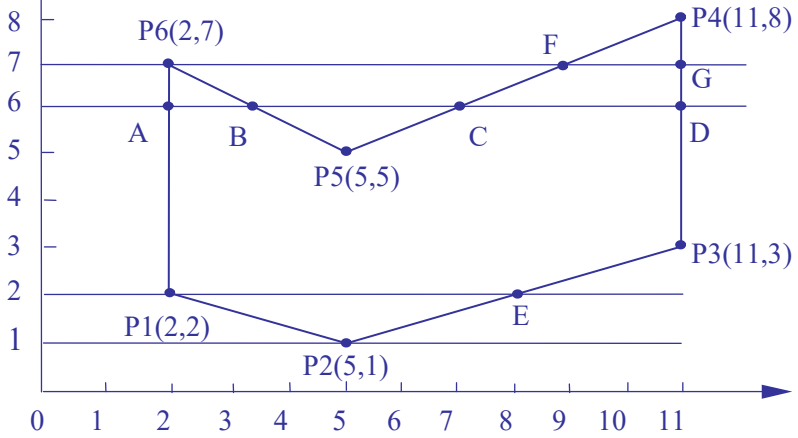
  x：当前扫描线与边的交点的***x***坐标

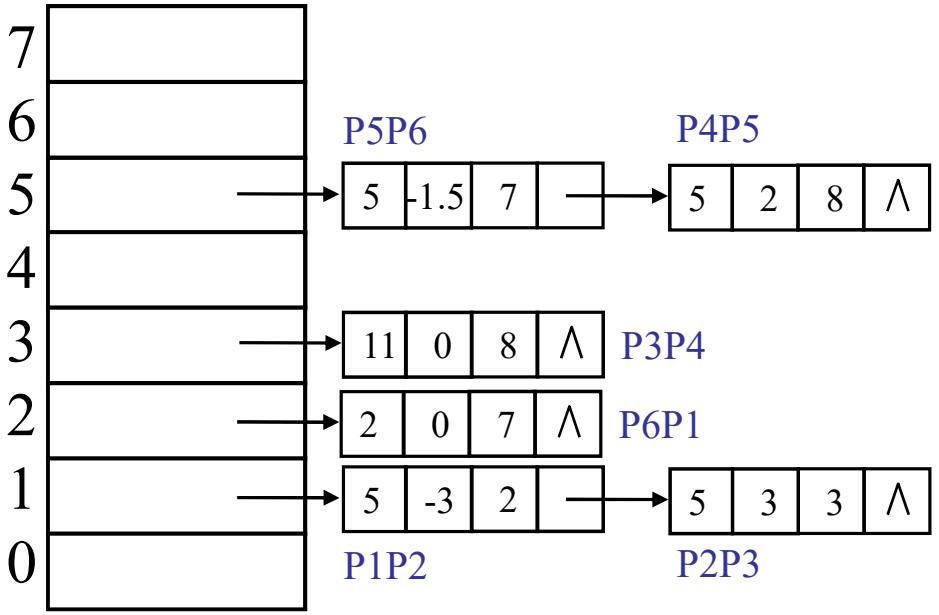
  △***x***：从当前扫描线到下一条扫描线间***x***的增量

  y_{max}：该边所交的最高扫描线号y_{max}



- 新边表(NET)：存放在该扫描线第一次出现的边。若某边的较低端点为y_{min}，则该边就放在扫描线y_{min}的新边表中





- 算法过程

void polyfill (polygon, color)

int color; 多边形 **polygon;**

{ for (各条扫描线**i)**

{ 初始化新边表头指针**NET [i];**

把y_{min} **= i** 的边放进边表**NET [i]; }**

y = 最低扫描线号**;**

初始化活性边表**AET**为空**;**

while(扫描线未处理完)

{

把新边表**NET[y]** 中的边结点用插入排序法插入**AET**表，使之按**x**坐标递增顺序排列；

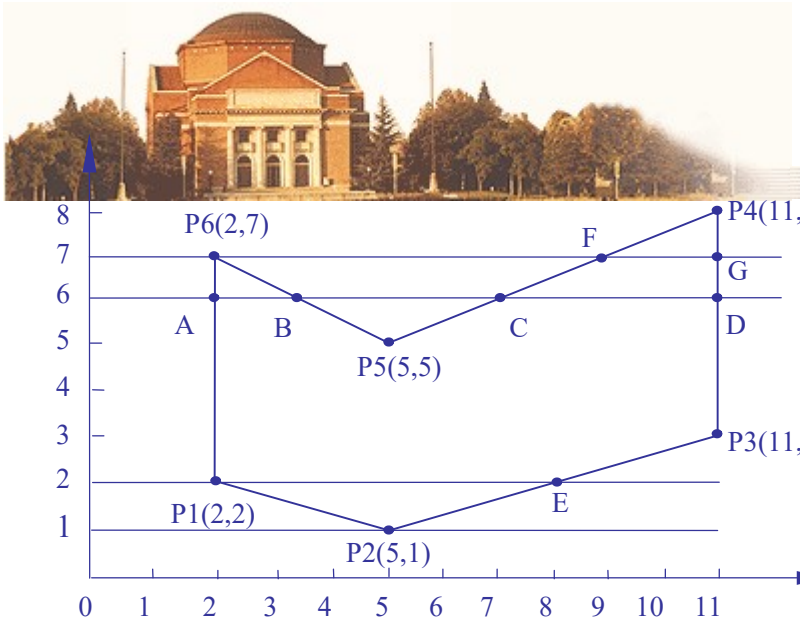
遍历**AET**表，把所有配对交点区间上的象素**(x, y)**，用**DrawPixel (x, y, color)** 改写象素颜色值；

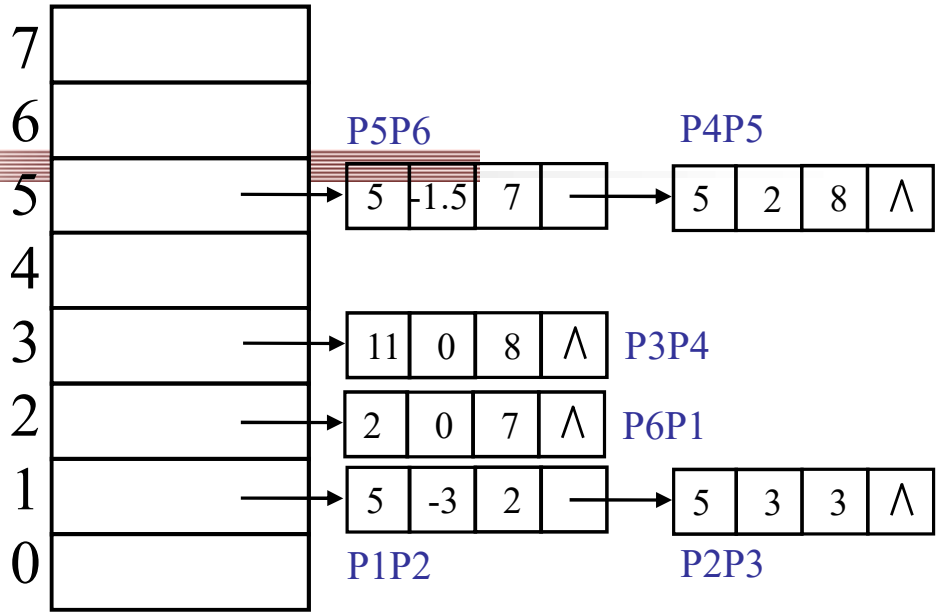
y=y+1;

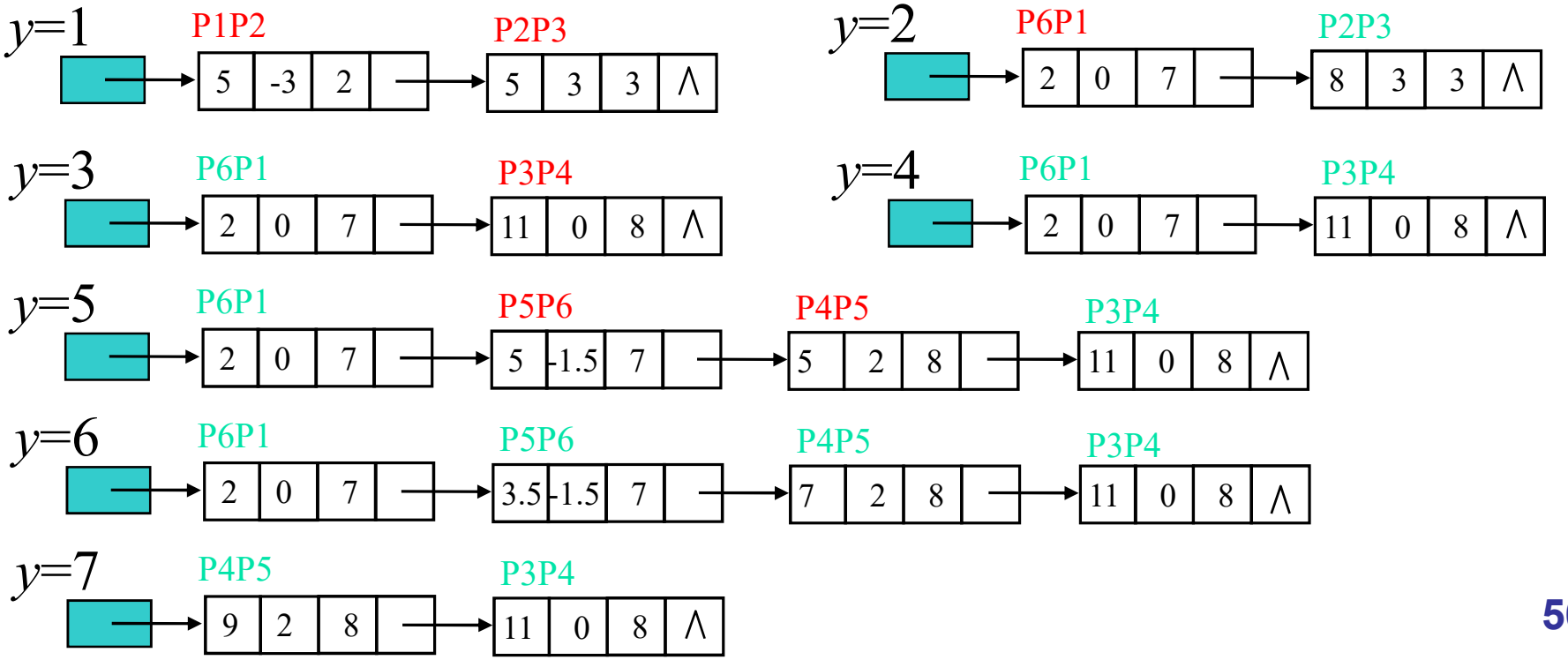
遍历**AET**表，把$y_{max} = y$ 的结点从**AET**表中删除，并把$y_{max} > y$ 结点的**x**值递增Δx；

}

}


P6(2,7)
P4(11,8)
F
G
A
B
C
D
P5(5,5)
P3(11,3)
P1(2,2)
E
P2(5,1)


7
6
5
4
3
2
1
0
P5P6
P4P5
5 -1.5 7
5 2 8 ∧
11 0 8 ∧
P3P4
2 0 7 ∧
P6P1
5 -3 2
5 3 3 ∧
P1P2
P2P3


y=1
P1P2
P2P3
5 -3 2
5 3 3 ∧
y=2
P6P1
P2P3
2 0 7
8 3 3 ∧
y=3
P6P1
P3P4
2 0 7
11 0 8 ∧
y=4
P6P1
P3P4
2 0 7
11 0 8 ∧
y=5
P6P1
P5P6
P4P5
P3P4
2 0 7
5 -1.5 7
5 2 8
11 0 8 ∧
y=6
P6P1
P5P6
P4P5
P3P4
2 0 7
3.5 -1.5 7
7 2 8
11 0 8 ∧
y=7
P4P5
P3P4
9 2 8
11 0 8 ∧

边缘填充法

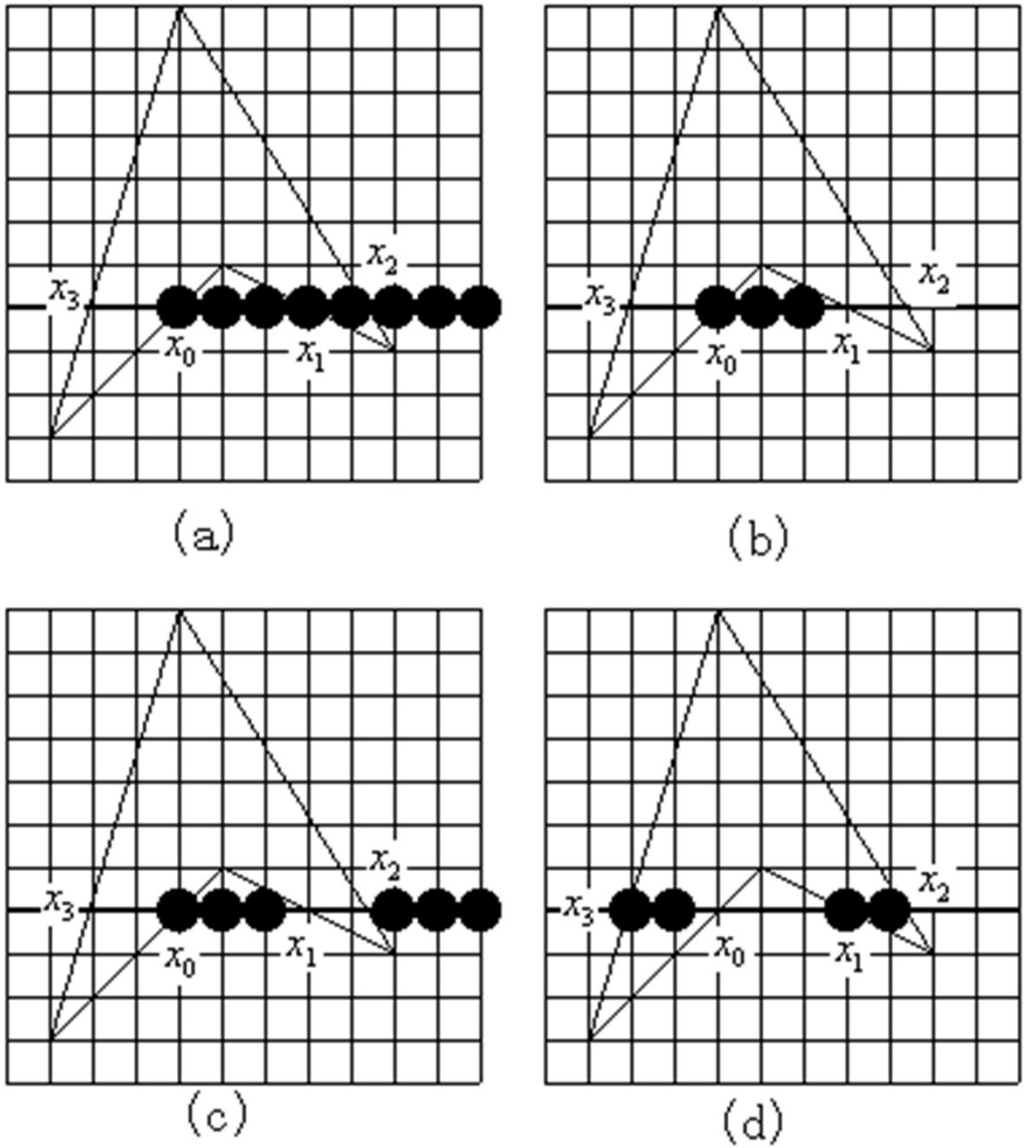
- 基本思想：对于每条扫描线和各多边形边的交点，将该扫描线上交点右方的所有象素颜色取补。对多边形的每条边作此处理，与边的顺序无关。
- 原理：以求补运算代替扫描线算法中的排序运算
- 求补**(A=0xFFFFFFFF)**

$$\overline{M} = A - M = M \ Xor \ A$$

- 算法1(以扫描线为中心的边缘填充)
  - 将当前扫描线上的所有象素着上背景色 $\overline{M}$
  - 求补：

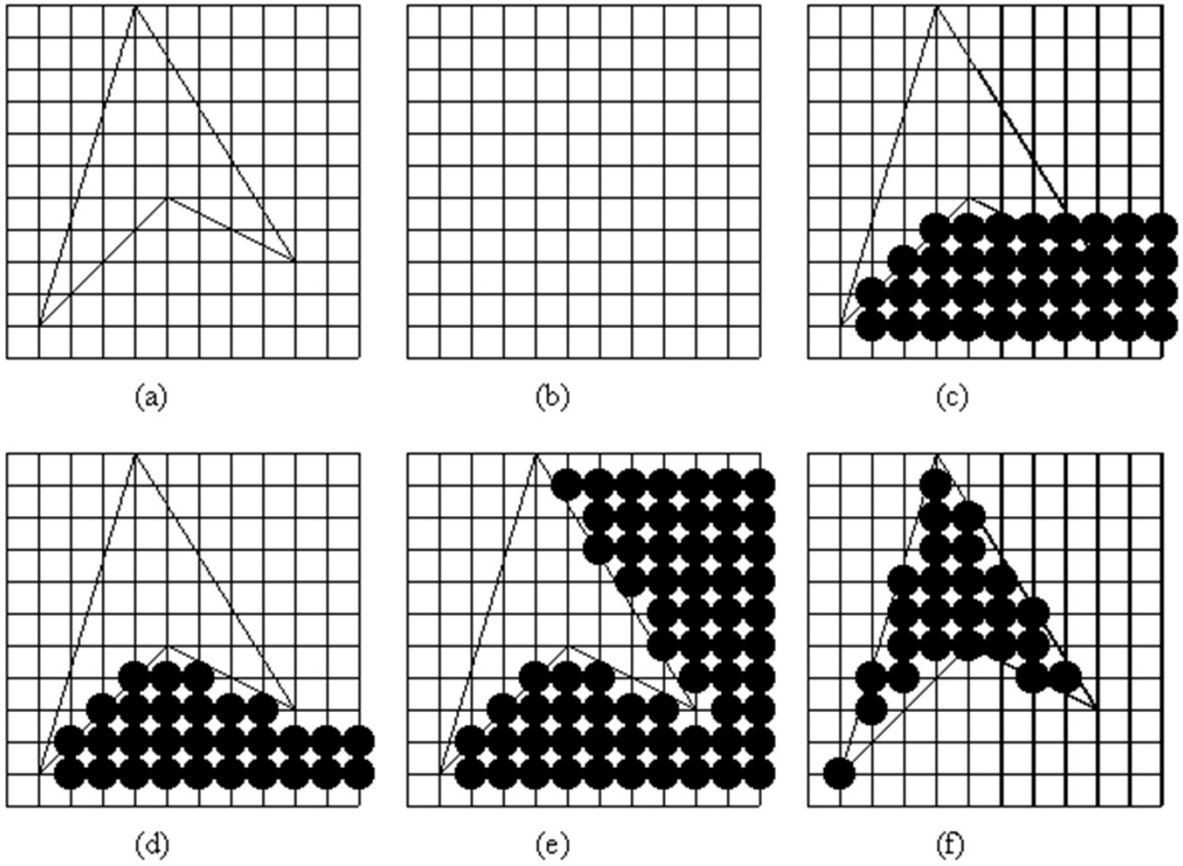
    for(i = 0; i <= m; i++)

    在当前扫描线上，从横坐标为x_i的交点向右的所有象素颜色求补


x_3
x_2
x_0
x_1
(a)
x_3
x_2
x_0
x_1
(b)
x_3
x_2
x_0
x_1
(c)
x_3
x_2
x_0
x_1
(d)

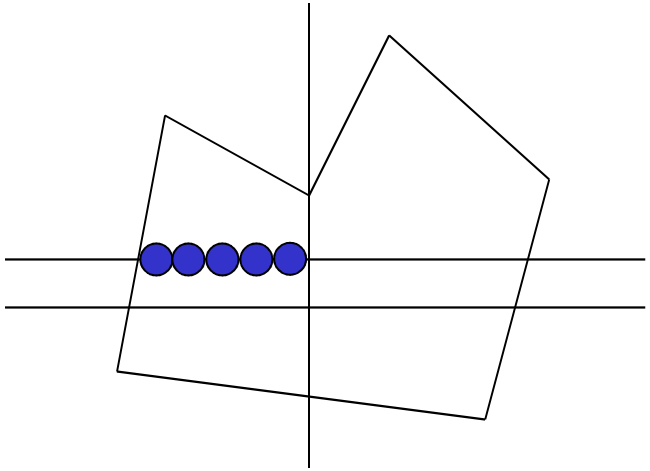
- 算法2(以边为中心的边缘填充)
  - 将绘图窗口所有象素置为背景色 $\overline{M}$
  - 对多边形的每一条非水平边

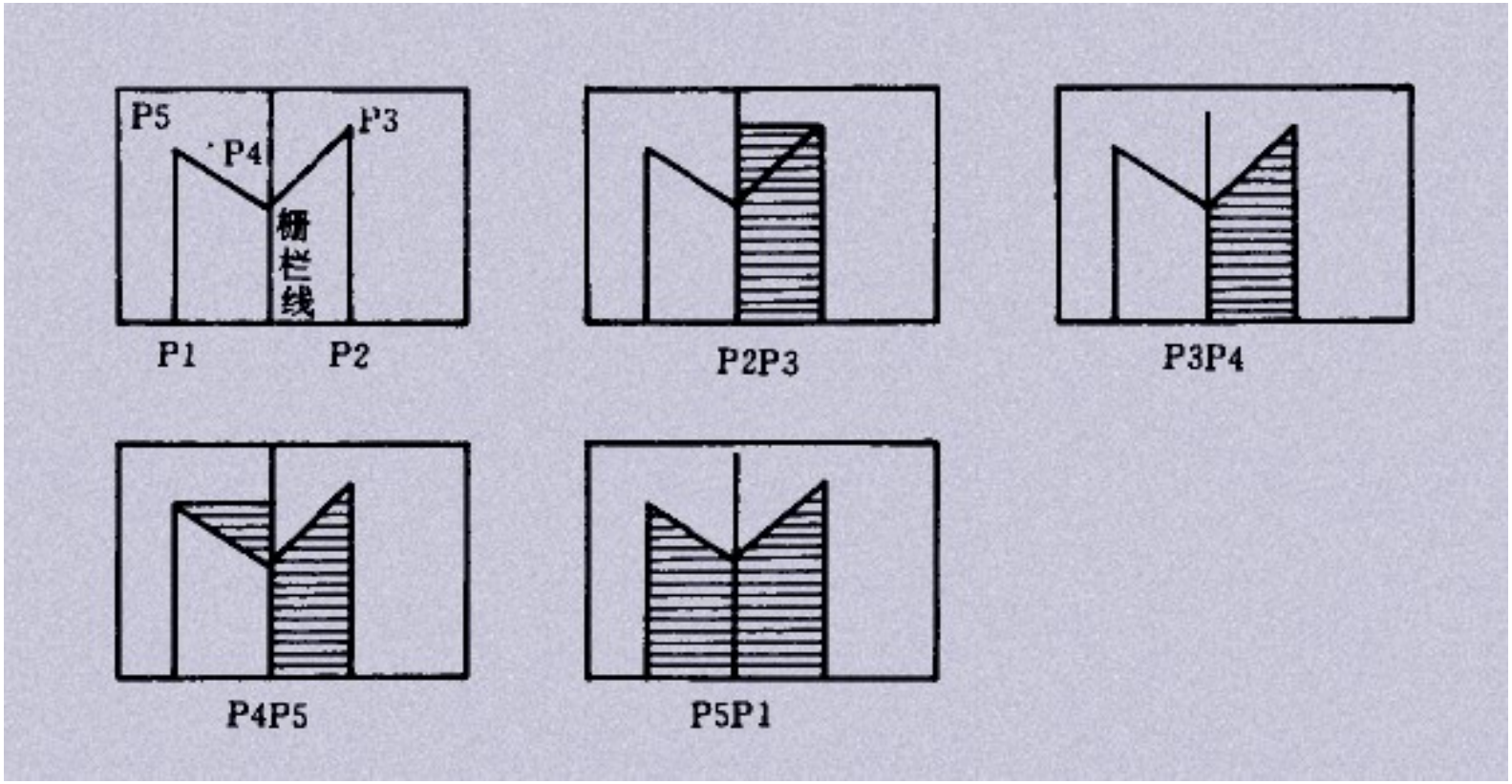
    从该边上的每个象素开始向右求补


(a)
(b)
(c)
(d)
(e)
(f)

- 条件？
- 优点：算法简单，不需排序
- 缺点：对于复杂图形，每一象素可能被访问多次，输入/输出的量比有序边表算法大得多。不适合图像填充。

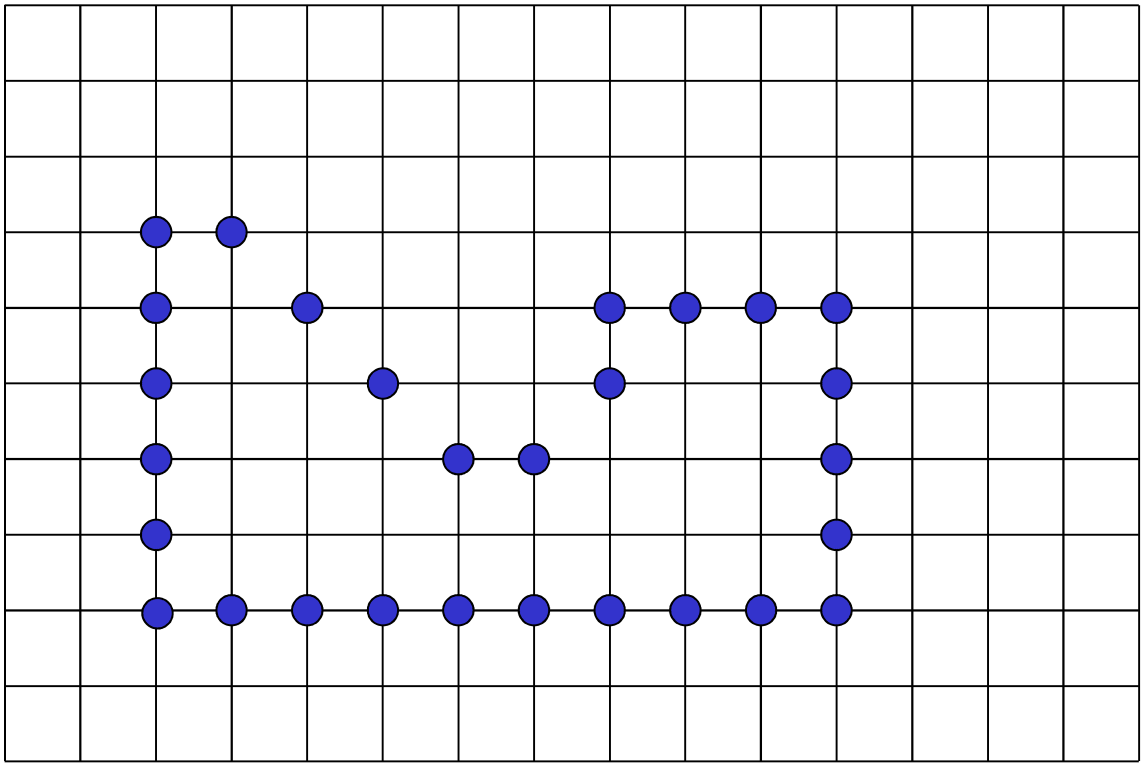
- 引入栅栏，以减少填充算法访问象素的次数。
  - 栅栏：与扫描线垂直的直线，通常过一顶点，且把多边形分为左右二半。
  - 基本思想：扫描线与多边形的边求交，将交点与栅栏之间的象素取补。
  - 减少了象素重复访问数目，但不彻底。




P5
P4
P3
栅栏线
P1
P2
P2P3
P3P4
P4P5
P5P1

边界标志法

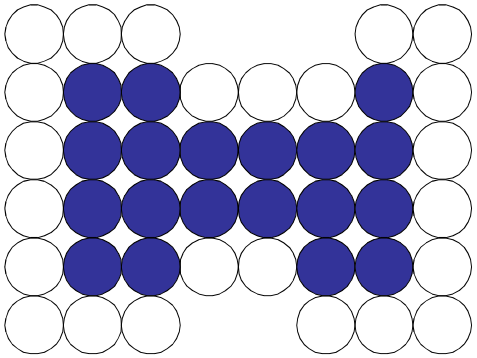
- 基本思想：
  - 对多边形的每条边进行直线扫描转换，亦即对多边形边界所经过的象素打上标志。
  - 然后再采用和扫描线算法类似的方法将位于多边形内的各个区段着上所需颜色。
  - 使用一个布尔量**inside**来指示当前点是否在多边形内的状态。



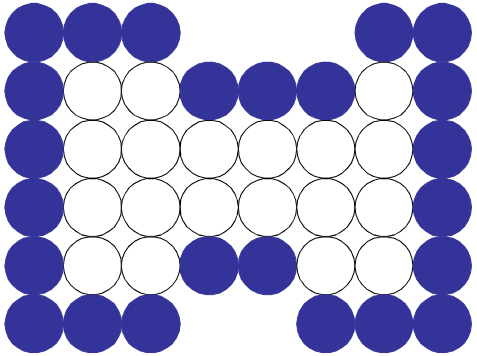
```
void edgemark_fill(polydef, color)
多边形定义  polydef;     int color;
{   对多边形polydef 每条边进行直线扫描转换，并打上边标记；
    for (每条与多边形polydef相交的扫描线y )
    {   inside = FALSE;
        for (扫描线上每个象素x )
        {
            if(象素 x 被打上边标志)
                inside = ! (inside);
            if(inside！ = FALSE)
                DrawPixel (x, y, color);
            else DrawPixel (x, y, background);
        }
    }
}
```

区域填充算法

- 区域指已经表示成点阵形式的填充图形，它是象素的集合。
- 区域可采用内点表示和边界表示两种表示形式。
- 区域填充指先将区域的一点赋予指定的颜色，然后将该颜色扩展到整个区域的过程。区域填充算法要求区域是连通的。

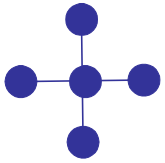


表示内点

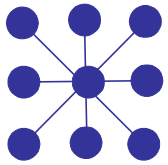


表示边界点

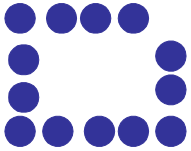
- **4向连通区域和8向连通区域**



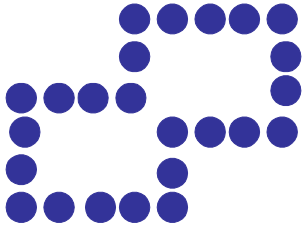
四个方向运动



八个方向运动



四连通区域



八连通区域

区域填充的递归算法

- 内点表示的**4**连通区域的递归填充算法

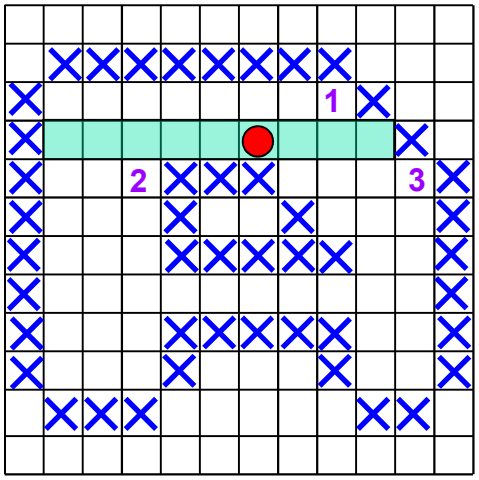
```
void FloodFill4(int x,int y,int oldcolor,int newcolor)
{   if(GetPixel(x,y)==oldcolor)   //属于区域内点
    {     DrawPixel(x,y,newcolor);
          FloodFill4(x,y+1,oldcolor,newcolor);
          FloodFill4(x,y-1,oldcolor,newcolor);
          FloodFill4(x-1,y,oldcolor,newcolor);
          FloodFill4(x+1,y,oldcolor,newcolor);
      }
}
```

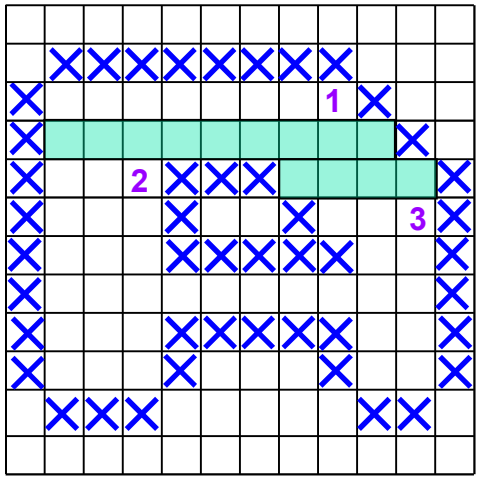
- 边界表示的**4**连通区域的递归填充算法

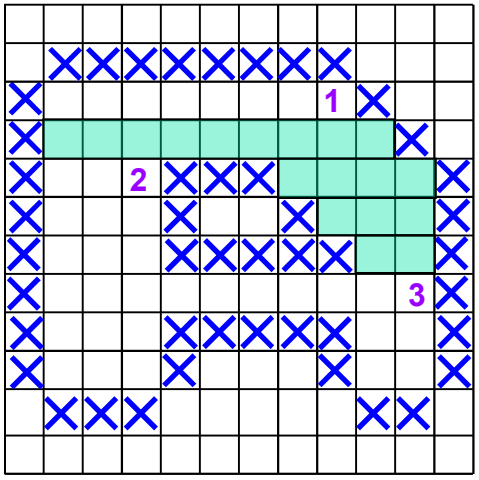
```
void BoundaryFill4(int x,int y,int boundarycolor,int newcolor)
{   int color=GetPixel(x,y);
    if(color!=newcolor && color!=boundarycolor)
    {     DrawPixel(x,y,newcolor);
          BoundaryFill4 (x,y+1, boundarycolor,newcolor);
          BoundaryFill4 (x,y-1, boundarycolor,newcolor);
          BoundaryFill4 (x-1,y, boundarycolor,newcolor);
          BoundaryFill4 (x+1,y, boundarycolor,newcolor);
    }
}
```

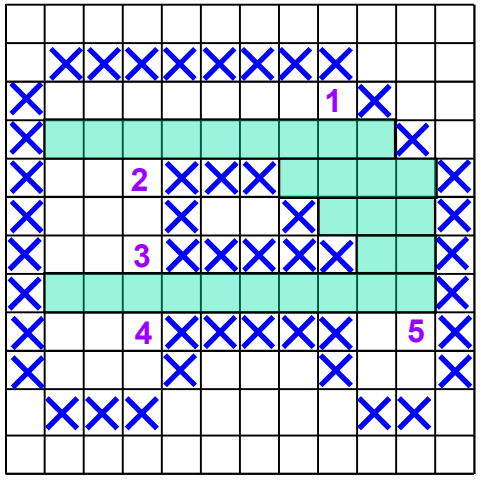
区域填充的扫描线算法

- 算法步骤：
  - 填充种子点所在的扫描线位于给定区域的一个区段
  - 然后确定与这一区段相连通的上、下两条扫描线上位于给定区域内的区段，并依次保存下来。
  - 反复这个过程，直到填充结束。


1
2
3


1
2
3


1
2
3


1
2
3
4
5

- 算法
  - 1初始化：堆栈置空。将种子点(x, y)入栈。
  - 2出栈：若栈空则结束。否则取栈顶元素(x, y)，以y作为当前扫描线。
  - 3填充并确定种子点所在区段：从种子点(x, y)出发，沿当前扫描线向左、右两个方向填充，直到边界。分别标记区段的左、右端点坐标为xl和xr。
  - 4并确定新的种子点：在区间[xl, xr]中检查与当前扫描线y上、下相邻的两条扫描线上的象素。若存在非边界、未填充的象素，则把每一区间的最右象素作为种子点压入堆栈，返回第2步。

多边形扫描转换与区域填充方法比较

- 都是光栅图形面着色，用于真实感图形显示。

- 不同点：
  - 基本思想不同；
  - 对边界的要求不同；
  - 基本的条件不同；