



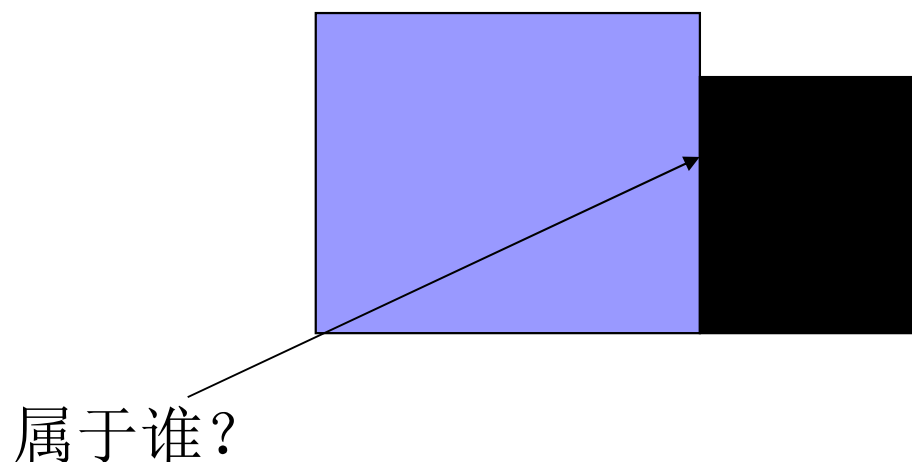
光栅图形的扫描转换与区域填充

- 扫描转换矩形
- 扫描转换多边形
- 区域填充

扫描转换矩形

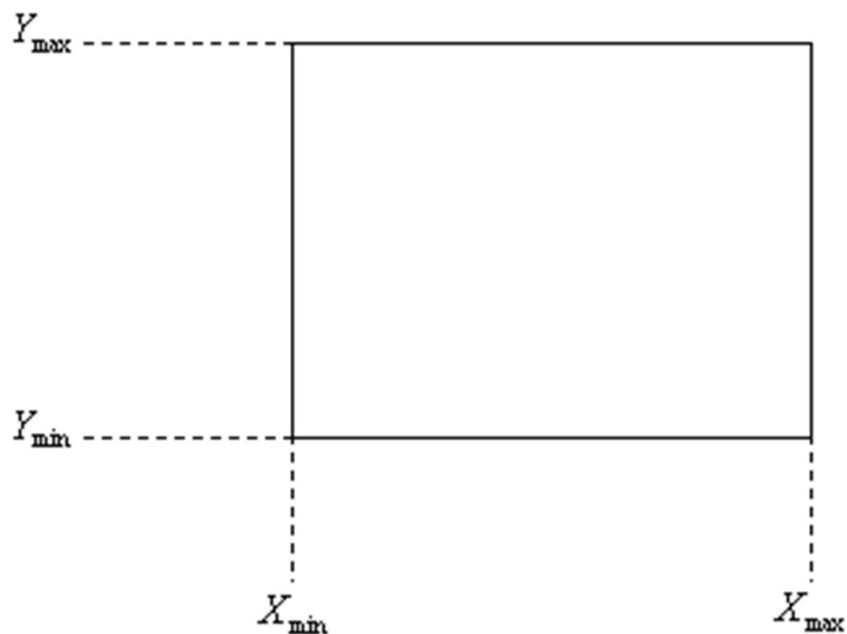
➤ 问题:

- 矩形是简单的多边形，那么为什么要单独处理矩形？
 - 比一般多边形可简化计算。
 - 应用非常多(窗口系统)。
- 共享边界如何处理？
 - **原则**：左闭右开，下闭上开



扫描转换矩形

方法:

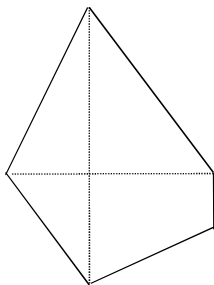


```
void FillRectangle(Rectangle *rect,int color)  
    { int x,y;  
      for(y = rect->ymin;y <= rect->ymax;y++)  
        for(x = rect->xmin;x <= rect->xmax;x++)  
          PutPixel(x,y,color);  
    }/*end of FillRectangle() */
```

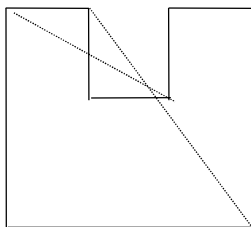


扫描转换多边形

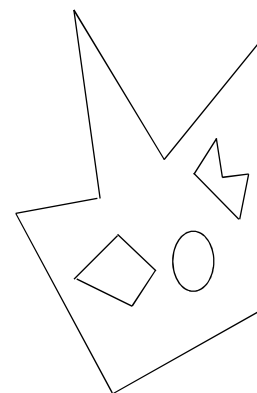
凸多边形



凹多边形



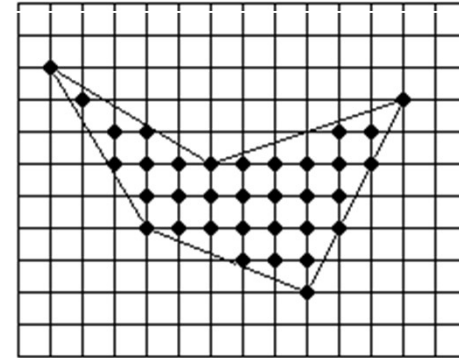
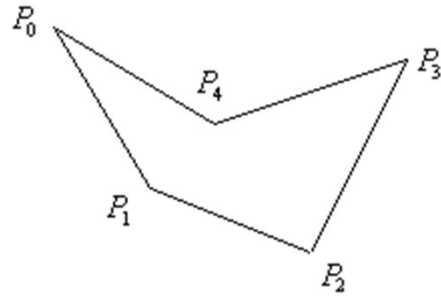
含内环的多边形



扫描转换多边形

➤ 多边形的表示

➤ 顶点表示



➤ 点阵表示

➤ **顶点表示**：用多边形顶点的序列来刻画多边形。直观、几何意义强、占内存少；不能直接用于面着色。

➤ **点阵表示**：用位于多边形内的象素的集合来刻画多边形。失去了许多重要的几何信息；便于运用帧缓冲存储器表示图形，易于面着色。



多边形的扫描转换

- **多边形的扫描转换**：把多边形的顶点表示转换为点阵表示，也就是从多边形的给定边界出发，求出位于其内部的各个象素，并给帧缓冲器内的各个对应元素设置相应的灰度和颜色，通常称这种转换为多边形的扫描转换。
- **几种方法**：逐点判断法；扫描线算法；边缘填充法；栅栏填充法；边界标志法。



逐点判断法

```
#define MAX 100
```

```
Typedef struct { int PolygonNum; // 多边形顶点个数  
                Point vertexces[MAX] //多边形顶点数组  
            } Polygon // 多边形结构
```

```
void FillPolygonPbyP(Polygon *P,int polygonColor)  
{ int x,y;
```

```
    for(y = ymin;y <= ymax;y++)
```

```
        for(x = xmin;x <= xmax;x++)
```

```
            if(IsInside(P,x,y))
```

```
                PutPixel(x,y,polygonColor);
```

```
            else
```

```
                PutPixel(x,y,backgroundColor);
```

```
    }/*end of FillPolygonPbyP() */
```



逐点判断法

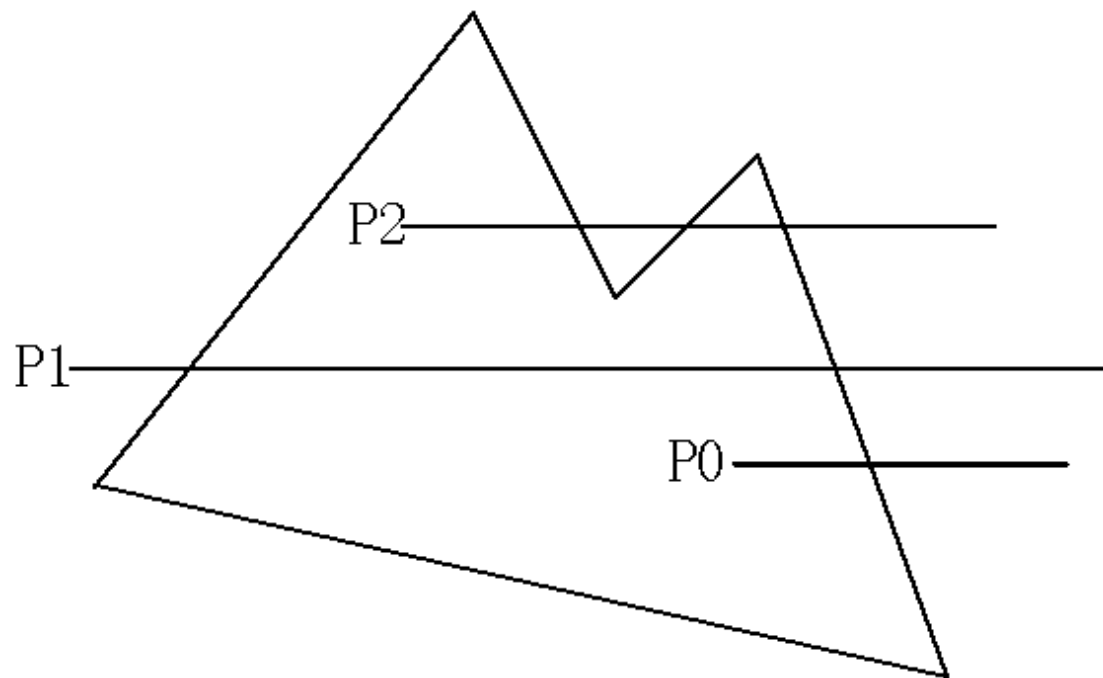
- 逐个判断绘图窗口内的像素
- 如何判断点在多边形的内外关系？
 - 1) 射线法；
 - 2) 累计角度法；
 - 3) 编码法；

逐点判断法

1) 射线法

➤ 步骤:

1. 从待判别点 v 发出射线
2. 求交点个数 k
3. k 的奇偶性决定了点与多边形的内外关系



逐点判断法

2) 累计角度法

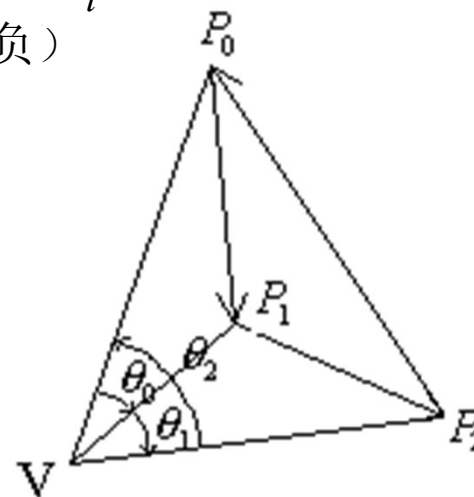
➤ 步骤

1. 从 v 点向多边形 P 顶点发出射线，形成有向角 θ_i
2. 计算有向角的和（逆时针为正，顺时针为负）

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

➤ 预处理

$$|\theta_i| \leq \pi$$



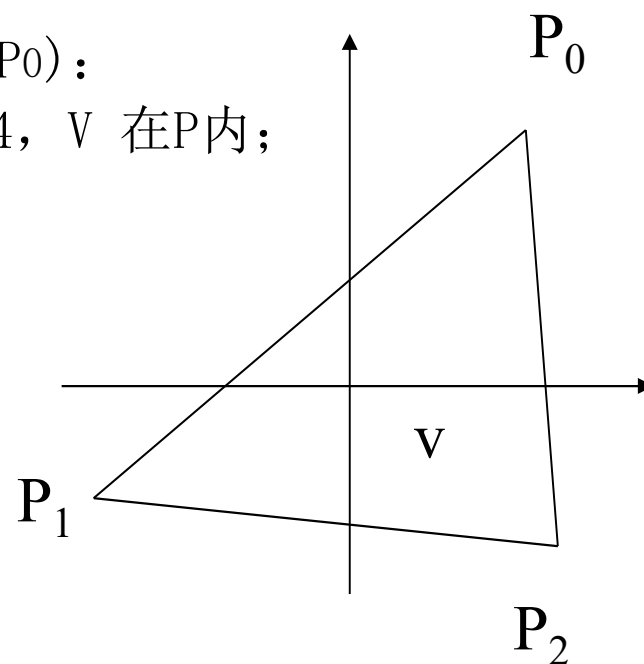
逐点判断法

3) 编码方法：累计角度方法的离散方法（略）

Step:

- 预处理，测试点在边上否？
- V为中点作局部坐标系, 对象限按逆时针（或顺时针）编码；
- 顶点编码 I_{pi} ,
- 边编码。 P_iP_{i+1} : $\Delta P_iP_{i+1} = I_{p_{i+1}} - I_{p_i}$
- 计算 $\Sigma \Delta P_iP_{i+1}$ （其中 $\Delta P_nP_{n+1} = \Delta P_nP_0$ ）:
若 Σ 为0, V在P外; 若 Σ 为 ± 4 , V 在P内;

逐点判断法程序简单，
速度太慢，效率低。





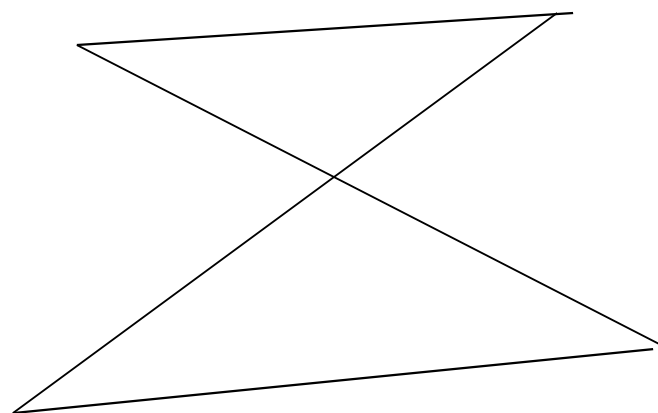
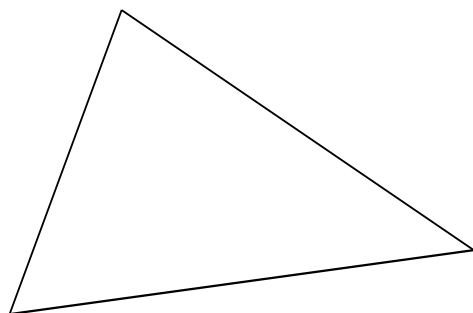
逐点判断法

- ▶ 逐点判断的算法虽然程序简单，但不可取。原因是速度太慢，主要是由于该算法割断了各像素之间的联系，孤立地考察各像素与多边形的内外关系，使得几十万甚至几百万个像素都要一一判别，每次判别又要多次求交点，需要做大量的乘除运算，花费很多时间。

扫描线算法

目标：利用相邻像素之间的连贯性，提高算法效率

➤ 处理对象：非自交多边形（边与边之间除了顶点外无其它交点）



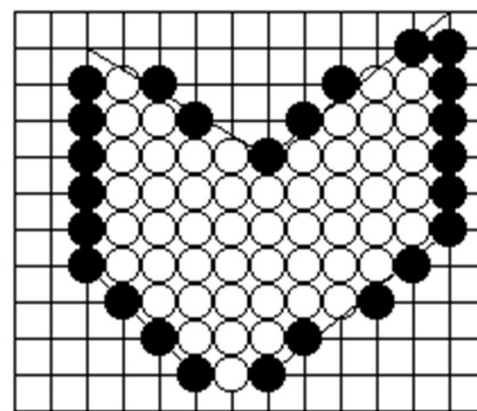
扫描线算法

➤ 交点的取整规则

➤ 要求：使生成的像素全部位于多边形之内

➤ 用于线画图元扫描转换的四舍五入原则导致部分像素位于多边形之外，从而不可用

➤ 假定非水平边与扫描线 $y=e$ 相交，交点的横坐标为 x ，规则如下



由中点算法求出的多边形边界上的像素

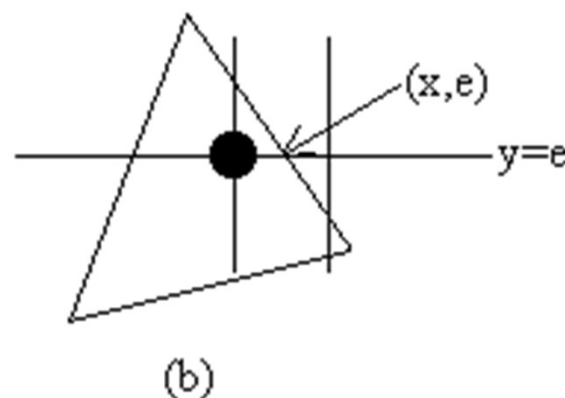
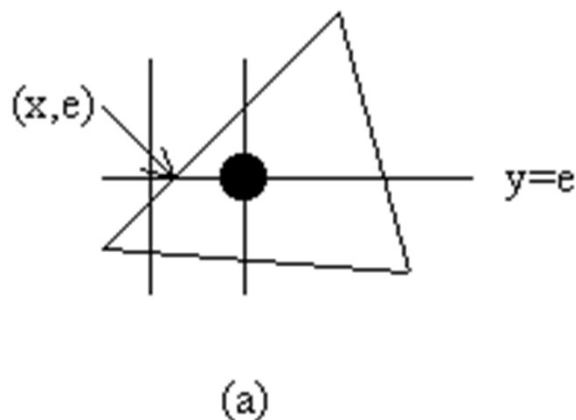
扫描线算法

规则1:

x 为小数，即交点落于扫描线上两个相邻像素之间

(a) 交点位于左边之上，向右取整

(b) 交点位于右边之上，向左取整



扫描线算法

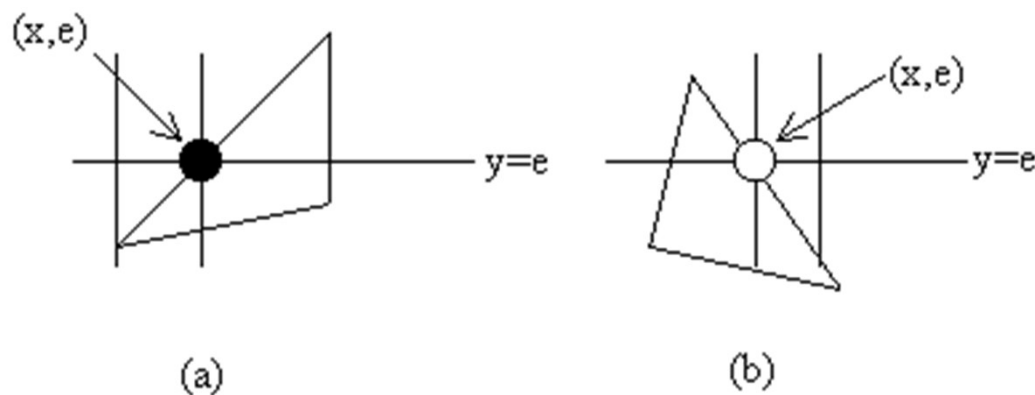
规则2:

边界上像素的取舍问题，避免填充扩大化。

解决方法:

边界像素：规定落在右上边界的像素不予填充。

具体实现时，只要对扫描线与多边形的相交区间左闭右开



扫描线算法

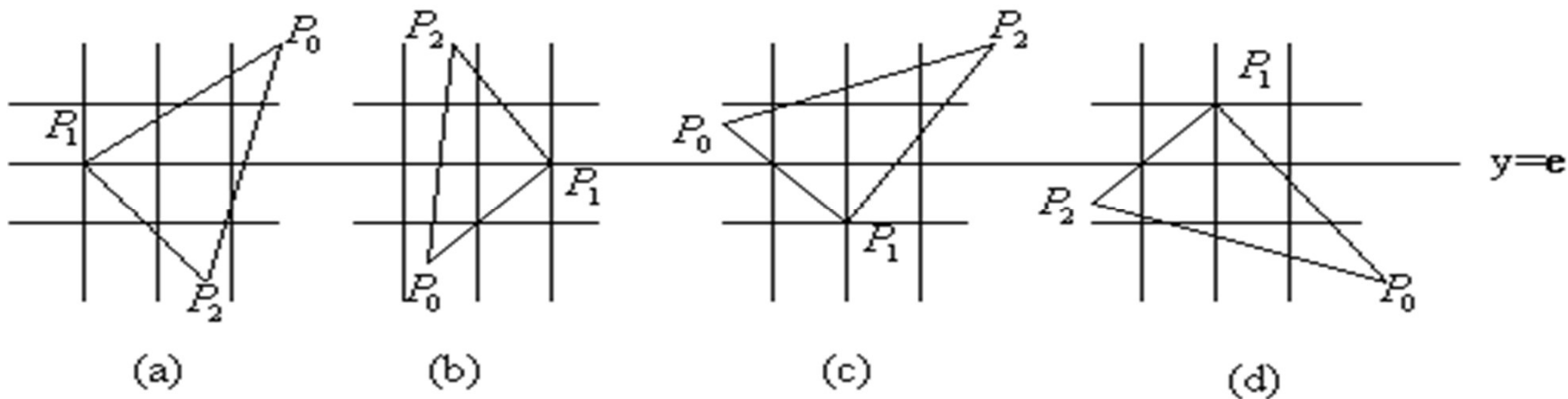
规则3:

扫描线与多边形的顶点相交时，交点的取舍，保证交点正确配对。

解决方法:

检查两相邻边在扫描线的哪一侧。

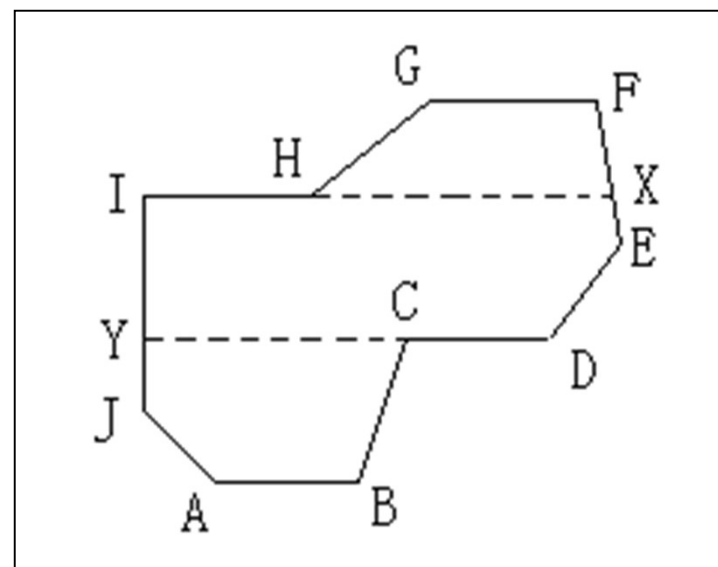
只要检查顶点的两条边的另外两个端点的Y值，两个Y值中大于交点Y值的个数是0, 1, 2，来决定取0, 1, 2个交点。



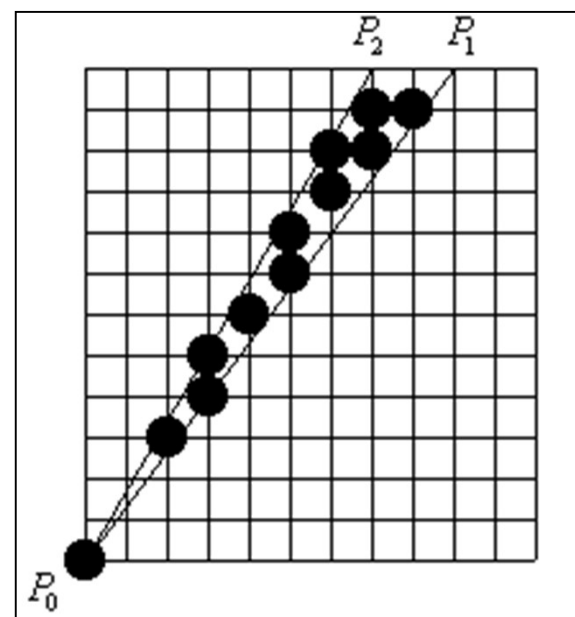
扫描线算法

□ 特殊情况处理

- 水平边: 扔掉!



- 尖角: 反混淆





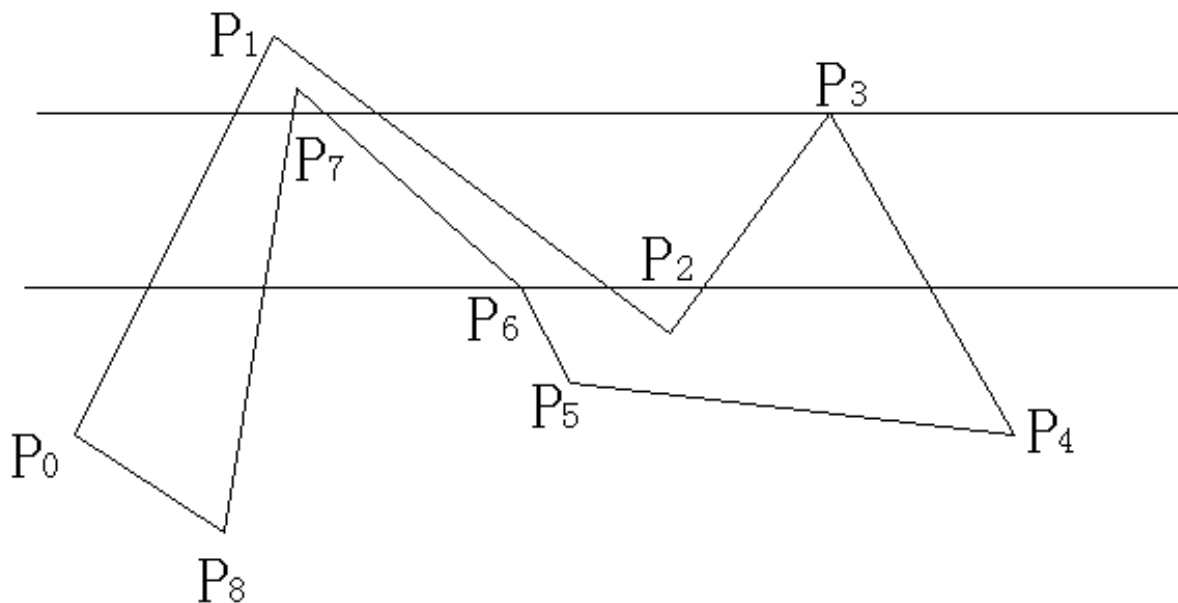
扫描线算法

扫描线算法是多边形扫描转换的常用算法。与逐点判断算法相比，扫描线算法充分利用了相邻像素之间的连贯性，避免了对像素的逐点判断和反复求交的运算，达到了减少了计算量和提高速度的目的。

开发和利用相邻像素之间的连贯性是光栅图形算法研究的重要内容。扫描转换算法综合利用了区域的连贯性、扫描线连贯性和边的连贯性等三种形式的连贯性。

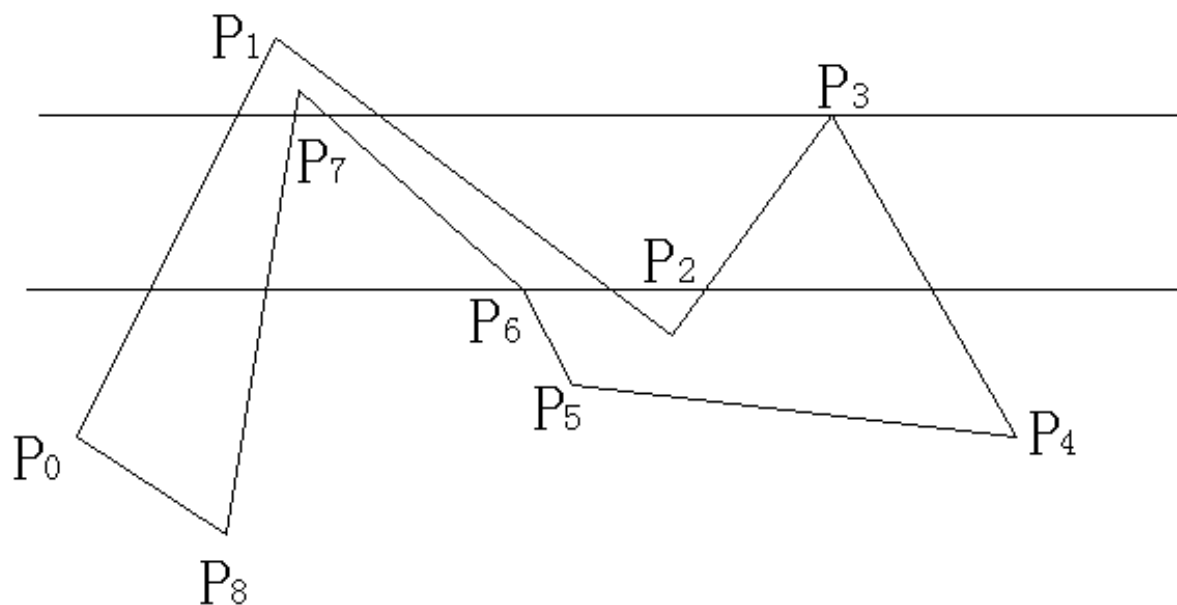
区域的连贯性

设多边形 P 的顶点 $P_i=(x_i, y_i)$, $i=0, 1, \dots, n$, 又设 $y_{i0}, y_{i1}, \dots, y_{in}$ 是各顶点 P_i 的坐标 y_i 的递减数列, 即 $y_{ik} \geq y_{ik+1}$, $0 \leq k \leq n-1$. 这样, 当 $y_{ik} \geq y_{ik+1}$, $0 \leq k \leq n-1$ 时, 屏幕上位于 $y=y_{ik}$ 和 $y=y_{ik+1}$ 两条扫描线之间的长方形区域被多边形 P 的边分割成若干梯形(三角形可看作其中一底边长为零的梯形), 它们具有下列性质:



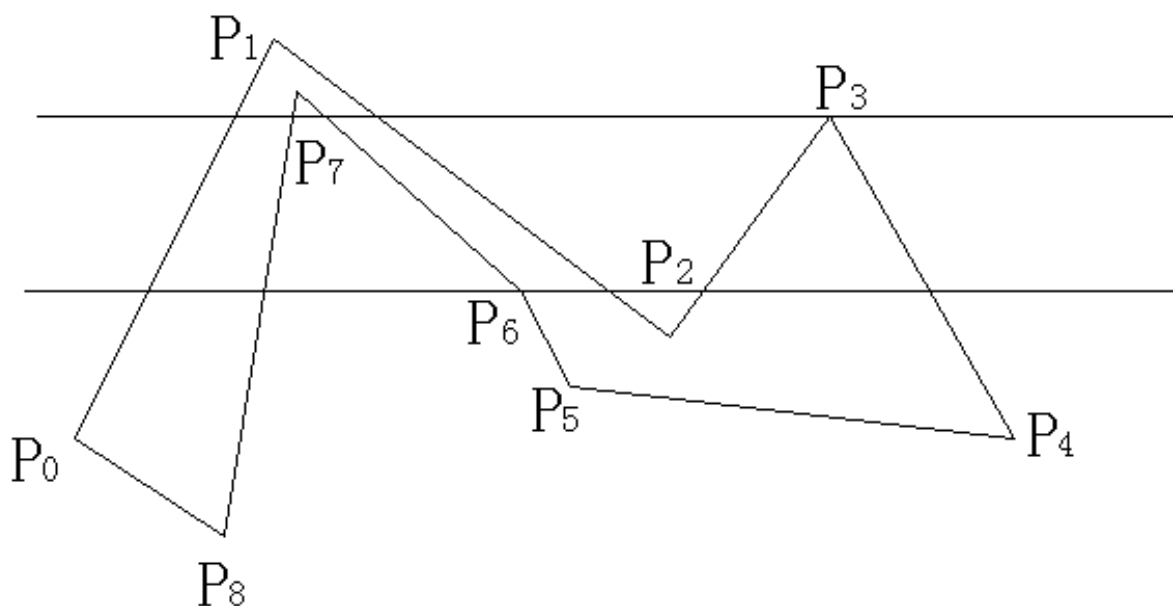
区域的连贯性

- 1) 梯形的两底边分别在 $y=y_{ik}$ 和 $y=y_{ik+1}$ 两条扫描线上，腰在多边形P的边上或在显示屏幕的边界上。
- 2) 这些梯形可分为两类：一类位于多边形P的内部；另一类在多边形P的外部。
- 3) 两类梯形在长方形区域 $\{y_{ik}, y_{ik+1}\}$ 内相间的排列，即相邻的两梯形必有一个在多边形P内，另一个在P外。



区域的连贯性

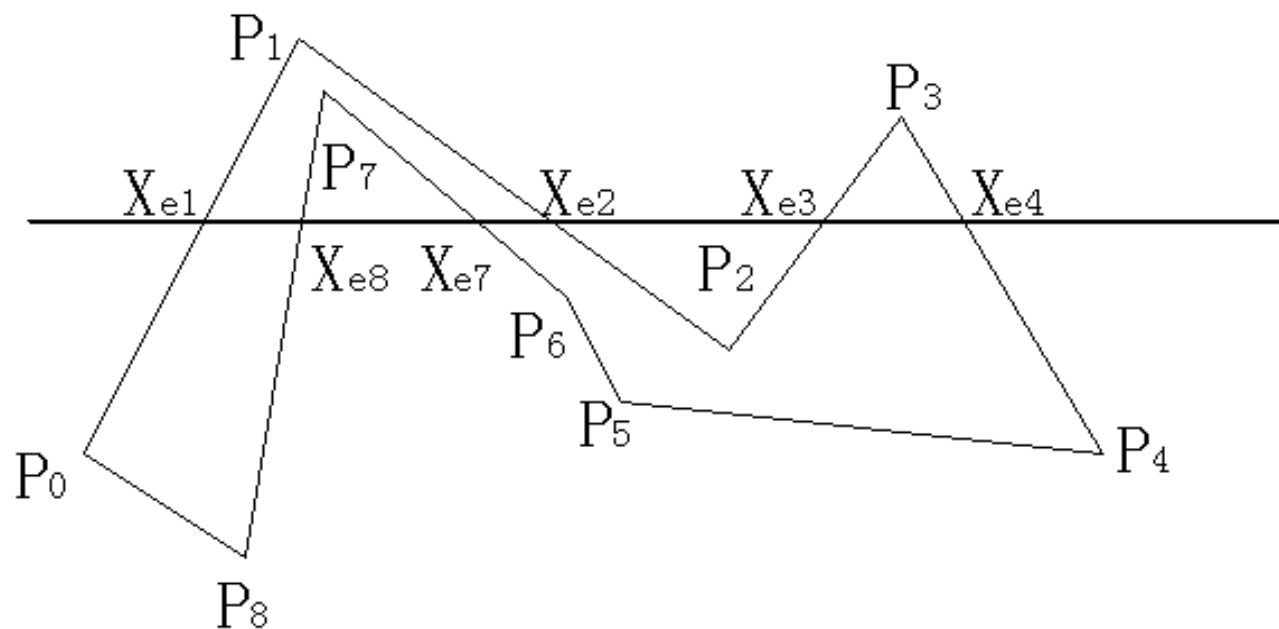
根据这些性质，实际上只需知道该长方形区域内任一梯形内一点关于多边形P的内外关系后，即可确定区域内所有梯形关于P的内外关系。



扫描线的连贯性

设 e 为一整数, $y_{i0} \geq e \geq y_{in}$ 。若扫描线 $y=e$ 与多边形 P 的 $P_{i-1}P_i$ 相交, 则记其交点的横坐标为 x_{ei} 。

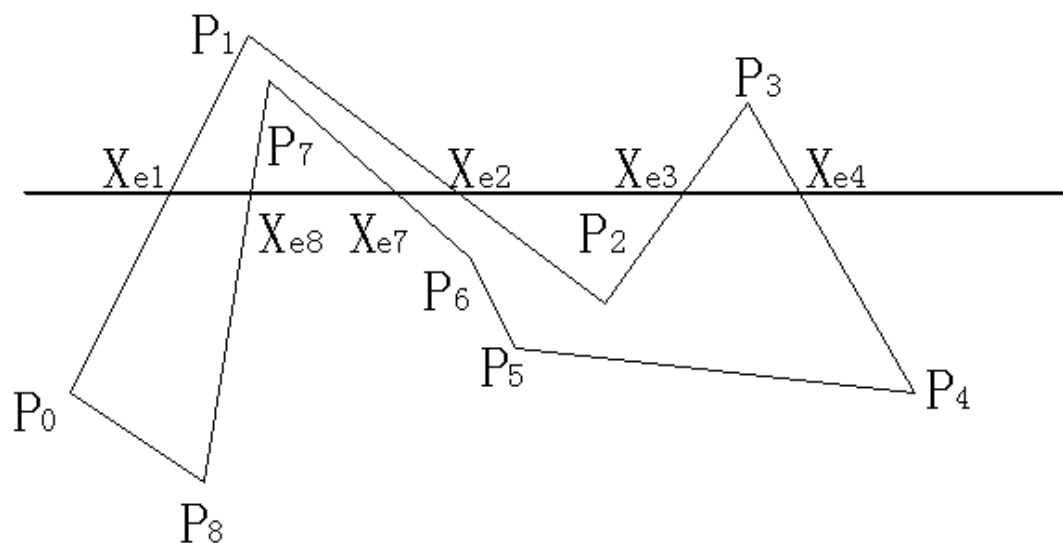
现设 $x_{ei1}, x_{ei2}, x_{ei3}, \dots, x_{eik}$ 是该扫描线与 P 的边界各交点横坐标的递增序列, 称此序列为**交点序列**。由区域的连贯性可知, 此交点序列具有以下性质:



扫描线的连贯性

- 1) 1是偶数。
- 2) 在该扫描线上，只有区段
 (x_{eik}, x_{eik+1}) , $k=1, 3, 5, \dots$, 1-1位于多边形P内,
其余区段都在P外。

以上性质称为扫描线的连贯性，它是多边形区域连贯性在一条扫描线上的反映。



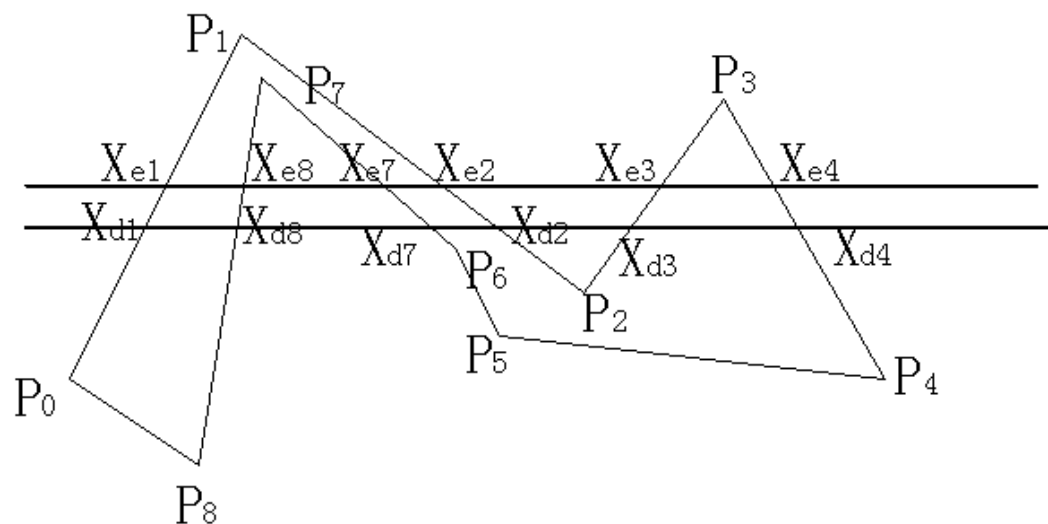
边的连贯性

设 d 为一整数, 并且 $d=e-1$, 并且 $y_{i0} \geq d \geq y_{in}$ 。设位于扫描线 $y=d$ 上的交点序列为 $x_{dj1}, x_{dj2}, x_{dj3}, \dots, x_{dj k}$

现在来讨论扫描线 d, e 交点序列之间的关系。若多边形 P 的边 $P_{r-1}P_r$ 与扫描线 $y=e, y=d$ 都相交, 则交点序列中对应元素 x_{er}, x_{dr} 满足下列关系:

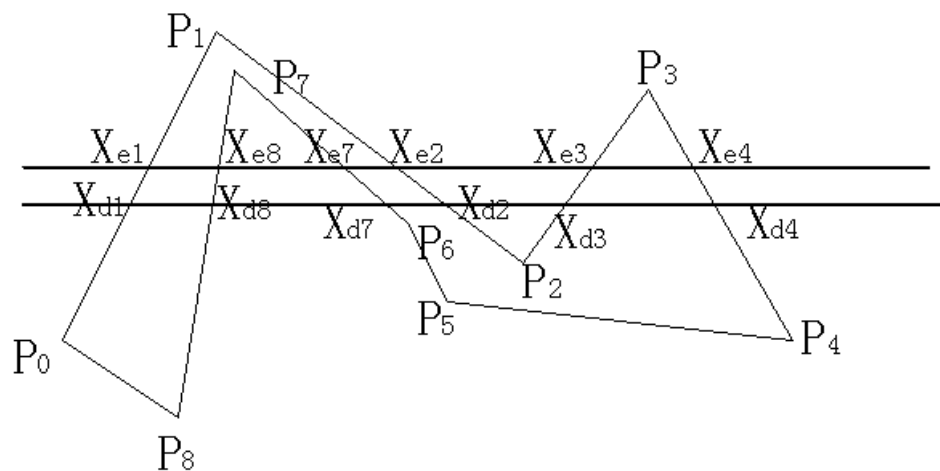
$$x_{er} = x_{dr} + 1/m_r \quad (1)$$

其中 m_r 为边 $P_{r-1}P_r$ 的斜率。



边的连贯性

于是，可利用 d 的交点序列计算 e 的交点序列，即先运用递推关系式(1)求得与扫描线 $y=e$ 和 $y=d$ 都相交的所有多边形上的交点 x_{er} ，再求得与扫描线 $y=d$ 不相交但与扫描线 $y=e$ 相交的所有边 P_qP_{q+1} 上的交点 x_{eq} 。如果 P 的顶点的坐标是整数，那么 $x_{eq}=x_q$ 或 $x_{eq}=x_{q+1}$ ，然后把这两部分按递增的顺序排列，即可得 e 的交点序列。



边的连贯性

特别是当存在某一个整数 k , $0 \leq k \leq n-1$, 使得

$$y_{ik} > e, \quad d > y_{ik+1}$$

成立时, 则由区域的连贯性可知 d 的交点序列和 e 的交点序列之间有以下关系:

1) 两序列元素的个数相等, 如上图所示。

2) 点 (x_{eir}, e) 与 (x_{djr}, d) 位于多边形 P 的同一边上, 于是 $x_{eir} = x_{djr} + 1/k_{jr}$ (2)

这样, 运用递推关系式(2)可直接由 d 的交点序列获得 e 的交点序列。

以上性质称为边的连贯性, 它是区域的连贯性在相邻两扫描线上的反映。

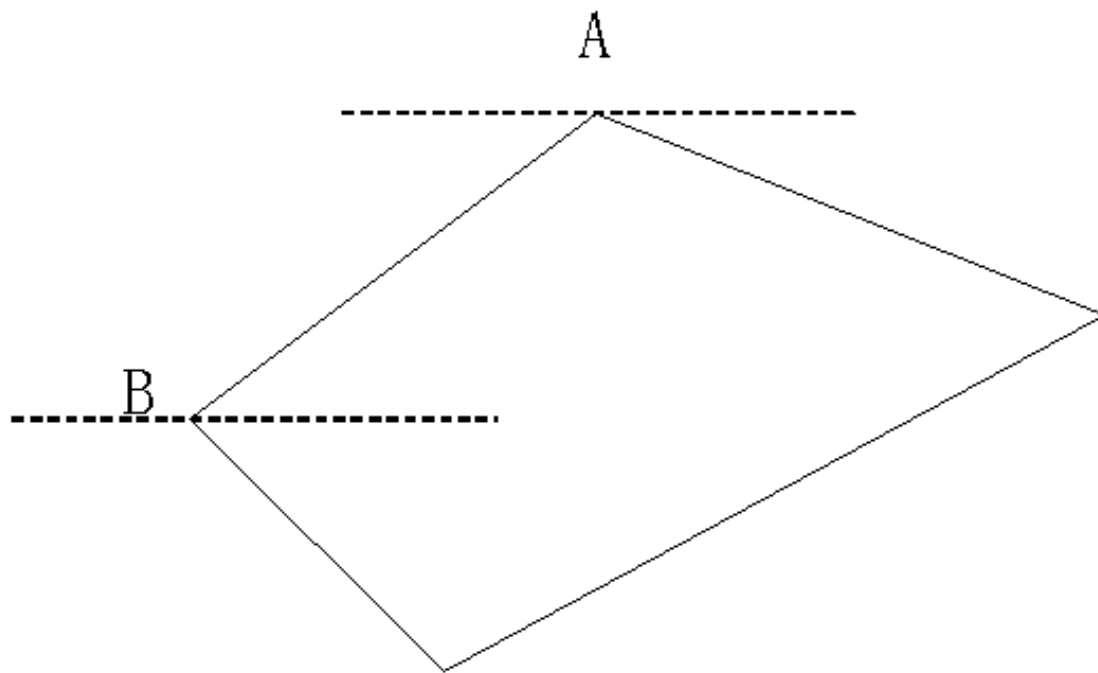


奇点的处理

- 当扫描线与多边形P的交点是P的顶点时，则称该交点为奇点。
- 以上所述多边形的三种形式的连贯性都基于这样的几何事实：每一条扫描线与多边形P的边界的交点个数都是偶数。但是如果把每一奇点简单地计为一个交点或者简单地计为两个交点，都可能出现奇数个交点。那么如何保证交点数为偶数呢？

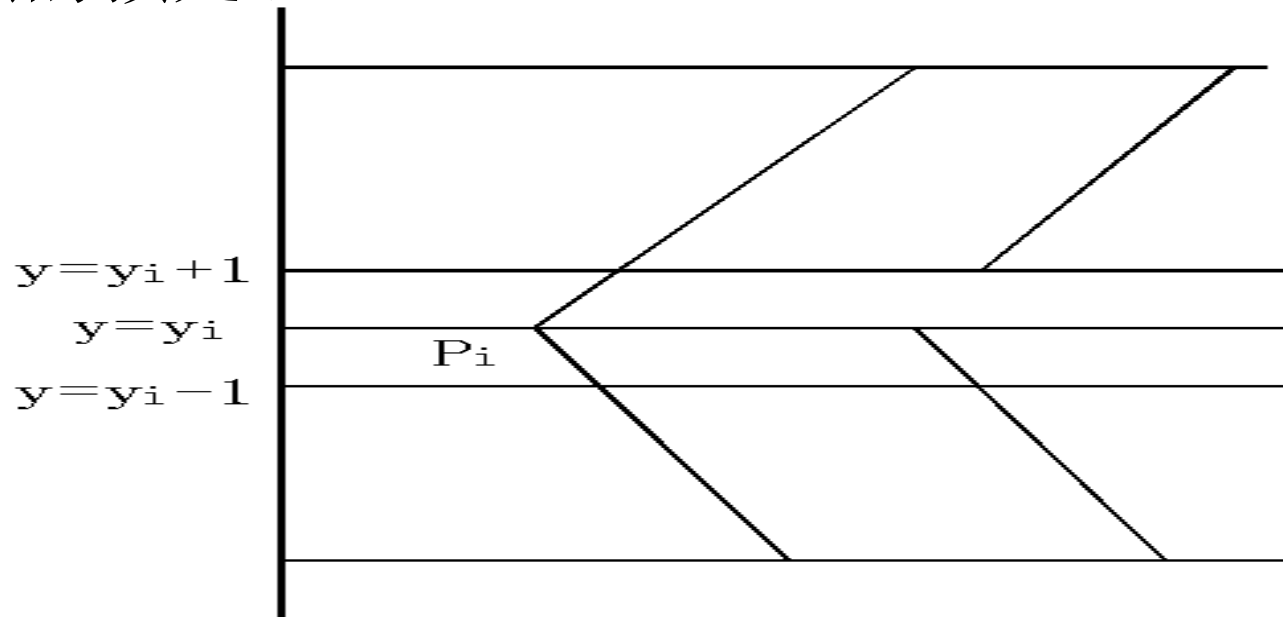
奇点的处理

- 若奇点做一个交点处理，则情况**A**，交点个数不是偶数。
- 若奇点做两个交点处理，则情况**B**，交点个数不是偶数。



奇点的处理

- 多边形P的顶点可分为两类：**极值奇点**和**非极值奇点**。如果 $(y_{i-1} - y_i)(y_{i+1} - y_i) \geq 0$ ，则称顶点 P_i 为极值点；否则称 P_i 为非极值点。
- **规定**：奇点是极值点时，该点按两个交点计算，否则按一个交点计算。
- 奇点的预处理：





数据结构与实现步骤

算法基本思想：首先取 $d=y_{in}$ 。容易求得扫描线 $y=d$ 上的交点序列为 $x_{dj1}, x_{dj2}, \dots, x_{djn}$ ，这一序列由位于扫描线 $y=d$ 上的多边形 P 的顶点组成。

由 y_{in} 的交点序列开始，根据多边形的边的连贯性，按从上到下的顺序求得各条扫描线的交点序列；根据扫描线的连贯性，可确定各条扫描线上位于多边形 P 内的区段，并表示成点阵形式。



数据结构与实现步骤

所有的边和扫描线求交，效率很低。因为一条扫描线往往只和少数几条边相交。

如何判断多边形的一条边与扫描线是否相交？

与当前扫描线相交的边称为**活性边**（active edge），把它们按与扫描线交点x坐标递增的顺序存入一个链表中，称为**活化边表**（AEL, Active edge table）。它记录了多边形边沿扫描线的交点序列。

只需对当前扫描线的活动边表作更新，即可得到下一条扫描线的活动边表。



数据结构与实现步骤

➤ 如何计算下一条扫描线与边的交点？

直线方程： $ax+by+c = 0$

当前交点坐标： (x_i, y_i)

下一交点坐标： (x_{i+1}, y_{i+1})

$$x_{i+1} = ((-by_{i+1}) - c) / a = ((-by_i + 1) - c) / a = x_i - b/a$$

活化边表中需要存放的信息：

x： 当前扫描线与边的交点

$dx = -b/a$ ： 从当前扫描线到下一条扫描线之间的x增量

ymax： 边所交的最高扫描线



数据结构与实现步骤

增加哪一条边呢？

为了方便边的活化链表的更新，建立另一个表-边表，存放在该扫描线第一次出现的边。

存放的信息：

x ：扫描线与该边的初始交点

dx ： x 的增量

y_{\max} ：该边的最大 y 值



数据结构与实现步骤

算法中采用较灵活的数据结构。它由边的分类表ET (Edge Table) 和活化边表AEL (Active Edge List) 两部分组成。

表结构ET和AEL中的基本元素为多边形的边。边的结构由以下四个域组成：

y_{\max}	边的上端点的y坐标；
x	在ET中表示边的下端点的x坐标，在AEL中则表示边与扫描线的交点的坐标；
Δx	边的斜率的倒数；
next	指向下一条边的指针。



数据结构与实现步骤

边的分类表ET是按边的下端点的y坐标对非水平边进行分类的指针数组。下端点的y坐标的值等于i的边归入第i类。有多少条扫描线，就设多少类。同一类中，各边按x值（x值相等时，按 Δx 的值）递增的顺序排列成行。

```
typedef struct {int ymax;  
                float x,deltax;  
                Edge *nextEdge;  
            }Edge;
```



算法实现步骤

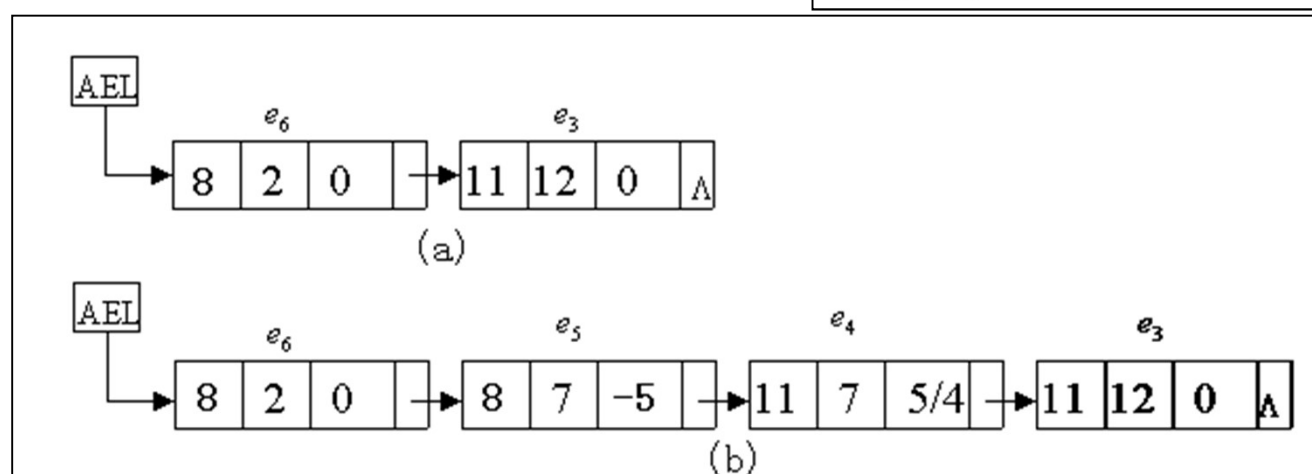
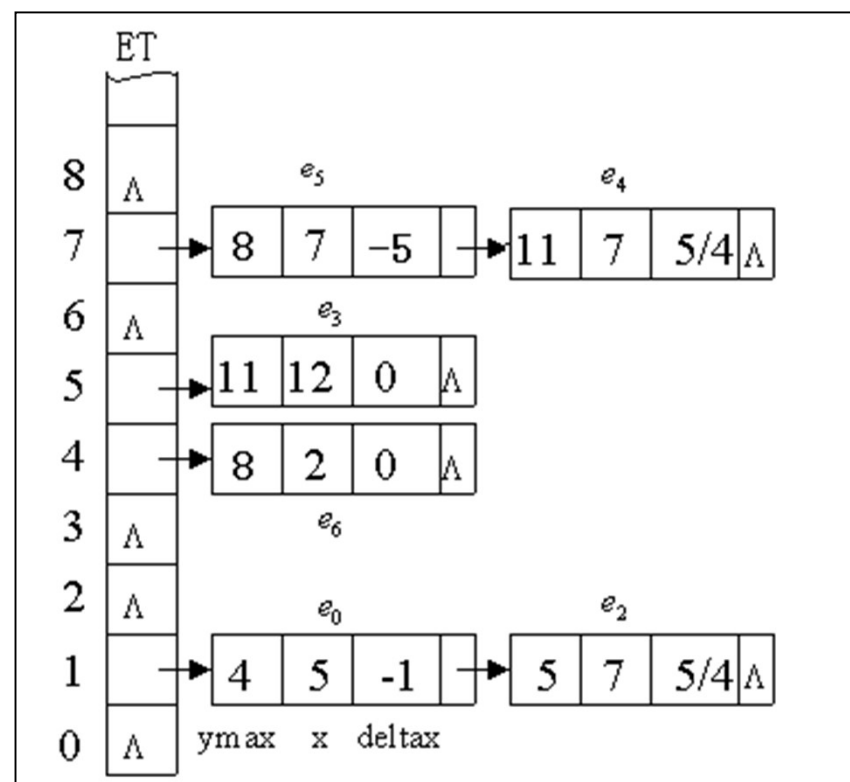
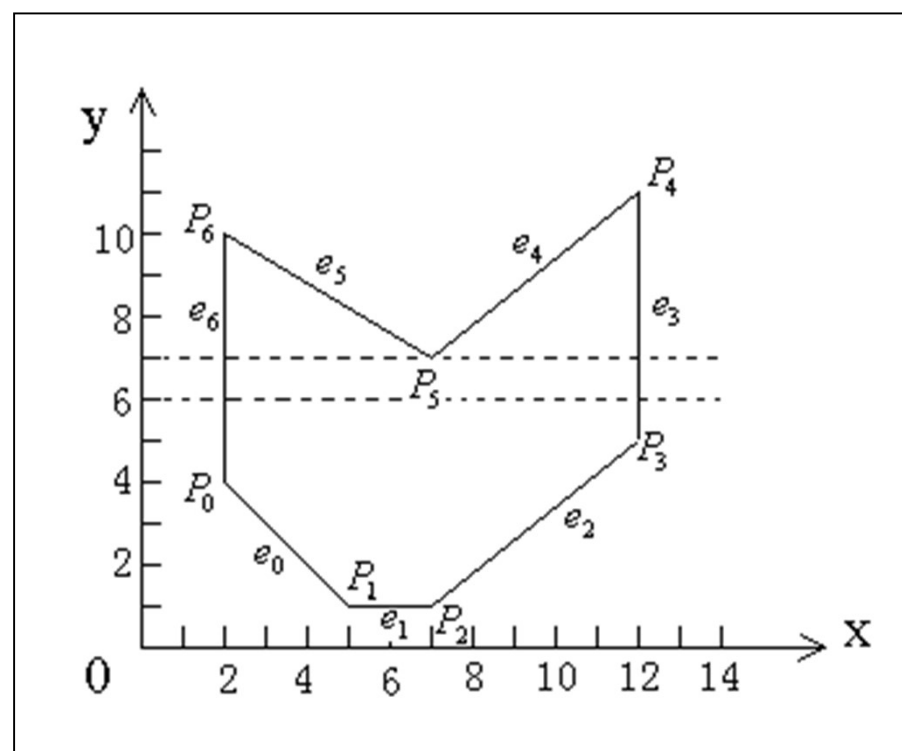
这样，当建立了边的分类表ET后，扫描线算法可按下列步骤进行：

- （1）取扫描线纵坐标 y 的初始值为ET中非空元素的最小序号。
- （2）将边的活化链表AEL设置为空。
- （3）按从下到上的顺序对纵坐标值为 y 的扫描线（当前扫描线）执行下列步骤，直到边的分类表ET和边的活化链表都变成空为止。



算法实现步骤

- 1) 如边分类表ET中的第 y 类元素非空, 则将属于该类的所有边从ET中取出并插入边的活化链表中, AEL中的各边按照 x 值(当 x 值相等时, 按 Δx 值)递增方向排序。
- 2) 若相对于当前扫描线, 边的活化链表AEL非空, 则将AEL中的边两两依次配对, 即1, 2边为一对, 3, 4边为一对, 依次类推。每一对边与当前扫描线的交点所构成的区段位于多边形内, 依次对这些区段上的点(像素)按多边形属性着色。
- 3) 将边的活化链表AEL中满足 $y=y_{\max}$ 的边删去。
- 4) 将边的活化链表AEL剩下的每一条边的 x 域累加 Δx , 即 $x:=x+\Delta x$ 。
- 5) 将当前的扫描线的纵坐标值 y 累加1, 即 $y:=y+1$ 。





扫描线算法

- 特点：算法效率比逐点填充法高很多。
- 缺点：对各种表的维持和排序开销太大，适合软件实现而不适合硬件实现。



扫描线算法

问题：

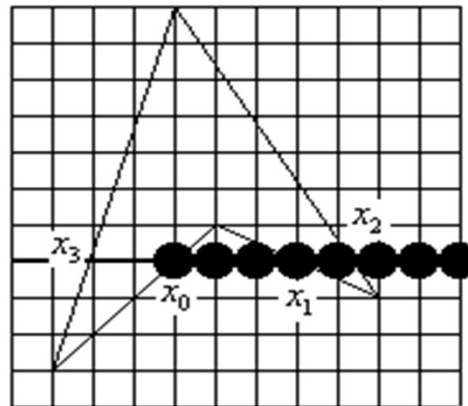
- 如何修改扫描线算法，使它能处理边自交的多边形？

边缘填充算法

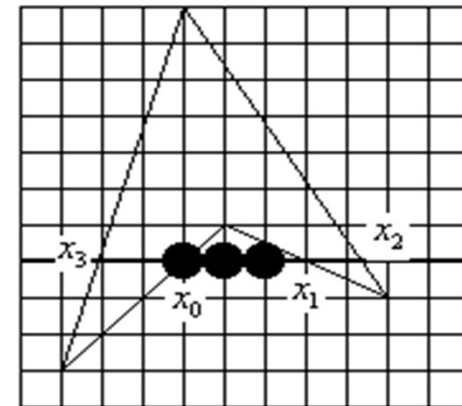
- **求余运算**：假定A为一个正整数，则正整数M的余定义为 $A-M$ ，记为 \bar{M} 。计算机中用n位表示M时，取A为n位能表示的最大整数。即 $A=0xFFFFFFFF$
- **由来**：光栅图形中，如果某区域已着上值为M的颜色值做偶数次求余运算，该区域颜色不变；而做奇数次求余运算，则该区域颜色变为值为 \bar{M} 的颜色。这一规律应用于多边形扫描转换，就称为边缘填充算法。
- **算法基本思想**：对于每条扫描线和每条多边形边的交点，将该扫描线上交点右方的所有像素取余。

算法1（以扫描线为中心的边 缘填充算法）

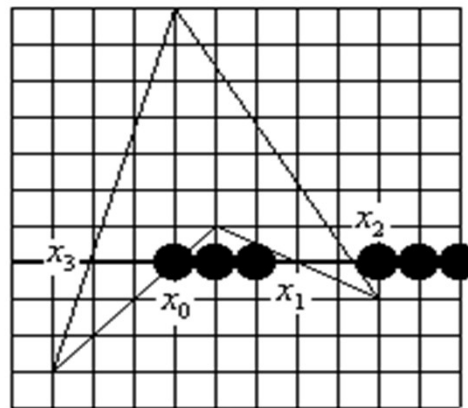
- 1、将当前扫描线上的所有像素着上 \overline{M} 颜色
- 2、求余：
for (i=0; i<=m; i++)
在当前扫描线上，
从横坐标为 x_i 的交点
向右求余；



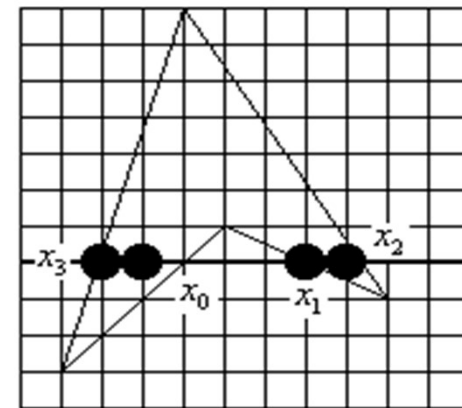
(a)



(b)



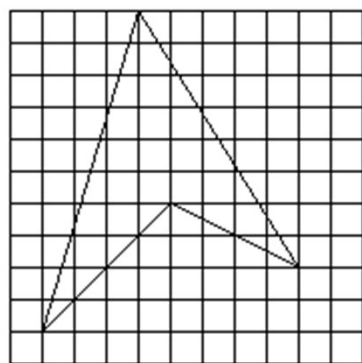
(c)



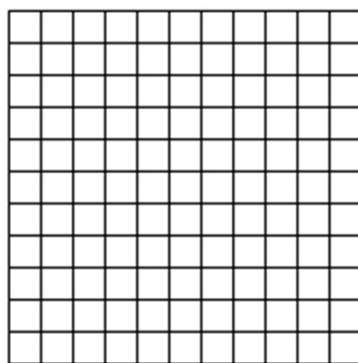
(d)

算法2（以边为中心的边缘填充算法）

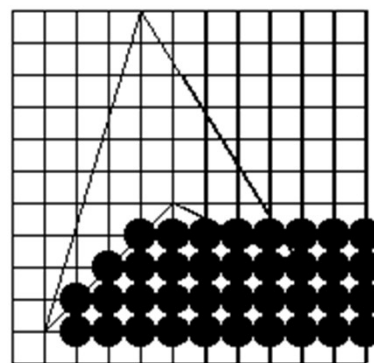
- 1、将绘图窗口的背景色置为 \overline{M} ；
- 2、对多边形的每一条非水平边做：
从该边上的每个象素开始向右求余；



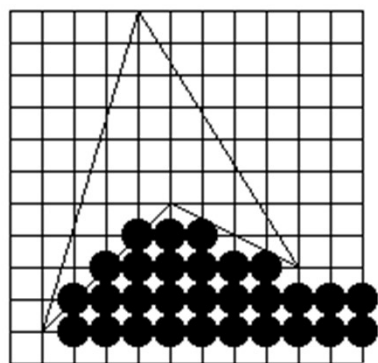
(a)



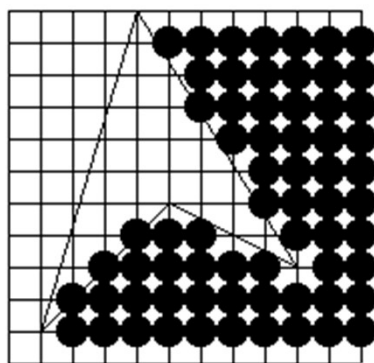
(b)



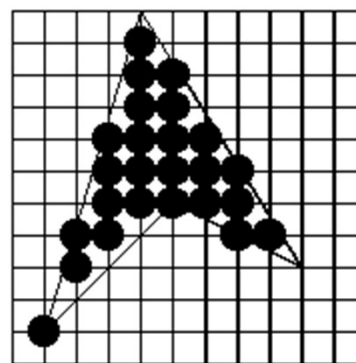
(c)



(d)

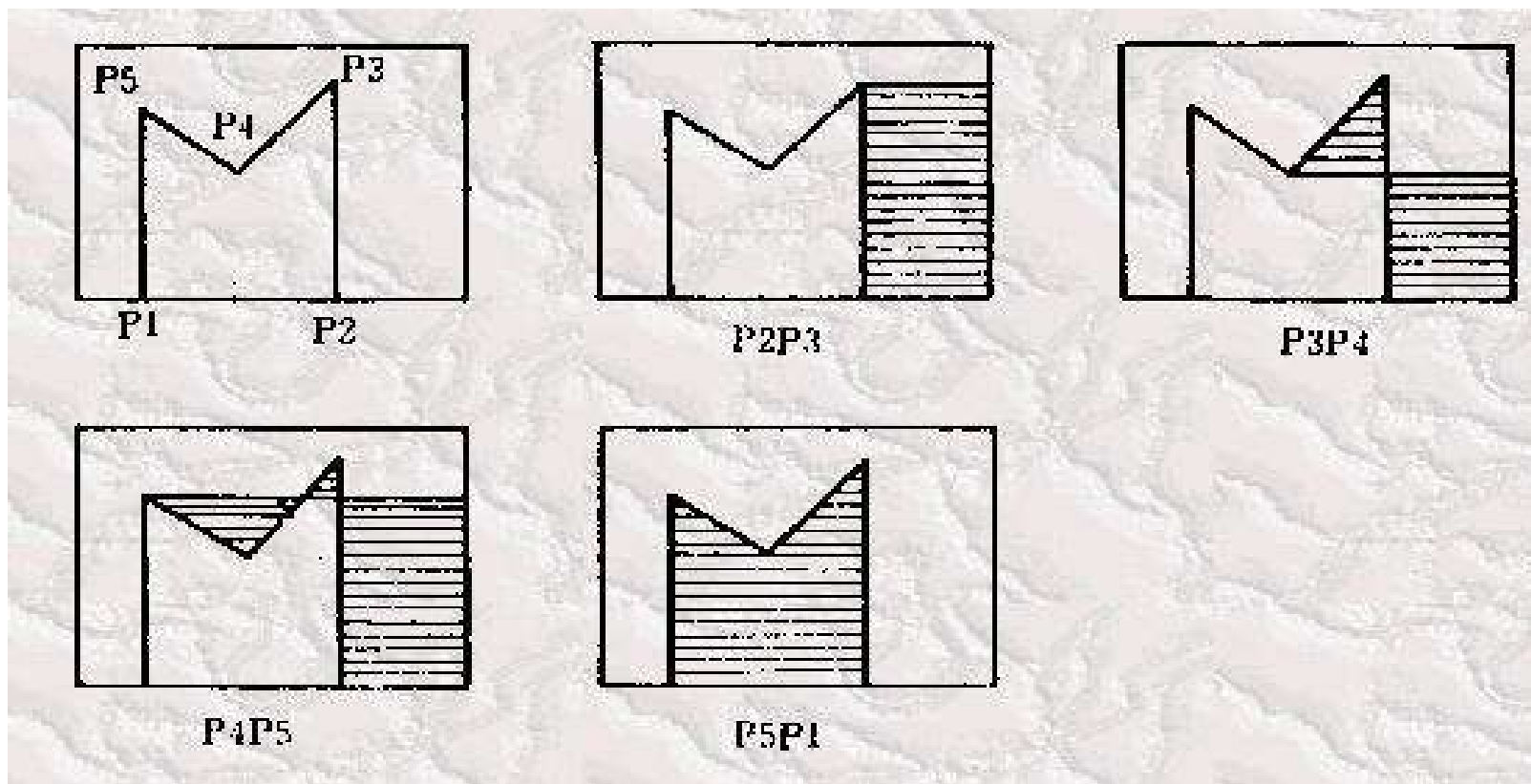


(e)



(f)

算法2（以边为中心的边缘填充算法）



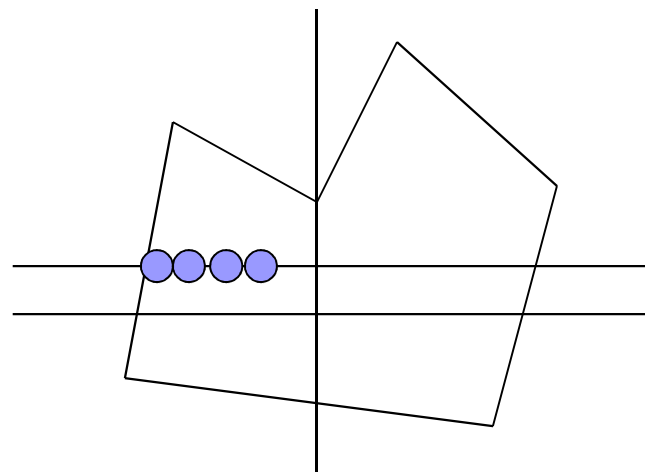


边缘填充算法

- 适合用于具有帧缓存的图形系统。处理后，按扫描线顺序读出帧缓存的内容，送入显示设备。
- 优点：算法简单
- 缺点：对于复杂图形，每一像素可能被访问多次，输入/输出的量比有序边表算法大得多。

栅栏填充算法

- 引入栅栏，以减少填充算法访问像素的次数。
- 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像素取补。
- 减少了像素重复访问数目，但不彻底。





边界标志算法

1. 对多边形的每一条边进行扫描转换，即对多边形边界所经过的像素作一个边界标志。

2. 填充。对每条与多边形相交的扫描线，按从左到右的顺序，逐个访问该扫描线上的像素。

取一个布尔变量`inside`来指示当前点的状态，若点在多边形内，则`inside`为真。若点在多边形外，则`inside`为假。

`Inside` 的初始值为假，每当当前访问像素为被打上标志的点，就把`inside`取反。对未打标志的点，`inside`不变。



边界标志算法: 算法过程

```
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;
    inside = FALSE;
    for (每条与多边形polydef相交的扫描线y )
    for (扫描线上每个像素x )
    { if(像素 x 被打上边标志)
        inside = ! (inside);
      if(inside != FALSE)
        drawpixel (x, y, color);
      else drawpixel (x, y, background);
    }
}
```



边界标志算法

- 用软件实现时，扫描线算法与边界标志算法的执行速度几乎相同，
- 但由于边界标志算法不必建立维护边表以及对它进行排序，所以边界标志算法更适合硬件实现，这时它的执行速度比有序边表算法快一至两个数量级。



边界标志算法

- 思考：如何处理边界的交点个数使其成为偶数？



区域填充算法

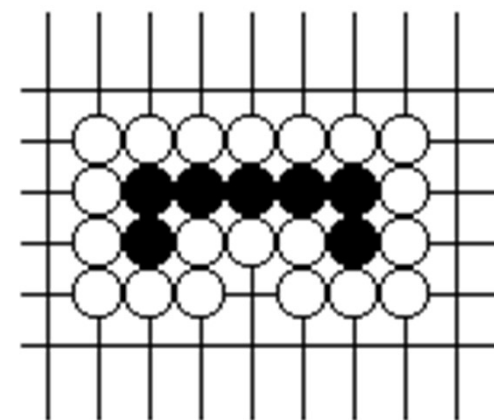
- **区域**指已经表示成点阵形式的填充图形，它是像素的集合。
- **区域填充**指先将区域的一点赋予指定的颜色，然后将该颜色扩展到整个区域的过程。区域填充算法要求区域是连通的。

区域填充

➤表示方法：内点表示、边界表示

➤内点表示

- 枚举处区域内部的所有像素
- 内部的所有像素着同一个颜色
- 边界像素着与内部像素不同的颜色



➤边界表示

- 枚举出边界上所有的像素
- 边界上的所有像素着同一颜色
- 内部像素着与边界像素不同的颜色

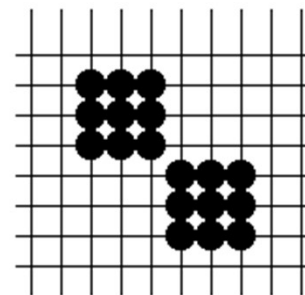
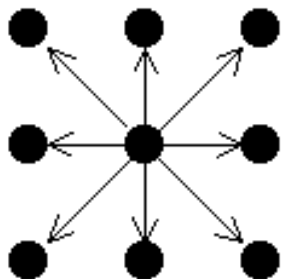
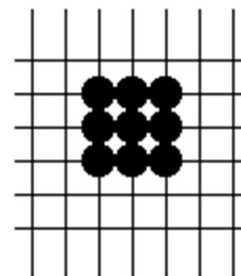
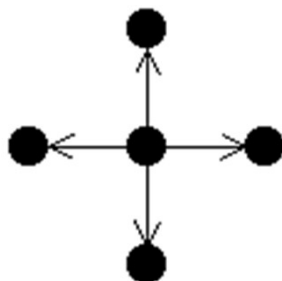
区域填充

区域填充要求区域是连通的（种子点）

➤ 连通性

4连通、8连通

➤ 4连通:



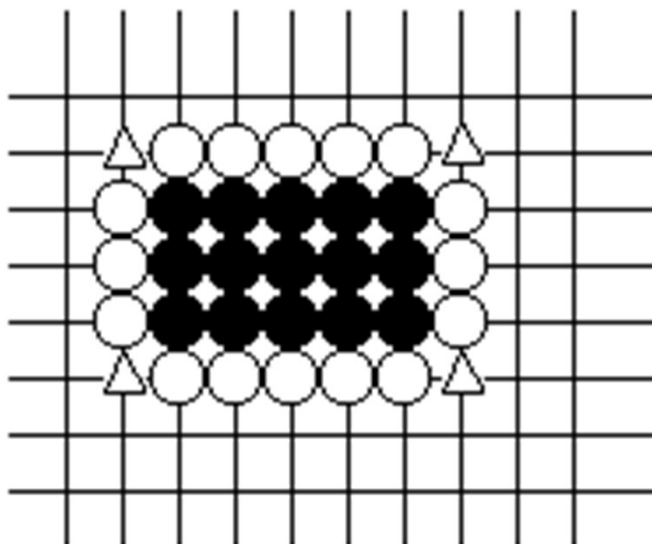
➤ 8连通

区域填充

➤ 4连通与8连通区域的区别

➤ 连通性： 4连通可看作8连通区域，但对边界有要求

➤ 对边界的要求





种子填充算法

A: 适合于内点表示区域的填充算法

设 G 为一内点表示的区域， (x, y) 为区域内一点， old_color 为 G 的原色。现取 (x, y) 为种子点对区域 G 进行填充：即先置像素 (x, y) 的颜色为 new_color ，然后逐步将整个区域 G 都置为同样的颜色。步骤如下：

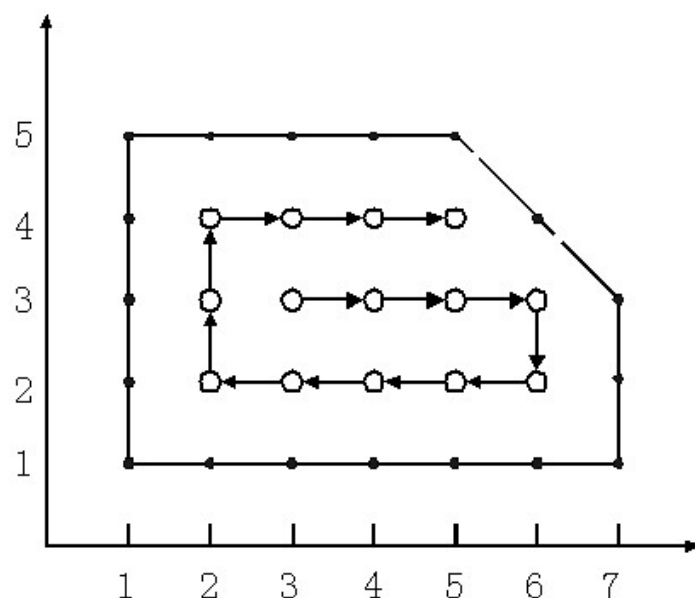
种子像素入栈，当栈非空时，执行如下三步操作：

- (1) 栈顶像素出栈；
- (2) 将出栈像素置成多边形色；
- (3) 按上、下、左、右的顺序检查与出栈像素相邻的四个像素，若其中某个像素不在边界上且未置成多边形色，则把该像素入栈。

种子填充算法

➤ 例：多边形由 $P_0P_1P_2P_3P_4$ 构成，
 $P_0(1,5)P_1(5,5)P_2(7,3)P_3(7,1)P_4(1,1)$

设种子点为 $(3, 3)$ ，搜索的方向是上、下、左、右。依此类推，最后像素被选中并填充的次序如图中箭头所示





种子填充算法

递归算法可实现如下

```
void FloodFill4(int x, int y, int oldColor, int newColor)  
{ if(GetPixel(x, y) == oldColor)  
  { PutPixel(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);  
  }  
} /*end of FloodFill4() */
```



种子填充算法

➤ 边界表示的4连通区域

```
void BoundaryFill4(int x,int y,int boundaryColor,int newColor)  
{  
    int color;  
    color = GetPixel(x,y);  
    if((color != boundaryColor) && (color != newColor))  
    {  
        PutPixel(x,y,newColor);  
        BoundaryFill4(x,y+1,oldColor,newColor);  
        BoundaryFill4(x,y-1,oldColor,newColor);  
        BoundaryFill4(x-1,y,oldColor,newColor);  
        BoundaryFill4(x+1,y,oldColor,newColor);  
    }  
} /*end of BoundaryFill4()    */
```



种子填充算法

- 该算法也可以填充有孔区域。
- 缺点：
 - (1) 有些像素会入栈多次，降低算法效率；栈结构占空间。
 - (2) 递归执行，算法简单，但效率不高，区域内每一像素都引起一次递归，进/出栈，费时费内存。
- 改进算法，减少递归次数，提高效率。

解决方法是用扫描线填充算法



扫描线算法

- **目标：**减少递归层次
- 适用于内点表示的4连通区域
- **算法思想：**在任意不间断区间中只取一个种子像素（不间断区间指在一条扫描线上一组相邻元素），填充当前扫描线上的该段区间；然后确定与这一区段相邻的上下两条扫描线上位于区域内的区段，并依次把它们保存起来，反复进行这个过程，直到所保存的每个区段都填充完毕。



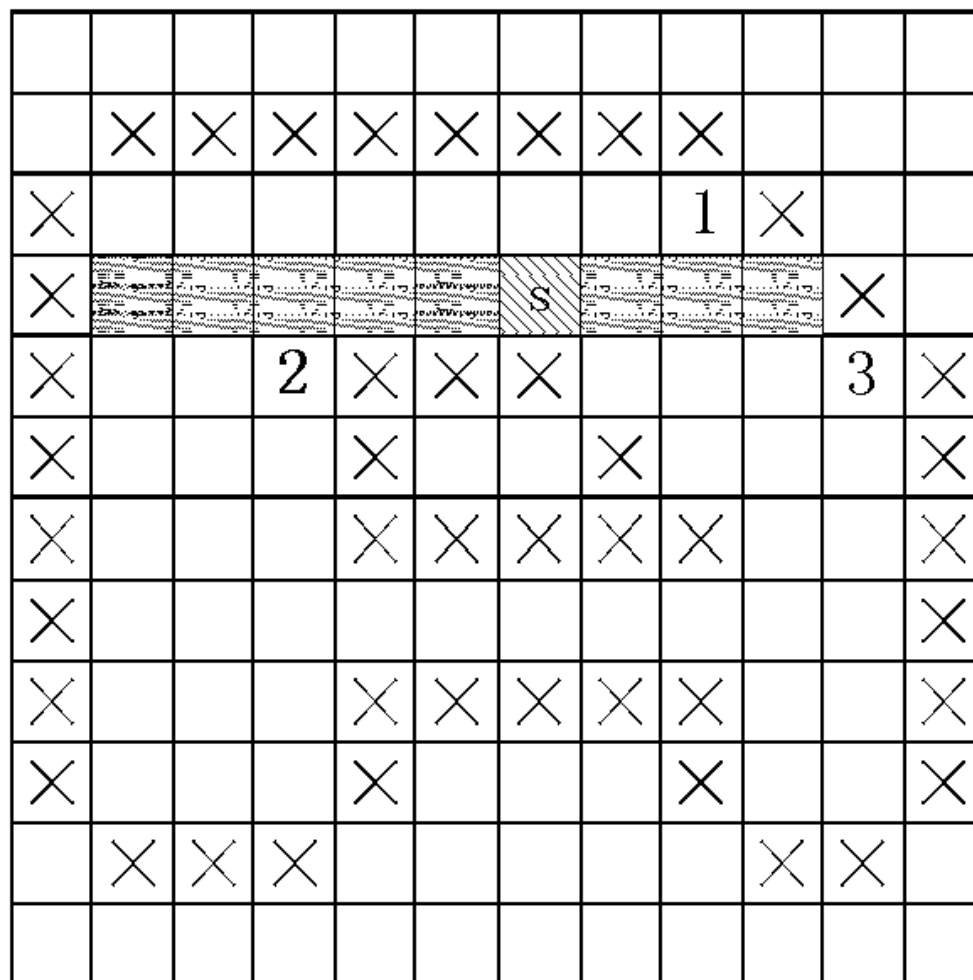
扫描线填充算法

- (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 x_l 和 x_r 。
- (4) 并确定新的种子点：在区间 $[x_l, x_r]$ 中检查与当前扫描线 y 上、下相邻的两条扫描线上的像素。若存在非边界、未填充的像素，则把每一区间的最右像素作为种子点压入堆栈，返回第 (2) 步。

上述算法对于每一个待填充区段，只需压栈一次；因此，扫描线填充算法提高了区域填充的效率。

扫描线算法分析（举例分析）

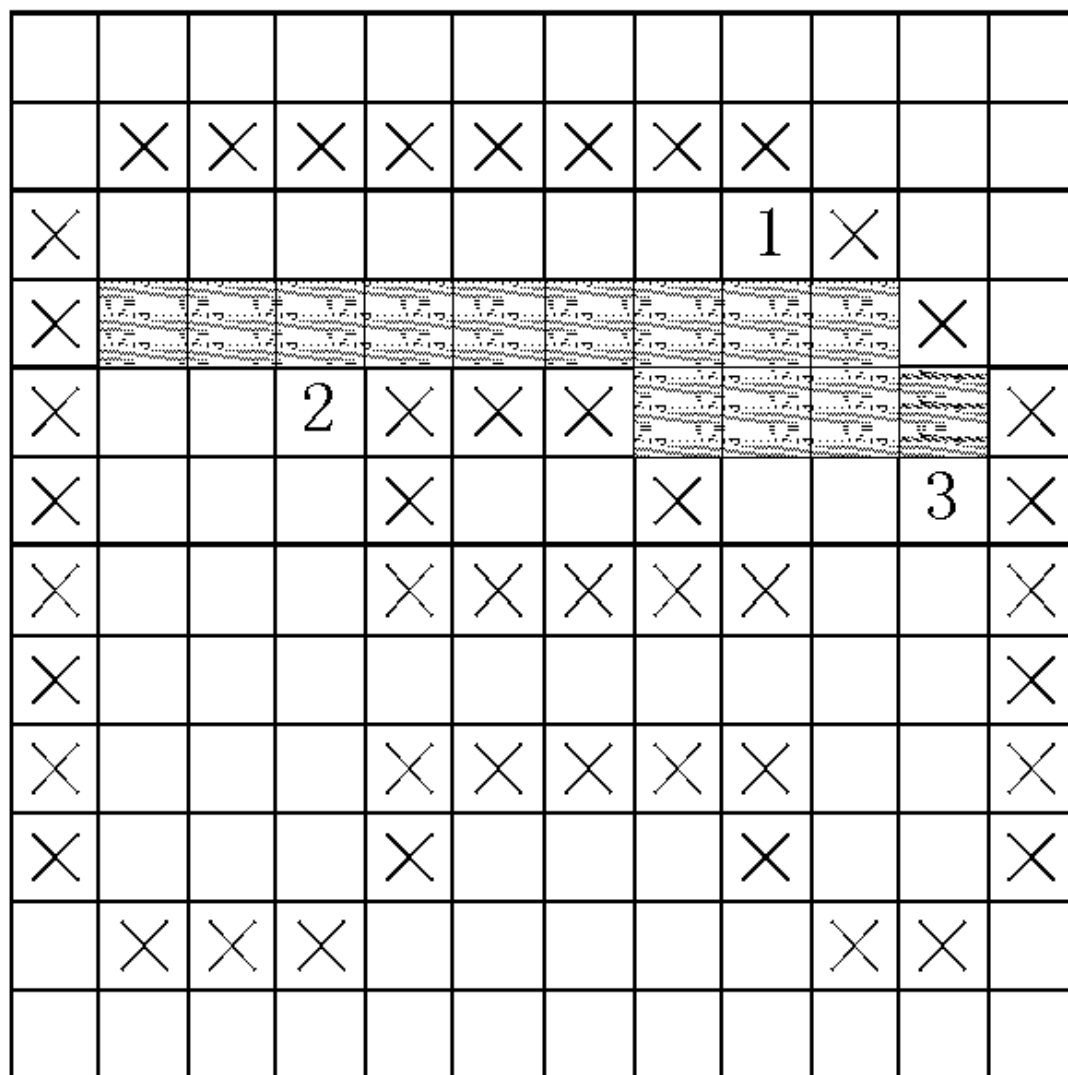
- 该算法也可以填充有孔区域。



(a)

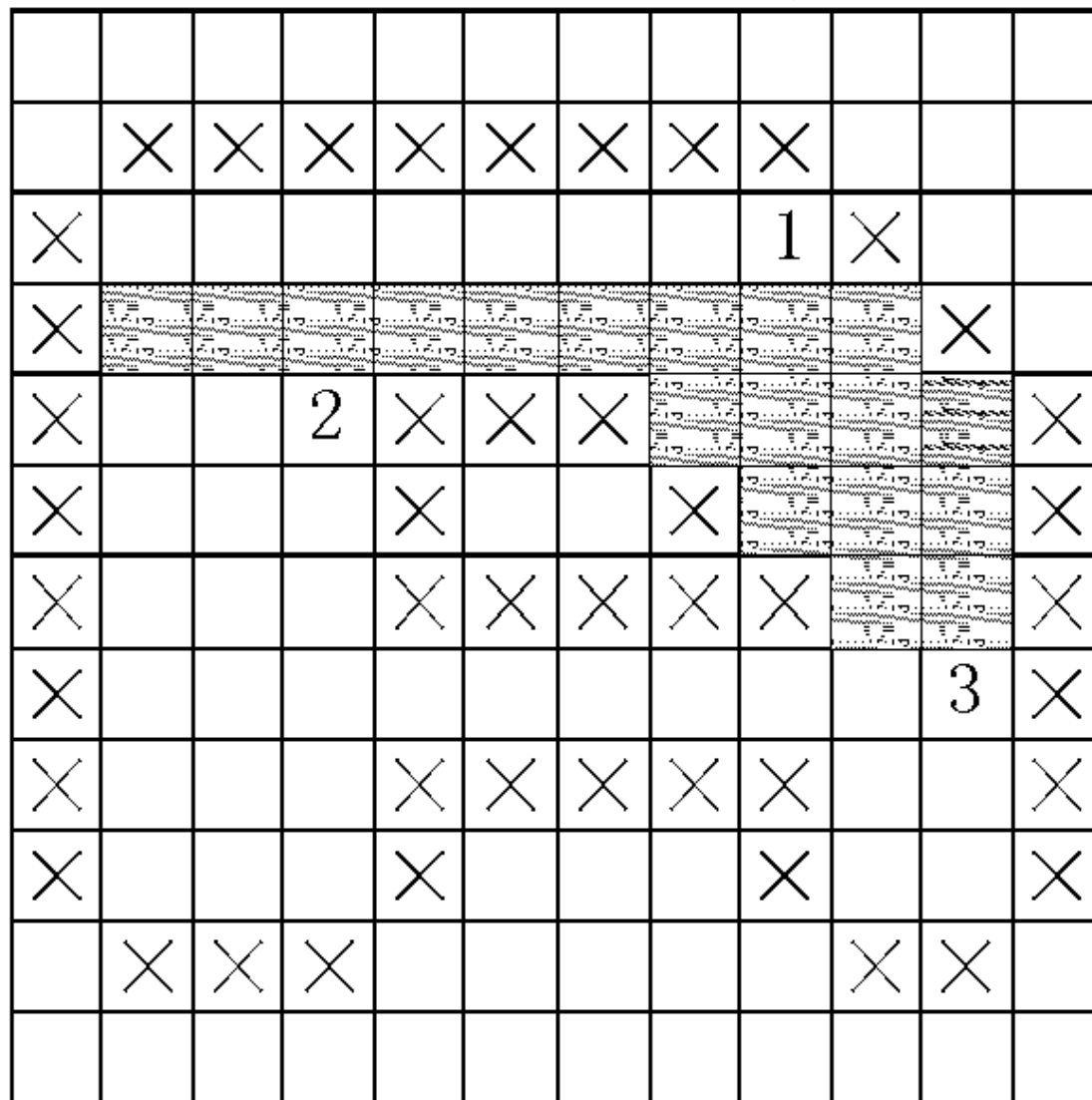
像素中的序号标指它所在区段位于堆栈中的位置

扫描线算法分析（举例分析）



(b)

扫描线算法分析（举例分析）



(c)

扫描线算法分析（举例分析）

	×	×	×	×	×	×	×	×			
×								1	×		
×										×	
×			2	×	×	×					×
×				×			×				×
×			3	×	×	×	×	×			×
×											×
×			4	×	×	×	×	×		5	×
×				×				×			×
	×	×	×						×	×	

(d)



多边形扫描转换与区域填充方法比较

- **联系**：都是光栅图形面着色，用于真实感图形显示。可相互转换。
- 多边形的扫描转换**转化**为区域填充问题：当给定多边形内一点为种子点，并用*Bresenham*或*DDA*算法将多边形的边界表示成八连通区域后，则多边形的扫描转换转化为区域填充。
- 区域填充**转化**为多边形的扫描转换：若已知给定多边形的顶点，则区域填充转化为多边形的扫描转换。



多边形扫描转换与区域填充方法比较

不同点:

1. 基本思想不同

前者是顶点表示转换成点阵表示，后者只改变区域内填充颜色，没有改变表示方法。

2. 对边界的要求不同

前者只要求扫描线与多边形边界交点个数为偶数。后者：区域封闭，防止递归填充跨界。

3. 基本的条件不同

前者：从边界顶点信息出发，后者：区域内种子点。