第三章 二维图元的填充

一般步骤

- 确定那些像素位于填充图元的内部；
- 确定以什么颜色填充这些像素；

主要内容

- 扫描转换矩形
- 扫描转换多边形:
  - 逐点判断法
  - 扫描线算法
  - 边缘填充算法
- 扫描转换扇形
- 区域填充(种子填充法)
  - 递归填充算法
  - 扫描线算法
- 以图像填充区域
- 字符的表示与输出

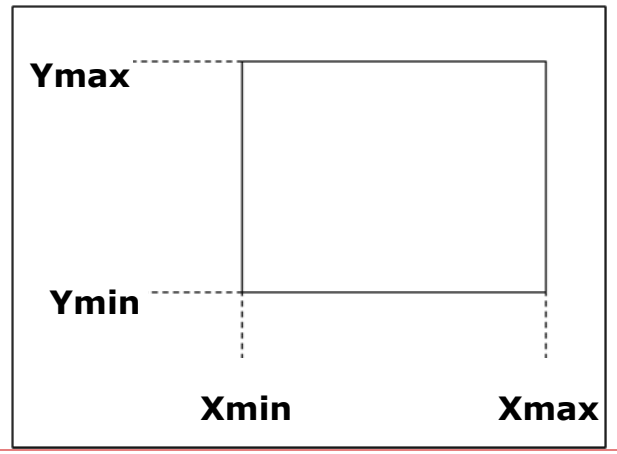
3.1 扫描转换矩形

- 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);
}
```

- 为减少函数调用次数，也可：

```
for (y= rect->ymin;  y<=rect->ymax; y++)
      DrawLine(xmin, y, xmax, y, color);
```

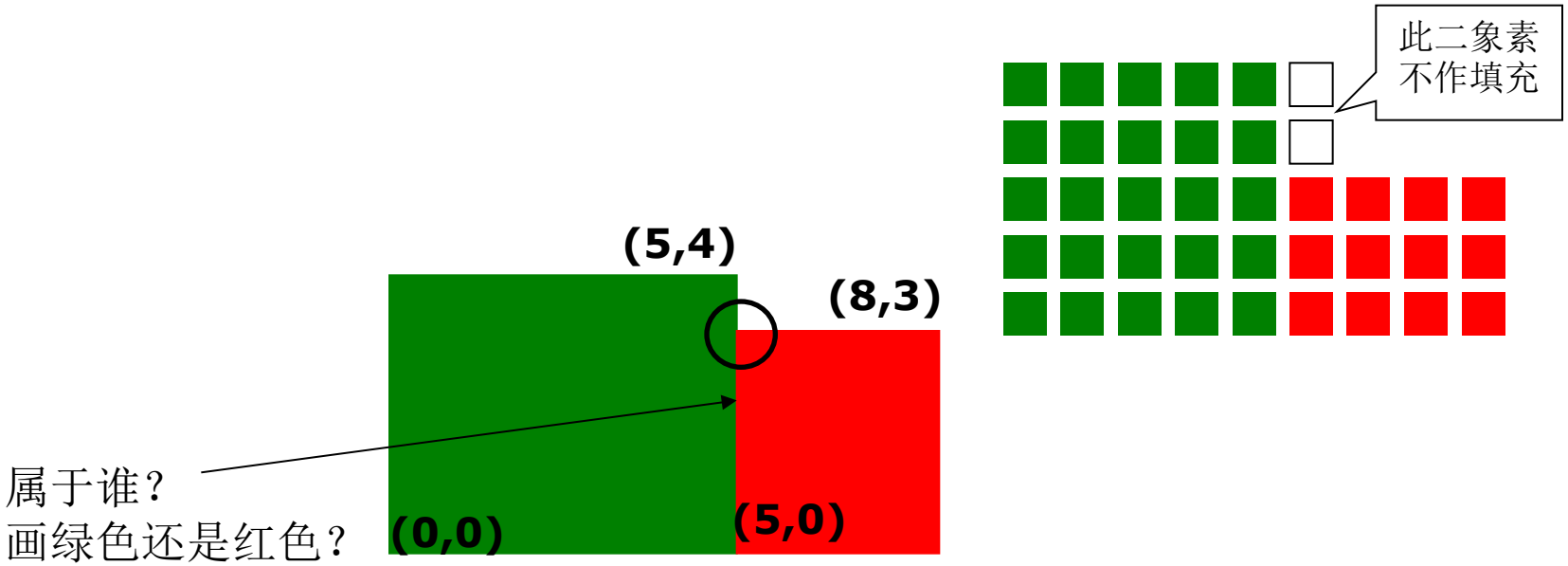


问题：

- 矩形是简单的多边形，可以用多边形填充方法，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，如窗口系统。

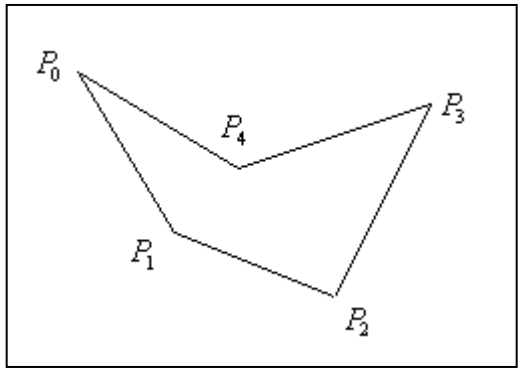
问题：

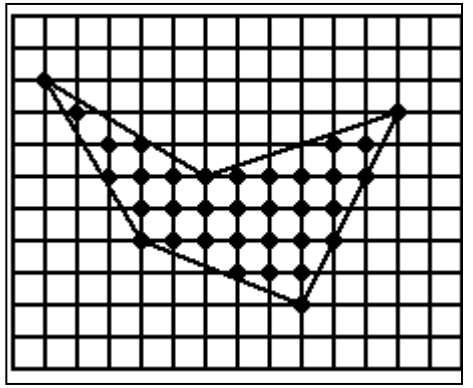
- 共享边界的处理
  - 原则：左闭右开，下闭上开。即矩形左边、下边的像素属于矩形。



3.2 扫描转换多边形

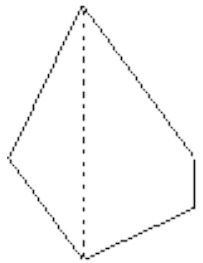
- 多边形的表示方法：
  - 顶点表示：用多边形的顶点序列来表示多边形。这种表示直观、几何意义强、占内存少，易于进行几何变换，但由于它没有明确指出哪些象素在多边形内故不能直接用于面着色。
  - 点阵表示：用位于多边形内的象素集合来刻画多边形。这种表示丢失了许多几何信息，但便于帧缓冲器表示图形，是面着色所需要的图形表示形式。



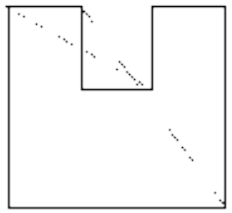


3.2 扫描转换多边形

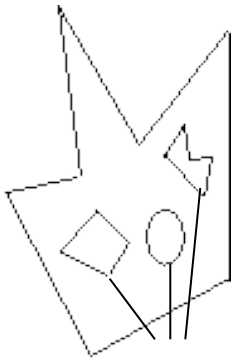
- 扫描转换多边形：将顶点表示形式转换成点阵表示形式，一般有三种方法：
  - 逐点判断法；
  - 扫描线算法；
  - 边缘填充法。
- 多边形分为凸多边形、凹多边形、含内环的多边形。



任意两顶点间的连线均在多边形内



任意两顶点间的连线有不在多边 形内的部分



内环

逐点判断法程序

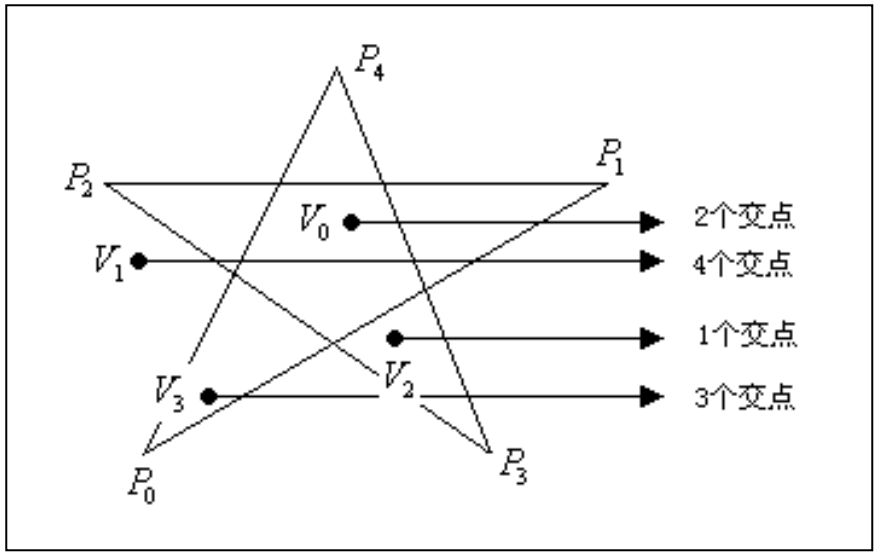
```
#define  MAX  100
typedef struct {  int   PolygonNum;        // 多边形顶点个数
                        Point vertexces[MAX]   //多边形顶点数组
                    }  Polygon;                    //  多边形结构
void FillPolygonPbyP(Polygon *P, int polygonColor)
{    int x,y;
       for(y = ymin;y <= ymax;y++)
            for(x = xmin;x <= xmax;x++) // 遍历图像包围盒中的所有像素
                 if  ( IsInside(P, x, y) )        //判断点是否在多边形内
                      PutPixel(x, y, polygonColor);     // 在多边形内部，画前景色
                 else
                      PutPixel(x, y, backgroundColor); // 在多边形内部，画背景色
}
```

判断点在多边形的内外关系

- 逐点判断法的关键在于点在多边形内外的判断方法
  - 射线法：
  - 累计角度法

射线法

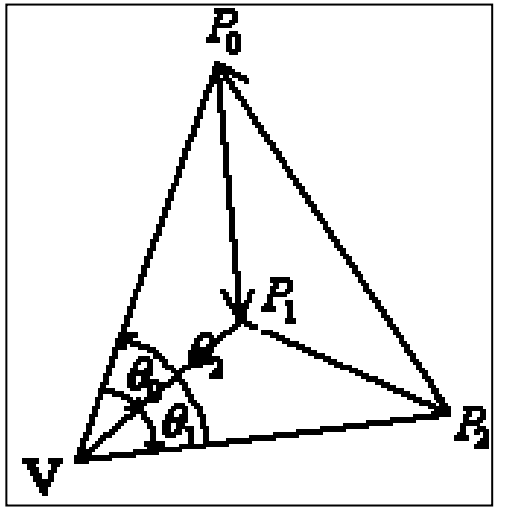
- 步骤：
  - 从待判别点v发出射线
  - 求交点个数k
  - K的奇偶性决定了点与多边形的内外关系
- 射线通过顶点的处理
  - 会造成误判
  - 强制这种情况不发生，如使射线通过中点。



累计角度法

- 步骤
  - 从v点向多边形P顶点发出射线，形成有向角 θ_i
  - 计算有向角的和，得出结论

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



- 问题：
  - 计算角度 θ_i 时要用三角函数、正弦余弦定律，而它们都是以 2π 为周期的函数，不唯一，可能形成误判。
  - 解决方法：θ_i 取绝对值不大于 π 有向角值。

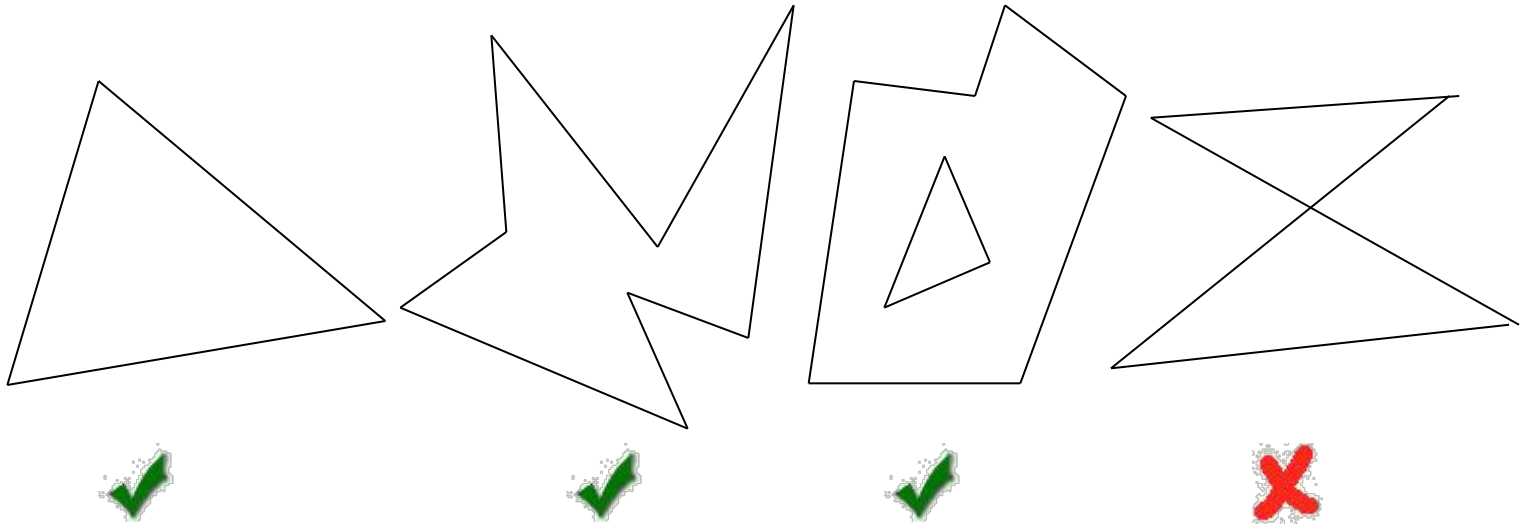
逐点判断法评价

- 优点：程序简单，
- 缺点：速度慢，效率低。
- 原因：没有利用相邻像素之间的连贯性

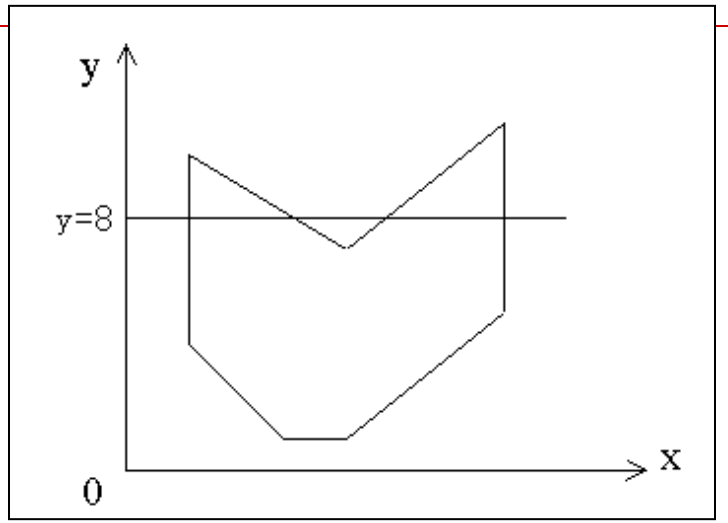
扫描线算法

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



- 基本原理

  一条扫描线与多边形的边有偶数个交点

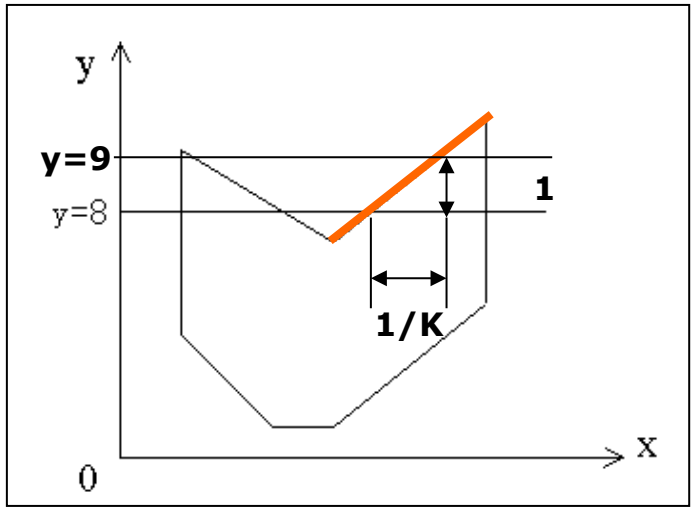
- 步骤(一般用水平扫描，对于每一条扫描线):

1. 求交：计算扫描线与多边形各边的交点；
2. 排序：把所有交点按x值递增顺序排序；
3. 配对：第一个与第二个，第三个与第四个等等；每对交点代表扫描线与多边形的一个相交区间;
4. 填色：把相交区间内的象素置成多边形颜色，把相交区间外的象素置成背景色。

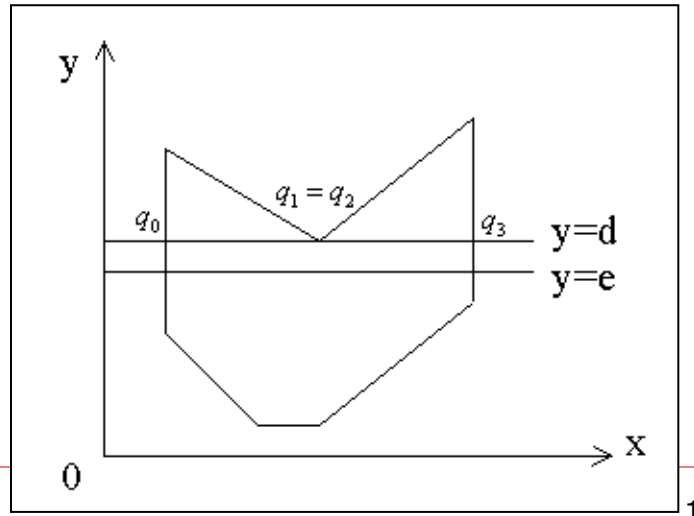
边的连贯性

- 相邻扫描线与边的交点的坐标有连贯性。计算方法与直线光栅化思想一致，我们用DDA算法。
- 设第i条扫描线与多边形的某条边E_j的交点的x坐标为x_{ij}，第i+1条扫描线与这条边的交点的x坐标为$x_{(i+1)j}$，这条边的斜率为k_j，则：

$$x_{(i+1)j} = x_{ij} + \frac{1}{k_j}$$



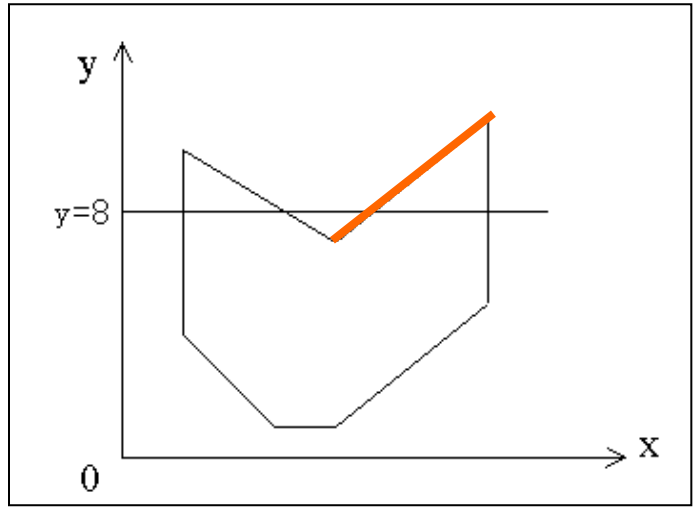
- 假定已求得扫描线y=e和多边形的所有交点，我们要递归出扫描线y=d=e+1和多边形的所有交点。这些交点可分为两类：
  - 第一类交点：位于同一条边上的后继交点(如q_0q_3)。
    - 此时，若x为某边与y=e的交点的横坐标，k为该边的斜率，
    - 则此边与y=d的交点的横坐标为x'=x+1/k
  - 第二类交点：新出现的边与扫描线的交点(如q_1q_2) 。
    - 此时，这些边的下端点就是交点，无需计算



边的数据结构

- 边的数据结构为

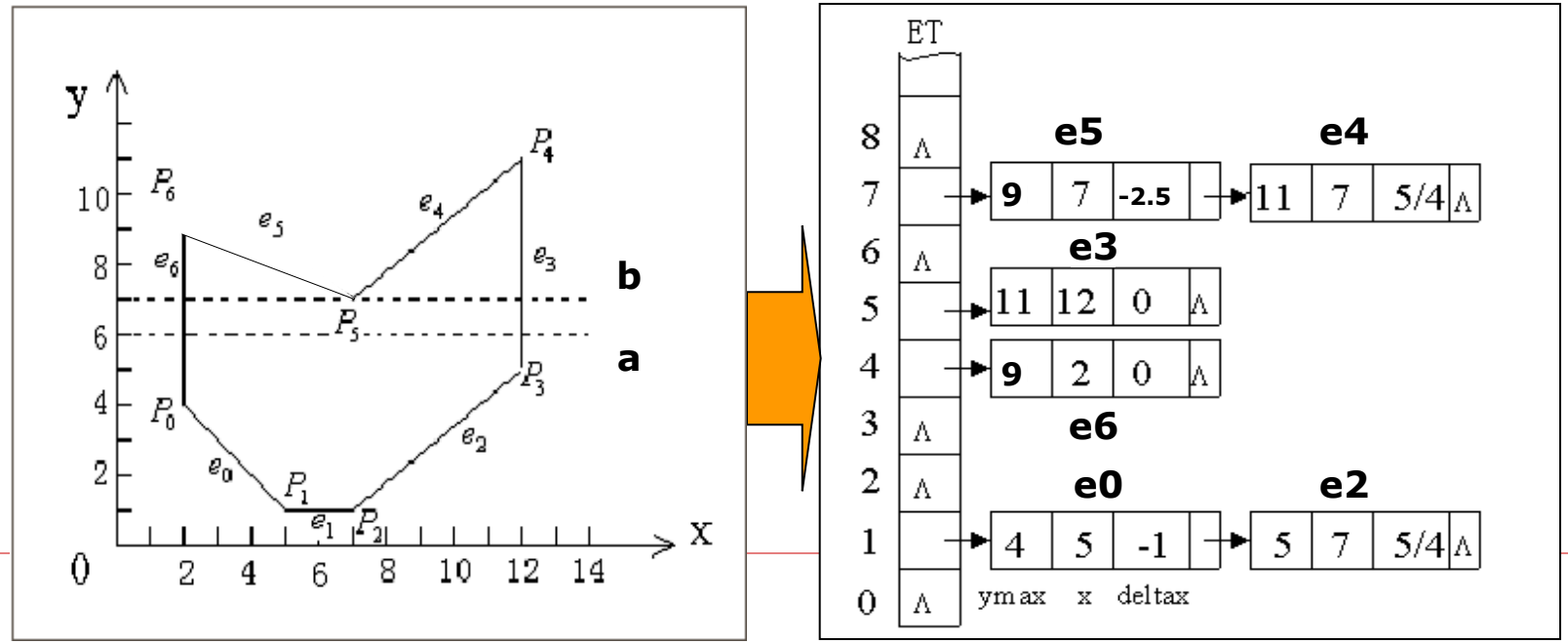
```
typedef  struct
{    int ymax;
     float x, deltax;
     struct Edge *nextEdge;
} Edge;
```



- 各参数含义如下：
  - ymax: 边的上端点y坐标;
  - x:初值为边下端点的x坐标，AEL中为当前扫描线与边的交点的X坐标;
  - deltax:边的斜率的倒数；
  - nextEdge:指向下一条边的指针。

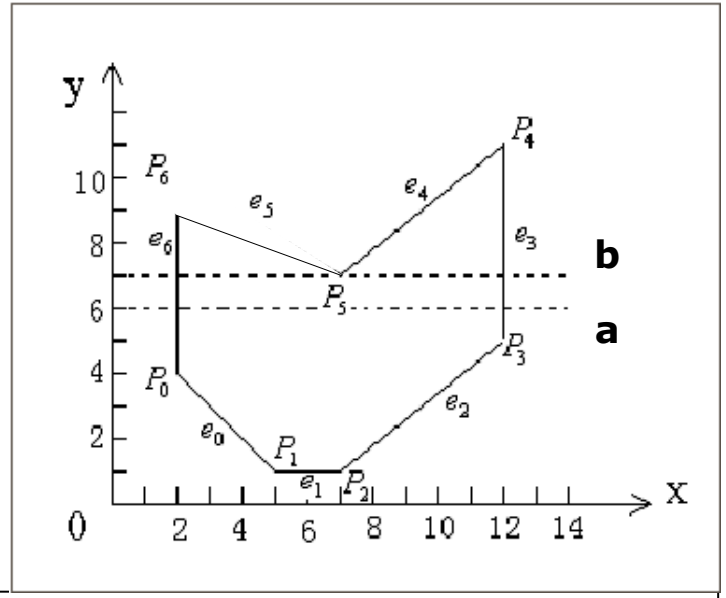
边的分类表(ET)

- 边的分类表：按照边的下端点y坐标对非水平边进行分类的指针数组，下端点y坐标值等于i的边属于第i类。
- 同一类中的边按x值(x相等的按deltax排)递增顺序排列。

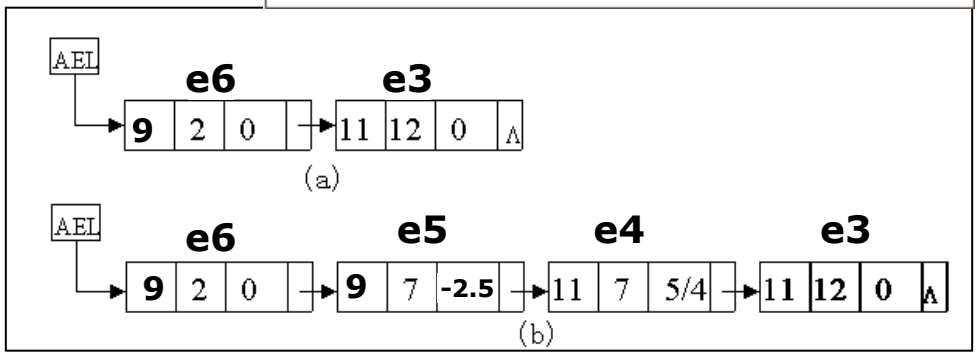


活性边表(AEL)

- 当处理一条扫描线时，仅对多边形与它相交的边进行求交运算。
- 我们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x坐标递增的顺序存放在一个链表中，称此链表为活性边表(AEL)。

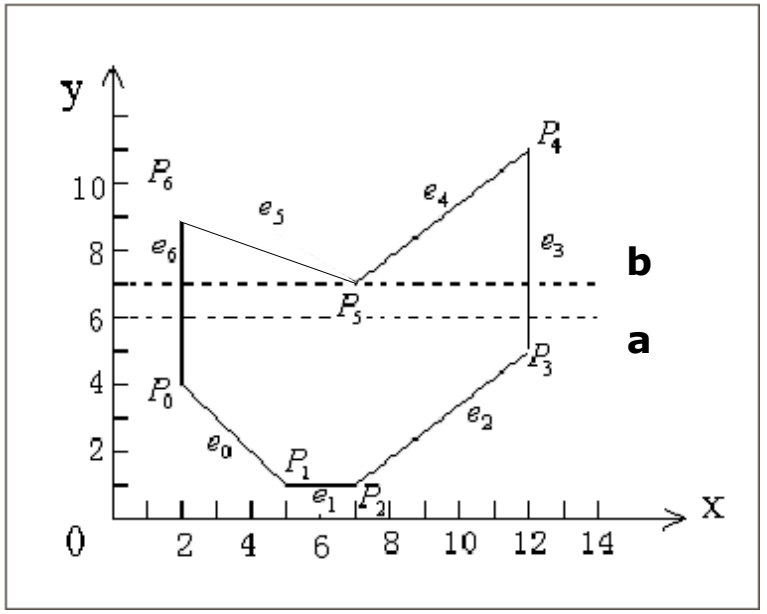



```
typedef  struct
{    int ymax;
     float x, deltax;
     struct Edge
          *nextEdge;
} Edge;
```



扫描线算法

1. 建立ET；
2. 将扫描线纵坐标y的初值置为ET中非空元素的最小序号，如在上图中，y=1；
3. 置AEL为空；



扫描线算法

4. 执行下列步骤直至ET和AEL都为空.

① 如ET中的第y类非空，则将其中的所有边取出并插入AEL中；

② 如果有新边插入AEL，则对AEL中各边排序；

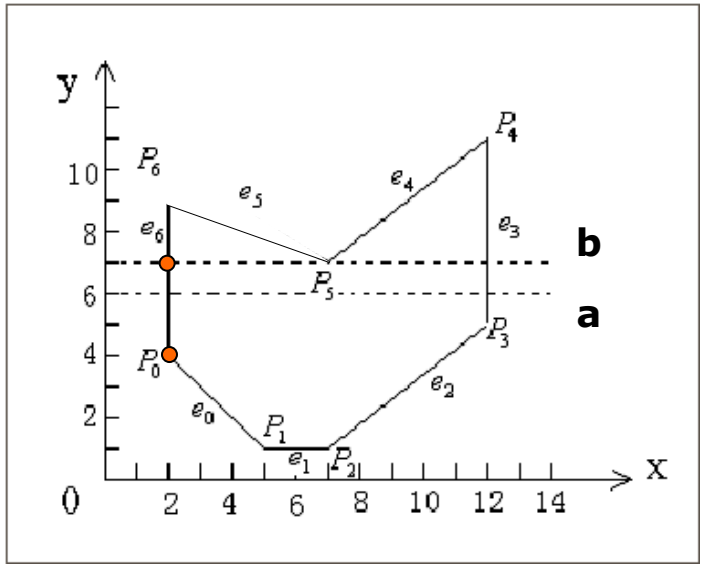
③ 对AEL中的边两两配对，（1和2为一对，3和4为一对，…），将每对边中x坐标按规则取整，获得有效的填充区段，再填充.

④ 将当前扫描线纵坐标y值递值1；

⑤ 将AEL中满足y=ymax边删去（因为每条边被看作下闭上开的）；

⑥ 对AEL中剩下的每一条边的x递增自身的deltax值，

即x = x+deltax.



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$\overline{M}$。计算机中取A为n位能表示的最大整数，即A=0xFFFFFFFF
- 由来：光栅图形中，如果某区域已填上颜色值M，当做偶数次求余运算，该区域颜色不变；做奇数次求余运算，则该区域颜色变为值为 $\overline{M}$ 的颜色。这一规律应用于多边形扫描转换，就为边缘填充算法。

边缘填充算法

- 求余运算可用异或显示模式实现。

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

$$\overline{\overline{M}} = (M \ \ XOR \ \ A) \ \ XOR \ \ A = M$$

- 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象素取余。

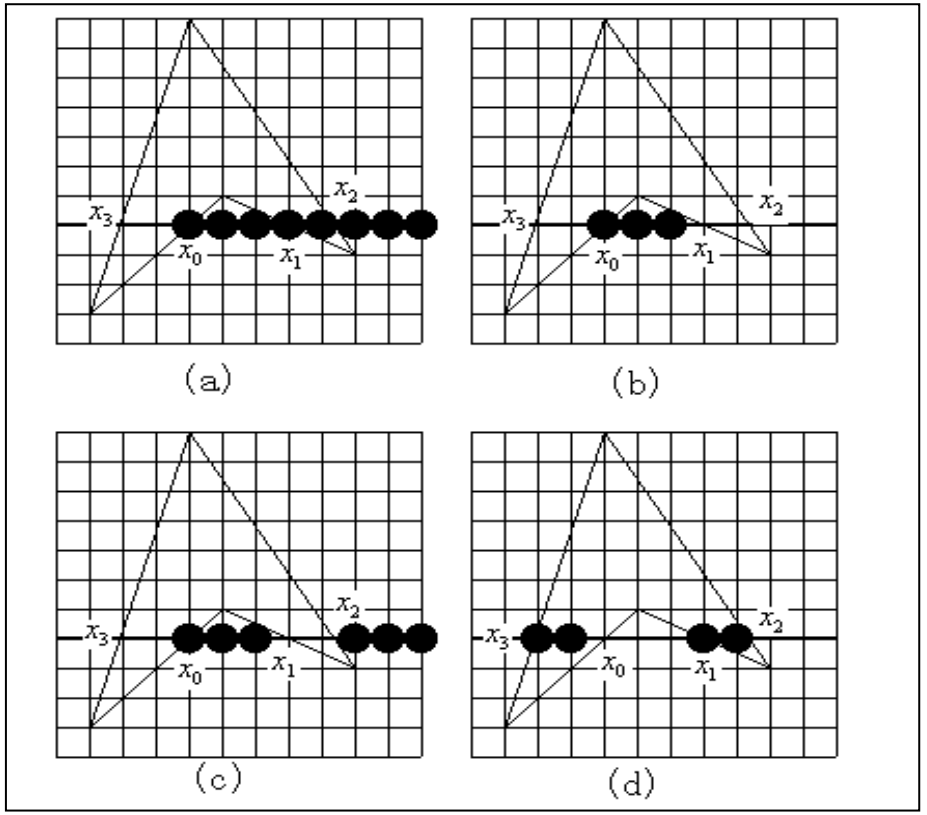
算法1(以扫描线为中心的边缘填充算法)

- 设x0,x1,…,xm是当前扫描线与多边形交点的x坐标串(不用排序)，按下面方法填充该扫描线：

1. 将当前扫描线上的所有象素着上$\overline{M}$颜色；
2. 求余：

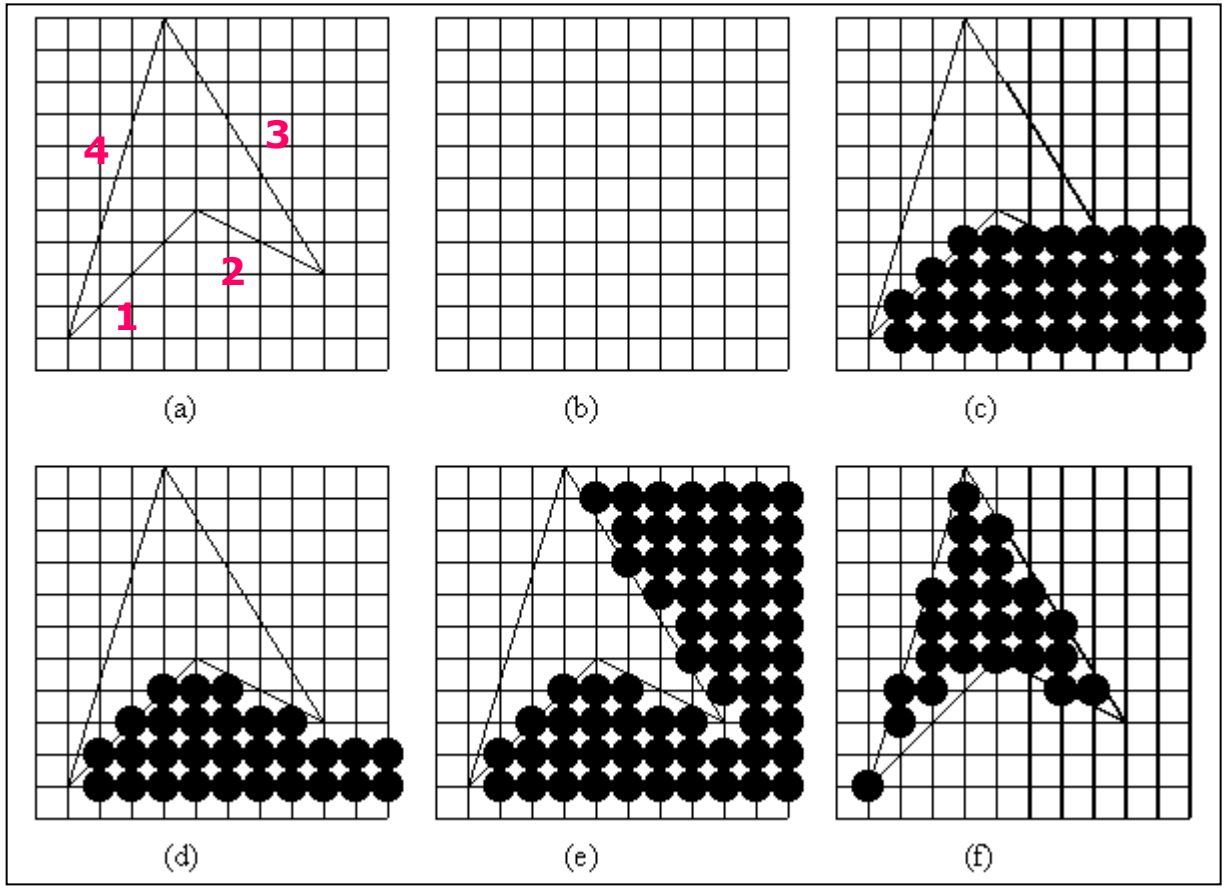
for(i=0;i<=m; i++)

在当前扫描线上，从横坐标为Xi的交点向右求余；



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；



边缘填充算法评价

- 优点：算法简单
- 缺点：对于复杂图形，每一象素可能被访问多次，输入/输出的量比扫描线算法大得多，造成速度较慢。
- 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。

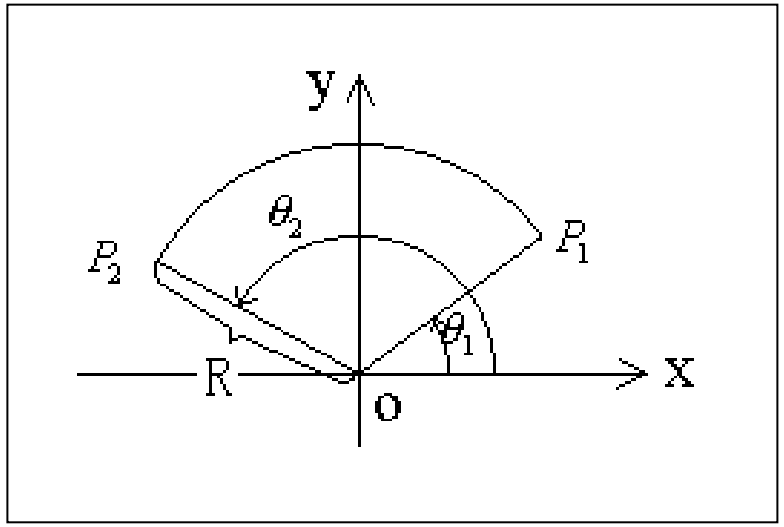
3.3 扫描转换扇形区域

- 扇形区域的描述:

  半径，起始角，终止角(设圆心在原点)
- 原理：同扫描转换多边形(圆弧与扫描线交点可求)
- 问题：如何确定扫描线与直线段和圆弧段的相交顺序
- 方法：分类

令点P1，P2满足：$\theta_1 < \theta_2$

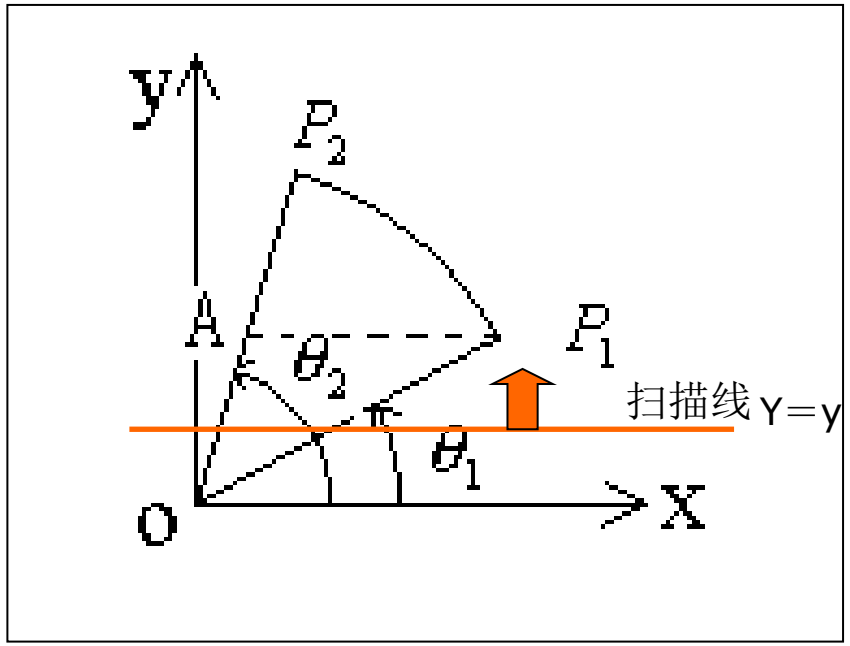
按P1和P2所处象限的不同，需要分成4×4=16种情况来讨论。



假设P1点落在第一象限 ,扇形区域的扫描转换分四种情况

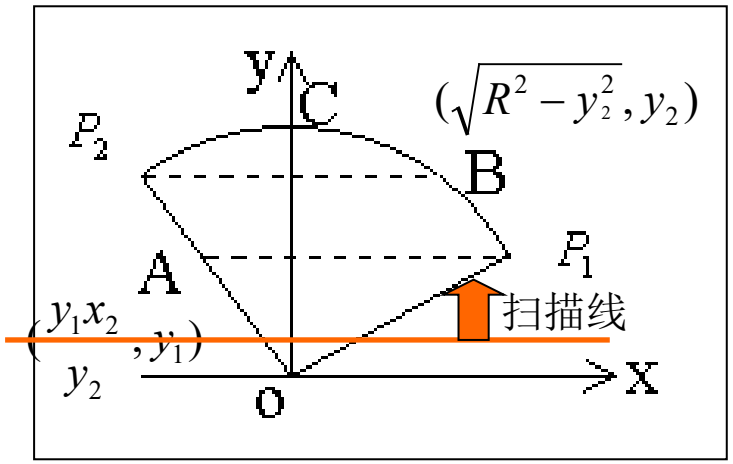
1、P2落在第一此时的相交顺序：

- $0<y<Y_{P1}$时，左点是扫描线与OP2交点，右点是扫描线与OP1交点。
- $Y_{P1}<y<Y_{P2}$时，左点是扫描线与OP2交点，右点是扫描线与圆弧P1P2交点。

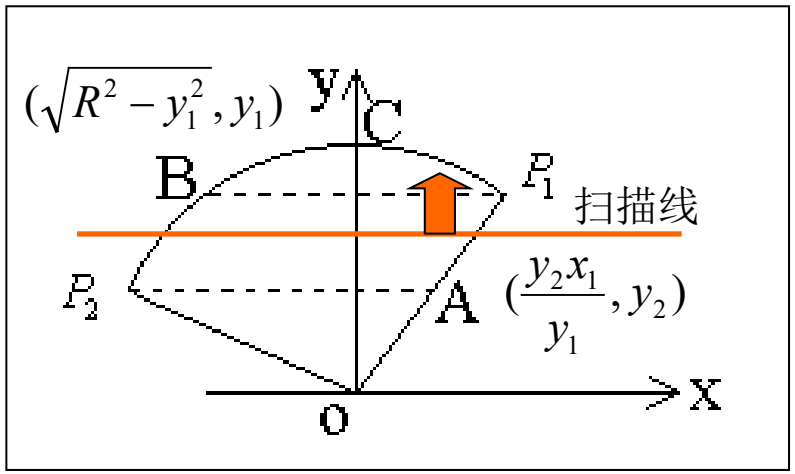


2、P2落在第二象限，此时又分为两种情况

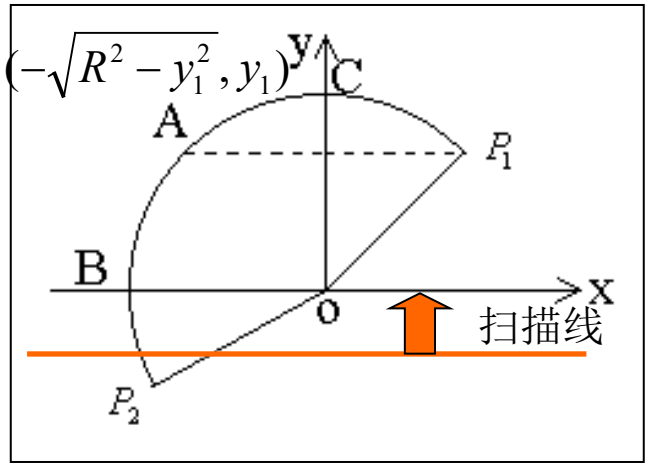
- 当Y1<Y2时



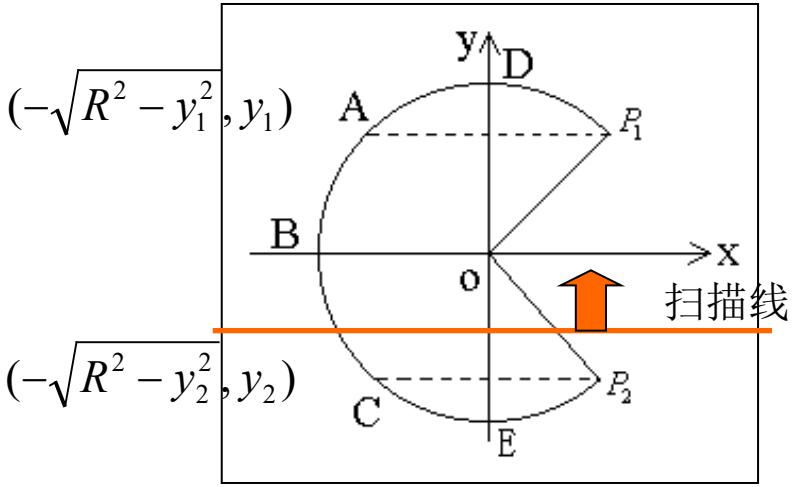
当Y1>Y2时



3、p2落在第三象限



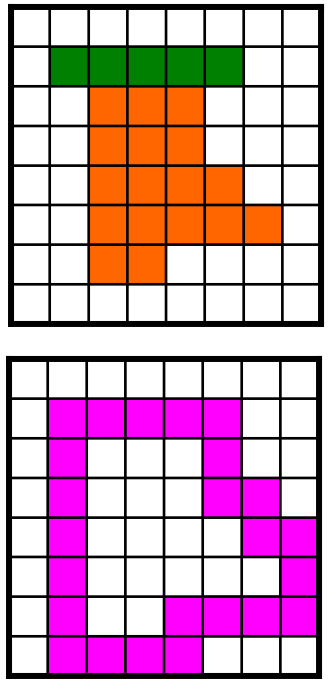
4、p2落在第四象限



- 遗留问题：当p1落在其它象限时?
  - 先将扇形顺时针转π/2(或π, 3π/2)，扫描完成后再逆时针转回原来位置。
- 扫描转换扇形的其它方法：
  - 先求出与扇形充分逼近的多边形，然后用扫描转化多边形的算法对其进行填充。

3.4 区域填充

- **区域**：点阵表示的图形，像素集合
- **表示方法**：内点表示、边界表示
  - **内点表示**
    - 枚举出区域内部的所有像素
    - 内部的所有像素着同一个颜色
    - 边界像素与内部像素不同颜色
  - **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
    - 内部像素与边界像素不同颜色

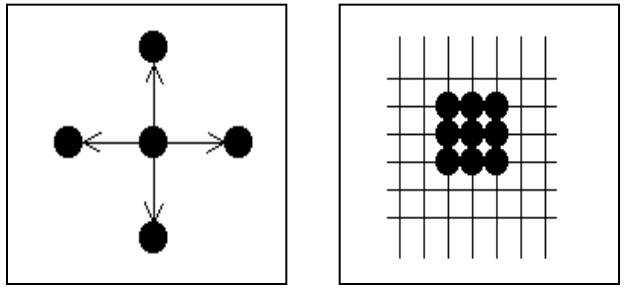


区域填充(种子填充法)

- 区域填充：对区域重新着色的过程
  - 将指定的颜色从种子点扩展到整个区域的过程
  - 区域填充算法要求区域是连通的
- 连通性：4连通、8连通

区域填充(种子填充法)

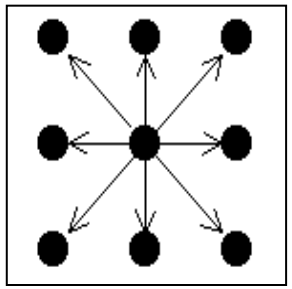
- 4连通
  - 两个像素点是上下或左右相连的，则称其为4连通的。
  - 一个像素点可通过四连通方式访问的像素点为上、下、左和右四个相邻像素点之一
  - 一个像素点最多与上、下、左和右四个相邻像素点具有连通关系，

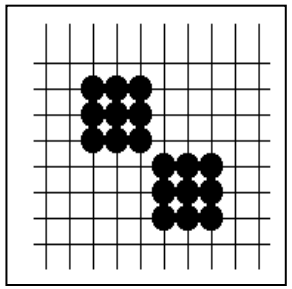


例子：PaintBrush

区域填充(种子填充法)

- 8连通
  - 两个像素点是上下或左右相连的，或者是对角线方向相连的，则称其为8连通的。
  - 一个像素点除可访问其上、下、左和右四个相邻像素点之外，还可访问两条对角线方向上四个相邻像素点，
  - 一个像素点最多与上、下、左和右四个及四个对角点共八个相邻像素点具有连通关系，



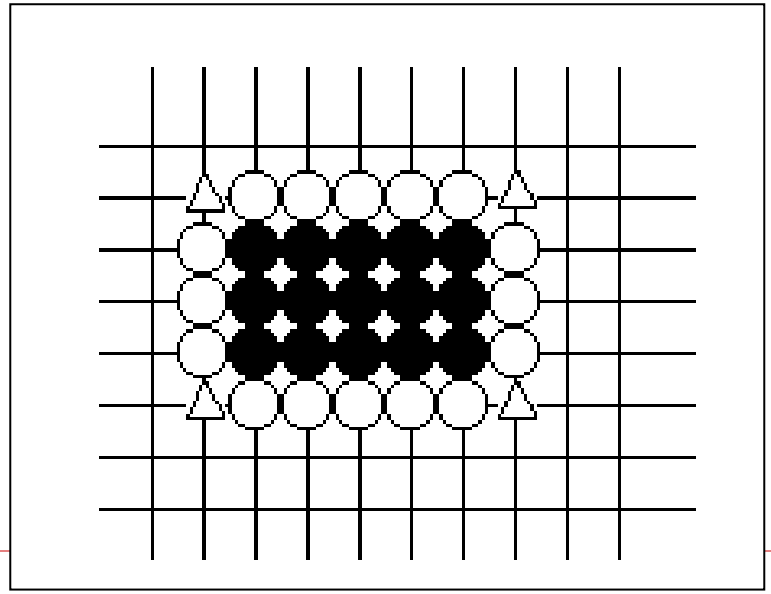


例子：PhotoShop

4连通要求比8连通高

区域填充(种子填充法)

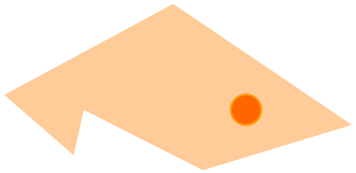
- 4连通与8连通区域的区别
  - 4连通也可看作8连通区域，但它作为4连通区域时，边界只要是8连通的就可以了；而作为8连通区域时，边界必须是4连通。
  - 右图黑点区域：
    - 看作4连通区域时，边界可不包含以△表示的象素。
    - 看作8连通区域时，边界必须包含以△表示的象素，否则会出现填充泄漏。



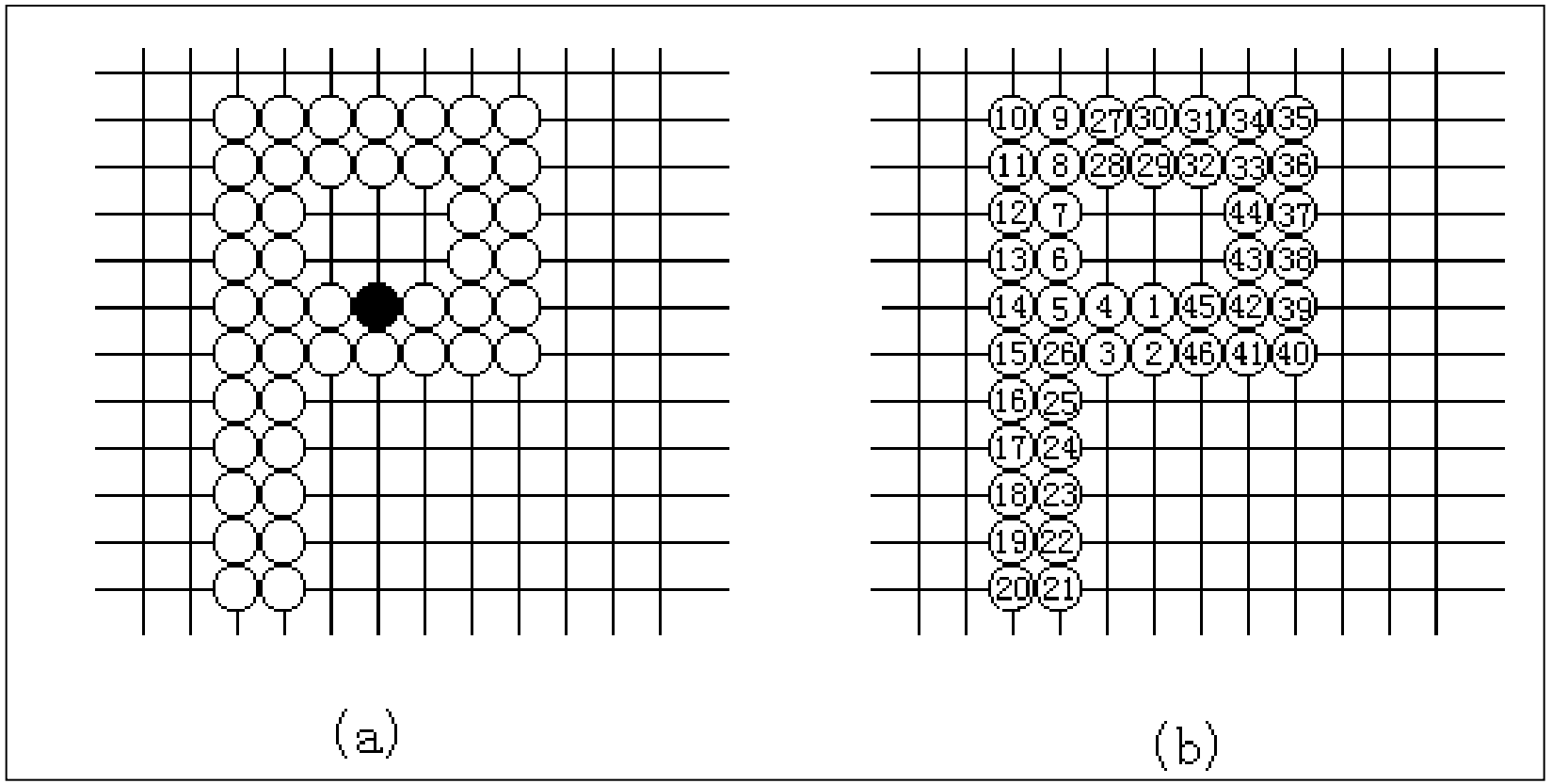
递归填充算法

- 内点表示的4连通区域(x,y种子点，画内点)

```
void FloodFill4(int x,int y,int oldColor,int newColor)
{                                        //区域原来填充着旧色
   if(GetPixel(x,y) == oldColor) //检测当前点，如颜色为旧色
   {    PutPixel(x,y,newColor);   //上新色
        FloodFill4(x,y+1,oldColor,newColor);   //检测上点
        FloodFill4(x,y-1,oldColor,newColor);   //检测下点
        FloodFill4(x-1,y,oldColor,newColor);   //检测左点
        FloodFill4(x+1,y,oldColor,newColor);   //检测右点
   }
}
```

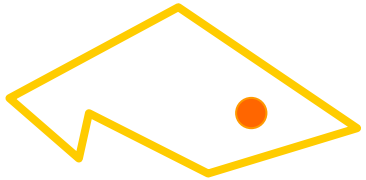


种子填充法流程


10 9 27 30 31 34 35
11 8 28 29 32 33 36
12 7 44 37
13 6 43 38
14 5 4 1 45 42 39
15 26 3 2 46 41 40
16 25
17 24
18 23
19 22
20 21
(a)
(b)

边界表示的4连通区域

```
void BoundaryFill4(int x,int y,int boundaryColor,int newColor)
{    //x,y种子点
     int color;
      color = GetPixel(x,y);
      if((color != boundaryColor) && (color != newColor))
      {                    // 前者说明已画到边界，后者说明该点已画过
          PutPixel(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);
      }
}
```



其他

- 问题：
  - 内点表示与边界表示的8连通区域的填充算法?
- 区域填充的优缺点：
  - 递归执行，算法简单，
  - 效率不高，区域内每一象素都引起一次递归，进/出栈，费时费内存。
  - 有些象素会入栈多次，降低算法效率
  - 栈结构占用额外空间。
- 方向：改进算法，减少递归次数，提高效率。
  - 方法之一：使用扫描线区域填充算法；

区域填充（扫描线算法)

- 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
  - 基本过程：

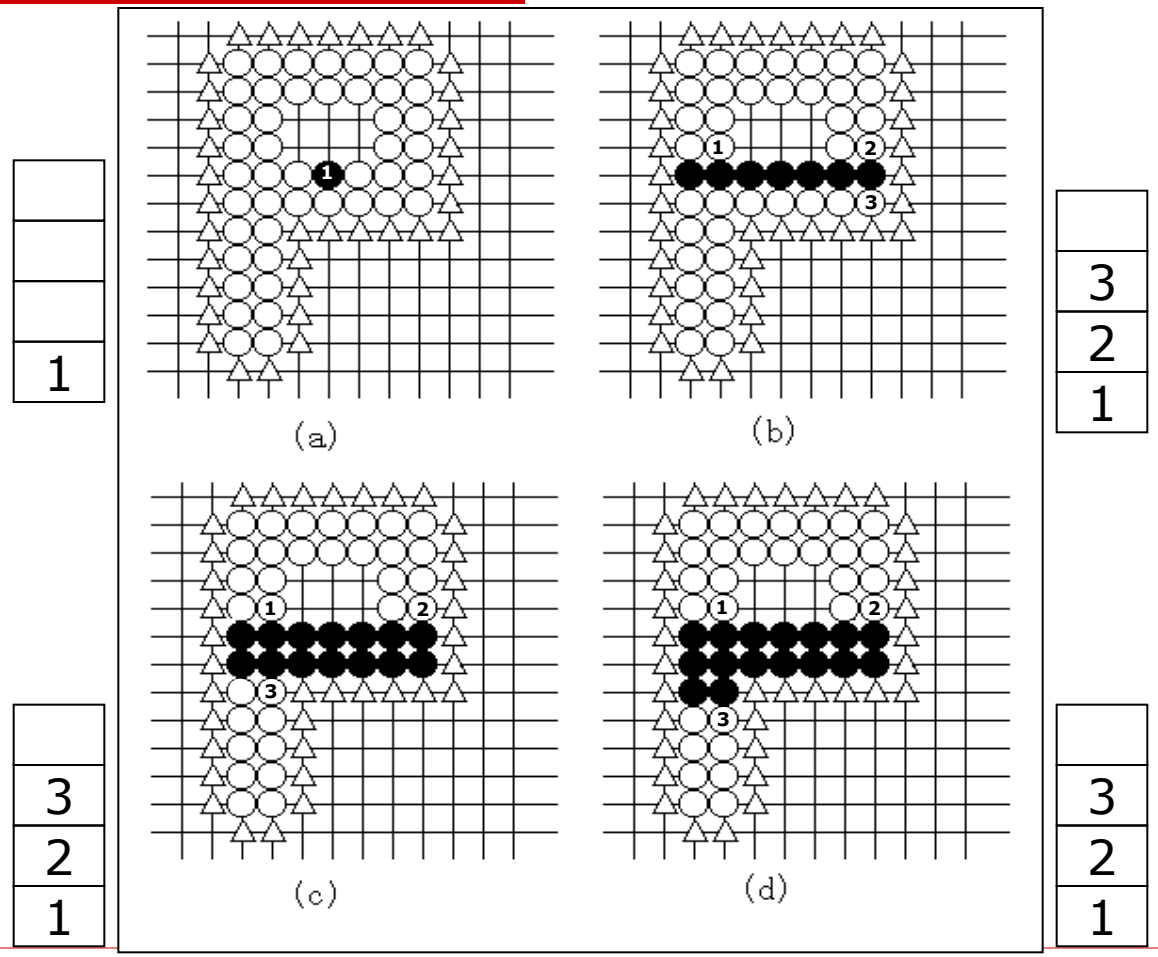
    当给定种子点(x,y)时，首先填充种子点所在的扫描线上的位于给定区域的一个区段，然后确定与这一区段相通的上下两条扫描线上位于给定区域内的区段，并依次保存下来。反复这个过程，直到填充结束。

算法步骤

- 区域填充的扫描线算法可由下列四个步骤实现：

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=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. 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。

区域填充的扫描线算法的流程



像素中的序号标指它所在区段位于堆栈中的位置

```
typedef struct{                          //记录种子点
   int x;
   int y;
} Seed;

void ScanLineFill4(int x,int y,int oldcolor,int newcolor)
{
      int xl, xr, i;
      bool spanNeedFill;
      Seed pt;
      setstackempty();          //设栈空
      pt.x =x; pt.y=y;
      stackpush(pt);            //将种子点压入堆栈
```

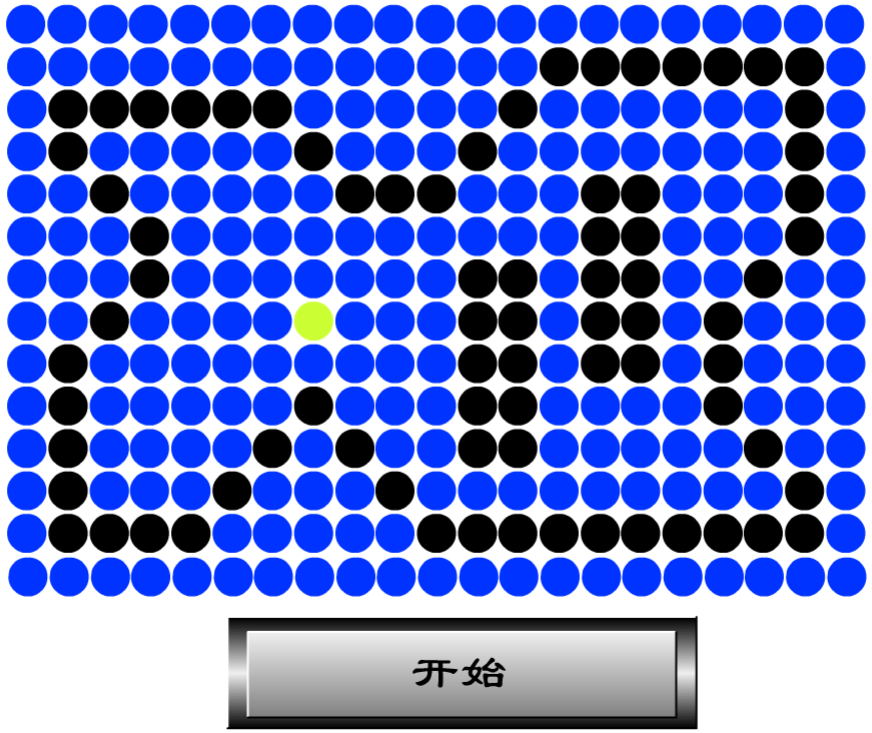
```
while(!isstackempty())          //检查堆栈状态，空返回F，否则返回T
{   pt = stackpop();            //取堆栈顶元素
     y=pt.y;     x=pt.x;
    while(getpixel(x,y)==oldcolor)     //向右填充
   {  Putpixel(x,y,newcolor);
        x++;
    }
    xr = x-1;                          //记下xr
    x = pt.x-1;
    while(getpixel(x,y)==oldcolor)     //向左填充
   {  Putpixel(x,y,newcolor);
      x--;
    }
   xl = x+1;                           //记下xl
```

```
x = xl;      y = y+1;                    //处理上面一条扫描线
while(x<xr)                              //在上线往右扫描
{  spanNeedFill=FALSE;
   while(getpixel(x,y)==oldcolor)  //到需填充部分的最右点
   { spanNeedFill=TRUE;
      x++;
   }
   if(spanNeedFill)                      //最右点压栈
   { pt.x=x-1;pt.y=y;
     stackpush(pt);
     spanNeedFill=FALSE;
    }
   while(getpixel(x,y)!=oldcolor && x<xr) x++;
}
```

```
    //处理下面一条扫描线，代码与处理上面一条扫描线类似
    x = xl;
    y = y-2;
    while(x<xr)
    {  ....
     }//End of while(i<xr)

}//End of while(!isstackempty())
```

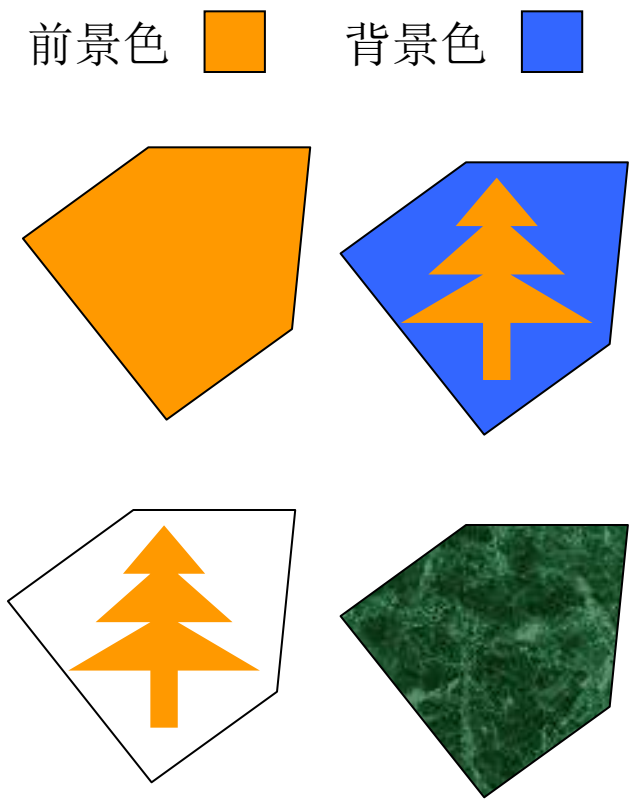
动画演示：区域填充的扫描线算法



- 上述算法非递归，对于每一个待填充区段，只需压栈一次。
- 而在递归算法中，每个象素都需要压栈。因此，扫描线填充算法提高了区域填充的效率。

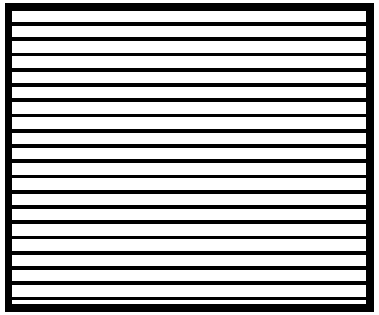
3.5 以图像填充区域

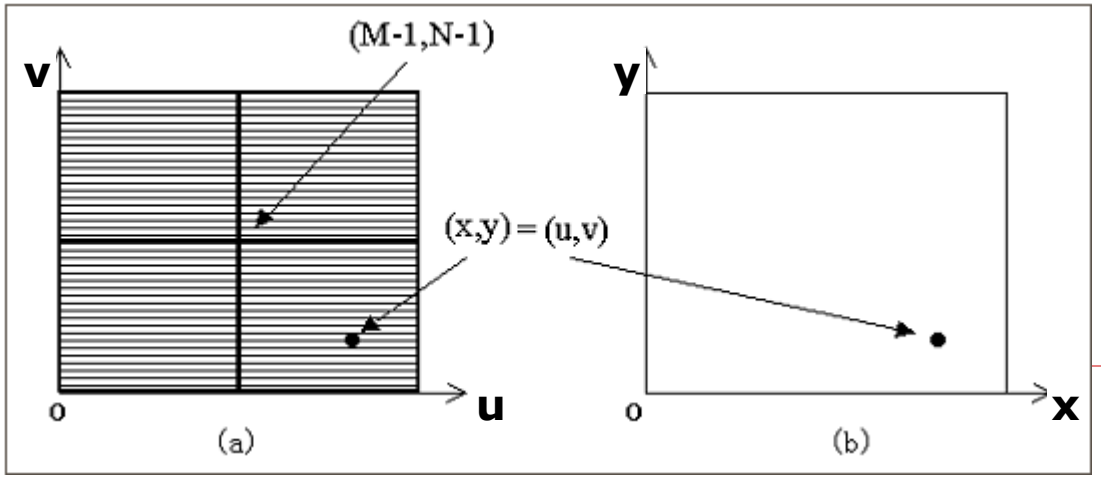
- 填充图元的四种方式：
  - 均匀着色方式： 将图元内部像素置成同一颜色
  - 位图不透明方式： 若像素对应的位图单元为1，则以前景色显示该像素；若为0，则以背景色显示该像素；
  - 位图透明方式： 若像素对应的位图单元为1，则以前景色显示该像素；若为0，则不做任何处理。
  - 像素图填充方式： 以像素对应的像素图单元的颜色值显示该像素。



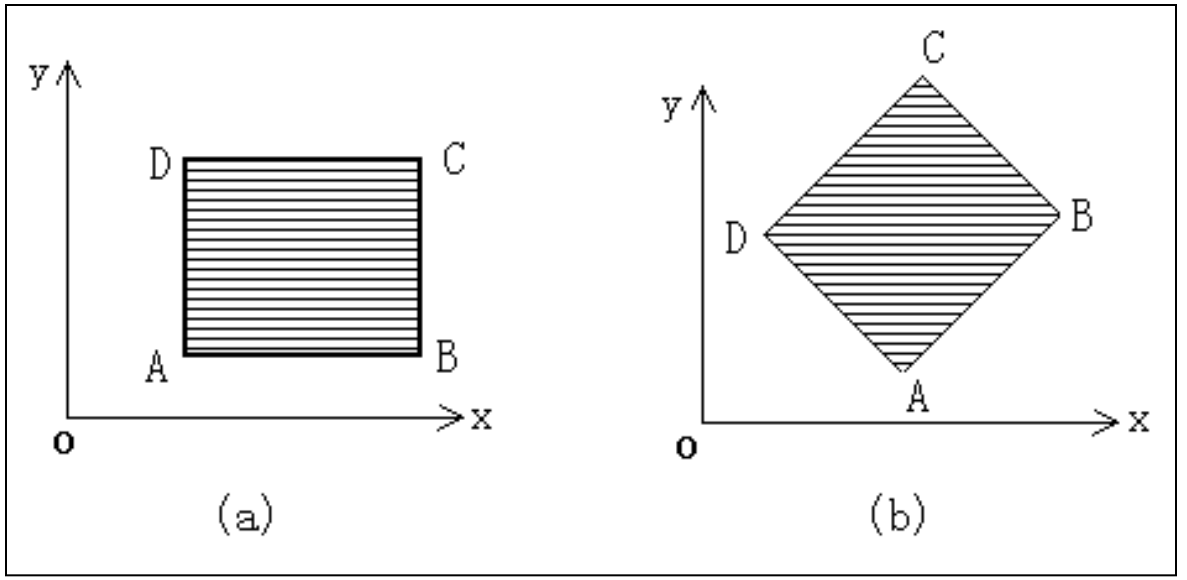
以图像填充区域

- 关键：建立区域与图像间的对应关系
  - 方法1：建立整个绘图空间与图像空间的1-1映射。
    - 图像p尺寸为M×N，将其瓦块式排列得图像空间(u,v)。
    - 绘图空间(x,y)与图像空间(u,v)的1-1映射x=u, y=v，则(x,y)的颜色值为p[x%M][y%N]。

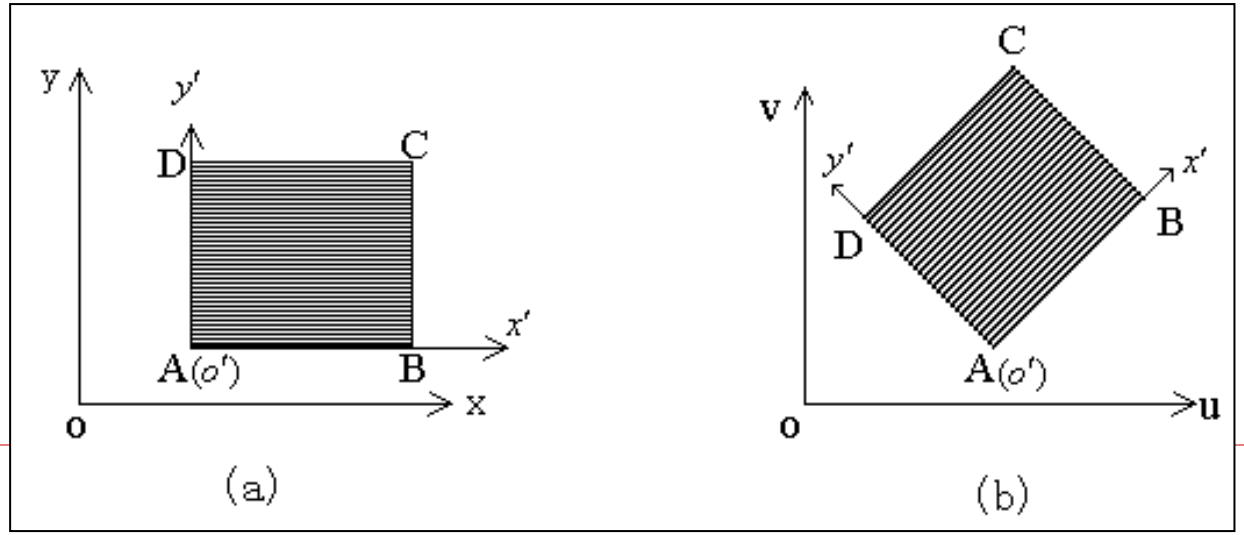




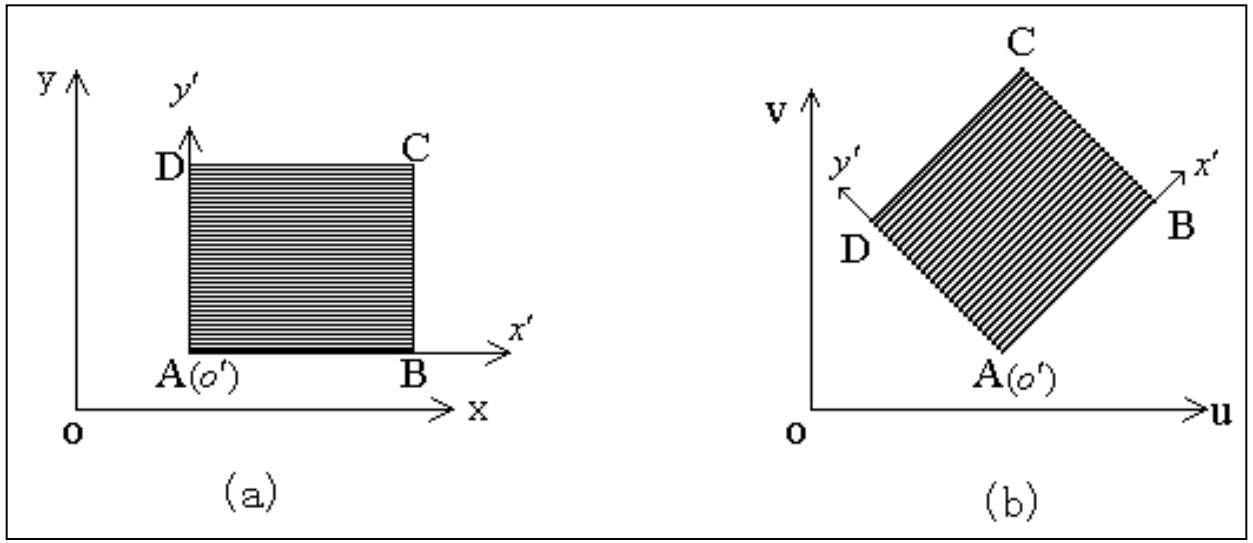
- 特点：区域运动时，其内部的图像不动。
- 适合于动画漫游，如透过车窗看外面景物。



- 方法2：建立区域局部坐标系和图像空间的1-1映射
  - 以矩形角点A为原点建立局部坐标系，其坐标(x',y')与图像空间(u,v)的1-1映射为x'=u,y'=v.
  - 对区域内像素(x,y)，先求局部坐标，再与该像素对应的颜色pattern[x'%M][y'%N]显示。



- 特点：区域运动时，局部坐标系跟着动，内部图像也动。
- 适用：图像作为区域表面属性的情况，例如桌面与其上的木纹。



3.6 字符的表示与输出

- 点阵字符
  - 每个字符由一个位图表示，并把它用一个称为字符掩膜的矩阵来表示
  - 点阵字符的显示分为两步：首先从字库中将它的位图检索出来，然后将检索到的位图写到帧缓冲器中。
  - 在实际应用中，同一个字符有多种字体（如宋体、楷体等），每种字体又有多种大小型号，因此字库的存储空间十分庞大。为了减少存储空间，一般采用压缩技术。

矢量字符

- 矢量字符记录字符的笔画信息而不是整个位图，具有存储空间小，美观、变换方便等优点。
- 对于字符的旋转、放大、缩小等几何变换，而矢量字符则只需要对其几何图素进行变换就可以了。
- 矢量字符的显示也分为两步。首先从字库中将它的字符信息，然后取出端点坐标，对其进行适当的几何变换，再显示字符。
- 轮廓字形法是当今国际上最流行的一种字符表示方法，其压缩比大，且能保证字符质量。

END