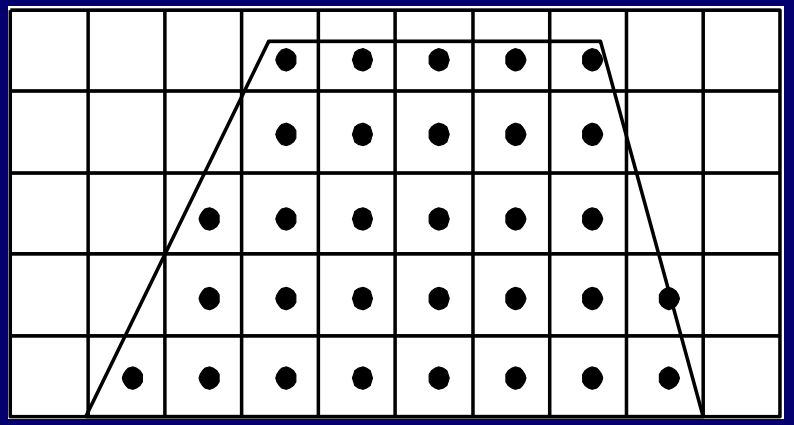
第六章　图形填充算法

王长波　教授

问题提出

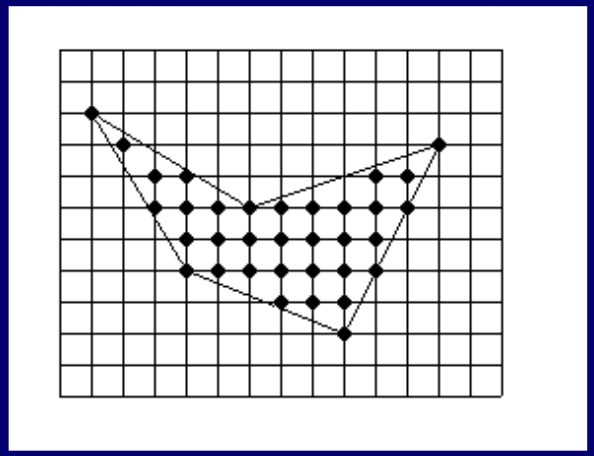
- 提出问题
  - 多边形顶点表示
  - 多边形点阵表示



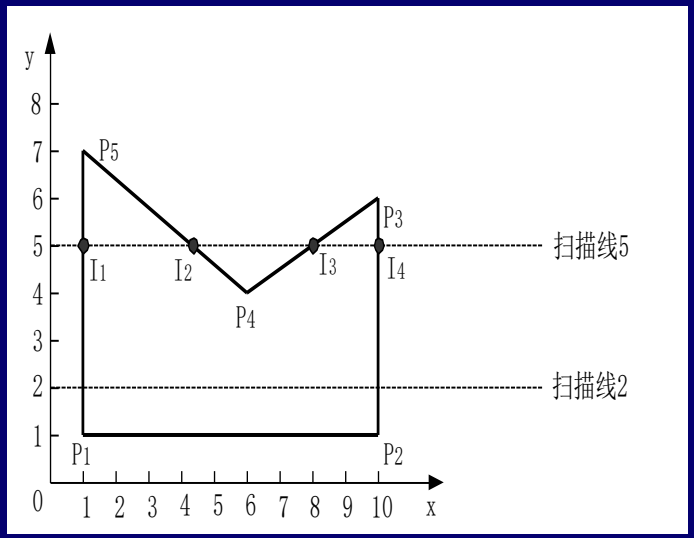
- **多边形填充**：将多边形的顶点表示转换成点阵模式
  - 从多边形的给定边界出发，求出位于其内部的各个像素，并将帧缓冲器内的各个对应元素设置响应的灰度和颜色。

6.1 多边形填充算法

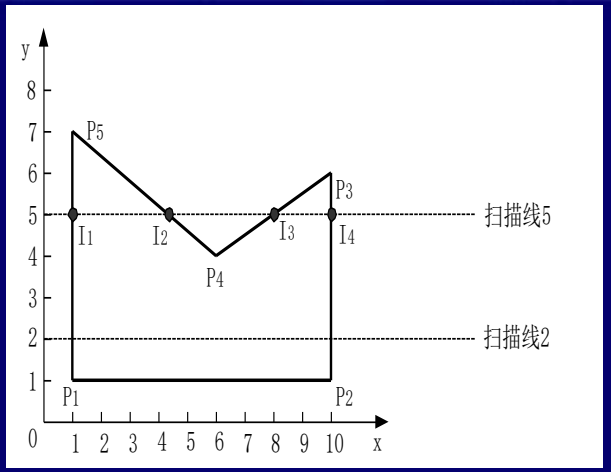
- 区域连贯性



- 对于一条扫描线：
  - 求交
  - 排序
  - 交点配对
  - 区间调色



6.1多边形填充算法



算法过程：

对于每一条扫描线的处理 ：

1）求交点：首先求出扫描线与多边形各边的交点；

2）交点排序：将这些交点按X坐标递增顺序排序；

3）交点匹配：即从左到右确定落在多边形内部的那些线段；

4）区间填充：填充落在多边形内部的线段。

6.1多边形填充算法

(1) 求交点的方法

- **最简单的办法：**将多边形的所有边放在一个表中，在处理每条扫描线时，从表中顺序取出所有的边，分别求这些边与扫描线的交点。
- **不使用该方法的原因：**将做一些无益的求交点动作，因为扫描线并不一定与多边形的边相交，扫描线只与部分甚至较少的边相交；因此，在进行扫描线与多边形边求交点时， 应只求那些与扫描线相交的边的交点。
- **确定与扫描线相交的边：**用边表来确定哪些边是下一条扫描线求交计算时应该加入运算的。

6.1多边形填充算法

(1)求交点的方法

当一条扫描线y_i 与多边形的某一边线有交点时，其相邻扫描线y_{i+1}一般也与该边线相交（除非y_{i+1}超出了该边线所在区间），而且扫描线y_{i+1}与该边线的交点，很容易从前一条扫描线y_i与该边的交点递推求得。

设某条直线边的方程为　　$y=mx+b$

该边两个端点坐标为（x_1, y_1）、（x_2, y_2）

且$x_1 \neq x_2$　则该边斜率为$m=(y_2-y_1)/(x_2-x_1)$

令扫描线y_i与该边交点为（x_i，y_i）

扫描线y_{i+1}与该边交点（x_{i+1}，y_{i+1}）

6.1多边形填充算法

(1)求交点的方法

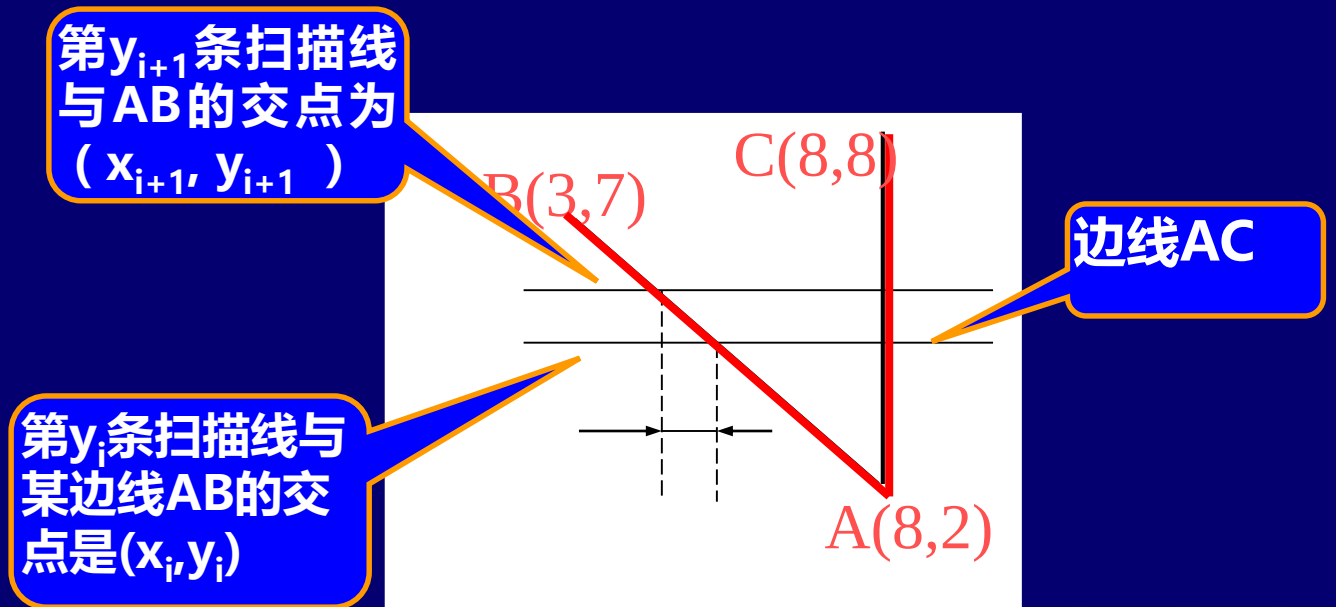
$$\because\ y_i = mx_i + b \qquad (1)$$

$$y_{i+1} = y_i + 1 = mx_{i+1} + b \qquad (2)$$

$$\therefore\ x_{i+1} = (y_{i+1} - b)/m = (y_i + 1 - b)/m$$

$$= (mx_i + b + 1 - b)/m = x_i + 1/m$$

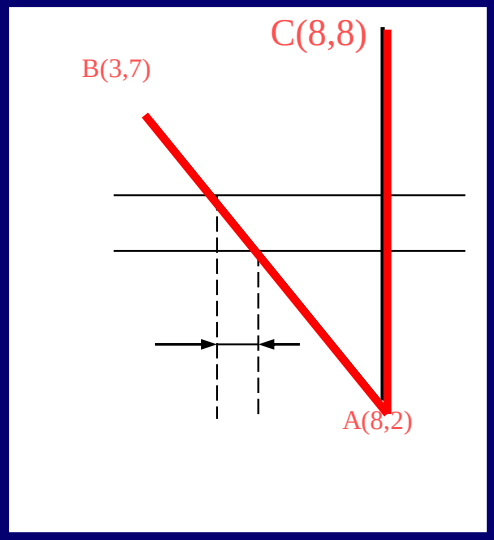
则第y_{i+1}条扫描线与AB的交点为$y_{i+1}=y_i+1$，$x_{i+1}=x_i+1/m$ 即由前一条扫描线交点X_i，Y_i求下一条扫描线交点X_{i+1}，Y_{i+1},式中，m是这条边的斜率。

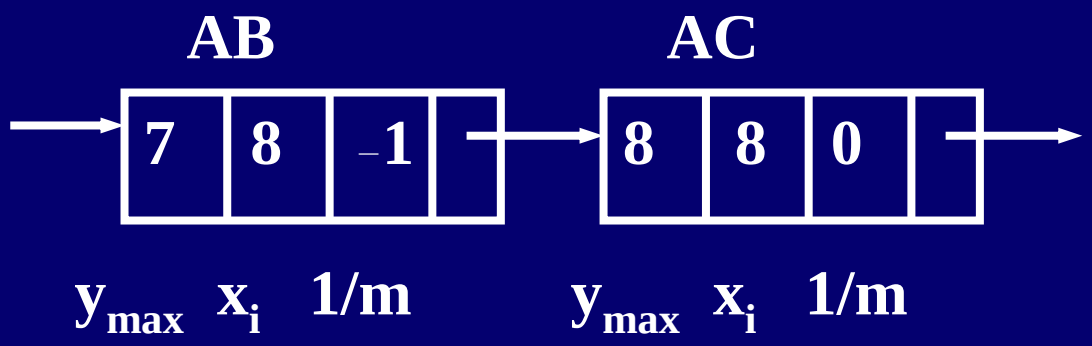


6.1多边形填充算法

(2)边记录

利用边的这种相关性，不必算出边线与各条扫描线的全部交点，只需以边线为单位，对每条边建立一个边记录，其内容包括：该边y的最大值y_{max}，该边底端的x坐标x_i，从一条扫描线到下一条扫描线间的x增量1/m，以及指示下一个边记录地址的指针

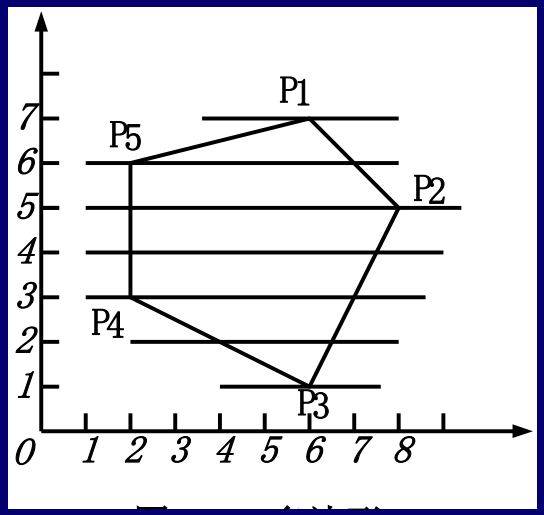


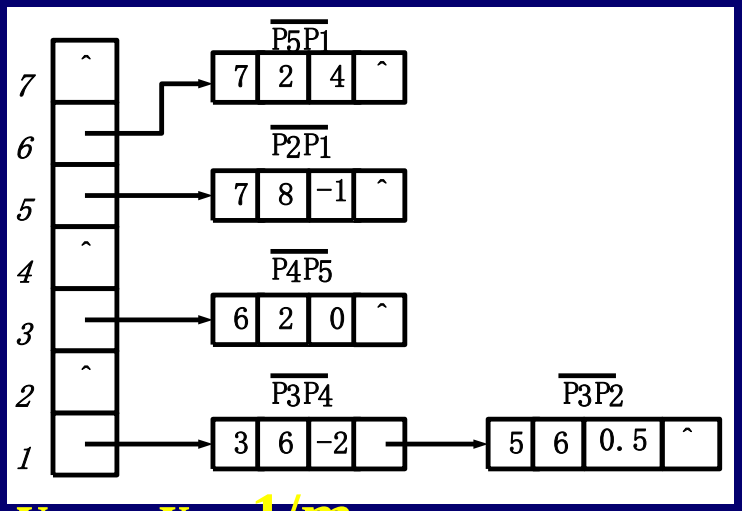


6.1多边形填充算法

（3）边表ET（Edge Table）

边表是一个包含多边形全部边记录的表，它按y坐标（与扫描线一一对应）递增（或递减）的顺序存放区域边界的所有边。每个y坐标值存放一个或者说几个边记录。当某条扫描线y_i碰到多边形边界的新边时（**以边线底端为准**），则在ET表中相应的y坐标值处写入一个边记录。当同时有多条边进入时，则在ET表中按链表结构写入相应数目的多个记录，**这些记录是按边线较低端点的x值增加的顺序排列**。当没有新边加入时，表中对应的y坐标值处存储内容为空白。

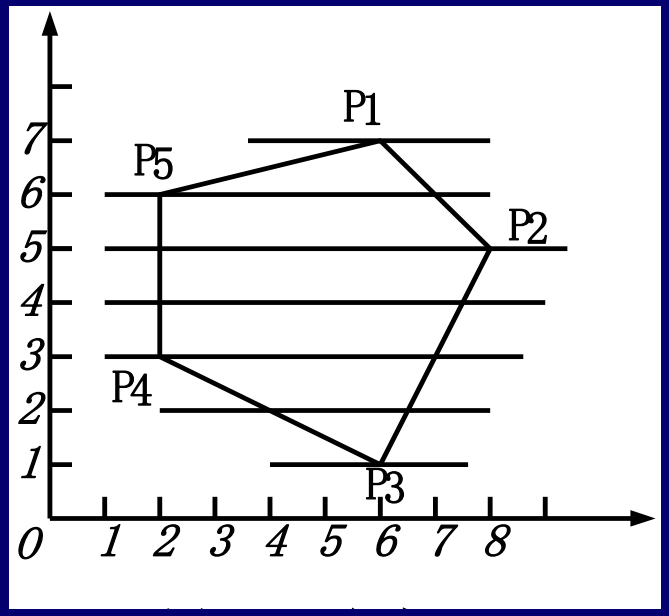


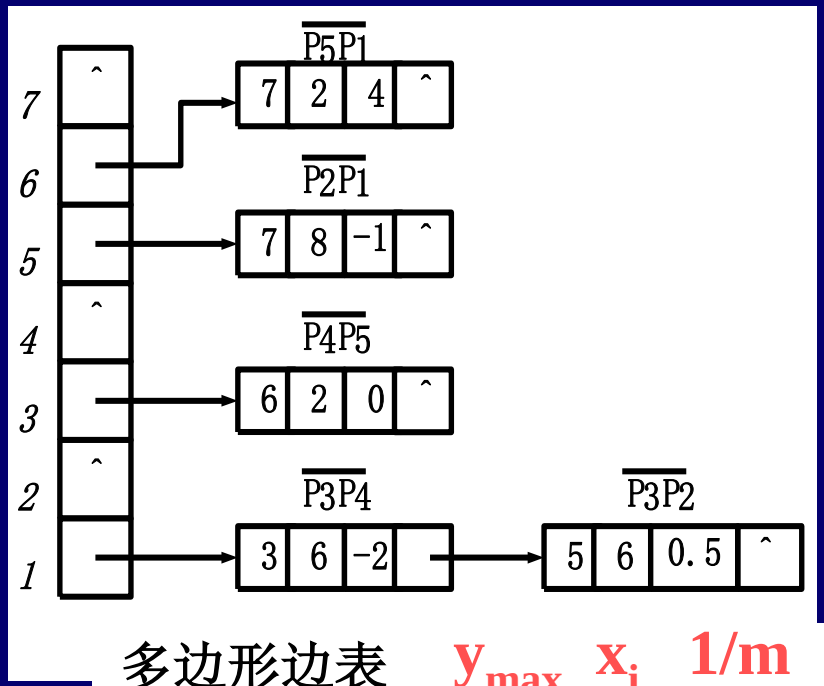


y_{max} x_i $1/m$

6.1多边形填充算法

注意：在ET表中：①与x轴平行的边不计入。②多边形的顶点分为两大类：一类是局部极值点，如下图中的P_1、P_3；另一类是非极值点，如下图中的P_2、P_4、P_5。当扫描线与第一类顶点相遇时，应看作两个点；当扫描线与第二类顶点相遇时，应视为一个点。





6.1多边形填充算法

(4) 活动边表AET(Activities Edge Table)

活动边表AET是一个只与当前扫描线相交的边记录链表。随着扫描线从一条到另一条的转换，AET表也应随之变动，利用 $y_{i+1}=y_i+1$， $x_{i+1}=x_i+1/m$ 可以算出AET表中x域中的新值x_i。

凡是与这一条扫描线相交的任何新边都加到AET表中，而与之不相交的边又被从AET表中删除掉了。下图列出了多边形图在扫描线为4、5、6时的AET表。AET表中的记录顺序仍是按x增大排序的。

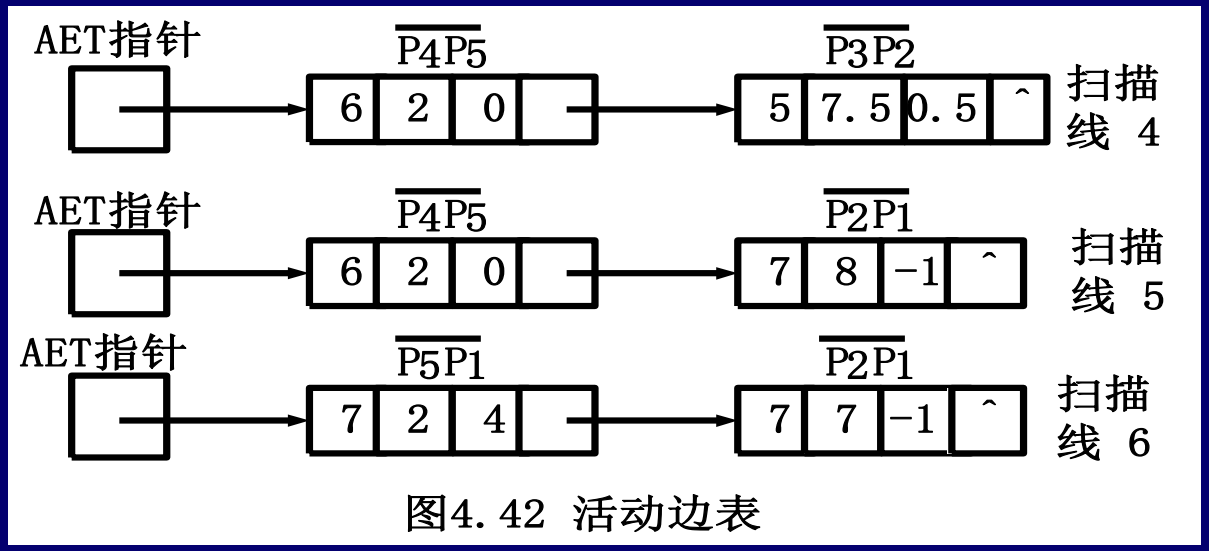


图4.42 活动边表

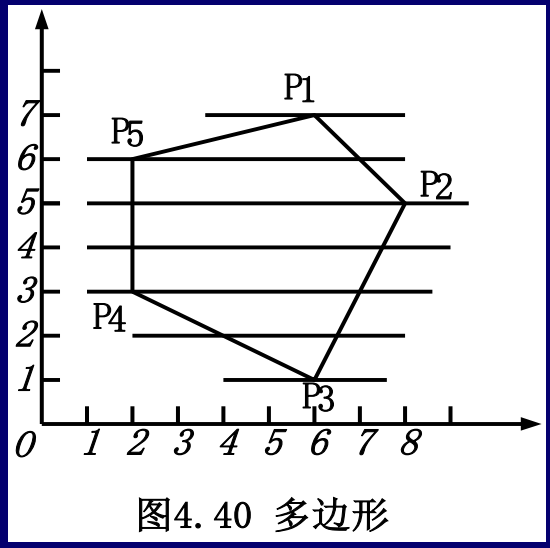


图4.40 多边形

6.1多边形填充算法

多边形区域填充算法过程

（1）根据给出的顶点坐标数据，按y递增顺序建立ET表

（2）根据AET指针，使之为空。

（3）使$y=Y_{min}$ （Y_{min}为顶点坐标中最小y值）。

（4）反复做下述各步，直至$y=Y_{max}$（顶点坐标中y的最大值）或ET与AET为空：

①将ET表加入到AET中，并保持AET链中的记录按x值增大排序。

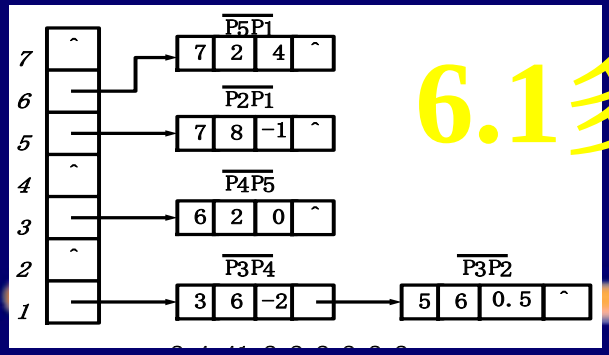
②对扫描线y_i依次成对取出AET中x_i值，并在每对x_i之间填上所要求的颜色或图案。

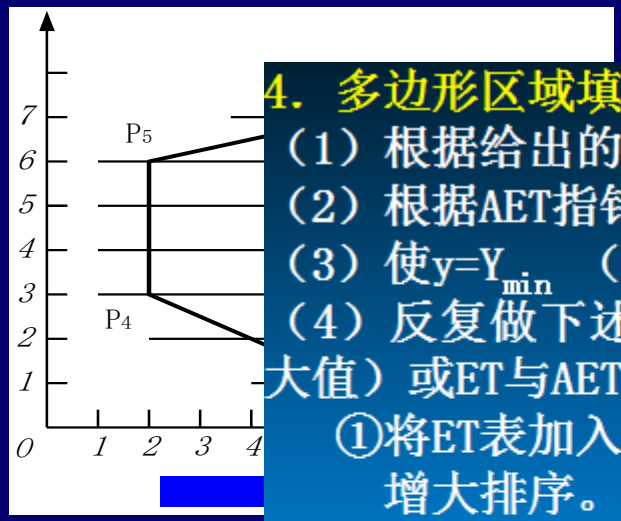
③从AET表中删去$y_i=y_{max}$的记录。

④对保留下来的AET中的每个记录，用x_i+1/m代替x_i，并重新按x递增排序。

⑤使y_i+1，以便进入下一轮循环。

6.1多边形填充算法





①开始y＝1，将ET表中y＝1结点加入至AET表，同

6 0.5 ^

以中

③上例由

就得将第

④对保留

如上例变

x递增排序。

AET

4. 多边形区域填充算法过程

（1）根据给出的顶点坐标数据，按y递增顺序建立ET表

（2）根据AET指针，使之为空。

（3）使$y=Y_{min}$ （Y_{min}为顶点坐标中最小y值）。

（4）反复做下述各步，直至$y=Y_{max}$（顶点坐标中y的最大值）或ET与AET为空：

①将ET表加入到AET中，并保持AET链中的记录按x值增大排序。

②对扫描线y_i依次成对取出AET中x_i值，并在每对x_i之间填上所要求的颜色或图案。

③从AET表中删去$y_i=y_{max}$的记录。

④对保留下来的AET中的每个记录，用x_i+1/m代替x_i，并重新按x递增排序。

⑤使y_i+1，以便进入下一轮循环。

6.1多边形填充算法

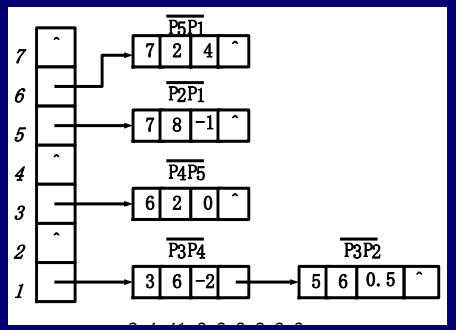
⑤使y_i+1，以便进入下一轮循环。即y＝2再进入以上循环

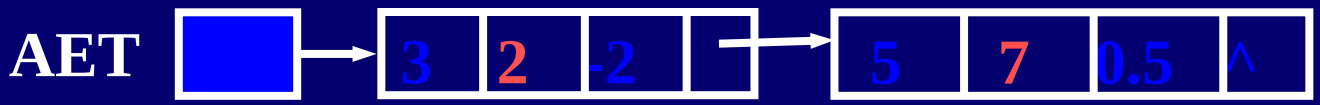
继续：①由y＝2，ET表中是空，所以不需ET表加入AET表

②取x＝4和x＝6.5,将4—6.5之间填上像素颜色。

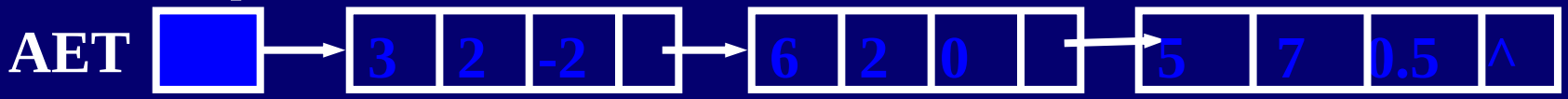
③由于y=2,不必删去结点。

④再改变x_i的值为



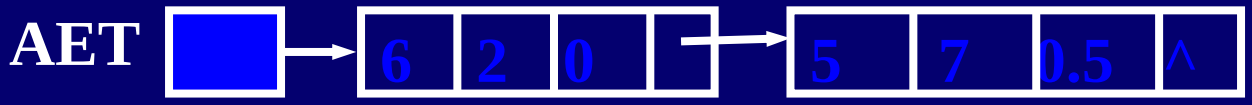


⑤使y_i ＝3，重复继续。继续：①由y＝3，将ET表中y＝3结点加入，即



②将2—7之间填上像素颜色。

③删去结点Y_{max}＝3 结点。



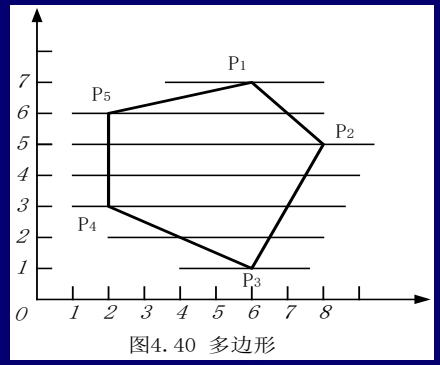
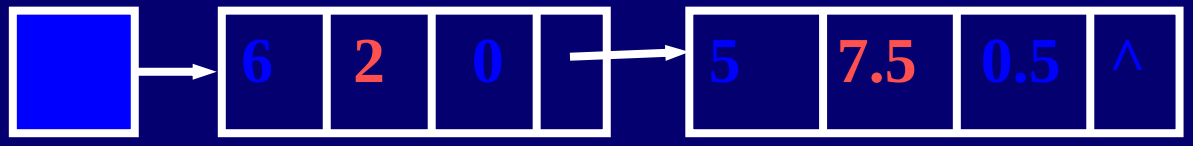


图4.40 多边形

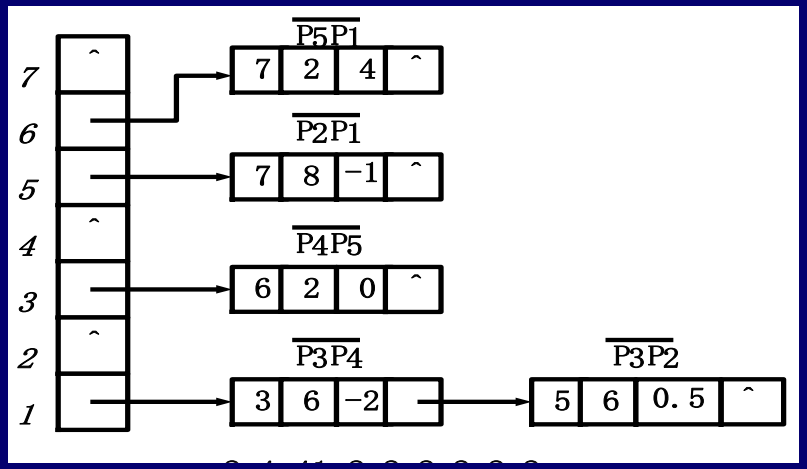
6.1多边形填充算法

④再改变x_i的值为



AET

⑤使y_i ＝4，重复继续。



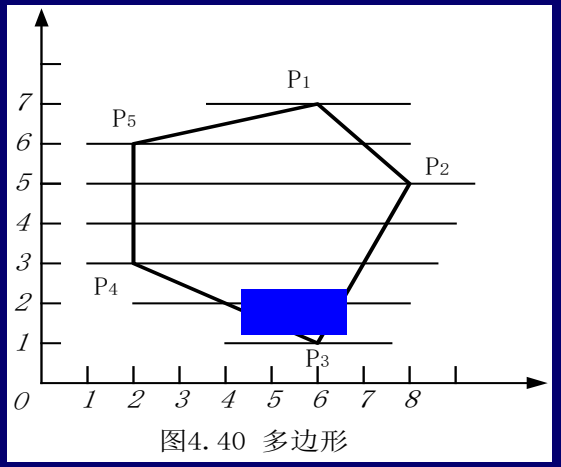


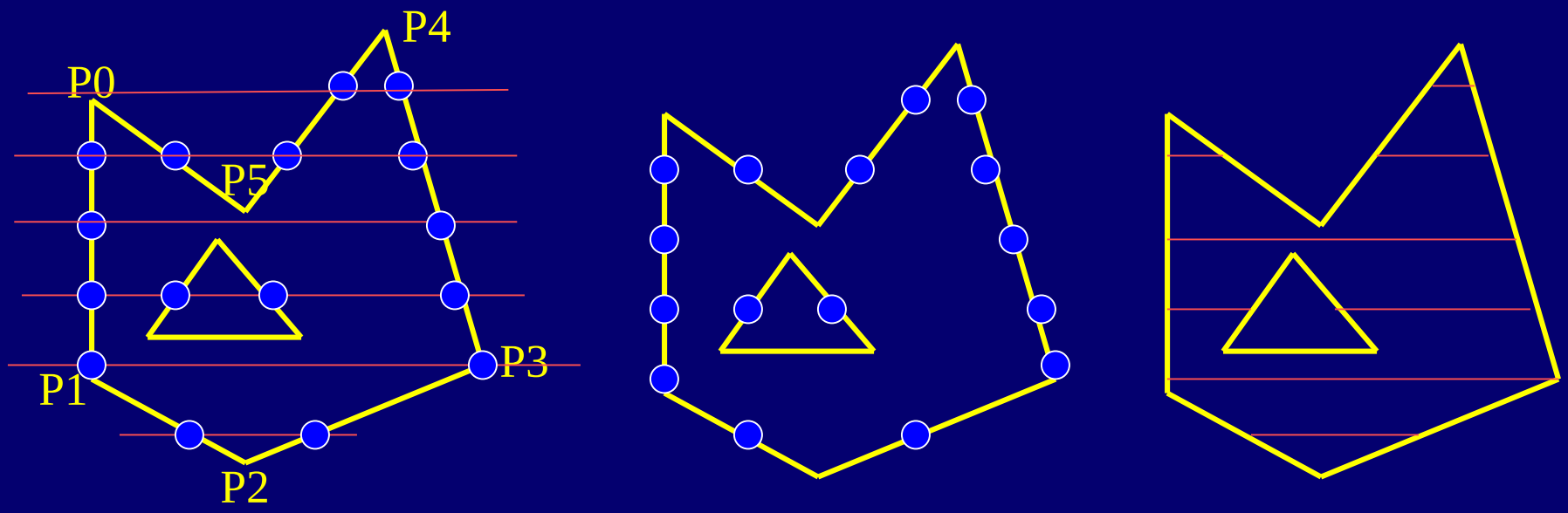
图4.40 多边形

6.1多边形填充算法

```
Polygonfill(polydef,color)
Int color
多边形定义 polydef
{    for(各条扫描线I)
     {    初始化新边表表头指针ET[I];
          把ymin=I的边放进边表ET[I];
      }
      y=最低扫描线号;
      初始化活化边表AEL为空;
      for(各条扫描线I)
      {    把新边表ET[I]中的边结点用插入排序法插入AEL表,使
           之按x递增顺序排列;
           遍历AET表,把配对交点之间的区间上的各像素(x,y)用待填颜色改写
           遍历AET表,把ymax=I的结点从AEL中删除,并把ymax>I的结点的
           x递增dx;
           若允许多边形的边自交,则用冒泡排序法对AEL表重新排序;
      }
}
```

6.1多边形填充算法

举例



6.1多边形填充算法

多边形填充算法特点：

（1）每个像素只访问一次，充分利用了扫描线、多边形边的连续性,避免了反复求交点的运算,是一种较快的填充方法

（2）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

6.2 种子填充算法

边填充

边填充算法的基本原理是：

（1）对多边形的每条边进行直线扫描转换，即对多边形边界经过的像素打上边标志；

（2）对多边形内部进行填充。填充时，对每条扫描线，依从左到右的顺序，逐个访问扫描线上的像素，用一个布尔量来标志当前点是在多边形内部还是外部（一开始设布尔量的值为假，当碰到设有边标志的点时，就把其值取反；对没有边标志的点，则其值保持不变）

（3）将其布尔量值为“真”的内部置为图形色，把其布尔量的值为“假”的外部点置为底色即可。

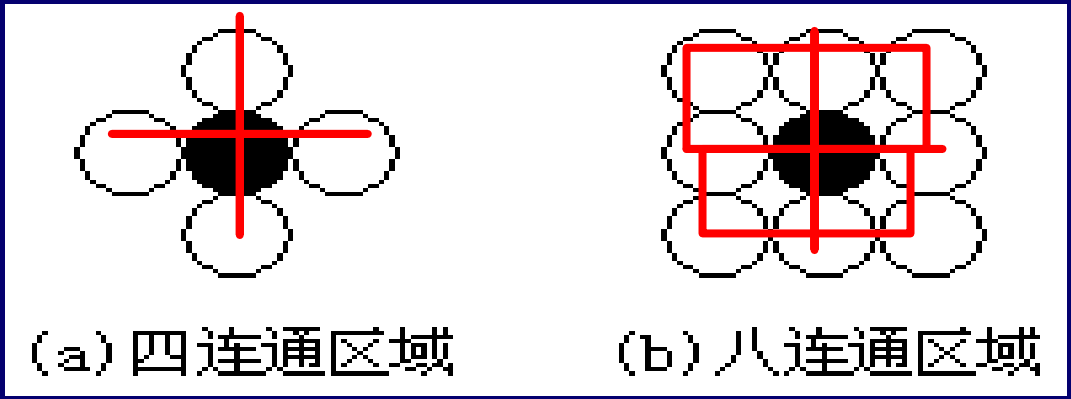
6.2 种子填充算法

（1）种子填充基本思路

首先假设在多边形区域的内部，至少有一个像素点（称为种子）是已知的，然后算法开始搜索与种子点相邻且位于区域内的其它像素。如果相邻点不在区域内，那么到达区域的边界；如果相邻点位于区域内，那么这一点就成为新的种子点，然后继续递归地搜索下去。

区域的连通情况可以分为四连通和八连通两种

四连通区域：各像素在水平和垂直四个方向上是连通的.八连通区域：各像素在水平、垂直以及四个对角线方向上都是连通的。



6.2 种子填充算法

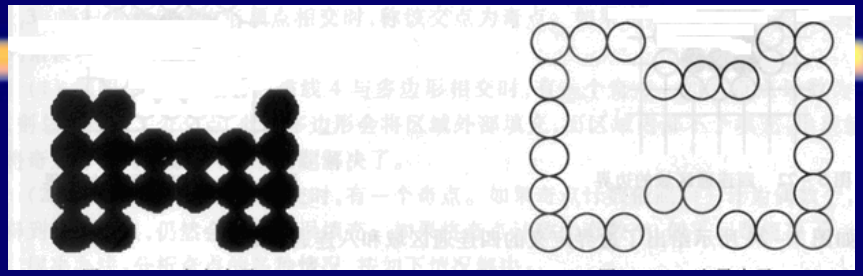
（1）种子填充基本思路

在种子填充算法中，如果允许从四个方向搜寻下一个像素点，则该算法称为四向算法；如果允许从八个方法搜寻下一个像素点，则该算法称为八向算法。

一个八向算法可以用在四连通区域的填充上，也可用在八连通区域的填充上；而一个四向算法只能用于填充四连通区域。

无论是四向算法还是八向算法，它们的填充算法基本思想是相同的。为简单起见，下面只讨论四向种子填充算法。

（1）种子填充基本思路



基本思想：假设在多边形区域内部至少有一个像素是已知的（此像素称为种子像素），由此出发找到区域内所有其他像素，并对其进行填充。

由于区域可采用边界定义和内点定义两种方式，区域按连通性又可分为四连通区域和八连通区域两类，所以常用的种子填充算法有：

- **边界表示的四连通区域种子填充算法**
- **内点表示的四连通区域种子填充算法**
- **边界表示的八连通区域种子填充算法**
- **内点表示的八连通区域种子填充算法**

6.2 种子填充算法

（2）边界表示的四连通区域种子填充算法

基本思想：从多边形内部任一点（像素）出发，依“左上右下”顺序判断相邻像素，若其不是边界像素且没有被填充过，对其填充，并重复上述过程，直到所有像素填充完毕。

可以使用栈结构来实现该算法，算法的执行步骤如下：

种子像素入栈，当栈非空时，重复执行如下三步操作：

1）栈顶像素出栈；

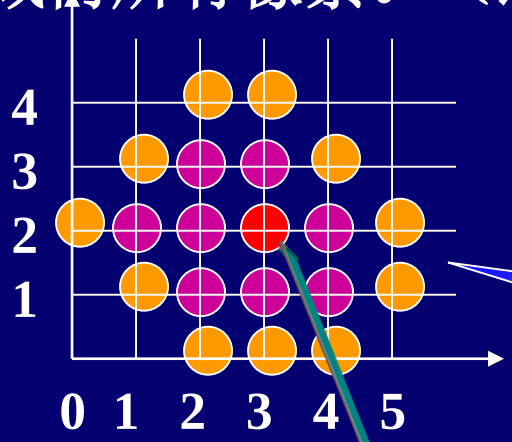
2）将出栈像素置成多边形填充的颜色；

3）按左、上、右、下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。

6.2 种子填充算法

边界填充算法（四连通域）

在区域有一个像素是已知（种子像素），由此像素从四个方向遍历区域内所有像素。（适用于交互绘图）



用什么方法实现？

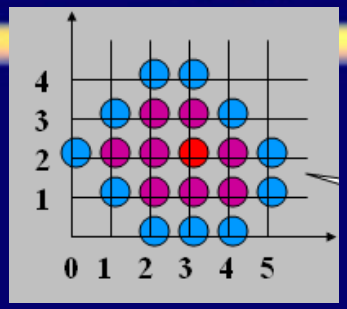
注意：栈中种子的次序，当前栈顶元素是哪个？

依“左上右下”

6.2 种子填充算法

以边界表示的四连通区域种子填充算法 (基于递归)

```
Void BoundaryFill4(int x,int y ,int boundarycolor,int newcolor)
    {
     if(getpixel(x,y)!= boundarycolor && getpixel(x,y)!= newcolor)
      {                          /* getpixel(x,y)取屏幕上像素(x,y)的颜色*/
        putpixel(x,y,newcolor);//着色
        BoundaryFill4(x-1,y, boundarycolor,  newcolor);
        BoundaryFill4(x,y+1, boundarycolor,  newcolor);
        BoundaryFill4(x+1,y, boundarycolor,  newcolor);
        BoundaryFill4(x,y-1, boundarycolor,  newcolor);
      }
}
```

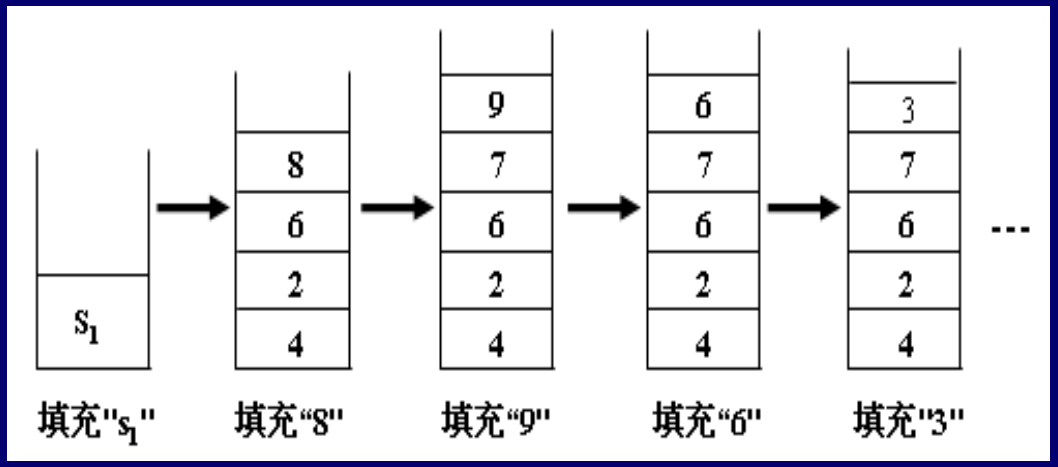


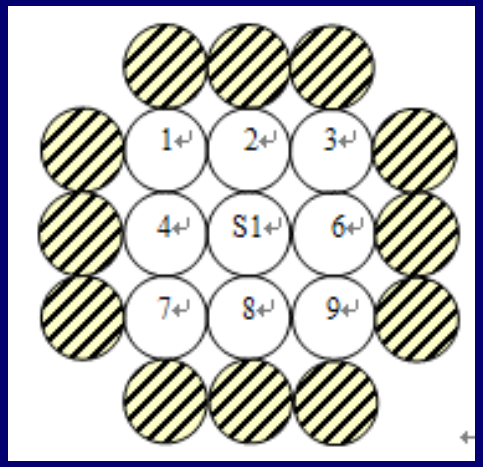
取(x,y)为种子点

基本思想：从多边形内部任一点（像素）出发，依“左上右下”顺序判断相邻像素，若其不是边界像素且没有被填充过，对其填充，并重复上述过程，直到所有像素填充完毕。

6.2 种子填充算法

- 例：填充如下区域，堆栈的变化如图所示。





6.2 种子填充算法

(3)内点表示的四连通区域种子填充算法

基本思想：从多边形内部任一点（像素）出发，依“左上右下”顺序判断相邻像素，如果是区域内的像素，则对其填充，并重复上述过程，直到所有像素填充完毕。它也常称为漫水法。

可以使用栈结构来实现该算法，算法的执行步骤如下：

种子像素入栈，当栈非空时，重复执行如下三步操作：

1）栈顶像素出栈；

2）将出栈像素置成多边形填充的颜色；

3）按左、上、右、下的顺序检查与出栈像素相邻的四个像素，若其中某个像素是区域内的像素，则把该像素入栈。

6.2 种子填充算法

内点表示的四连通区域种子填充算法(基于递归)

```
Void FloodFill4(int x,int y ,int oldcolor,int newcolor)
        {
          if(getpixel(x,y)==oldcolor)
          {                          /* getpixel(x,y)取屏幕上像素(x,y)的颜色,      oldcolor
                                                                          为区域的原色*/
            putpixel(x,y,newcolor);//着色
            FloodFill4(x-1,y, oldcolor,  newcolor);
            FloodFill4(x,y+1, oldcolor,
            FloodFill4(x+1,y, oldcolor,
            FloodFill4(x,y-1, oldcolor,
          }
        }
```

取(x,y)为种子点

基本思想：从多边形内部任一点（像素）出发，依“左上右下”顺序判断相邻像素，如果是区域内的像素，则对其填充，并重复上述过程，直到所有像素填充完毕

6.2 种子填充算法

种子填充算法特点：

(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。

方法之一使用扫描线填充算法；

6.2 种子填充算法

- 上述简单种子填充算法操作过程非常简单，却要进行深度的递归，这不仅要花费许多时间，降低了算法的效率，而且还要花费许多空间要构造堆栈结构。因此出现了改进的扫描线种子填充算法。

■ 扫描线种子填充算法

扫描线种子填充算法适用于边界定义的四连通区域。

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。

6.2 种子填充算法

- **借助于堆栈，算法可分为以下五步实现：**

(1) 初始化。将算法设置的堆栈置为空。将给定的种子（x, y）压入堆栈。

(2) 出栈:如果堆栈为空，算法结束。否则从包含种子像素的堆栈中取出栈顶元素（x, y）作为种子像素。

(3) 区间填充:沿当前扫描线对种子像素的左右像素进行填充（像素值为new_color），直至遇到边界像素为止，从而填满包含种子像素的区间。

(4) 定范围:以x_l和x_r分别表示步骤（3）区间内最左和最右的两个像素。

(5) 进栈:在$x_l \leqslant x \leqslant x_r$中，检查与当前扫描线相邻的上下两条扫描线是否全为边界像素(boundary_color)或者前面已经填充过的像素（new_color），是则转到步骤(2)，否则在$x_l \leqslant x \leqslant x_r$中把每一个区间的最右像素作为种子像素压入堆栈，再转到步骤（2）继续执行。

6.3 反走样

- 用离散量表示连续量引起的失真现象称之为**走样**(aliasing)。

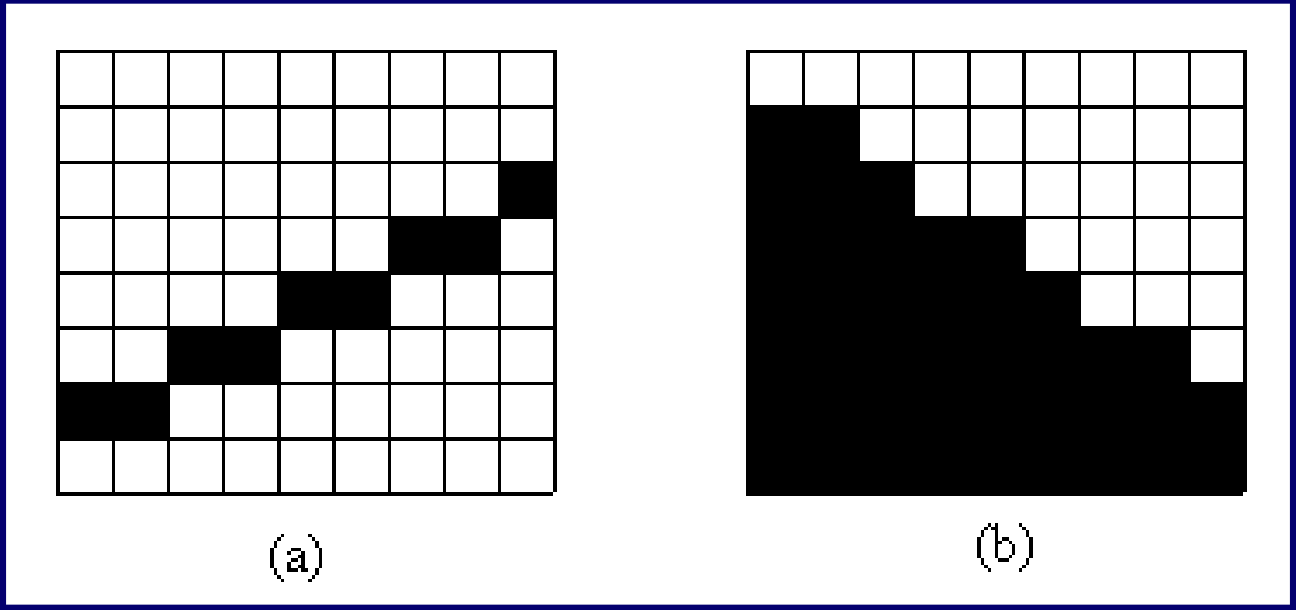
光栅图形的走样现象

- 阶梯状边界；
- 图形细节失真；
- 狭小图形遗失：动画序列中时隐时现，产生闪烁。

6.3 反走样

走样现象举例

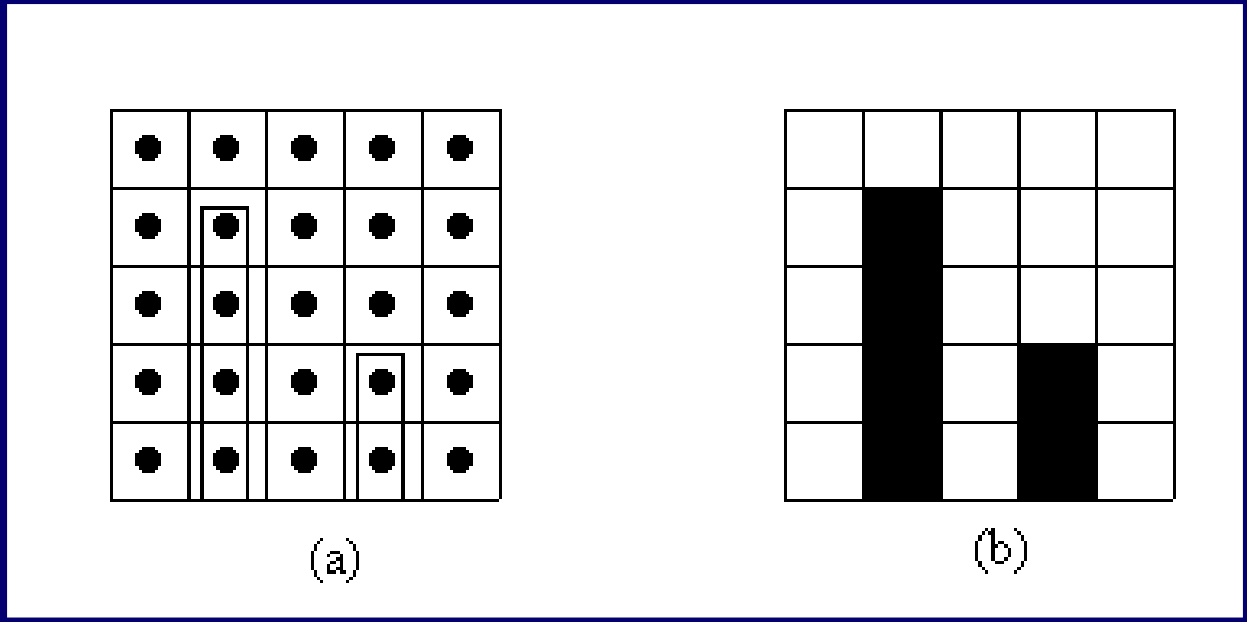
- 不光滑(阶梯状）的图形边界



6.3 反走样

走样现象举例

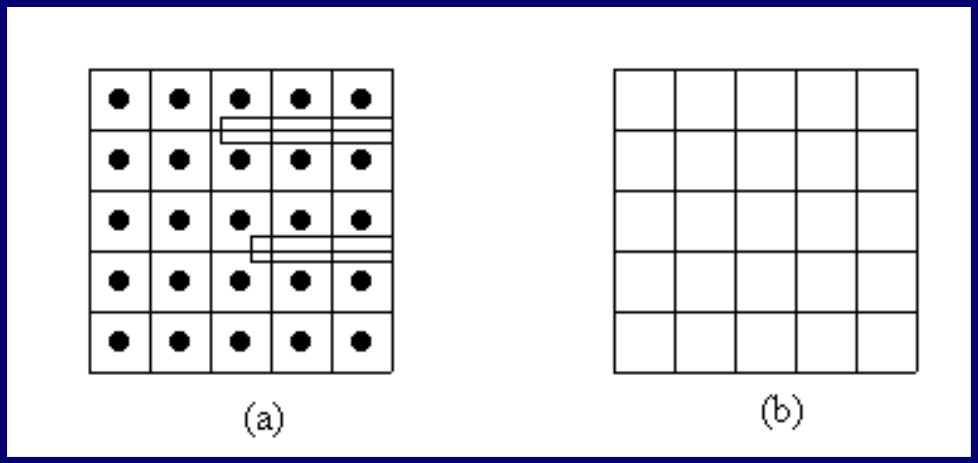
- 图形细节失真

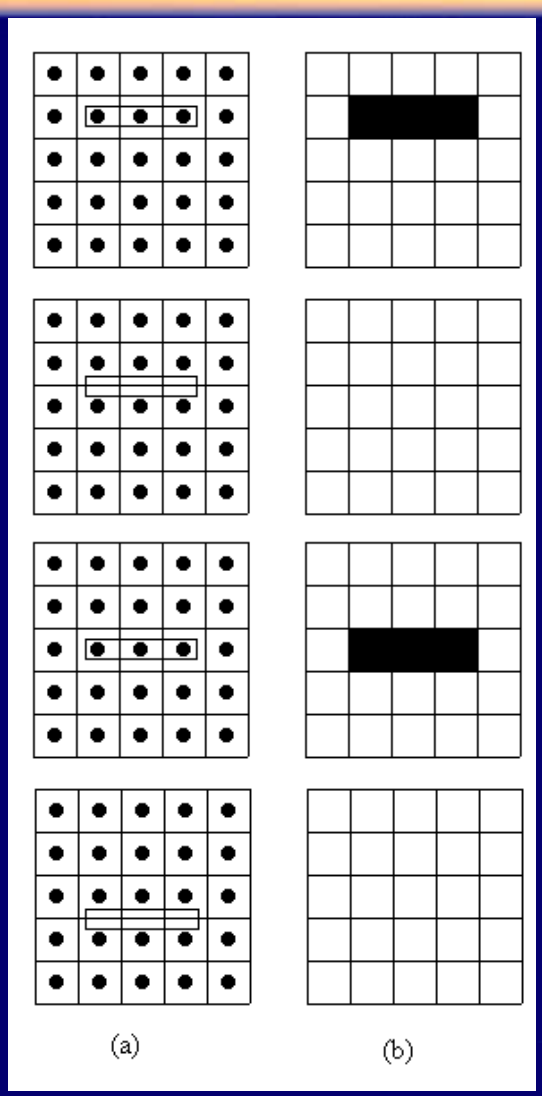


6.3 反走样

走样现象举例

- 狭小图形的遗失与动态图形的闪烁





6.3 反走样

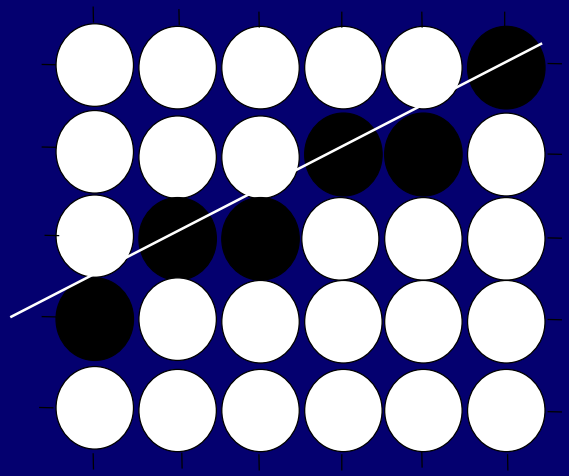
反走样概念及方法

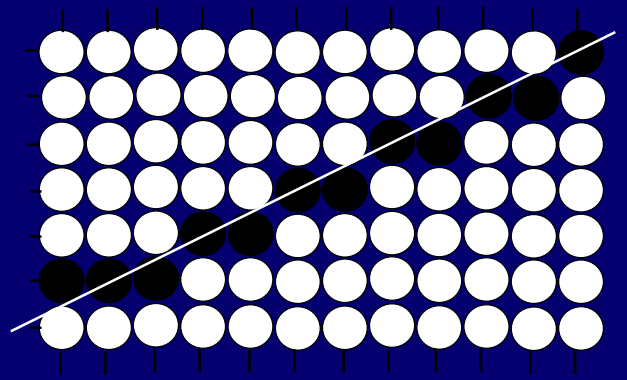
- 用于减少或消除走样现象的技术称为**反走样**(antialiasing)
  - 提高分辨率
  - 简单区域取样
  - 加权区域取样

6.3 反走样

(1) 提高分辨率

- 把显示器分辨率提高一倍，
  - 直线经过两倍的象素，锯齿也增加一倍，
  - 但同时每个阶梯的宽度也减小了一倍，
  - 所以显示出的直线段看起来就平直光滑了一些。





6.3 反走样

(1)提高分辨率

- 方法简单，但代价非常大。显示器的水平、竖直分辨率各提高一倍，则显示器的点距减少一倍，帧缓存容量则增加到原来的4倍，而扫描转换同样大小的图元却要花4倍时间。
- 而且它也只能减轻而不能消除锯齿问题

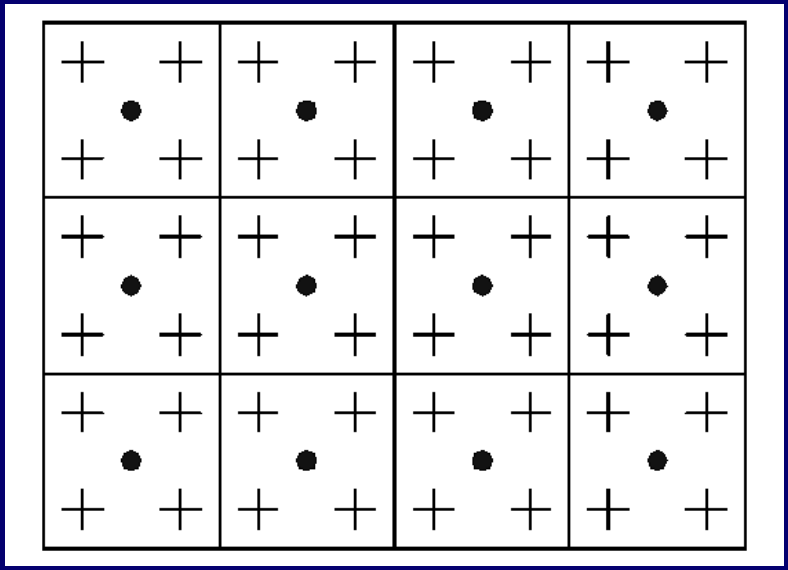
另一种方法（软件方法）：

用较高的分辨率的显示模式下计算，（对各自像属下计算，再求（非）加权平均的颜色值），在较低的分辨率模式下显示。只能减轻而不能消除锯齿问题。

6.3 反走样

软件方法1

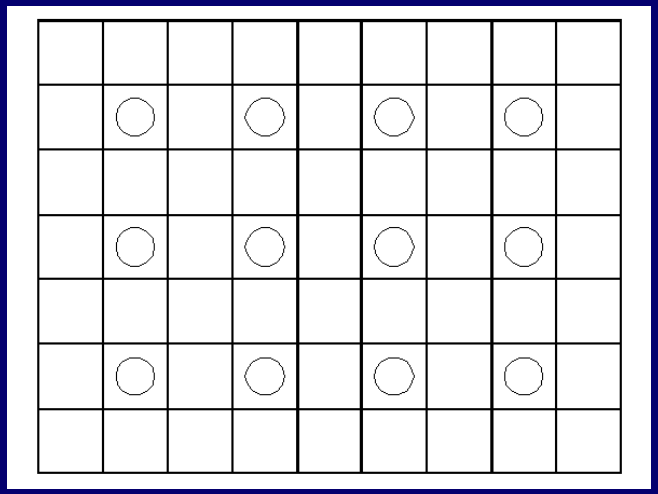
- 把每个像素分为四个子像素，扫描转换算法求得各子像素的灰度值，然后对四像素的灰度值简单平均，作为该像素的灰度值。



6.3 反走样

软件方法2

- 设分辨率为$m \times n$,把显示窗口分为$(2m+1) \times (2n+1)$个子像素，对每个子像素进行灰度值计算，然后根据权值表所规定的权值，对位于像素中心及四周的九个子像素加权平均，作为显示像素的颜色。
- 设$m=4, n=3$



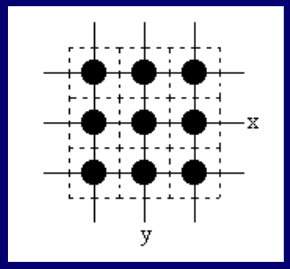
6.3 反走样

(2）简单区域取样

- 方法由来
  - 两点假设

    1、象素是数学上抽象的点，它的面积为0，它的亮度由覆盖该点的图形的亮度所决定；

    2、直线段是数学上抽象直线段，它的宽度为0。
  - 现实
    - 像素的面积不为0；
    - 直线段的宽度至少为1个像素；



  - 假设与现实的矛盾是导致混淆出现的原因之一

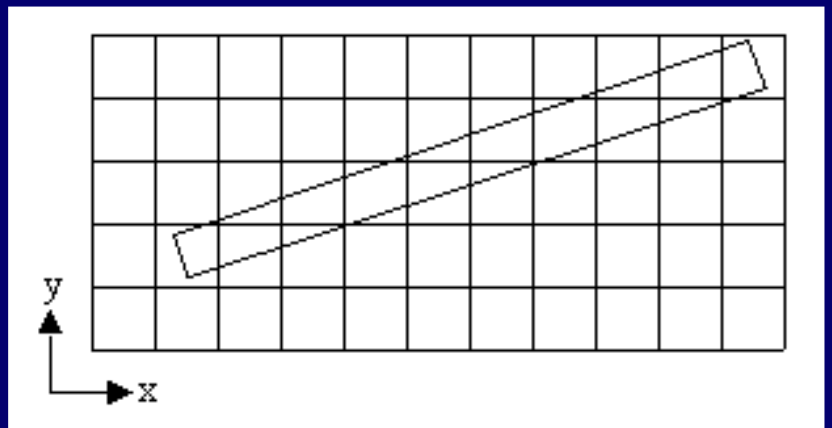
6.3 反走样

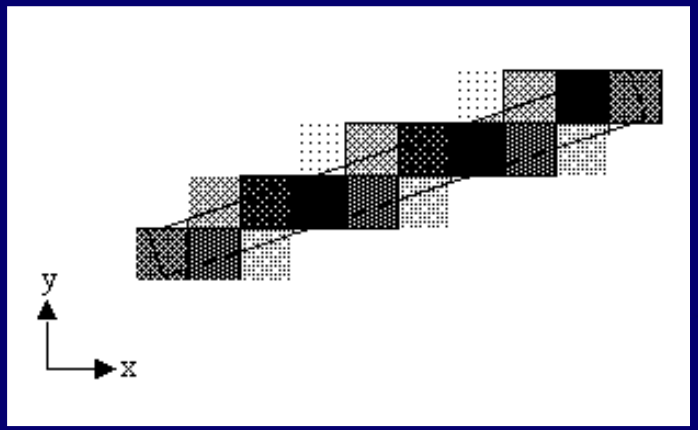
(2）简单区域取样

– 解决方法：改变直线段模型，由此产生算法

– 方法步骤：

1、将直线段看作具有一定宽度的狭长矩形；
2、当直线段与某象素有交时，求出两者相交区域的面积；
3、根据相交区域的面积，确定该象素的亮度值

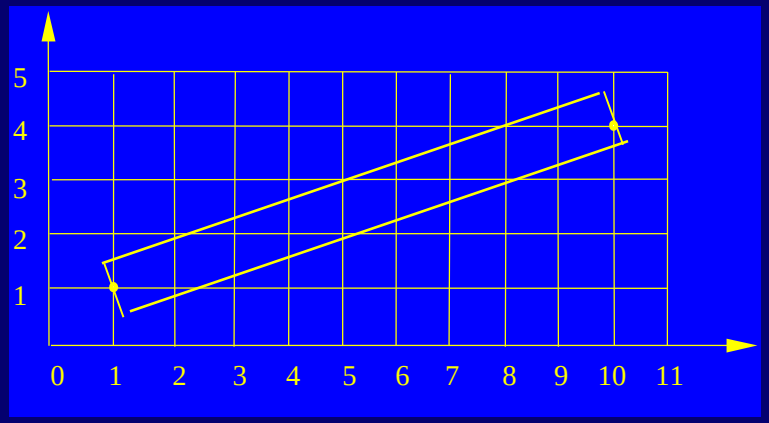




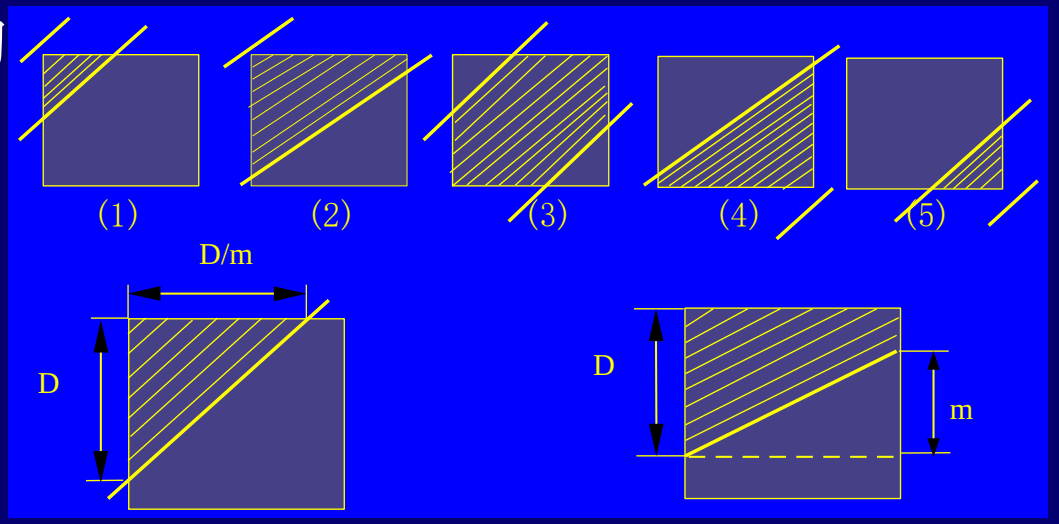
6.3 反走样

(2）简单区域取样

- 基本思想：
  - 每个象素是一个具有一定面积的小区域，将直线段看作具有一定宽度的狭长矩形。当直线段与象素有交时，求出两者相交区域的面积，然后根据相交区域面积的大小确定该象素的



有宽度的线条轮廓



象素相交的五种情况及用于计算面积的量

6.3 反走样

(2）简单区域取样

- 面积计算
  - 情况(1)（5)阴影面积为：$D^2/2m$;
  - 情况(2)（4)阴影面积为：$D - m/2$;
  - 情况(3)阴影面积为：$1 - D^2/m$
- 为了简化计算可以采用离散的方法

6.3 反走样

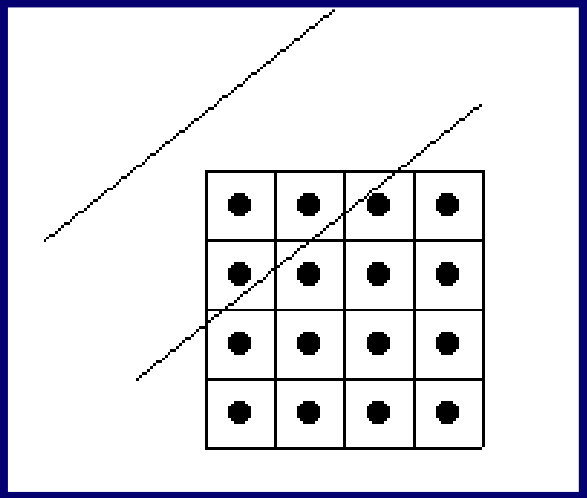
(2）简单区域取样

– 求相交区域的近似面积的离散计算方法

1、将屏幕象素分割成n个更小的子象素；

2、计算中心点落在直线段内的子象素的个数，记为k，

3、k/n为线段与象素相交区域面积的近似值



目的：简化计算

n = 16, k = 3

近似面积 = 3/16

6.3 反走样

(2）简单区域取样

- 简单区域取样采用的是一个盒式滤波器，它是一个二维加权函数，以w表示。
- w =1　若在当前像素所代表的正方形上
- w =0　其它区域上
- 直线条经过该像素时，该像素的灰度值可以通过在像素与直线条的相交区域上对w求积分获得。
- 此时，面积值=体积值

6.3 反走样

(2）简单区域取样

- 缺点：

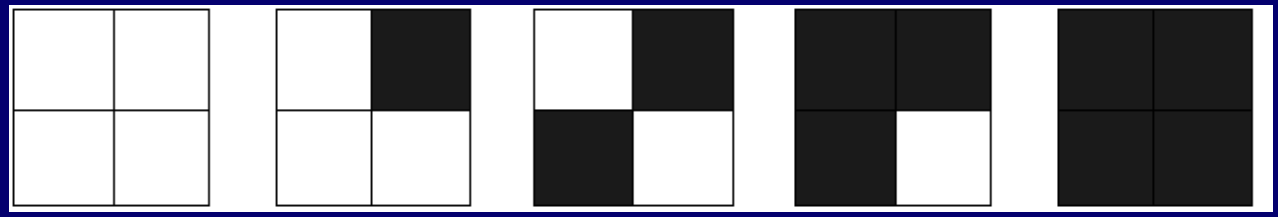
- 象素的亮度与相交区域的面积成正比，而与相交区域落在象素内的位置无关，这仍然会导致锯齿效应。

- 直线条上沿理想直线方向的相邻两个象素有时会有较大的灰度差。

6.3 反走样

(3）半色调技术

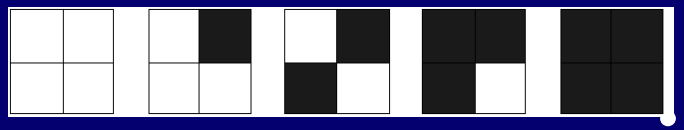
- 简单区域取样和加权区域取样技术的前提是多级灰度，利用多级灰度来提高视觉分辨率。但是，若只有两级灰度呢？能否使用上述技术呢？
- 对于给定的分辨率，通过将几个像素组合成一个单元来获得多级灰度。
- 例：在一个显示器中将四个像素组成一个单元，可产生5种光强。



6.3 反走样

(3）半色调技术

- 可用如下矩阵来表示：



$$\begin{pmatrix} 3 & 1 \\ 2 & 4 \end{pmatrix}$$

它表示黑色像素填入2×2个位置中的次序，每一级灰度再添上一个黑色像素就得到下一级灰度。

- 注意：

1．要尽量避免连成一条直线的花样。

2．花样是可以选择的。

单元也可以是长方形，如

$$\begin{pmatrix} 4 & 1 & 5 \\ 6 & 3 & 2 \end{pmatrix}$$

6.3 反走样

(3）半色调技术

- 一般来说，对于两级灰度显示器可能构成的灰度数等于单元中像素个数加1

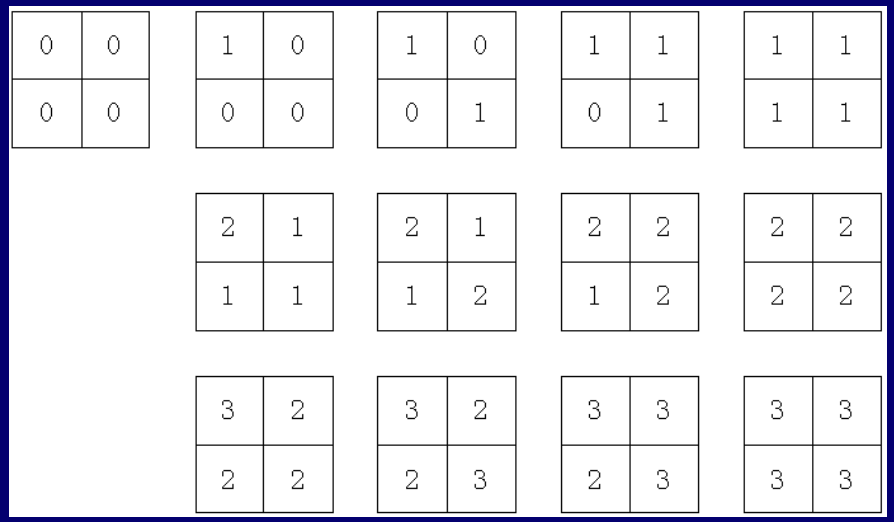
  ∴单元越大，灰度级别越高

- 它是以牺牲空间分辨率为代价的。

6.3 反走样

(3）半色调技术

- 例：灰度级别=4,每个单元=2*2



- 若有*m*级灰度，*n*×*n*个像素组成一个单元，则灰度级别数为*n*×*n*×(*m*-1)+1

END