# 第三章 基本图形生成算法(Ⅱ)

- 如何在指定的输出设备上根据坐标描述构造基本二维几何图形（点、直线、圆、椭圆、多边形域、字符串及其相关属性等）。

# 图形生成的概念

- 图形的生成：是在指定的输出设备上，根据坐标描述构造二维几何图形。
- 图形的扫描转换：在光栅显示器等数字设备上确定一个最佳逼近于图形的象素集的过程。

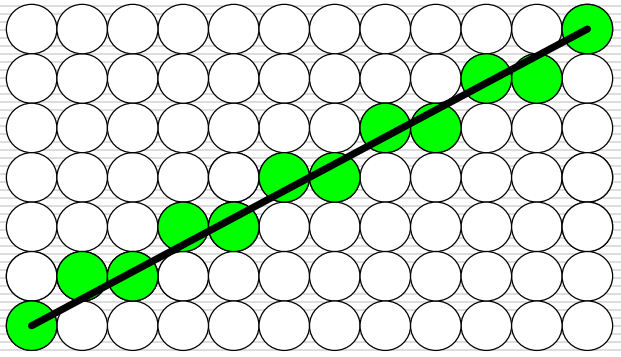


图3.1 用象素点集逼近直线

# 3.1 多边形的扫描转换与区域填充

- 多边形的扫描转换主要是通过确定穿越区域的扫描线的覆盖区间来填充。
- 区域填充是从给定的位置开始涂描直到指定的边界条件为止。

# 多边形的扫描转换与区域填充

- 多边形的扫描转换
- 边缘填充算法
- 区域填充
- 相关概念

# 多边形的扫描转换

- 顶点表示用多边形的顶点序列来刻划多边形。
- 点阵表示是用位于多边形内的象素的集合来刻划多边形。
- 扫描转换多边形：从多边形的顶点信息出发，求出位于其内部的各个象素，并将其颜色值写入帧缓存中相应单元的过程。

# 多边形的扫描转换

- X—扫描线算法
- 改进的有效边表算法

# X－扫描线算法——原理

- 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所有象素。

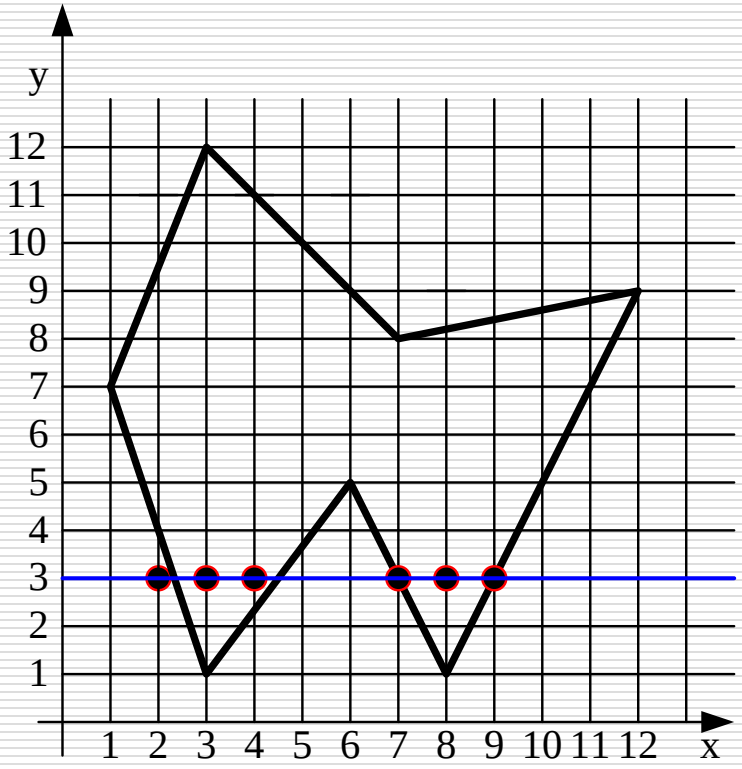



图3.2 x-扫描线算法填充多边形

# X－扫描线算法——算法步骤

1.确定多边形所占有的最大扫描线数，得到多边形顶点的最小和最大y值（$y_{min}$和$y_{max}$）。

2.从$y=y_{min}$到$y=y_{max}$，每次用一条扫描线进行填充。

3.对一条扫描线填充的过程可分为四个步骤：

求交；排序；交点配对；区间填色。

# X－扫描线算法——取整规则

- 交点的取整规则：使生成的像素全部位于多边形之内。（用于直线等图元扫描转换的四舍五入原则可能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：

- 规则1：X为小数，即交点落于扫描线上两个相邻像素之间时：
  - 交点位于左边界之上，向右取整；
  - 交点位于右边界之上，向左取整；

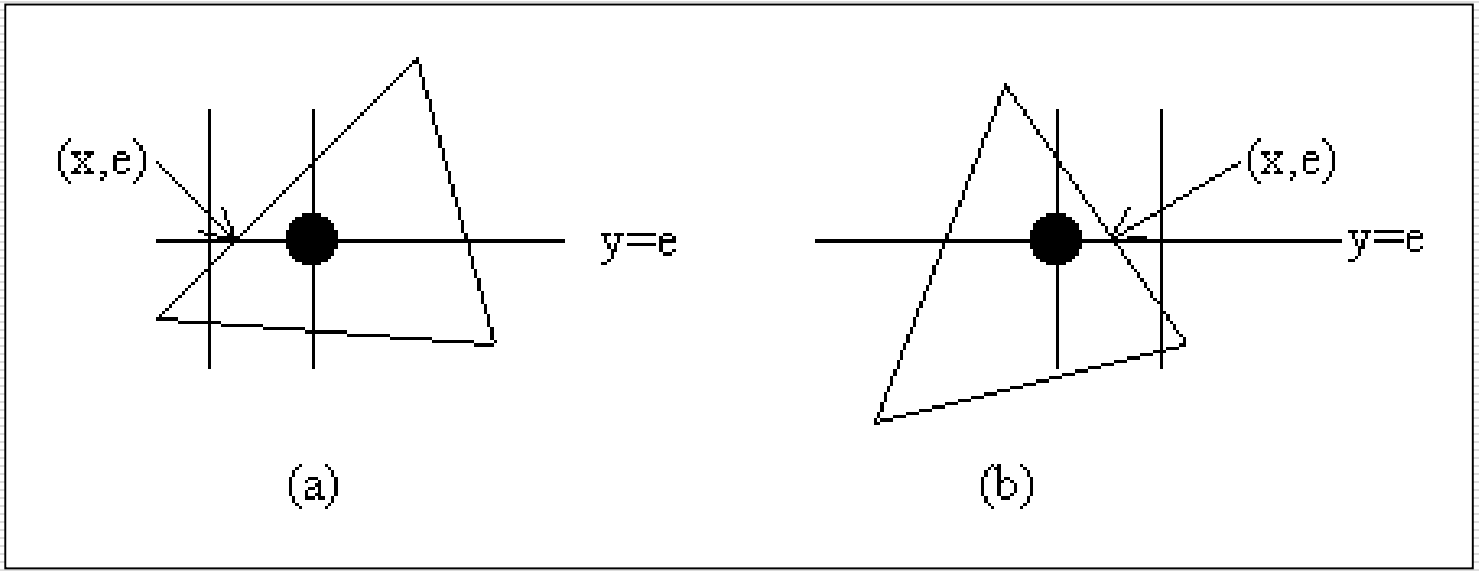



图3.3 取整规则1

# X－扫描线算法——取整规则

- 规则2：边界上象素的取舍问题，避免填充扩大化。规定落在右边边界上的像素不予填充。（具体实现时，只要对扫描线与多边形的相交区间左闭右开）

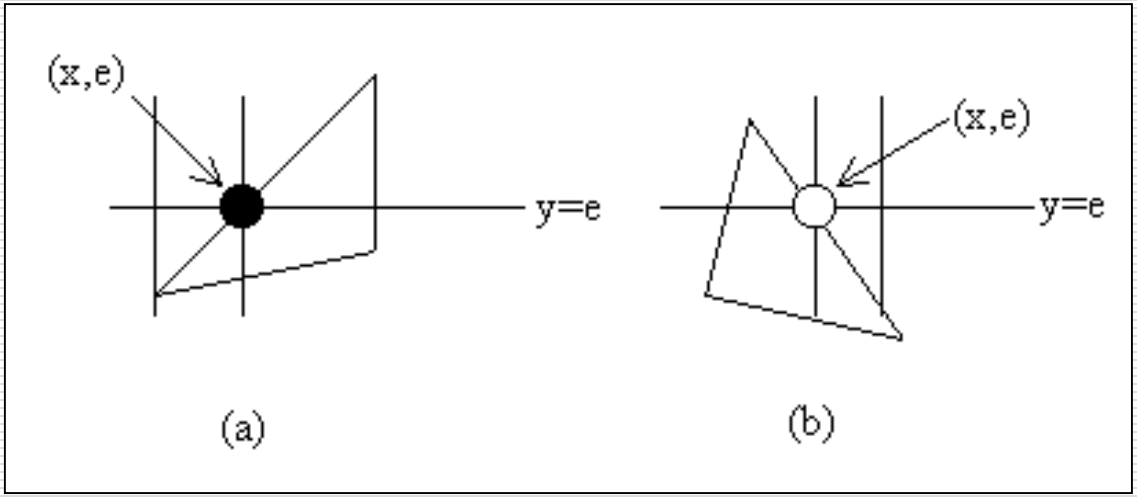



图3.4 取整规则2

# X－扫描线算法——取整规则

- 规则3：当扫描线与多边形顶点相交时，交点的取舍，保证交点正确配对。

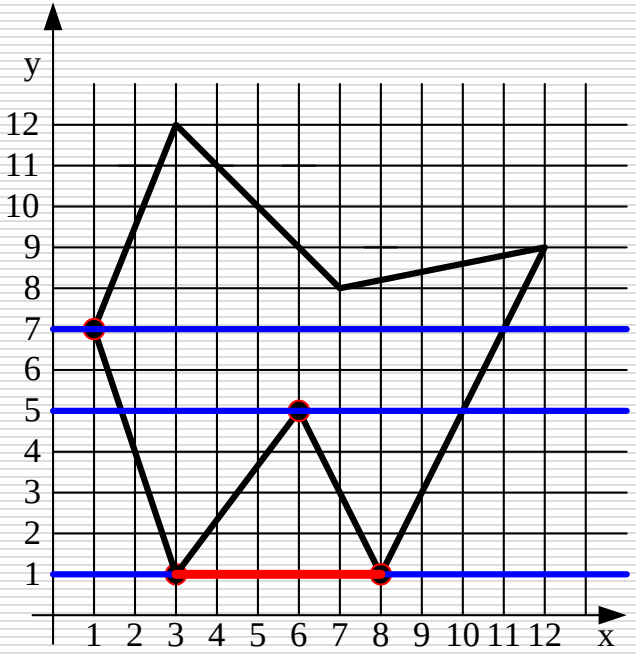



图3.5 取整规则3

# X－扫描线算法——取整规则

*解决方法*:

当扫描线与多边形的顶点相交时，

- 若共享顶点的两条边分别落在扫描线的两边，交点只算一个；
- 若共享顶点的两条边在扫描线的同一边，这时交点作为零个或两个。



# X－扫描线算法——取整规则

实际处理：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。

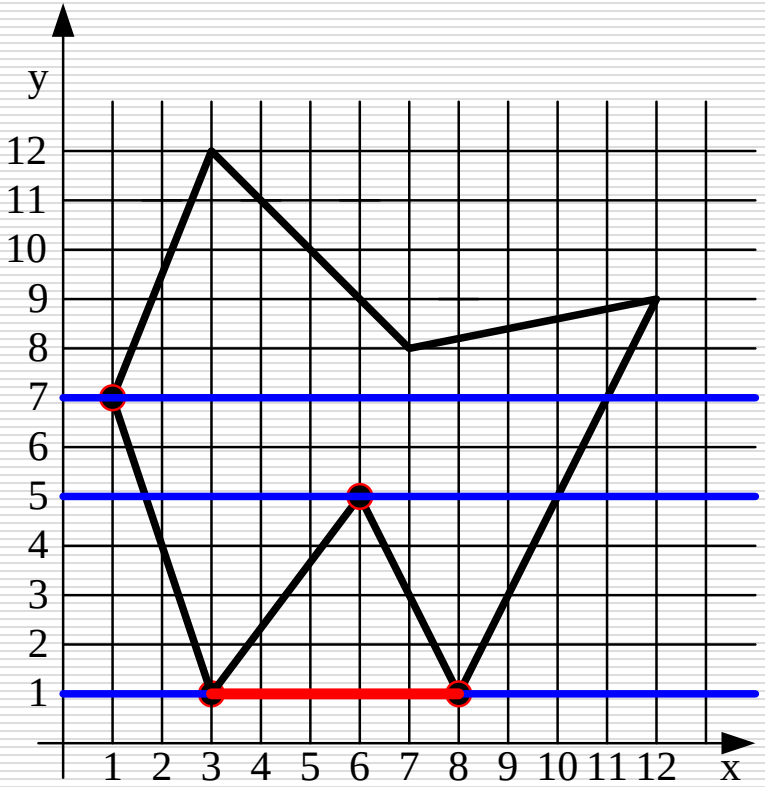



图3.6 取整规则3

# X－扫描线算法——取整规则

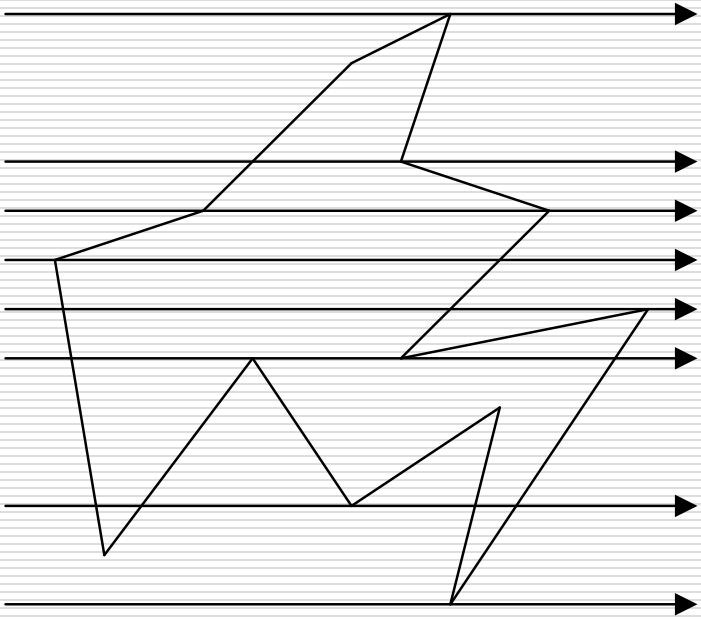


图5.21 与扫描线相交的多边形顶点的交点数

[填充过程实例]

# 改进的有效边表算法（Y连贯性算法）

改进原理：

- 处理一条扫描线时，仅对有效边求交。
- 利用扫描线的连贯性。
- 利用多边形边的连贯性。

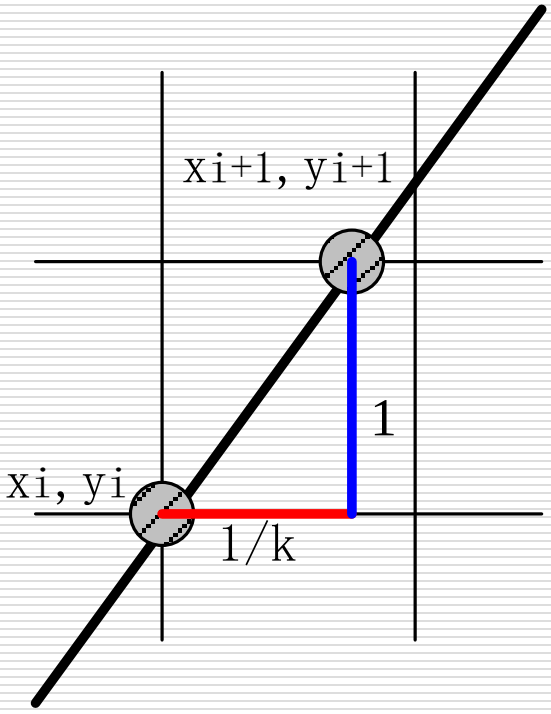



图3.7 与多边形边界相交的两条连续扫描线交点的相关性

# 改进的有效边表算法（Y连贯性算法）

- 有效边（Active Edge）：指与当前扫描线相交的多边形的边，也称为活性边。
- 有效边表（Active Edge Table，AET）：把有效边按与扫描线交点x坐标递增的顺序存放在一个链表中，此链表称为有效边表。
- 有效边表的每个结点：

$$x \quad y_{max} \quad 1/k \quad next$$

# 改进的有效边表算法——构造边表

- 首先构造一个纵向链表，链表的长度为多边形所占有的最大扫描线数，链表的每个结点，称为一个桶，则对应多边形覆盖的每一条扫描线。
- 将每条边的信息链入与该边最小y坐标（$y_{min}$）相对应的桶处。也就是说，若某边的较低端点为$y_{min}$，则该边就放在相应的扫描线桶中。



# 改进的有效边表算法——构造边表

- 每条边的数据形成一个结点，内容包括：该扫描线与该边的初始交点x（即较低端点的x值），1/k，以及该边的最大y值$y_{max}$。

$x|_{ymin}$ $y_{max}$ 1/k NEXT

- 同一桶中若干条边按$X|_{ymin}$由小到大排序，若$X|_{ymin}$ 相等，则按照1/k由小到大排序。

# 解决顶点交点计为1时的情形：

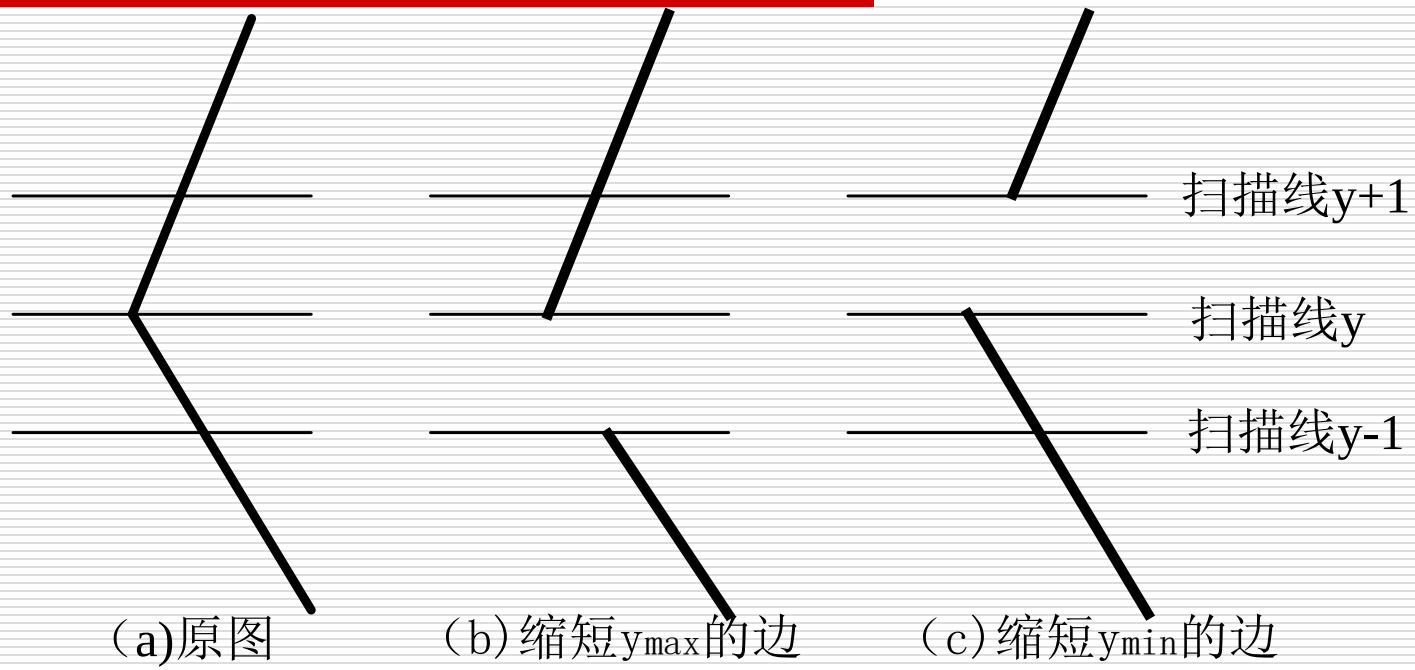



**图3.8 将多边形的某些边缩短以分离那些应计为1个交点的顶点**

| | | | | |
|---|---|---|---|---|
| p3p2 | 3 | 6 | −1/3 | |
| p3p4 | 3 | 5 | 3/4 | |
| p5p4 | 8 | 5 | −1/2 | |
| p5p6 | 8 | 9 | 1/2 | |
| p2p1 | 1 | 12 | 2/5 | |
| p0p1 | 7 | 12 | −1 | |
| p0p6 | 7 | 9 | 5 | |

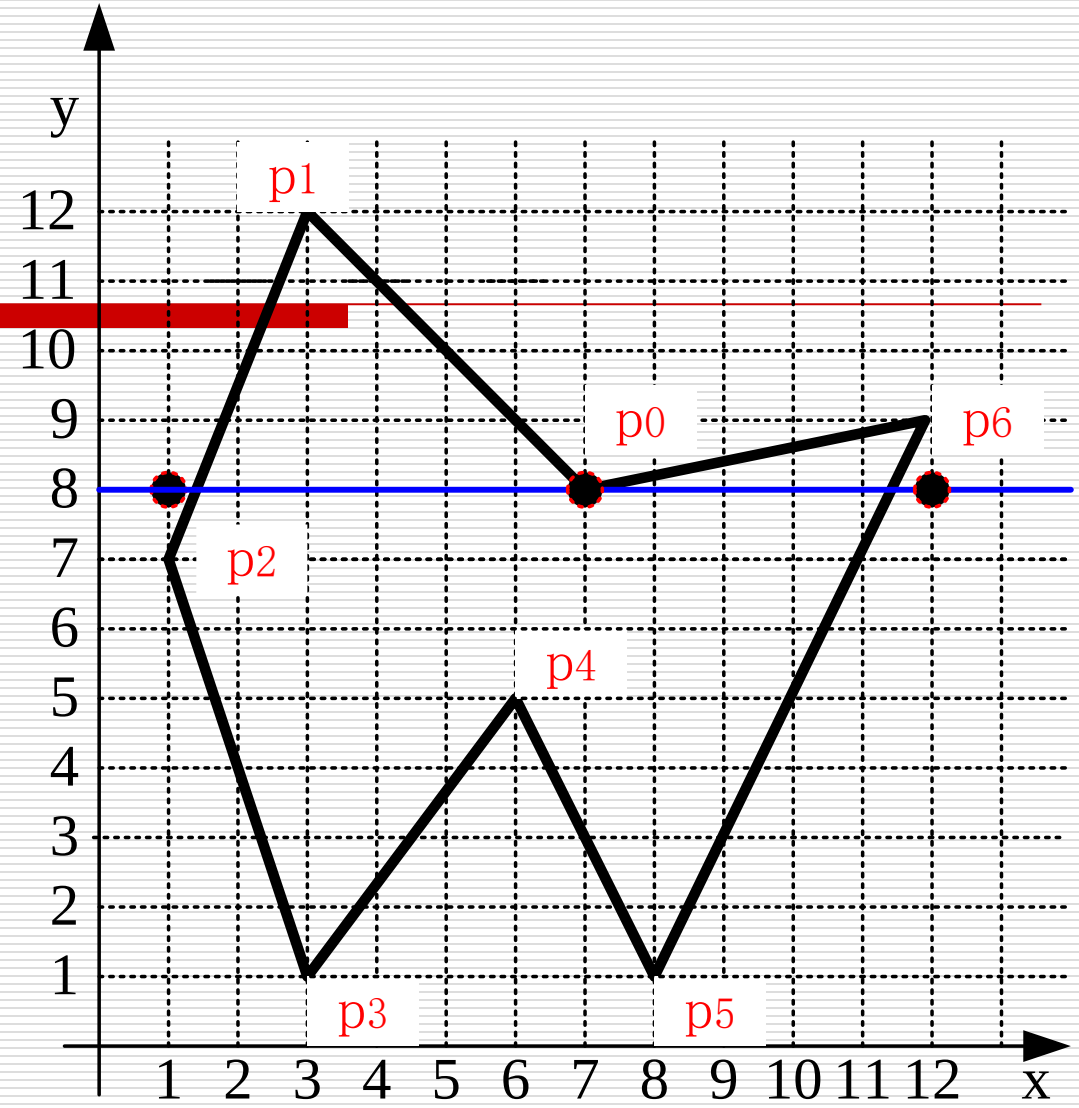



**图3.9 多边形$P_0P_1P_2P_3P_4P_5P_6$**

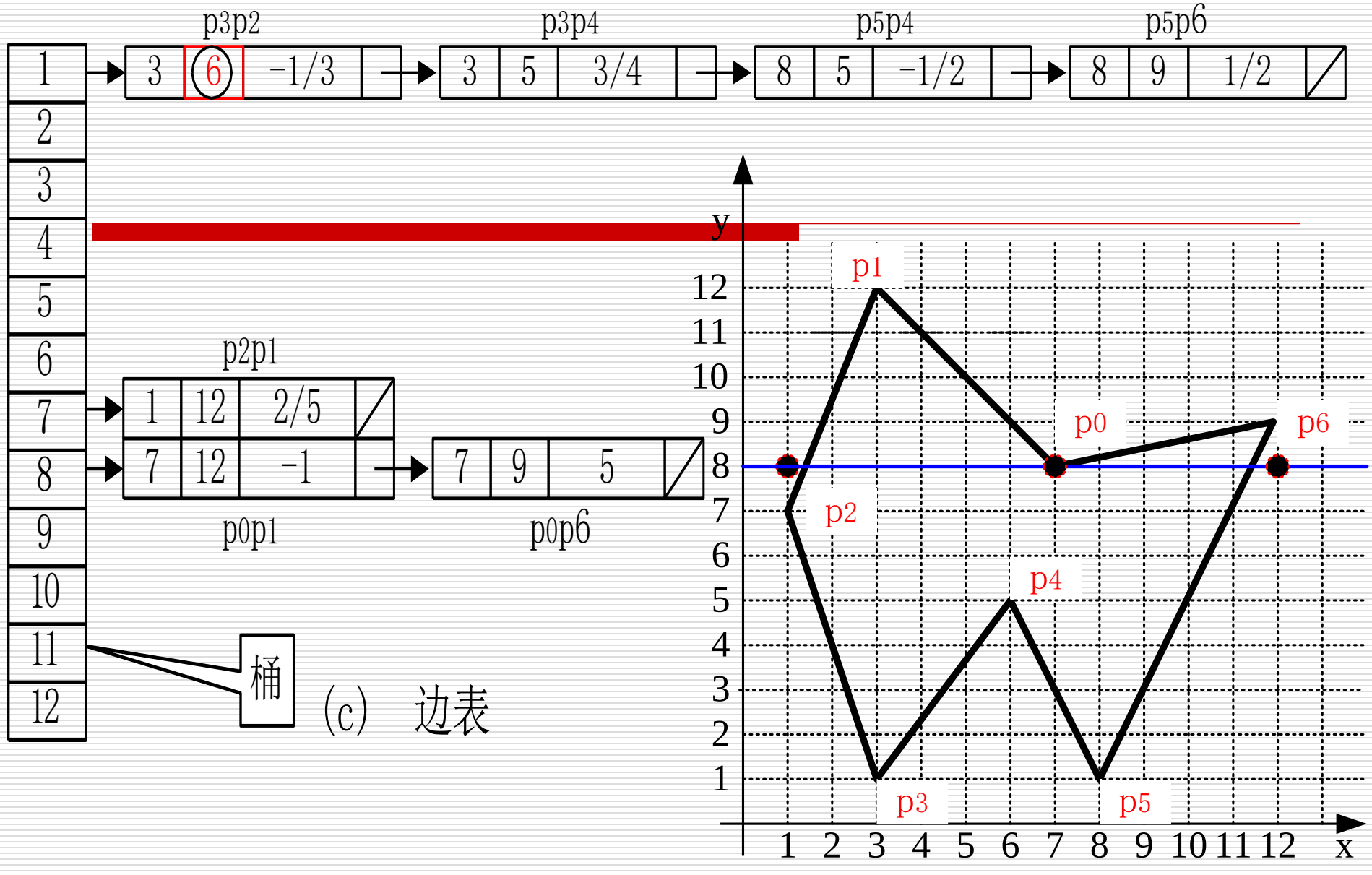



**图3.10 多边形$P_0P1P_2P_3P_4P_5P_6$**

# 改进的有效边表算法———算法步骤

(1)初始化：构造边表，AET表置空；

(2)将第一个不空的ET表中的边与AET表合并；

(3)由AET表中取出交点对进行填充。填充之后删除$y=y_{max}$的边；

(4)$y_{i+1}=y_i+1$，根据$x_{i+1}=x_i+1/k$计算并修改AET表，同时合并ET表中$y=y_{i+1}$桶中的边，按次序插入到AET表中，形成新的AET表；

(5)AET表不为空则转(3)，否则结束。

# 边缘填充算法

- 基本思想：按任意顺序处理多边形的每条边。

  处理时，先求出该边与扫描线的交点，再对扫描线上交点右方的所有象素取反。

- 算法简单，但对于复杂图型，每一象素可能被访问多次

# 栅栏填充算法

- 栅栏指的是一条过多边形顶点且与扫描线垂直的直线。它把多边形分为两半。
- 基本思想：按任意顺序处理多边形的每一条边，但处理每条边与扫描线的交点时，将交点与栅栏之间的象素取反。
- 这种算法尽管减少了被重复访问象素的数目，但仍有一些象素被重复访问。

# 边标志算法

- 基本思想：先用特殊的颜色在帧缓存中将多边形的边界勾画出来，然后将着色的象素点依x坐标递增的顺序配对，再把每一对象素构成的区间置为填充色。
- 分为两个步骤：打标记-多边形扫描转化；填充。
- 当用软件实现本算法时，速度与改进的有效边表算法相当，但本算法用硬件实现后速度会有很大提高。

# 区域填充（种子填充）

- 基本概念
- 区域的表示方法
- 区域的分类
- 区域填充算法

# 基本概念

- 区域填充是指从区域内的某一个象素点（种子点）开始，由内向外将填充色扩展到整个区域内的过程。
- 区域是指已经表示成点阵形式的填充图形，它是相互连通的一组像素的集合。

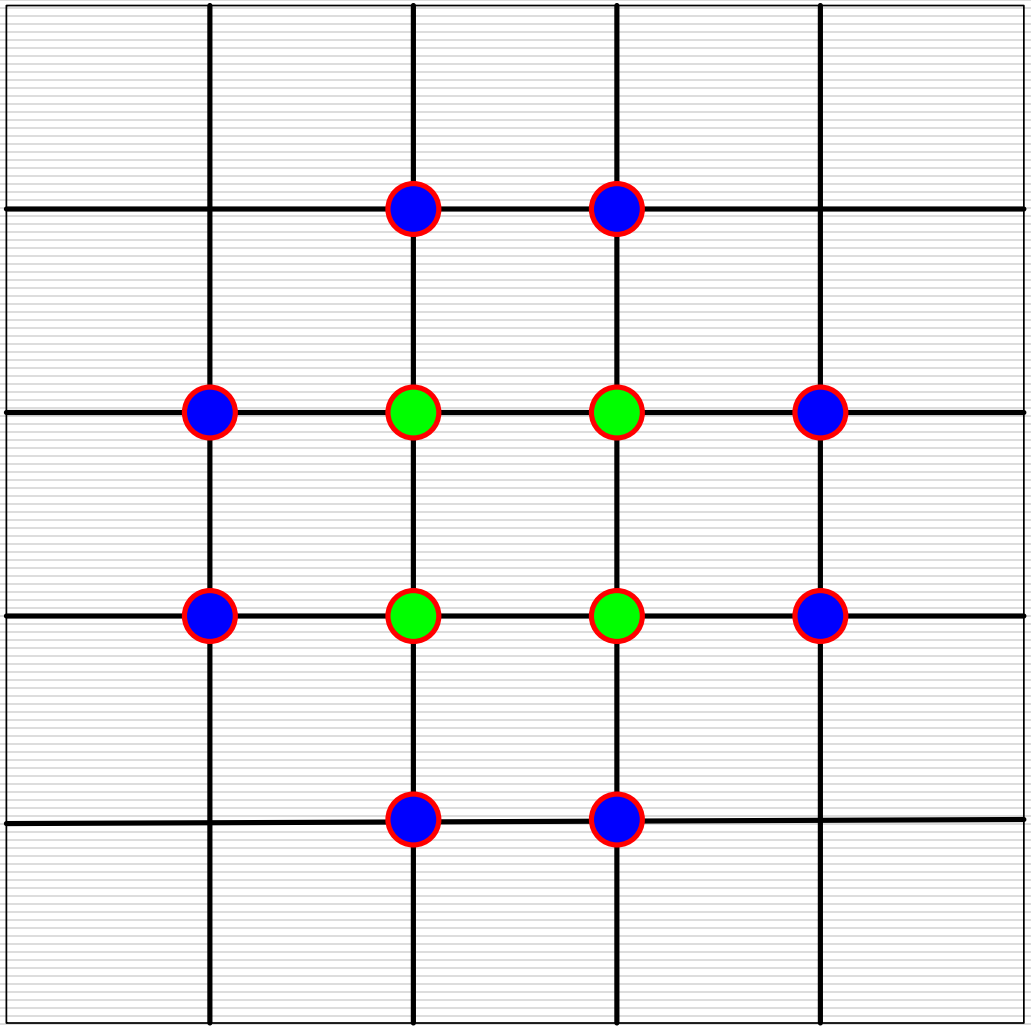


图3.12 区域的概念

# 区域的表示方法

- 边界表示法：把位于给定区域的边界上的象素一一列举出来的方法。
- 边界表示法中，由于边界由特殊颜色指定，填充算法可以逐个象素地向外处理，直到遇到边界颜色为止，这种方法称为边界填充算法（Boundary-fill Algorithm）。

- 内点表示法：枚举出给定区域内所有象素的表示方法。以内点表示法为基础的区域填充算法称为泛填充算法（Flood-fill Algorithm）。

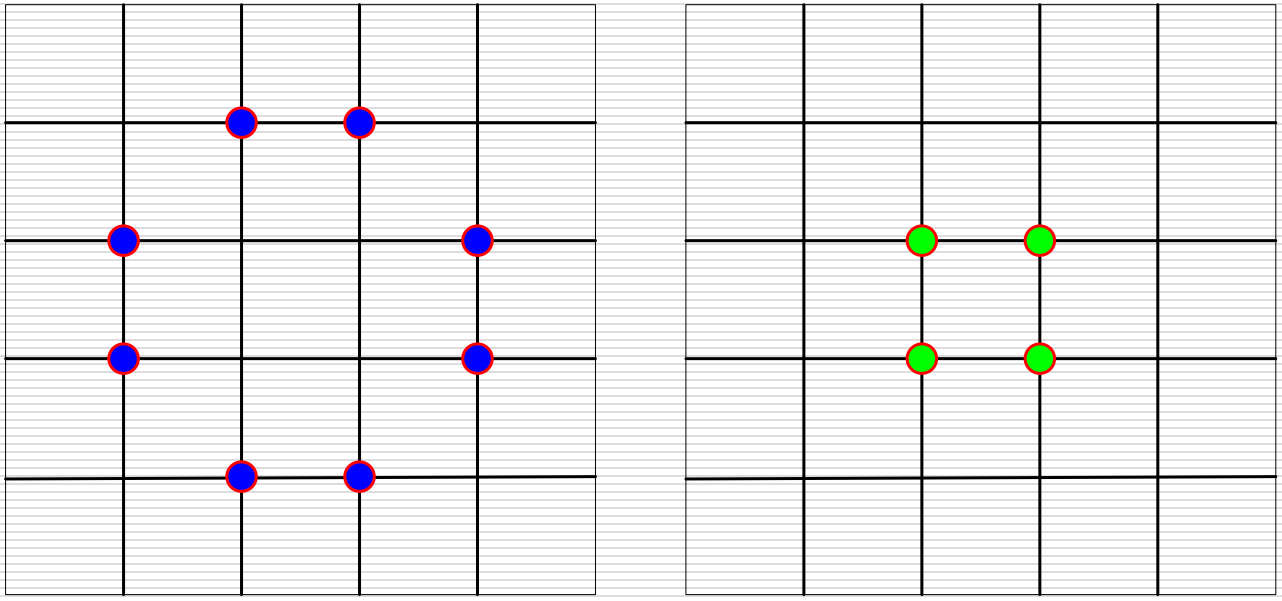
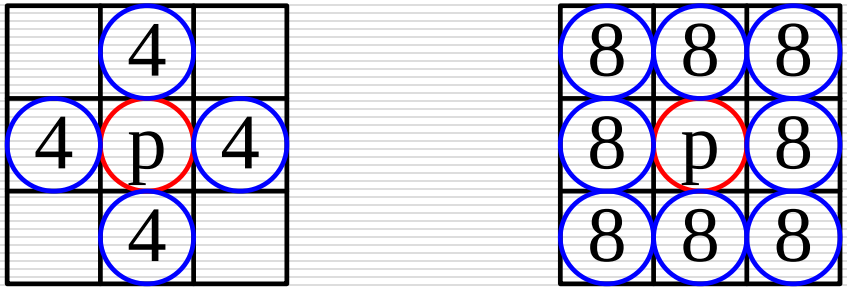


图3-13 区域的表示方法

# 区域的分类

## 4-连通区域，8-连通区域




（a） 4—邻接点　　（b）8—邻接点

图3-14 4—邻接点与8—邻接点

# 区域的分类

- **4-连通区域**：从区域上的一点出发，通过访问已知点的4-邻接点，在不越出区域的前提下，遍历区域内的所有象素点。
- **8-连通区域**：从区域上的一点出发，通过访问已知点的8-邻接点，在不越出区域的前提下，遍历区域内的所有象素点。

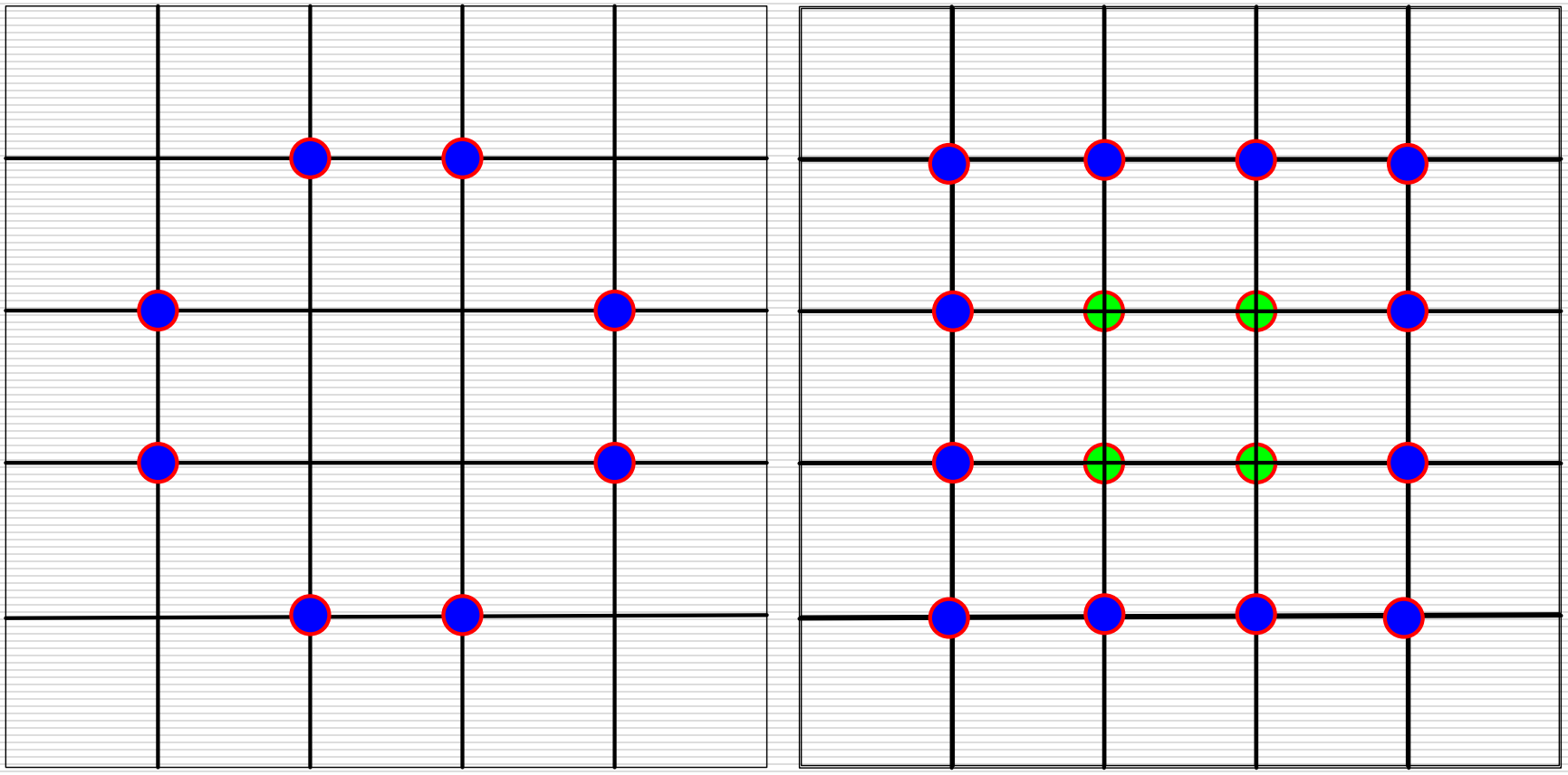


图3.15　4－连通与8－连通区域

# 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。

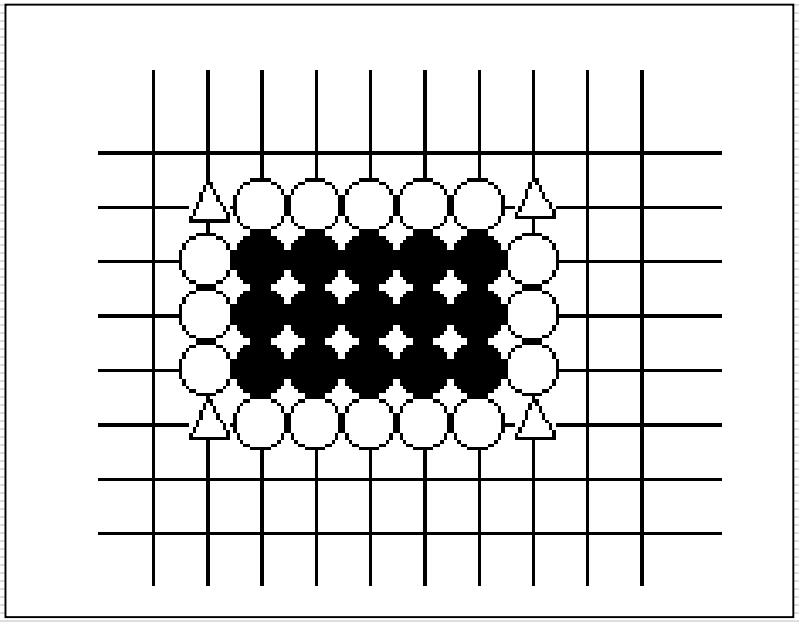


图3-16 4－连通与8－连通区域

# 区域填充算法

- 区域填充算法（边界填充算法和泛填充算法）是根据区域内的一个已知象素点（种子点）出发，找到区域内其他象素点的过程，所以把这一类算法也成为种子填充算法。

# 区域填充算法——边界填充算法

- 算法的输入：种子点坐标(x,y)，填充色以及边界颜色。
- 利用堆栈实现简单的种子填充算法

  算法从种子点开始检测相邻位置是否是边界颜色，若不是就用填充色着色，并检测该像素点的相邻位置，直到检测完区域边界颜色范围内的所有像素为止。

# 区域填充算法——边界填充算法

栈结构实现4-连通边界填充算法的算法步骤为：

种子象素入栈；当栈非空时重复执行如下三步操作：

(a)栈顶象素出栈；

(b)将出栈象素置成填充色；

(c)检查出栈象素的4-邻接点，若其中某个象素点不是边界色且未置成多边形色，则把该象素入栈。

# 区域填充算法——边界填充算法

栈结构实现8-连通边界填充算法的算法步骤为：

种子象素入栈；当栈非空时重复执行如下三步操作：

(a)栈顶象素出栈；

(b)将出栈象素置成填充色；

(c)检查出栈象素的8-邻接点，若其中某个像素点不是边界色且未置成多边形色，则把该像素入栈。

# 区域填充算法——边界填充算法

- 可以用于填充带有内孔的平面区域。
- 把太多的像素压入堆栈，降低了效率，同时需要较大的存储空间。
- 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。
- 通过沿扫描线填充水平象素段，来代替处理4-邻接点和8-邻接点。

# 区域填充算法——边界填充算法

- 扫描线种子填充算法：扫描线通过在任意不间断扫描线区间中只取一个种子像素的方法使堆栈的尺寸极小化。不间断区间是指在一条扫描线上的一组相邻像素。

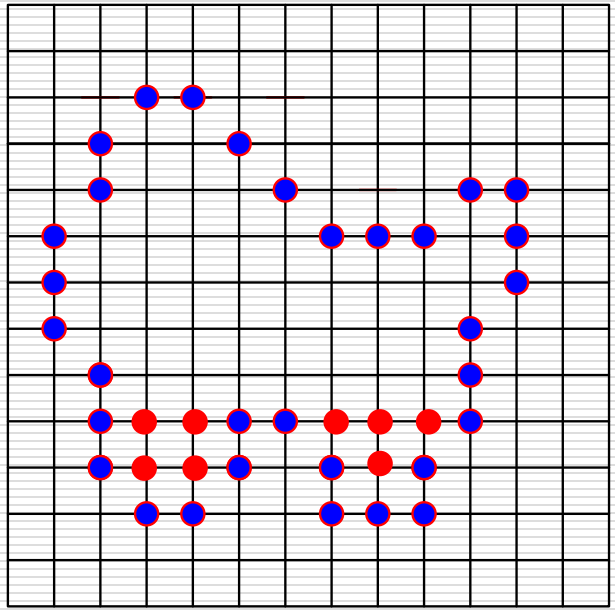


图3-17 扫描线种子填充算法

# 区域填充算法——边界填充算法

- 基本过程：当给定种子点时，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。

# 区域填充算法——边界填充算法

- 扫描线种子填充算法：我们可以在任意一个扫描线与多边形的相交区间中，只取一个种子像素，并将种子像素入栈，当栈非空时作如下四步操作：

- （1）栈顶像素出栈；
- （2）沿扫描线对出栈像素的左右像素进行填充，直至遇到边界像素为止，也就是对包含出栈像素的整个区间进行填充；
- （3）上述区间内最左最右的像素分别记为$x_l$和$x_r$；
- （4）在区间［$x_l$，$x_r$］中检查与当前扫描线相邻的上下两条扫描线的有关像素是否全为边界像素或已填充的像素，若存在非边界、非填充的像素，则把每一区间的最右像素入栈。

上述改进之后的算法称之为扫描线种子填充算法，其伪C语言描述如下：

```
void scanline-seed-fill(polydef, color, x, y)
  多边形定义 polydef;
  int color,  x,  y;  / * (x,  y)为种子像素*/

   {
     int x, y, x0, xl, xr
     push(seed(x, y));  /* 种子像素x栈 */
```

```
while (栈非空)
{  pop (pixel(x, y)); /* 栈顶像素出栈，并置为
   (x, y) */
    putpixel(x, y, color);
    x0=x+1;
    while (getpixel(x0, y)的值不等于边界像素颜
色值) /* 填充出栈像素的右方像素*/
     { putpixel(x0, y, color);
      x0=x0+1;
     }
    xr=x0-1;  /*最右像素*/
    x0=x-1;
```

```
while(getpixel(x0, y)的值不等于边界像素颜色值)
  /* 填充出栈像素的左方 像素*/
      { putpixel(x0, y, color);
       x0=x0-1;
      }
     x1=x0+1; /*最左像素*/
     /*  检查上一条扫描线，若存在非边界且未填充
  的像素，则将各连续区间的最右像素入栈 */
     x0=x1;  y=y+1;
```

```
while(x0<=xr)
   { flag=0;
     while(( getpixel(x0, y)的值不等于边界像素颜色值)&& (getpixel(x0, y)的值不等于多边形内像素颜色值)&& (x0<xr))
       { if (flag==0) flag=1;
         x0++
       }
     if(flag==1)
       { if(( x0==xr)&&(getpixel(x0, y)的值不等于边界像素颜色值)&& (getpixel(x0, y) 的值不等于多边形内像素颜色值))   push((x0, y));
```

```
    else   push((x0-1, y));


     flag=0;
      }
    xnextspan = x0;
     while(( pixel(x0, y)等于边界像素颜色值
)||(pixel(x0, y)等于多边形内像素颜色值
)&&(x0<=xr))
        x0++;
   if(xnextspan ==x0)     x0++;
} /* while(x0<=xr) */
```

```
 /*  检查下一条扫描线，若存在非边界，未填充的像素，则将各连续区间的最右像素入栈；处理与前面一条扫描线的算法相同，只要把y+1换为y-1即可 */

    } /*  while (栈非空) */
}
```

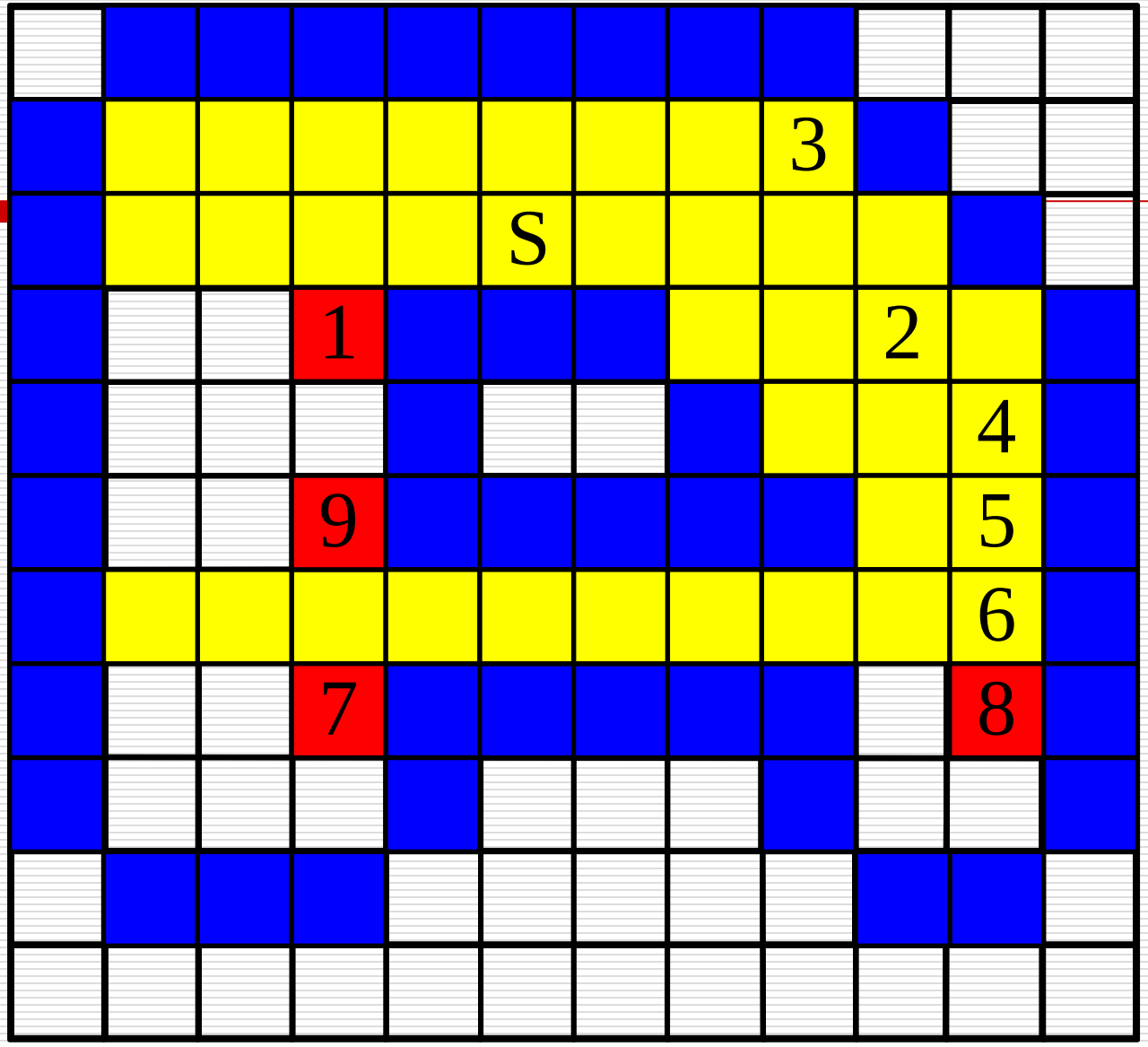



图3.18 扫描线种子填充算法过程

# 相关概念——角外测试

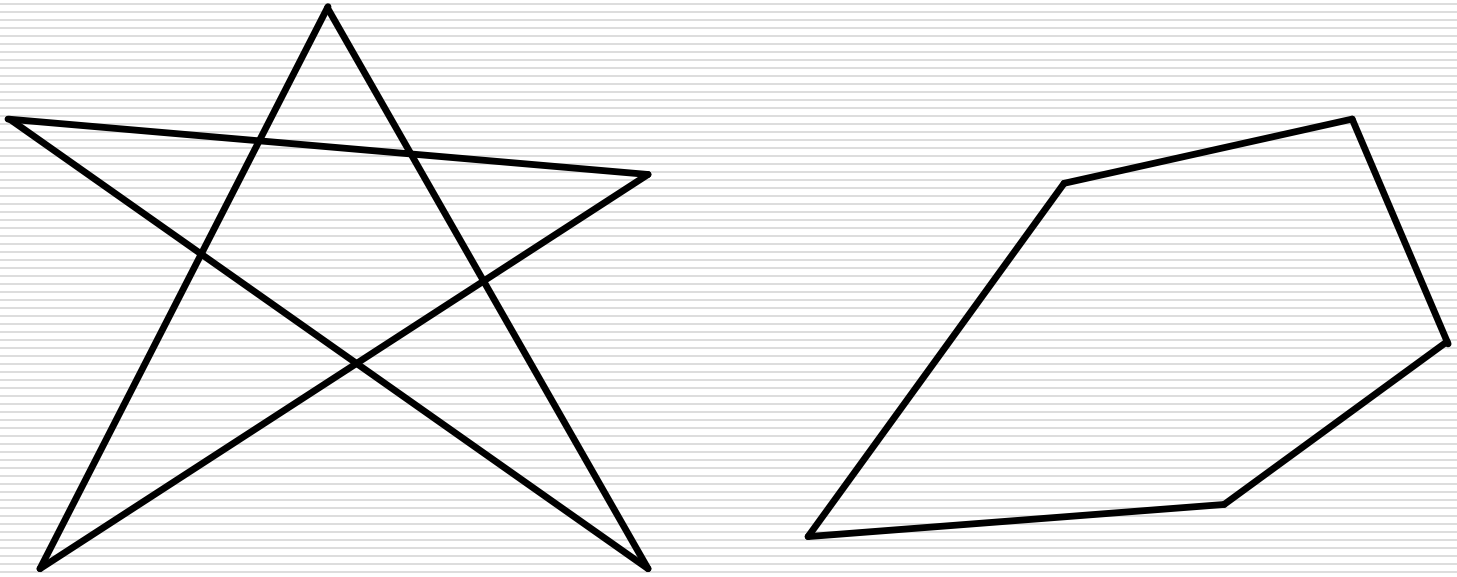
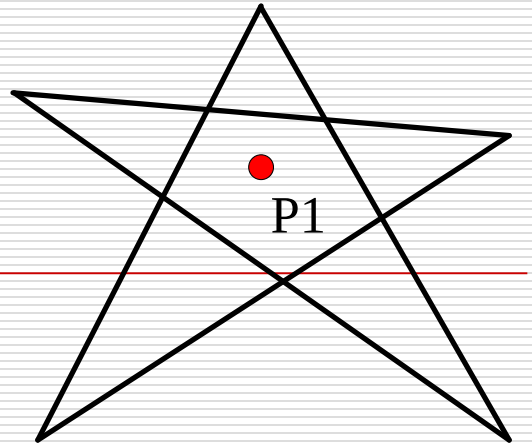


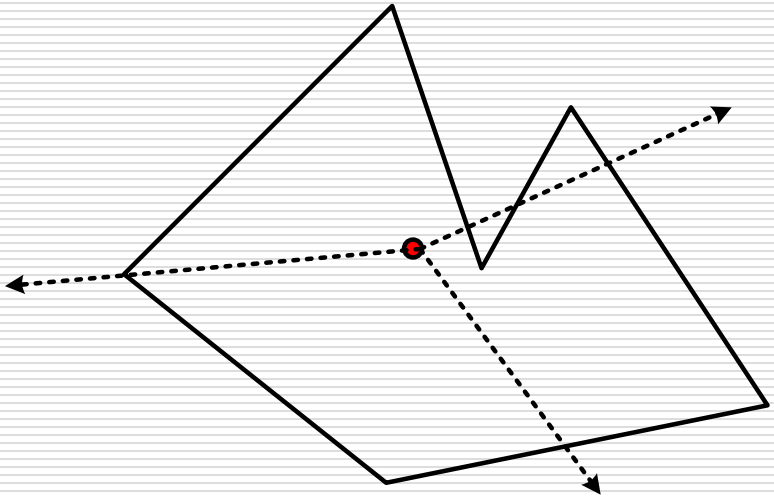
图3.19 不自交的多边形与自相交的多边形

# 相关概念——内外测试




- 奇-偶规则（Odd-even Rule）

从任意位置p作一条射线，若与该射线相交的多边形边的数目为奇数，则p是多边形内部点，否则是外部点。



# 相关概念——内外测试

非零环绕数规则（Nonzero Winding Number Rule）

- 首先使多边形的边变为矢量。
- 将环绕数初始化为零。
- 再从任意位置p作一条射线。当从p点沿射线方向移动时，对在每个方向上穿过射线的边计数，每当多边形的边从右到左穿过射线时，环绕数加1，从左到右时，环绕数减1。
- 处理完多边形的所有相关边之后，若环绕数为非零，则p为内部点，否则，p是外部点。

# 相关概念——曲线边界区域填充

- 相交计算中包含了非线性边界。
- 对于简单曲线：

（1）计算曲线相对两侧的两个扫描线交点

（2）简单填充在曲线两侧上的边界点间的水平象素区间。

# 3.2 字符处理

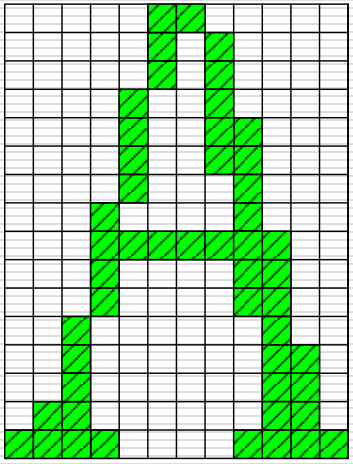
- ASCII码：“美国信息交换用标准代码集”（American Standard Code for Information Interchange），简称ASCII码。
- 国际码：“中华人民共和国国家标准信息交换编码，简称为国际码，代号GB2312－80。
- 字库：字库中储存了每个字符的图形信息。
- 矢量字库和点阵字库。

# 字符处理——点阵字符

- 在点阵表示中，每个字符由一个点阵位图来表示。
- 显示时：形成字符的像素图案。

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

(a)字符A的点阵位图



(a)字符A的象素图案

图3.20 字符A的点阵表示

# 字符处理——点阵字符

- 定义和显示直接、简单。
- 存储需要耗费大量空间。
  - 从一组点阵字符生成不同尺寸和不同字体的其他字符。
  - 采用压缩技术。

# 字符处理——矢量字符

- 矢量字符采用直线和曲线段来描述字符形状，矢量字符库中记录的是笔划信息。
- 显示时：解释字符的每个笔划信息。

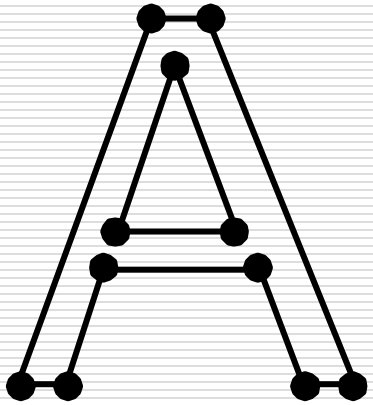


图3.21 字符A的矢量表示

# 字符处理——矢量字符

- 轮廓字型法采用直线或二、三次曲线的集合来描述一个字符的轮廓线，轮廓线构成了一个或若干个封闭区域，显示时只要填充这些区域就可产生相应的字符。
- 显示时可以“无极放缩”。
- 占用空间少，美观，变换方便。

# 3.3 属性处理

- 图素或图段的外观由其属性决定。
- 图形软件中实现属性选择的方法是提供一张系统当前属性值表，通过调用标准函数提供属性值表的设置和修改。进行扫描转换时，系统使用属性值表中的当前属性值进行显示和输出。

# 属性处理

- 线型与线宽
- 区域填充属性

# 线型与线宽——线型

- 线型的显示可用象素段方法实现：针对不同的线型，画线程序沿路径输出一些连续象素段，在每两个实心段之间有一段中间空白段，他们的长度（象素数目）可用象素模板(pixel mask)指定。
- 存在问题：如何保持任何方向的划线长度近似地相等。

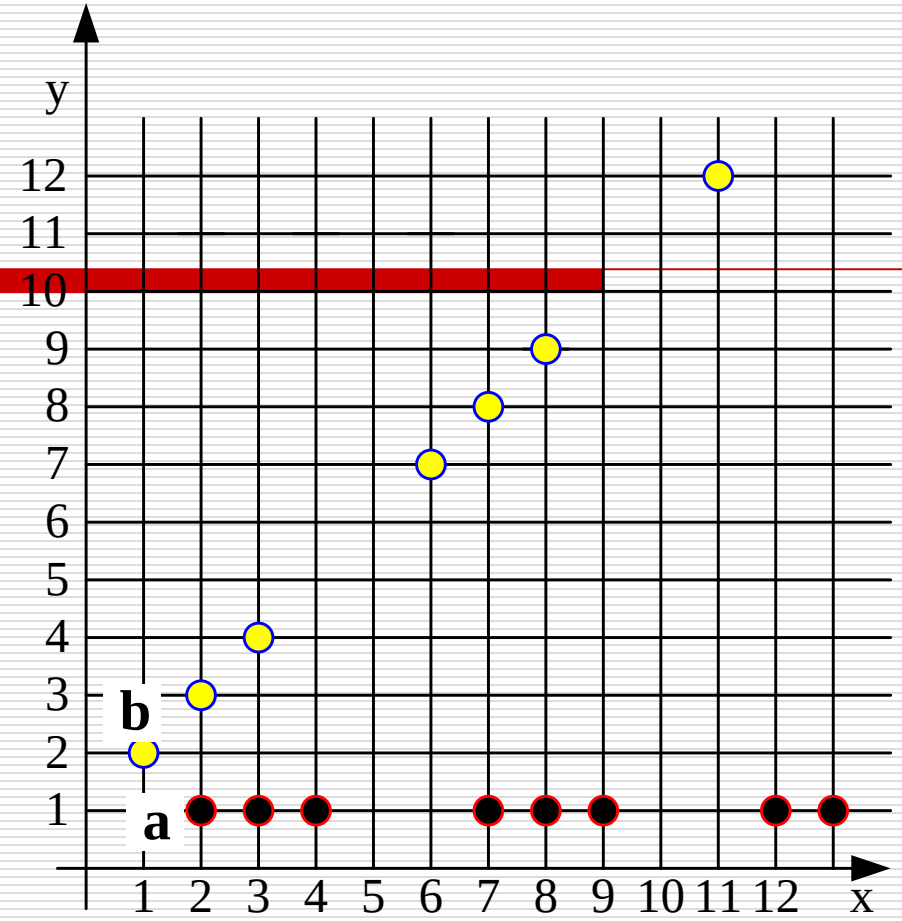
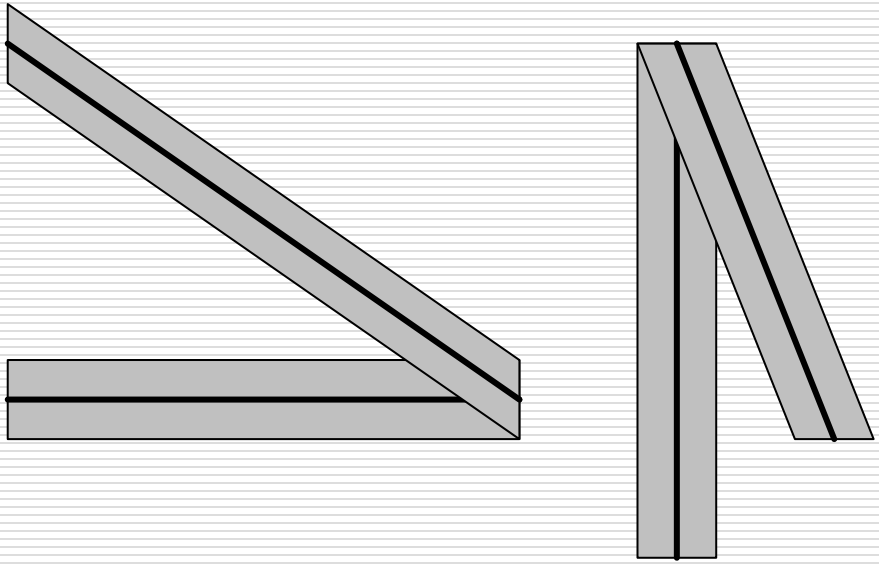



图3.22 相同数目象素显示的不等长划线

- 可根据线的斜率来调整实心段和中间空白段的像素数目。

# 线型与线宽——线宽

- 线刷子：垂直刷子、水平刷子



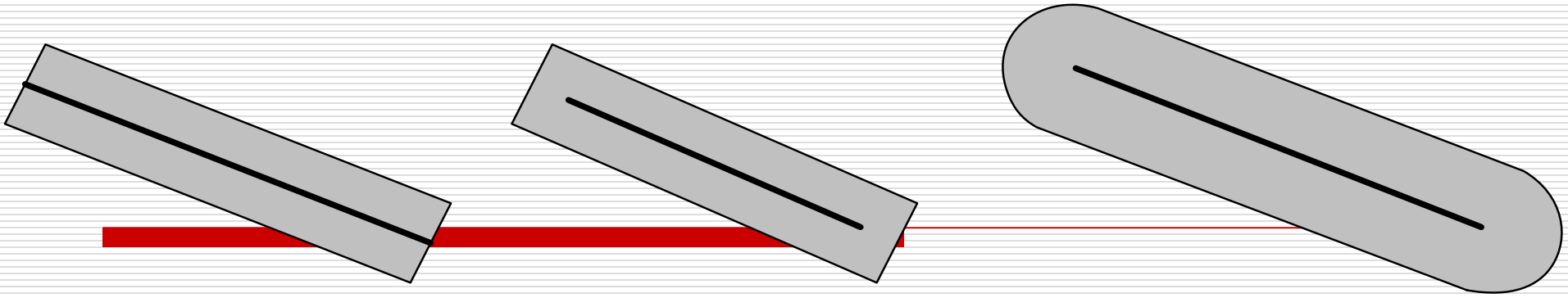
(a) (b)

**图3.23 线刷子**

# 线型与线宽——线型

- 实现简单、效率高。
- 斜线与水平(或垂直)线不一样粗。
- 当线宽为偶数个象素时，线的中心将偏移半个象素。
- 利用线刷子生成线的始末端总是水平或垂直的，看起来不太自然。

  解决：添加“线帽（line cap）”



(a)方帽　　(b)凸方帽　　(c)圆帽

**图3.24 线帽**

- 方帽：调整端点位置，使粗线的显示具有垂直于线段路径的正方形端点。
- 凸方帽：简单将线向两头延伸一半线宽并添加方帽。
- 圆帽：通过对每个方帽添加一个填充的半圆得到。

# 线型与线宽——线型

- 当比较接近水平的线与比较接近垂直的线汇合时，汇合处外角将有缺口。

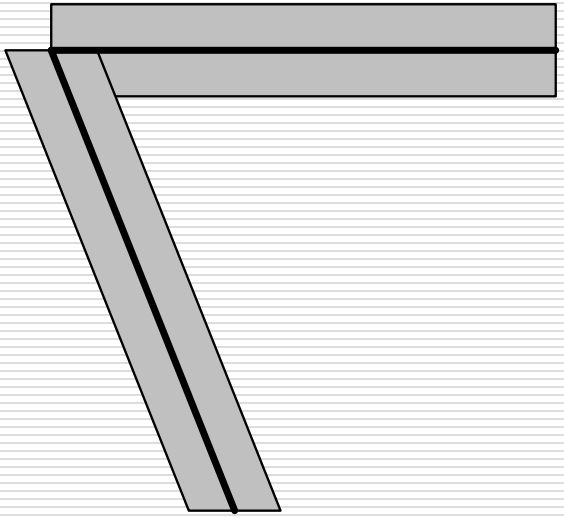
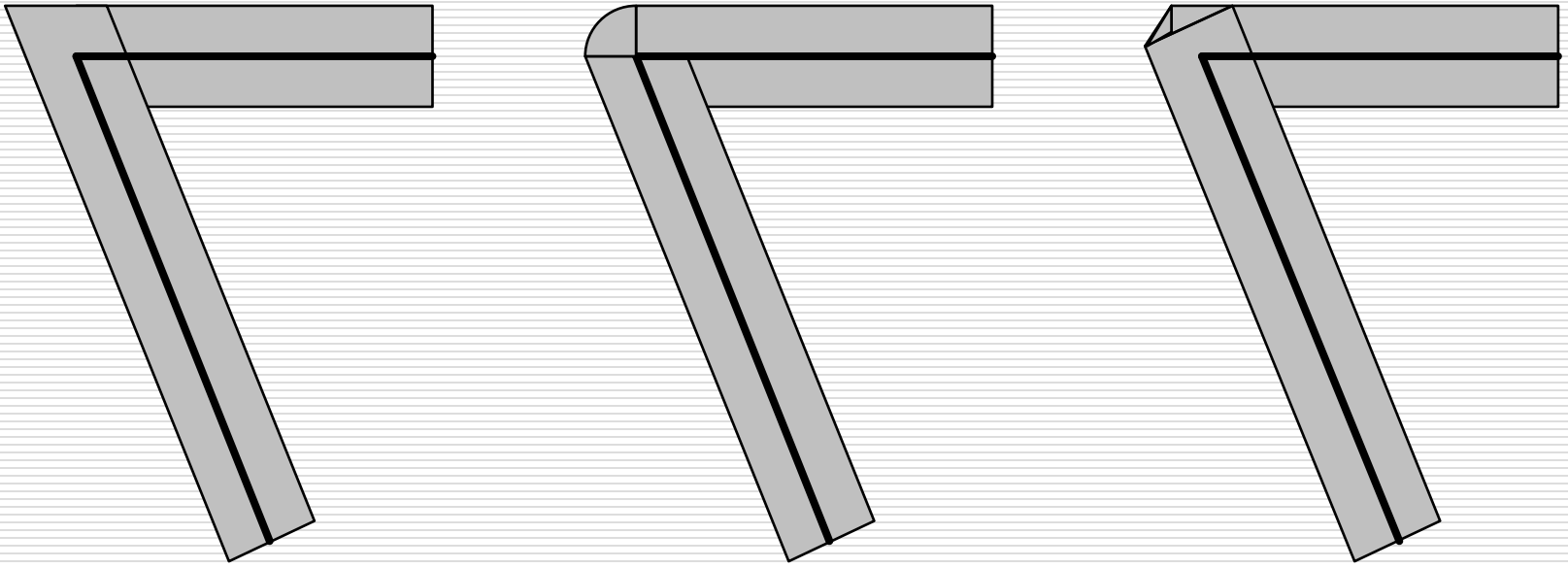


图3.25 线刷子产生的缺口

- 解决：斜角连接（miter join）、圆连接（round join）、斜切连接（bevel join）



(a)斜角连接　(b)圆连接　(c)斜切连接

**图3.26 线刷子产生的缺口**

# 方刷子

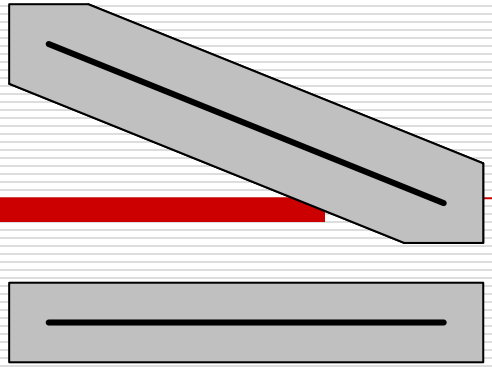


图3.27 方刷子

特点：

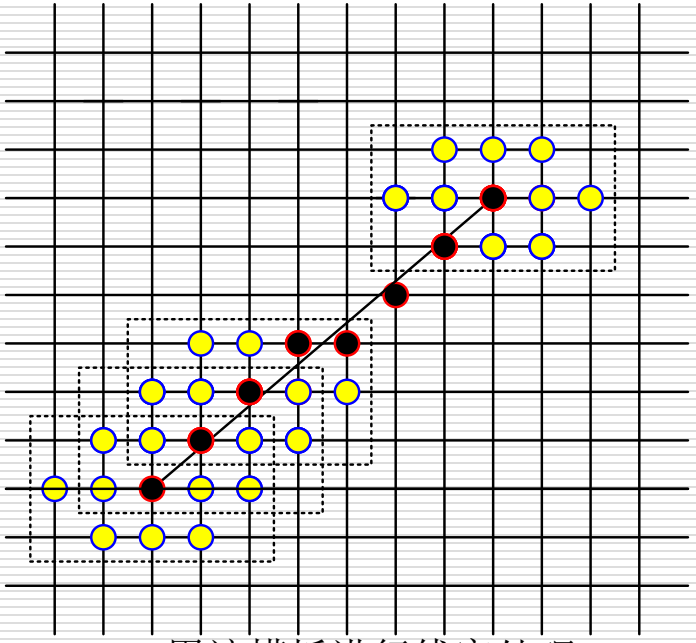
- 方刷子绘制的线条（斜线）比用线刷子所绘制的线条要粗一些。
- 方刷子绘制的斜线与水平（或垂直）线不一样粗。
- 方刷子绘制的线条自然地带有一个“方线帽”。

# 线型与线宽——线型

- 区域填充
- 改变刷子形状

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

(a)象素模板



(b)用该模板进行线宽处理

**图3.28 利用像素模板进行线宽处理**

# 线型与线宽——曲线的线型和线宽

- 线型：可采用像素模板的方法。

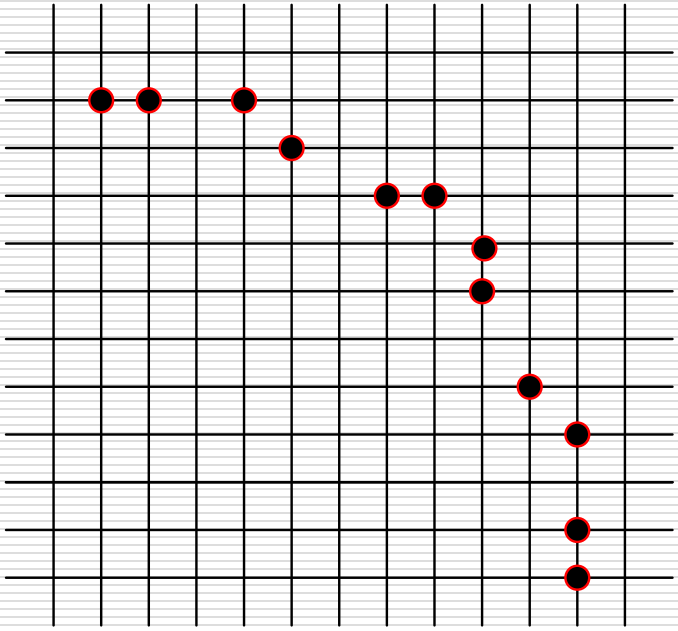


图3.29 利用模板110进行圆的线型处理

# 线型与线宽——曲线的线型和线宽

- 从一个八分象限转入下一个八分象限时要交换象素的位置，以保持划线长度近似相等。
- 在沿圆周移动时调整画每根划线的像素数目以保证划线长度近似相等。
- 改进：可以采用沿等角弧画像素代替用等长区间的像素模板来生成等长划线。

# 线型与线宽——曲线的线型和线宽

线宽：

- 线刷子：经过曲线斜率为1和－1处，必须切换刷子。

  曲线接近水平与垂直的地方，线条更粗。

- 方刷子：接近水平垂直的地方，线条更细

  要显示一致的曲线宽度可通过旋转刷子方向以使其在沿曲线移动时与斜率方向一致

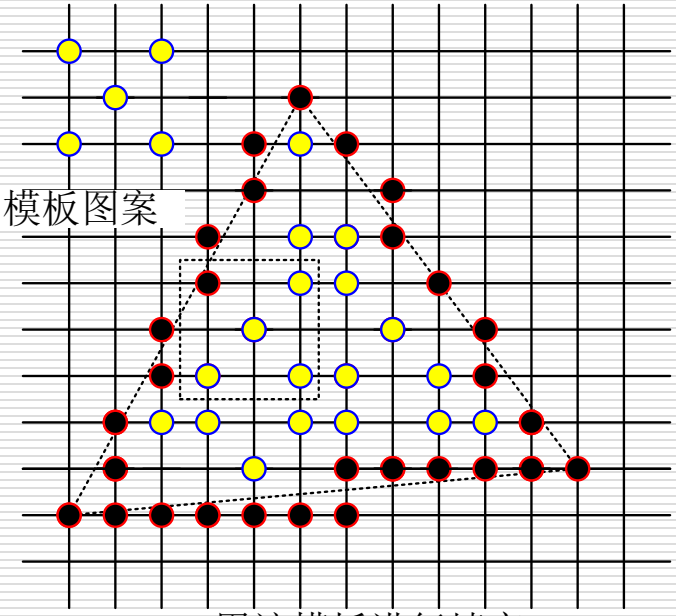
- 圆弧刷子
- 采用填充的办法。

# 区域填充属性

- 区域填充属性选择包括颜色、图案和透明度。

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

(a)图案模板位图




(b)用该模板进行填充

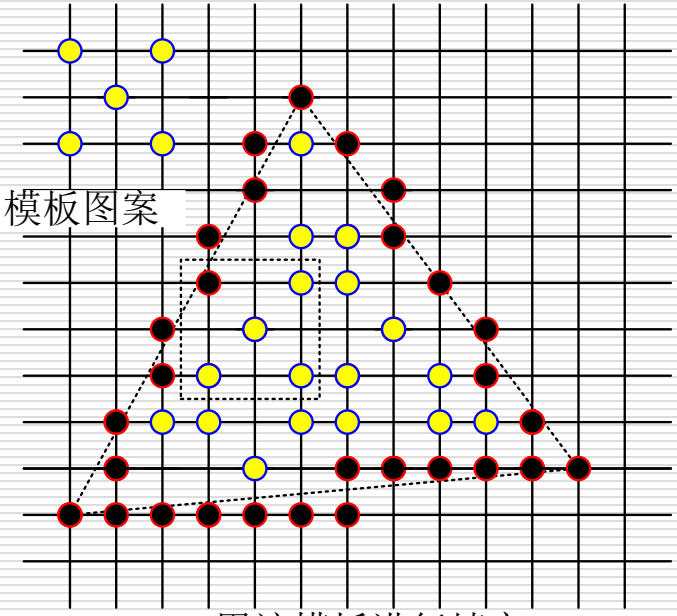
**图3.30 利用图案模板进行三角形的填充**

# 区域填充属性

- 确定区域与模板之间的位置关系（对齐方式）

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

(a)图案模板位图




(b)用该模板进行填充

**图3.31 利用图案模板进行三角形的填充**

# 3.4 反走样

- 用离散量表示连续量引起的失真，就叫做走样（Aliasing）。

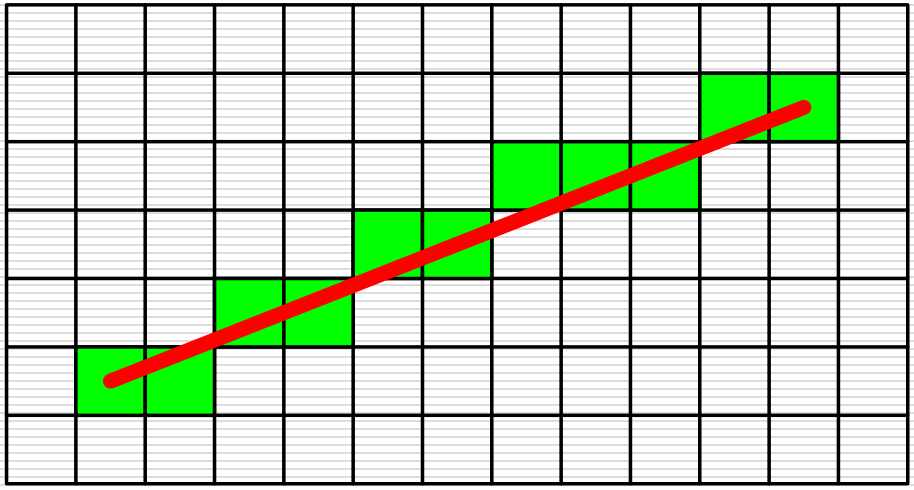


图3.32 绘制直线时的走样现象

# 反走样

产生原因：

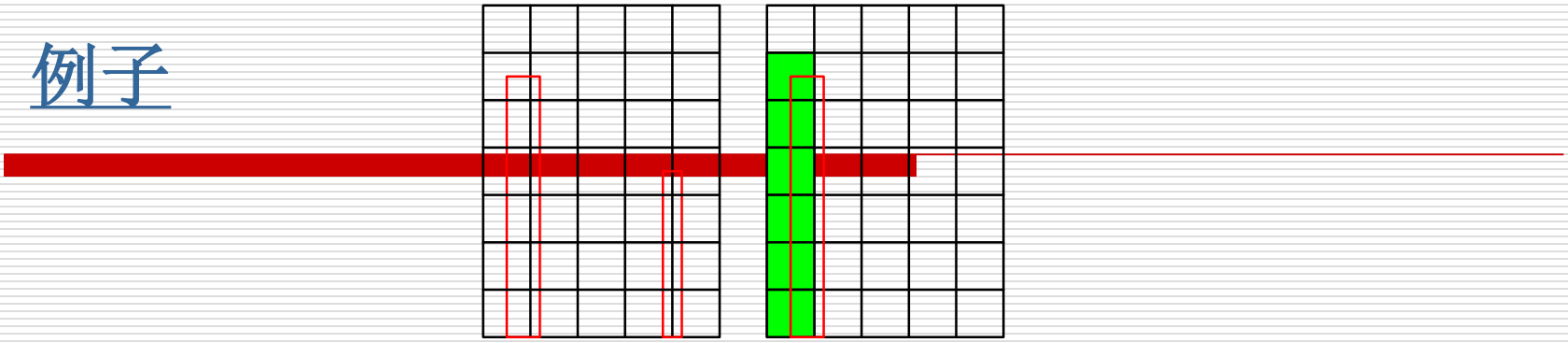
数学意义上的图形是由无限多个连续的、面积为零的点构成；但在光栅显示器上，用有限多个离散的，具有一定面积的象素来近似地表示他们。

# 反走样

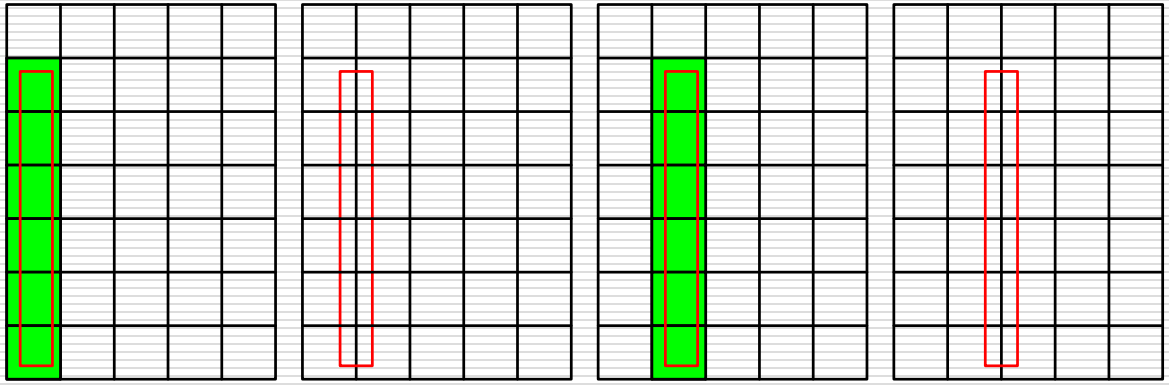
走样现象：

- 一是光栅图形产生的阶梯形。
- 一是图形中包含相对微小的物体时，这些物体在静态图形中容易被丢弃或忽略，在动画序列中时隐时现，产生闪烁。

# 例子



(a)需显示的矩形　(b)显示结果



(a)显示　(b)不显示　(c)显示　(d)不显示

**图3.33 丢失细节与运动图形的闪烁**

- 用于减少或消除这种效果的技术，称为反走样（antialiasing，图形保真）。
- 一种简单方法：

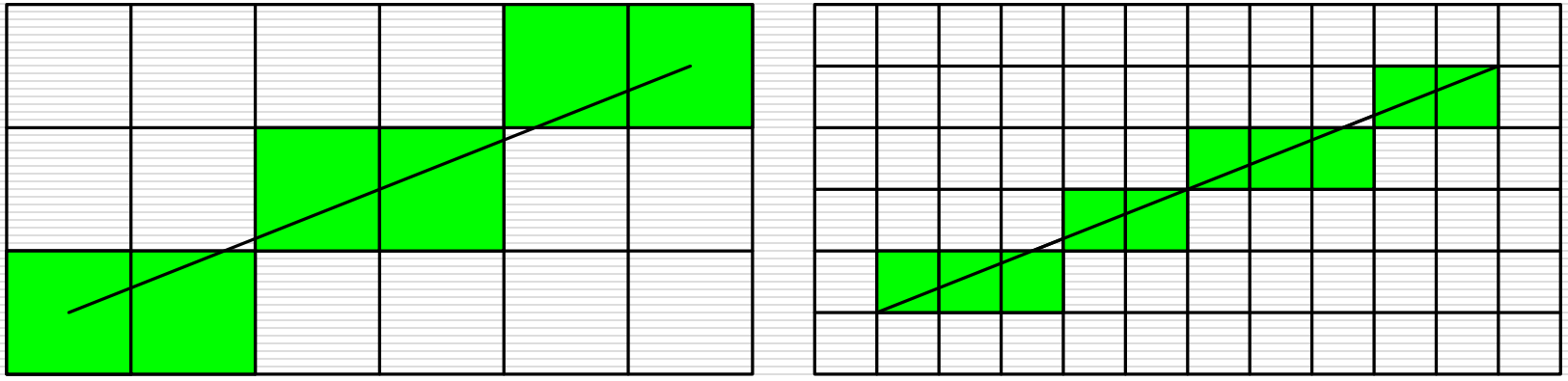


图3.34 分辨率提高一倍，阶梯程度减小一倍

- 过取样（supersampling），或后滤波
- 区域取样（area sampling），或前滤波

# 反走样——过取样（super sampling）

- 过取样：在高于显示分辨率的较高分辨率下用点取样方法计算，然后对几个象素的属性进行平均得到较低分辨率下的象素属性。

## 简单过取样

在x，y方向把分辨率都提高一倍，使每个象素对应4个子象素，然后扫描转换求得各子象素的颜色亮度，再对4个象素的颜色亮度进行平均，得到较低分辨率下的象素颜色亮度。

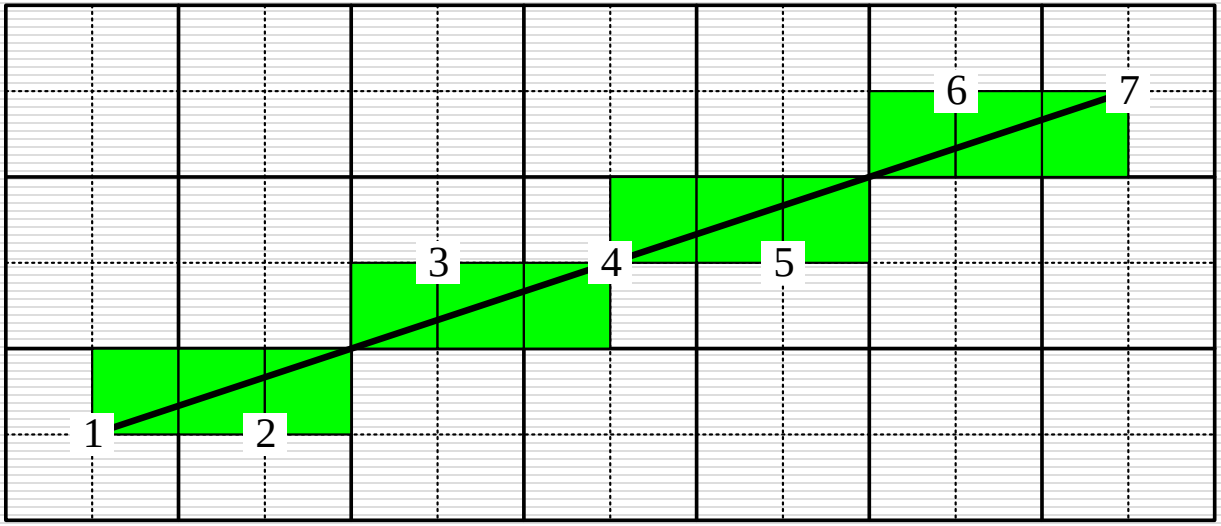
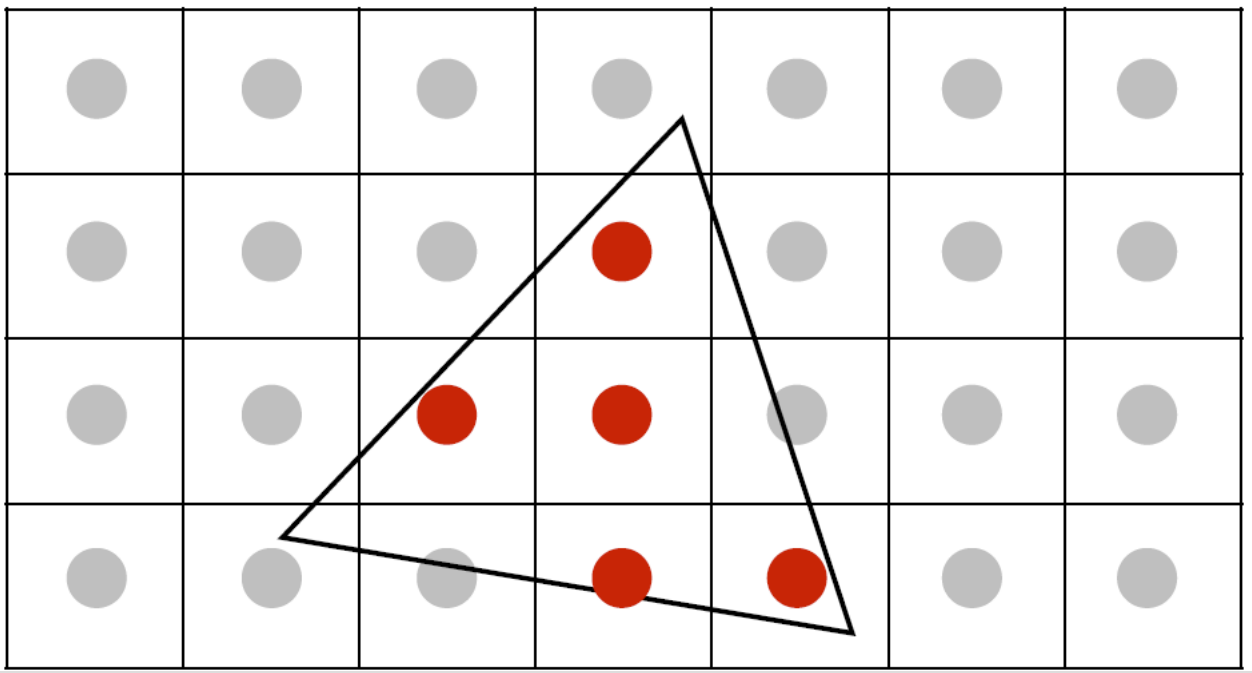



图3.35 简单的过取样方式

# 过取样

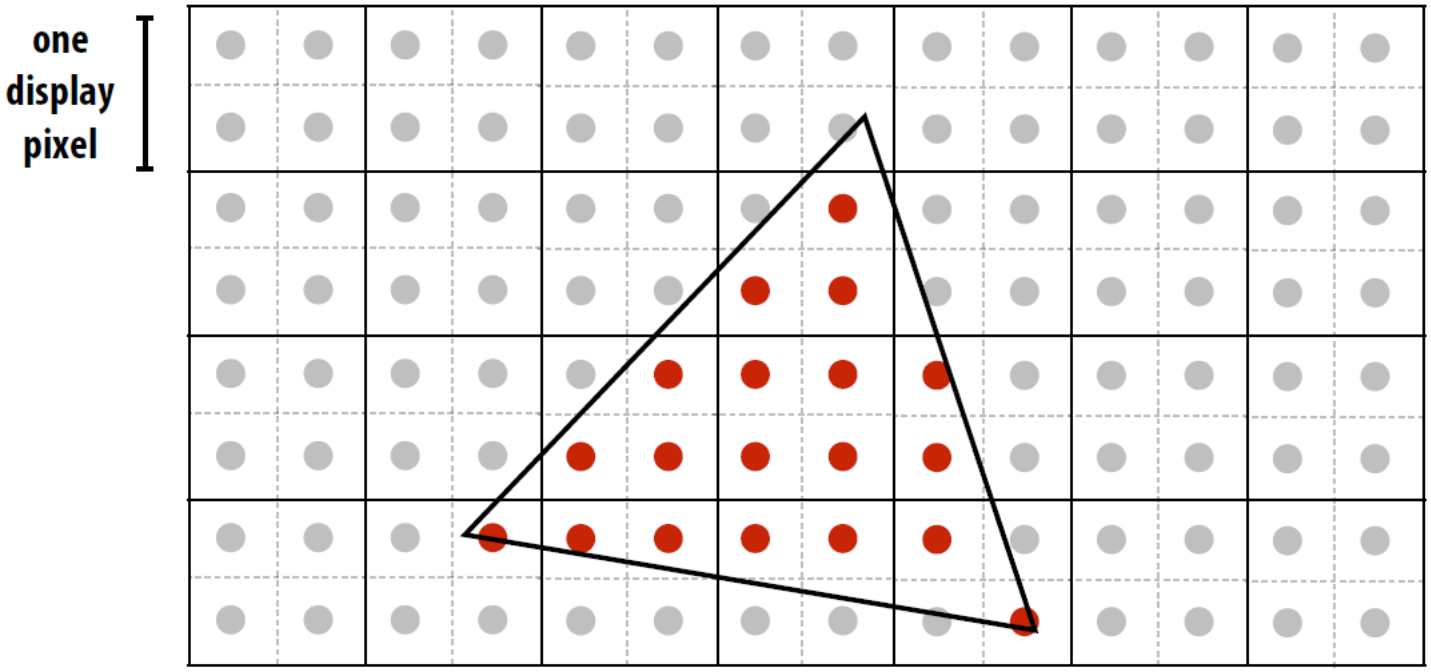
## 点取样：每个像素仅取一个采样点



# Supersampling: step 1

## Take NxN samples in each pixel

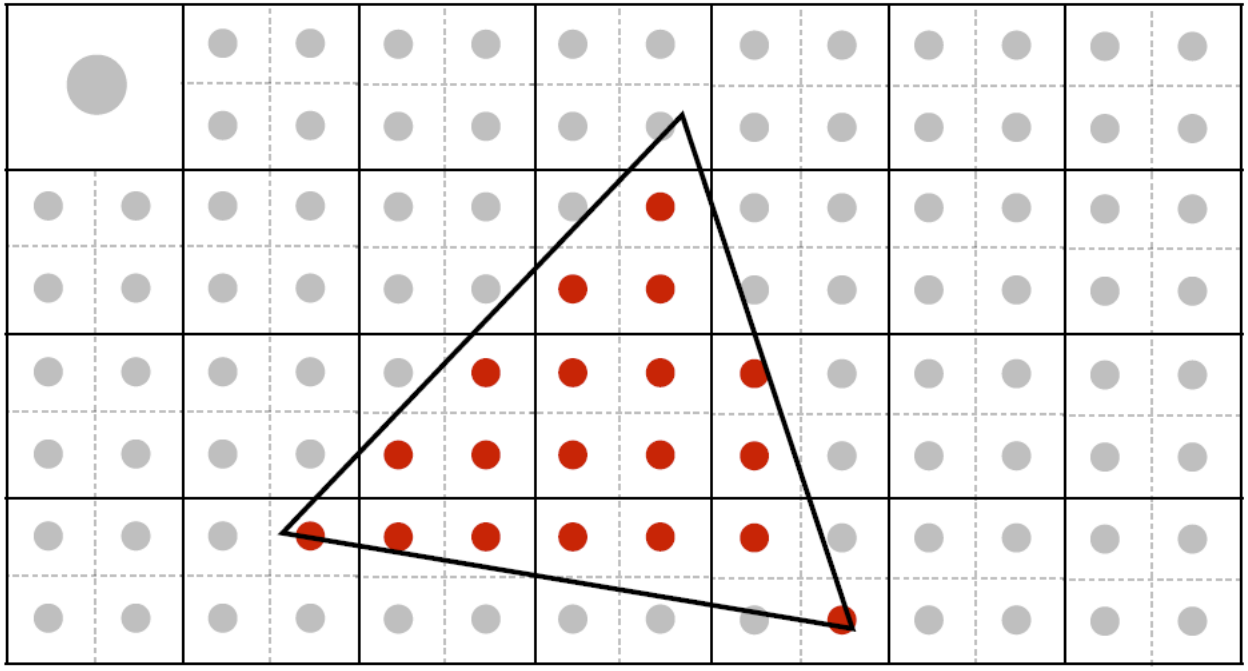
(but… how do we use these samples to drive a display, since there are four times more samples than display pixels!)




2x2 supersampling

# Supersampling: step 2

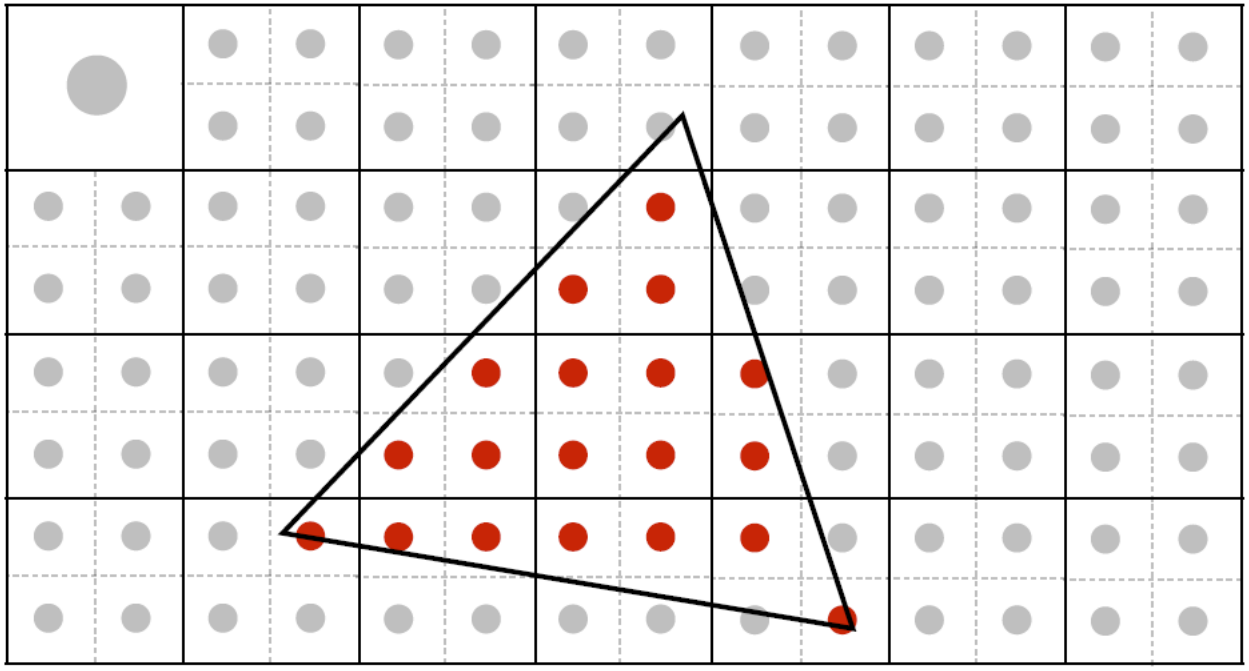
Average the NxN samples "inside" each pixel



Averaging down

# Supersampling: step 2

## Average the NxN samples "inside" each pixel

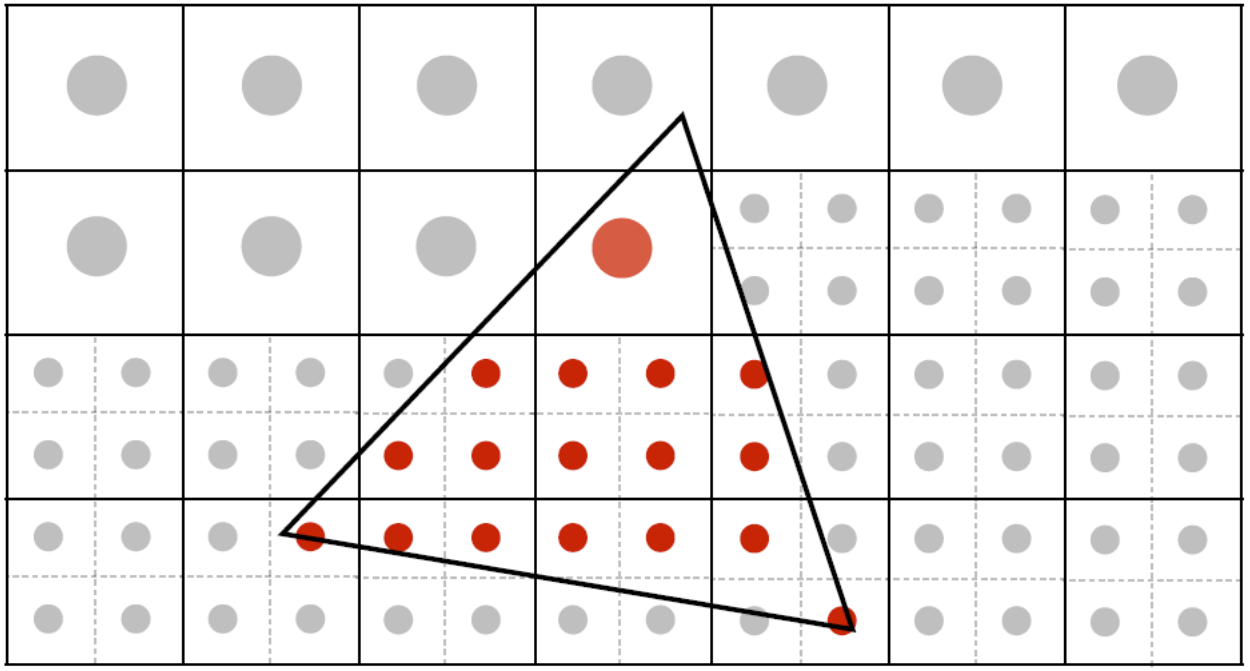


Averaging down

过取样

# Supersampling: step 2

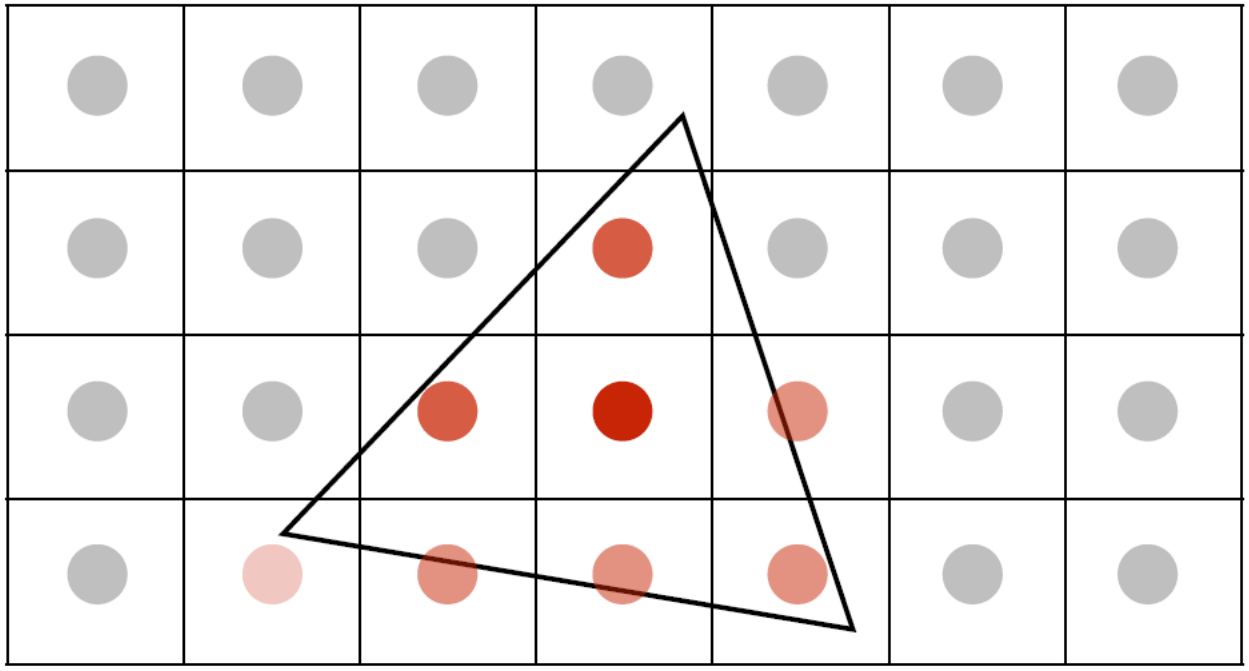
Average the NxN samples "inside" each pixel



Averaging down

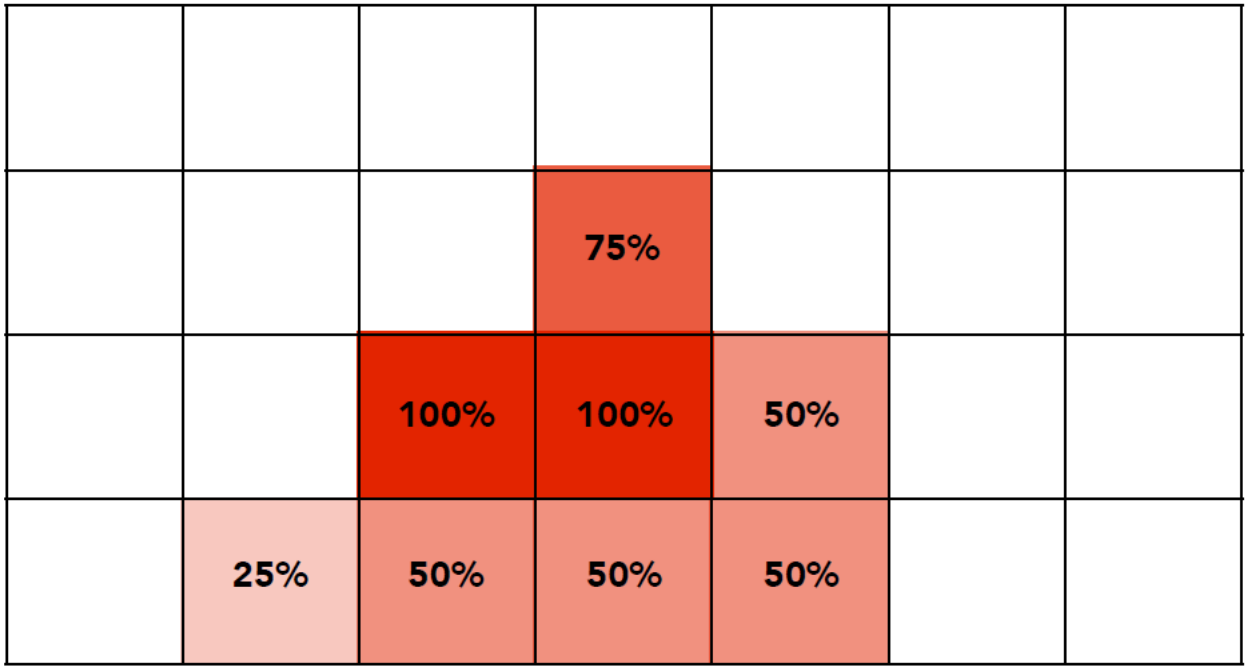
# Supersampling: step 2

## Average the NxN samples "inside" each pixel

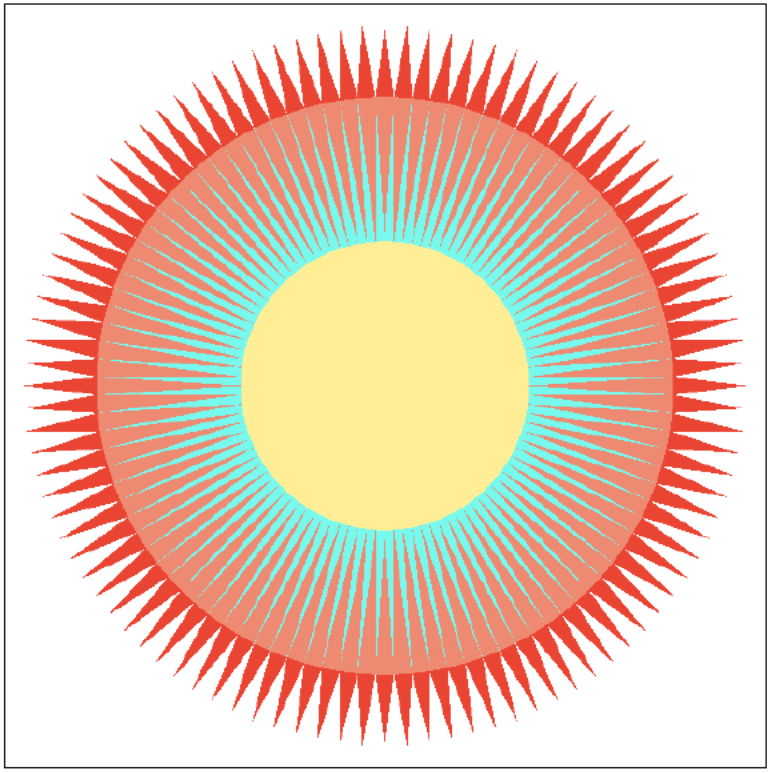


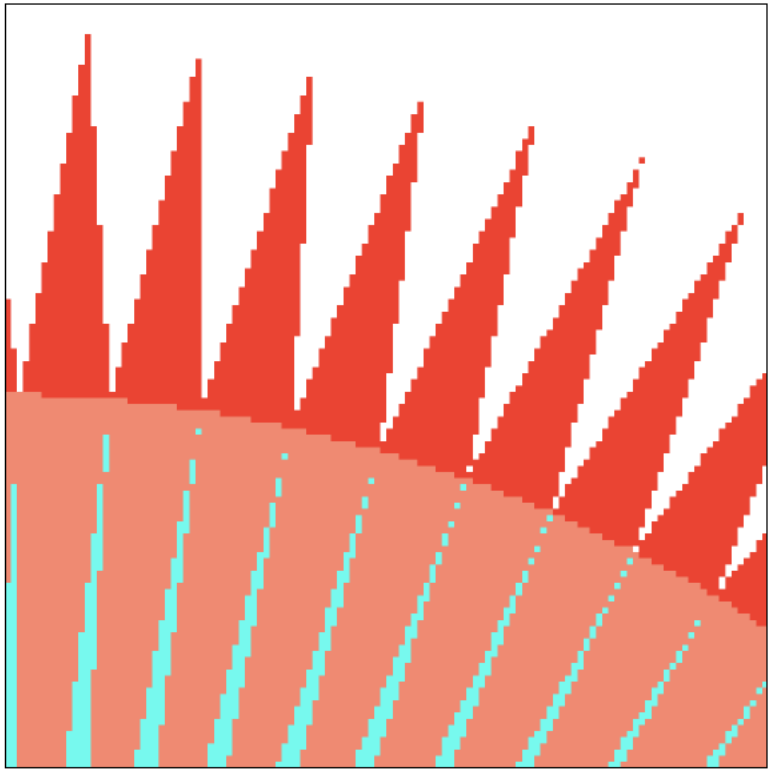
# Supersampling: result

**This is the corresponding signal emitted by the display**



# Point sampling

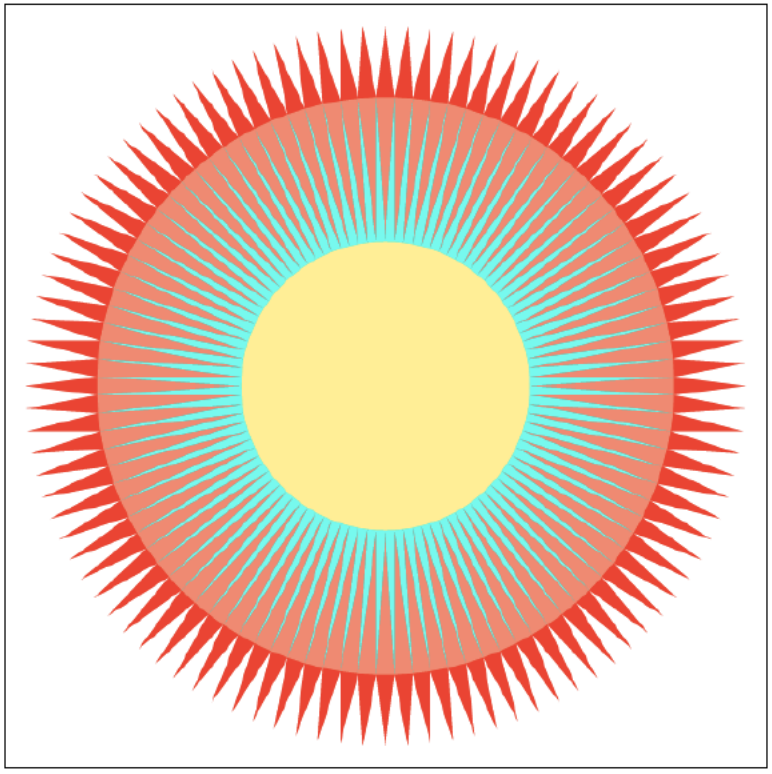


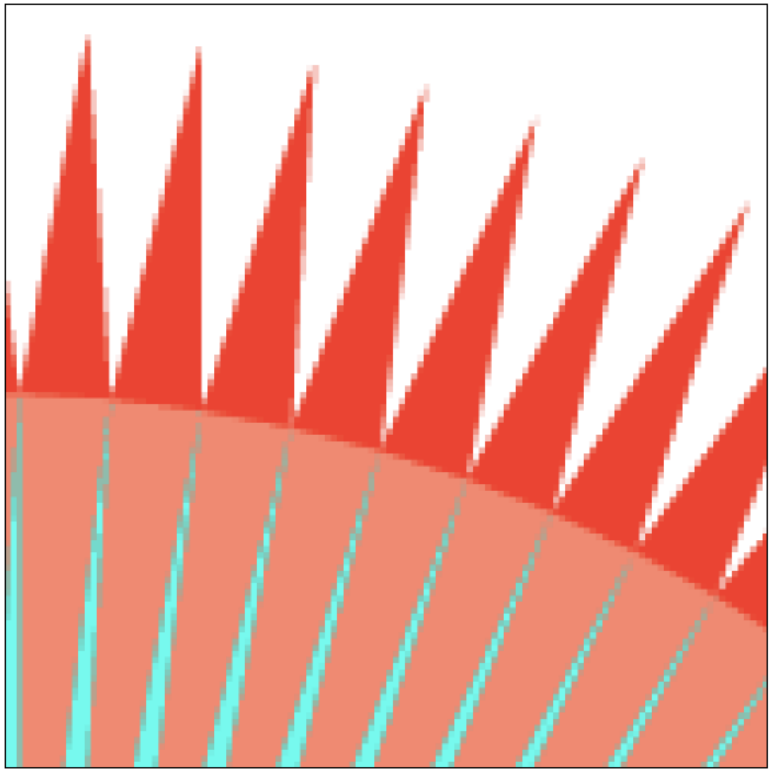


**One sample per pixel**

Stanford CS248, Winter 2020

# 4x4 supersampling + downsampling





**Pixel value is average of 4x4 samples per pixel**

- **重叠过取样**：为了得到更好的效果，在对一个像素点进行着色处理时，不仅仅只对其本身的子像素进行采样，同时对其周围的多个像素的子像素进行采样，来计算该点的颜色属性。

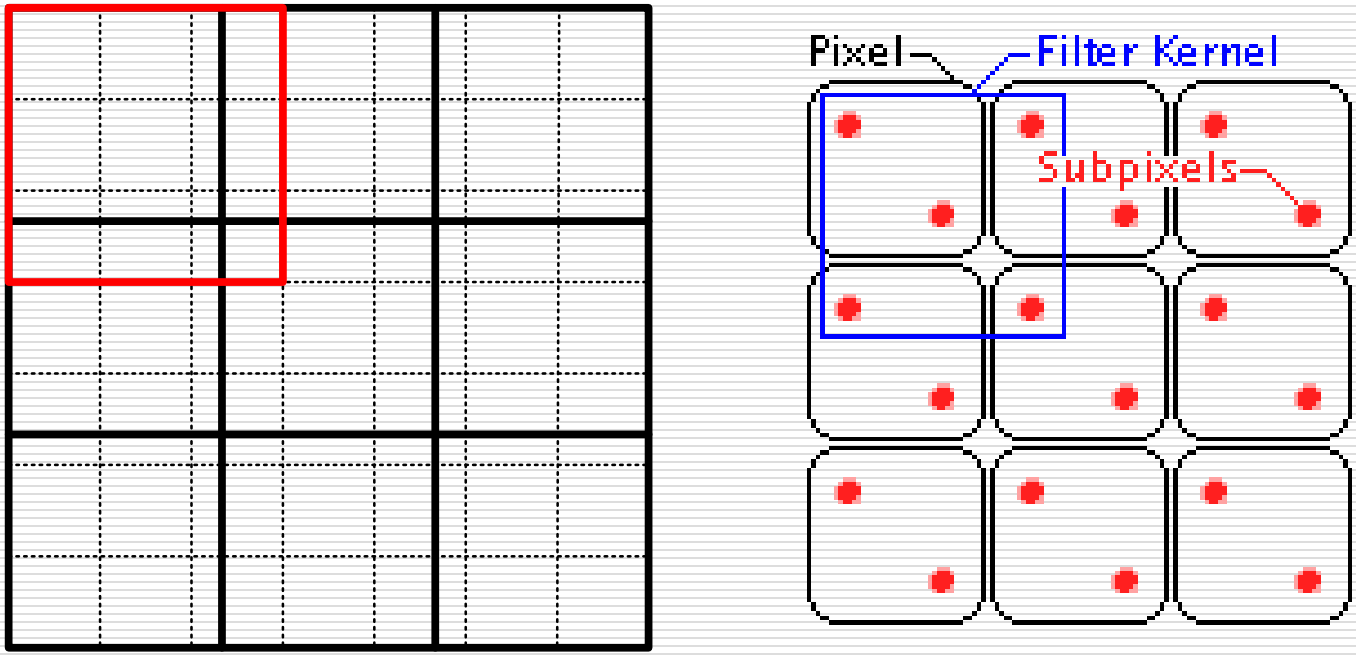



图3.36 重叠过取样

- 基于加权模板的过取样：前面在确定像素的亮度时，仅仅是对所有子像素的亮度进行简单的平均。更常见的做法是给接近像素中心的子像素赋予较大的权值，即对所有子像素的亮度进行加权平均。

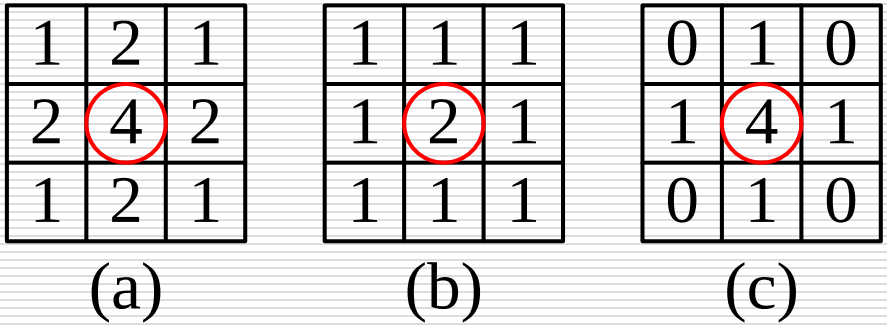



图3.37 常用的加权模板

# 反走样——简单的区域取样

- 在整个像素区域内进行采样，这种技术称为区域取样。又由于像素的亮度是作为一个整体被确定的，不需要划分子像素，故也被称为前置滤波。

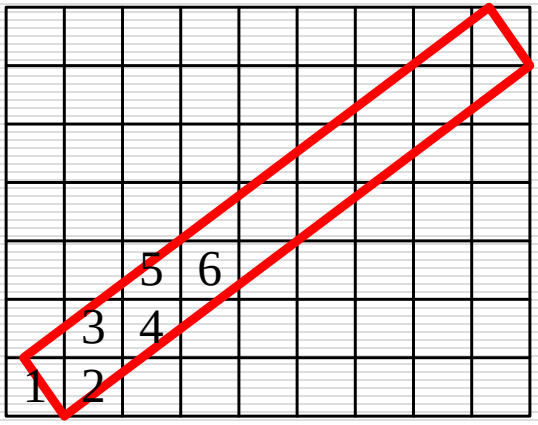



图3.38 有宽度的直线段

# 反走样——简单的区域取样

如何计算直线段与像素相交区域的面积？

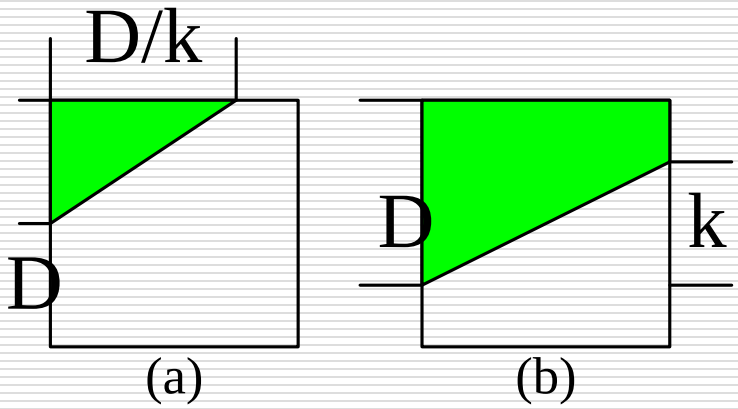



**图3.39 重叠区域面积的计算**

# 反走样——简单的区域取样

- 可以利用一种求相交区域的近似面积的离散计算方法：

  (1)将屏幕像素分割成n个更小的子像素，

  (2)计算中心落在直线段内的子像素的个数m，

  (3)m/n为线段与像素相交区域面积的近似值。

- 直线段对一个像素亮度的贡献与两者重叠区域的面积成正比。
- 相同面积的重叠区域对像素的贡献相同。

# 反走样——加权区域取样

- 过取样中，我们对所有子像素的亮度进行简单平均或加权平均来确定像素的亮度。
- 在区域取样中，我们使用覆盖像素的连续的加权函数（Weighting Function）或滤波函数(Filtering Function)来确定像素的亮度。

## 加权区域取样原理

加权函数W(x,y)是定义在二维显示平面上的函数。对于位置为(x,y)的小区域dA来说，函数值W(x,y)（也称为在(x,y)处的高度）表示小区域dA的权值。将加权函数在整个二维显示图形上积分，得到具有一定体积的滤波器（Filter），该滤波器的体积为1。将加权函数在显示图形上进行积分，得到滤波器的一个子体，该子体的体积介于0到1之间。用它来表示像素的亮度。

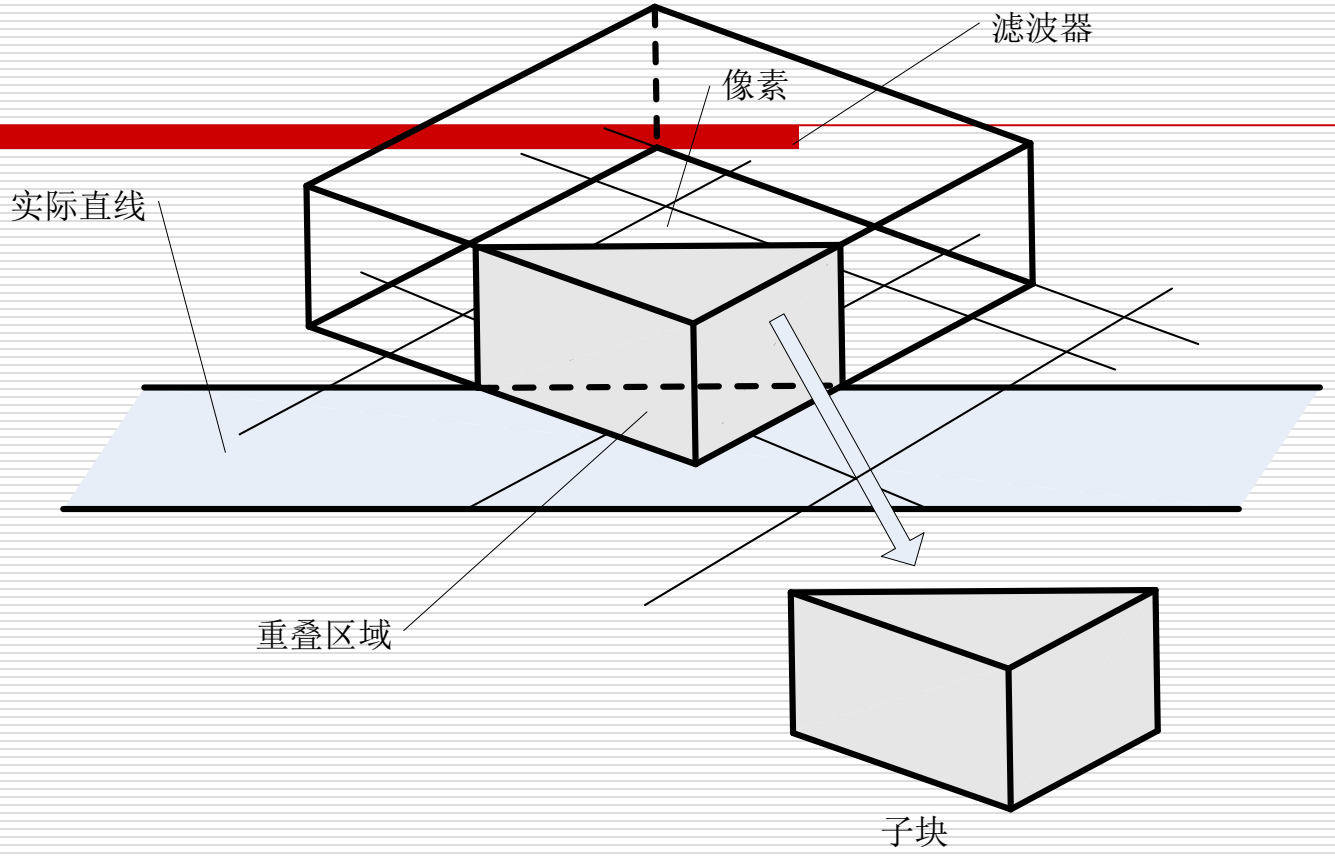
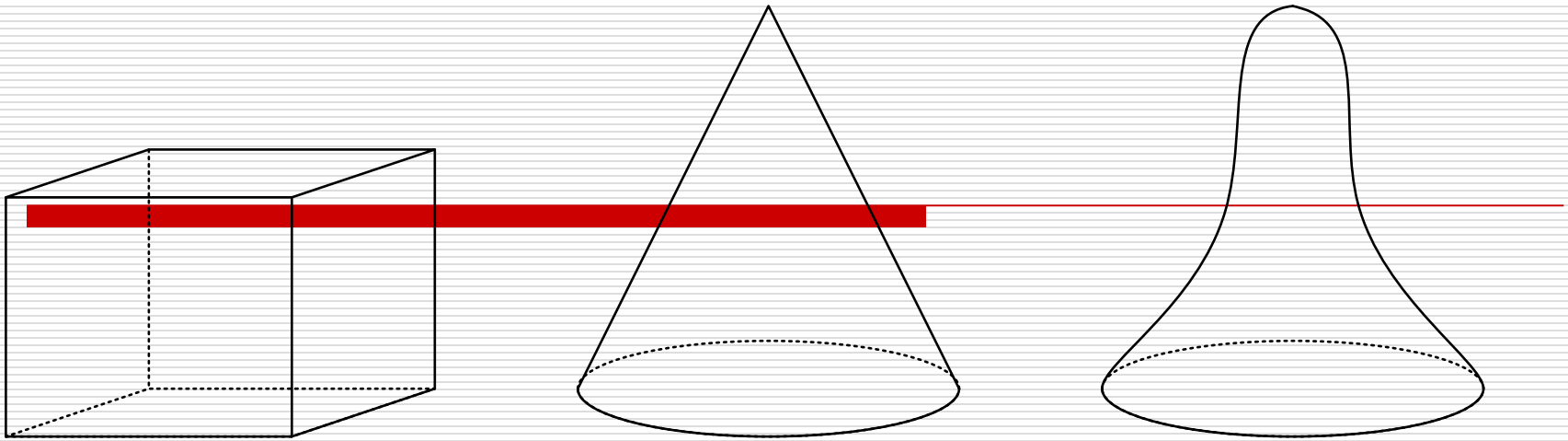
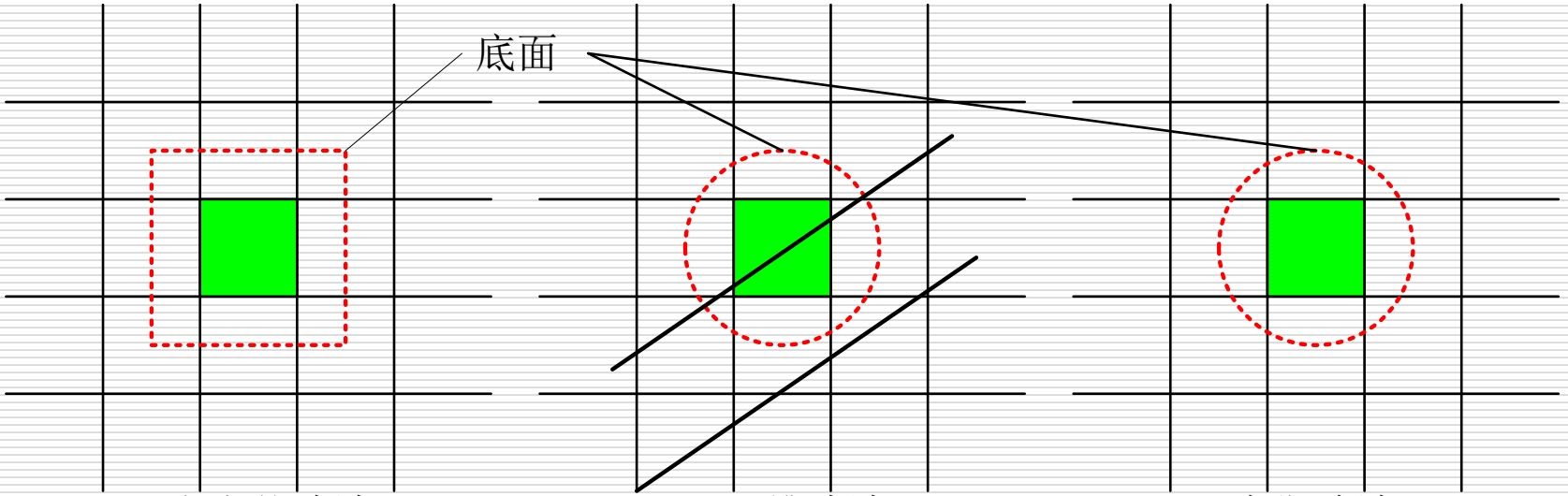



**图3.40　盒式滤波器的加权区域取样**






(a)立方体滤波　(b)圆锥滤波　(c)高斯滤波

**图3.41 常用的滤波函数**

# 反走样——加权区域取样

特点：

- 接近理想直线的象素将被分配更多的灰度值；
- 相邻两个象素的滤波器相交，有利于缩小直线条上相邻象素的灰度差。

# 3.5 在OpenGL中绘图

- 点的绘制
- 直线的绘制
- 多边形面的绘制
- **OpenGL中的字符函数**
- **OpenGL中的反走样**



# 点的绘制

- 点的绘制

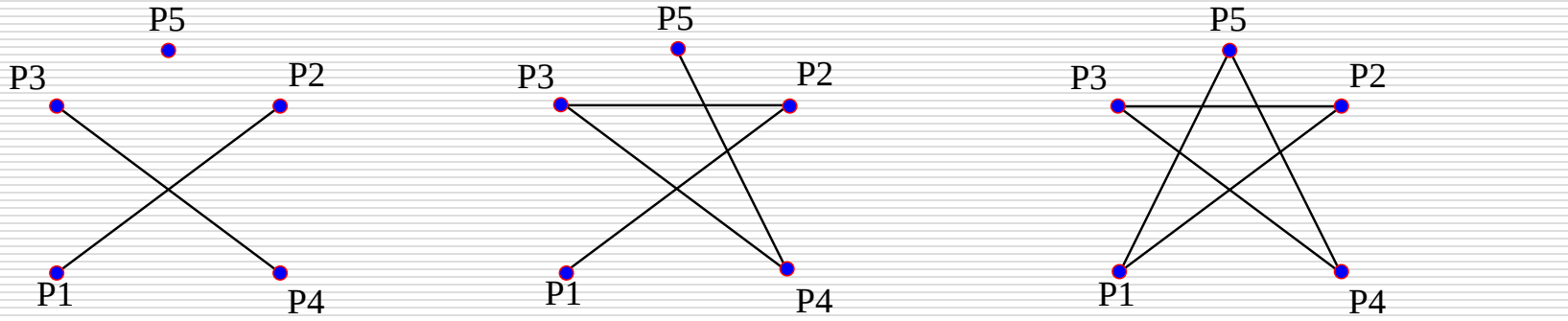
```
glBegin(GL_POINTS);
    glVertex3f(0.0f, 0.0f, 0.0f);
    glVertex3f(10.0f, 0.0f, 0.0f);
glEnd();
```

- 点的属性（大小）

```
void glPointSize(GLfloat size);
```

# 直线的绘制

- 直线的绘制模式
  - GL_LINES
  - GL_LINE_STRIP
  - GL_LINE_LOOP




（a）GL_LINES画线模式　（b）GL_LINE_LOOP画线模式　（c）GL_LINE_STRIP画线模式

图3.42 OpenGL画线模式

# 直线的绘制

- 直线的属性
  - 线宽

    **void glLineWidth(GLfloat width)**
  - 线型

    **glEnable(GL_LINE_STIPPLE);**

    **glLineStipple(GLint factor，GLushort pattern);**

# 直线的绘制

模式：0X00FF = 255

二进制表示：0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1

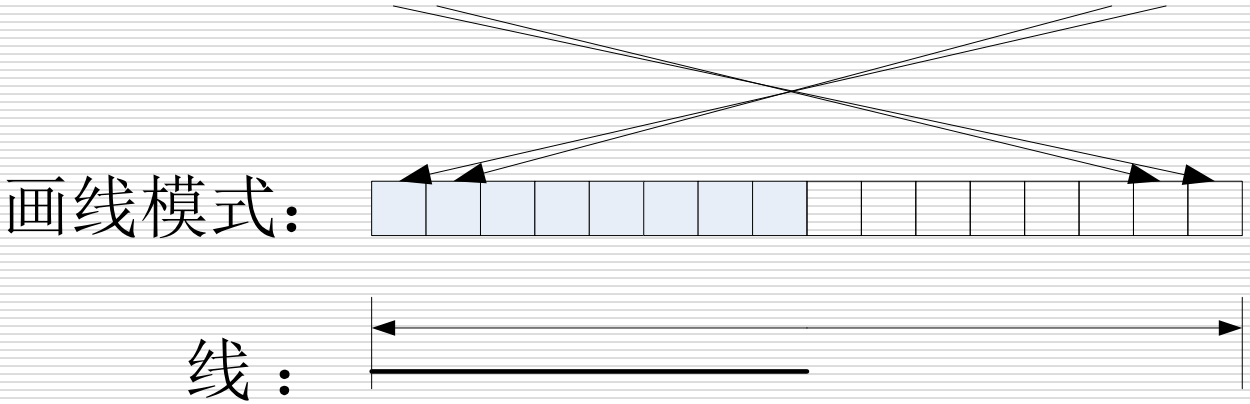



**图3.43　画线模式用于构造线段**

# 多边形面的绘制

- 三角形面的绘制
  - GL_TRIANGLES
  - GL_TRIANGLE_STRIP
  - GL_TRIANGLE_FAN
- 四边形面的绘制
  - GL_QUADS
  - GL_QUADS_STRIP
- 多边形面的绘制（GL_POLYGON）

# 多边形面的绘制

- 多边形面的绘制规则
  - 所有多边形都必须是平面的。
  - 多边形的边缘决不能相交，而且多边形必须是凸的。
- 解决：对于非凸多边形，可以把它分割成几个凸多边形（通常是三角形），再将它绘制出来。

# 多边形面的绘制

- 问题：轮廓图形状态会看到组成大表面的所有小三角形。处理OpenGL提供了一个特殊标记来处理这些边缘，称为边缘标记。

  **glEdgeFlag（True）**

  **glEdgeFlag（False）**

# 多边形面的属性

- 多边形面的正反属性（绕法）

  指定顶点时顺序和方向的组合称为“绕法”。绕法是任何多边形图元的一个重要特性。一般默认情况下，**OpenGL**认为逆时针绕法的多边形是正对着的。

  **glFrontFace(GL_CW);**

# 多边形面的属性

- 多边形面的颜色
  - **glShadeModel(GL_FLAT)** 用指定多边形最后一个顶点时的当前颜色作为填充多边形的纯色，唯一例外是**GL_POLYGON**图元，它采用的是第一个顶点的颜色。
  - **glShadeModel(GL_SMOOTH)** 从各个顶点给三角形投上光滑的阴影，为各个顶点指定的颜色之间进行插值。

# 多边形面的属性

- 多边形面的显示模式

  **glPolygonMode(GLenum face，GLenum mode)；**

  - 参数**face**用于指定多边形的哪一个面受到模式改变的影响。
  - 参数**mode**用于指定新的绘图模式。

# 多边形面的属性

- 多边形面的填充

多边形面既可以用纯色填充，也可以用32×32的模板位图来填充。

```
void glPolygonStipple(const GLubyte *mask);
glEnable(GL_POLYGON_STIPPLE);
```

# 多边形面的属性

- 多边形面的法向量
  - 法向量是垂直于面的方向上点的向量，它确定了几何对象在空间中的方向。
  - 在OpenGL中，可以为每个顶点指定法向量。

```
void glNormal3{bsidf} ( TYPE nx, TYPE ny, TYPE nz);
void glNormal3{bsidf}v (const TYPE* v);
```

# OpenGL中的字符函数

- GLUT位图字符

  void glutBitmapCharacter(void *font, int character);

- GLUT矢量字符

  void glutStrokeCharacter(void *font, int character);

# OpenGL中的反走样

- 启用反走样

```
glEnable(primitiveType);
```

- 启用OpenGL颜色混和并指定颜色混合函数

```
glEnable(GL_BLEND);
glBlendFunc(GL_SCR_ALPHA,
GL_ONE_MINUS_SRC_ALPHA);
```

# OpenGL中的反走样

- 颜色混和函数用于计算两个相互重叠的对象的颜色。
- 在RGBA颜色模式（A表示透明度）中，已知源像素的颜色值为（$S_r$，$S_g$，$S_b$，$S_a$），目标像素的颜色值为（$D_r$，$D_g$，$D_b$，$D_a$），颜色混合后像素的颜色为：

  （$R_S.S_r + R_D.D_r$，$G_S.S_g + G_D.D_g$，$B_S.S_b + B_D.D_b$，$A_S.S_a + A_D.D_a$）

# OpenGL中的反走样

- 定义混合因子

**void glBlendFunc(GLenum srcfactor, GLenum destfactor);**

表3-1　源混和因子和目标混合因子

| 常量 | RGB混合因子 | Alpha混合因子 |
|---|---|---|
| GL_ZERO | (0，0，0) | 0 |
| GL_ONE | (1，1，1) | 1 |
| GL_SRC_COLOR | （$R_S$, $G_S$，$B_S$） | $A_S$ |
| GL_ONE_MINUS_SRC_COLOR | (1，1，1)－（$R_S$, $G_S$，$B_S$） | $1-A_S$ |
| GL_DST_COLOR | （$R_D$，$G_D$，$B_D$） | $A_D$ |
| GL_ONE_MINUS_DST_COLOR | (1，1，1)－（$R_D$，$G_D$，$B_D$） | $1-A_D$ |
| GL_SCR_ALPHA | ($A_S$，$A_S$，$A_S$) | $A_S$ |
| GL_ONE_MINUS_SRC_ ALPHA | (1，1，1)－($A_S$，$A_S$，$A_S$) | $1-A_S$ |
| GL_DST_ALPHA | ($A_D$，$A_D$，$A_D$) | $A_D$ |
| GL_ONE_MINUS_DST_ ALPHA | (1，1，1)－($A_D$，$A_D$，$A_D$) | $1-A_D$ |