第2章　光栅扫描图形学

光栅扫描图形设备需要采用专门算法来生成图形、画直线或曲线，以及填充多边形以产生实区域效果，本章将研究与这些问题相关的算法。

2.1　直线生成算法

由于可以把阴极射线管光栅显示器看作是由离散可发光的有限区域单元（像素）组成的矩阵，因此难以直接从一点到另一点画一条直线。确定最佳逼近某直线的像素的过程通常称为光栅化。当和按扫描线顺序绘制图形的过程组合在一起时，它被称为扫描转换。对于水平线、垂直线以及45°线，选择哪些光栅元素是显而易见的，而对于其他方向的直线，像素的选择则较为困难（见图2-1）。

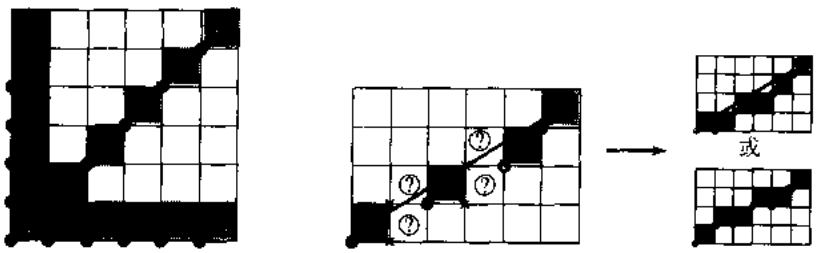


图2-1　直线的光栅化

在讨论具体的画线算法之前，需要先研究这类算法的一般要求，即所生成的直线应具有哪些特征。最起码的要求是所画的直线应该外观笔直，且具有精确的起点和终点。此外，所显示的亮度应沿直线保持不变，且与直线的长度和方向无关。最后，直线的生成速度要快。然而，如同大多数设计准则一样，难以同时完全满足所有这些要求。光栅扫描显示器的本质决定它难以生成完美的直线，也不能保证直线精确通过指定位置的起点和终点，当然特例除外。但是，如果显示器具有足够的分辨率，那么还是可以获得满意的结果的。

只有水平线、垂直线和45°线的亮度沿整个线段不变，而所有其他方向的直线只能产生不均匀的亮度。如图2-1所示，即使对于上述特殊情形，直线的亮度也与其方向有关，例如45°线上像素的有效间距大于垂直线和水平线像素的间距。这使垂直线和水平线较45°线亮。为使不同长度和方向的直线沿着线段具有同等亮度就要使用开方运算，但是这是以像素可用多个亮度等级来表示为前提的。这使运算速度变慢。一般可采用下列折衷方法：即只计算直线长度的近似值，采用整数运算将计算量减到最小，并用增量方法来简化运算。

设计画线算法要考虑的问题还有：端点次序，即光栅化线段P_2P_1与光栅化线段P_1P_2产生的结果相同；端点形状,即控制光栅化线段的端点形状，例如方头、圆头、箭头、叉头等，这关系到线段连接时是否精确吻合和美观；使用多个位平面显示这一性能来控制线段的亮度，使其成为线段斜率和方向的函数；生成宽度大于一个像素的线段，即所谓的粗线(参见[Bres90])；反走样粗线以改进外观；当一系列的短线连接起来构成折线时，保证端点像素不

画两次。下面只讨论简单的画线算法。

2.2 数字微分分析法

实现直线光栅化的方法之一是解直线的微分方程式，即

$$\frac{\mathrm{d}y}{\mathrm{d}x}=\text{常数} \quad \text{或} \quad \frac{\Delta y}{\Delta x}=\frac{y_2-y_1}{x_2-x_1}$$

其有限差分近似解

$$y_{i+1}=y_i+\Delta y$$

$$y_{i+1}=y_i+\frac{y_2-y_1}{x_2-x_1}\Delta x \tag{2-1}$$

式中x_1,y_1和x_2,y_2是所求直线的端点坐标，y_i是直线上某一步的初值。事实上，公式（2-1）表示所求直线上y值的逐次递归关系。此式用于直线光栅化时，称为数字微分分析器（DDA）[⊖]。简单的DDA选择Δx或Δy中的较大者为一个光栅单位。下面是一个适合于所有象限的简单DDA算法。

digital differential analyzer *(DDA) routine for rasterizing a line*

the line end points are (x_1, y_1) *and* (x_2, y_2)*, assumed not equal*

Integer *is the integer function. Note: Many Integer functions are floor functions; i.e., Integer*$(-8.5) = -9$ *rather than* -8*. The algorithm assumes this is the case.*

Sign *returns* $-1, 0, 1$ *for arguments* $< 0, = 0, > 0$*, respectively*

```
    approximate the line length
    if abs(x2 − x1) ≥ abs(y2 − y1) then
        Length = abs(x2 − x1)
    else
        Length = abs(y2 − y1)
    end if
    select the larger of Δx or Δy to be one raster unit
    Δx = (x2 − x1)/Length
    Δy = (y2 − y1)/Length
    round the values rather than truncate, so that center pixel addressing
      is handled correctly
    x = x1 + 0.5
    y = y1 + 0.5
    begin main loop
    i = 1
    while (i ≤ Length)
        setpixel (Integer(x), Integer(y))
        x = x + Δx
        y = y + Δy
        i = i + 1
```

⊖ 数字微分分析器是通过同时在x方向和y方向分别增加与dx和dy成正比的小数值来积分微分方程的机械设备。

end while
finish

算法举例如下。

例2.1　在第一象限使用简单的DDA算法。

考虑从（0, 0）到（5, 5）的直线段。用这个简单DDA算法光栅化该线段。按算法中的步骤计算如下。

初值计算:

$x_1 = 0$
$y_1 = 0$
$x_2 = 5$
$y_2 = 5$
Length $= 5$
$\Delta x = 1$
$\Delta y = 1$
$x = 0.5$
$y = 0.5$

增量执行主循环过程如下:

i	setpixel	x	y
		0.5	0.5
1	(0,0)		
		1.5	1.5
2	(1,1)		
		2.5	2.5
3	(2,2)		
		3.5	3.5
4	(3,3)		
		4.5	4.5
5	(4,4)		
		5.5	5.5

其结果见图2-2。由图可见，所选像素沿直线等距分布，且直线的外形令人满意。显然，端点(0, 0)是精确的，但是端点(5, 5)没有显示，使得线段看起来过短。如果i的初值不是所给的1而是0，那么位于（5, 5）的像素被选中，但是这可能导致意外的结果。如果像素的地址是由其中心的整数坐标表示的，那么，当画一系列连接的线段时，位于（5, 5）的像素会被选中两次；一次是作为线段的终点，另一次是作为下一线段的起点。这一像素会显得亮一些或者呈现与众不同的颜色。　□

下面给出本算法在第三象限的一个实例。

例2.2　简单的第三象限DDA算法。

在第三象限从（0, 0）到（-8,-4）画一线段。其算法步骤如下。

初值计算:

$x_1 = 0$
$y_1 = 0$
$x_2 = -8$
$y_2 = -4$
Length $= 8$
$\Delta x = -1$

$\Delta y = -0.5$
$x = -0.5$
$y = -0.5$

假设使用向下取整函数，增量执行主循环过程如下：

i	setpixel	x	y
		0.5	0.5
1	(0,0)		
		−0.5	0
2	(−1,0)		
		−1.5	0.5
3	(−2,−1)		
		−2.5	−1.0
4	(−3, 1)		
		−3.5	−1.5
5	(−4,−2)		
		−4.5	2.0
6	(−5,−2)		
		−5.5	−2.5
7	(−6,−3)		
		−6.5	−3.0
8	(−7,−3)		
		−7.5	−3.5

其结果如图2-3所示。 □

尽管图2-3的结果还相当可以，但是若采用本算法画（0,0）到（−8,4）以及（8,−4）之间的线段，那么光栅化的直线就会位于实际线段的一侧。如用四舍五入取整代替算法中所用的向下取整函数，结果就不一样了。解决这一问题可以采用更为复杂的慢速算法，或者对位置的精确性作出一些让步。本算法的另一缺点是必须采用浮点运算。下一节将给出一个更适用的算法。

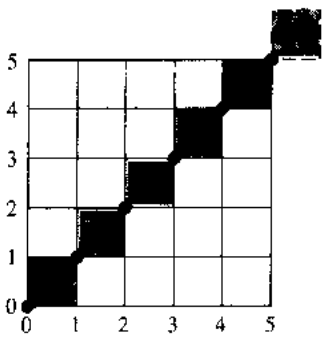


图2-2 在第一象限简单DDA算法产生的结果

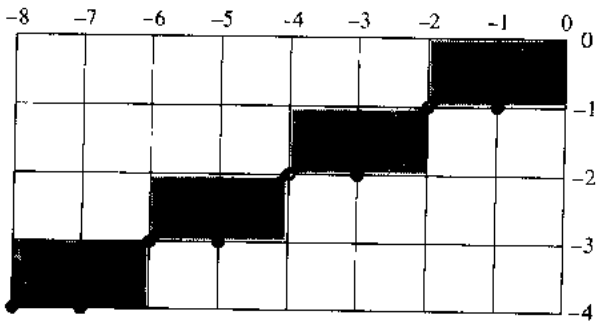


图2-3 在第三象限简单DDA算法生成的结果

2.3 Bresenham算法

虽然Bresenham算法（见[Bres65]）最初是为数字绘图仪而提出的，但它同样适用于CRT光栅显示器。算法的目标是选择表示直线的最佳光栅位置。为此，算法根据直线的斜率来确定或者选择变量在x或y方向上每次递增一个单位。另一方向的增量为0或1，它取决于实际直线与最近网格点位置的距离。这一距离称为误差。

算法的构思巧妙，使得每次只需检查误差项的符号即可。如图2-4所示，直线位于第一八分圆域，即它的斜率介于0和1之间。由图2-4可见，若通过（0，0）的所求直线的斜率大于$\frac{1}{2}$，它与 $x=1$ 直线的交点离直线 $y=1$ 较近，离直线 $y=0$ 较远。因此光栅点（1，1）比（1，0）更逼近于该直线。如果斜率小于$\frac{1}{2}$，则反之。当斜率恰好等$\frac{1}{2}$时，没有确定的选择标准。但本算法将光栅点选在（1，0）。

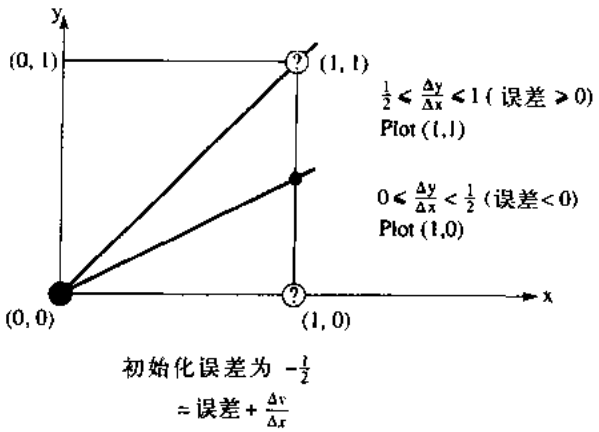


图2-4 Bresenham算法基础

并非所有直线能正好通过光栅点。图2-5是一斜率为$\frac{3}{8}$的直线，其起点落在（0，0）光栅点上，然后越过三个像素。图中也给出了用离散像素表示直线时误差的计算。由于只需检查误差项的符号，因此可置其初值为$-\frac{1}{2}$。这样，如果直线的斜率大于等于$\frac{1}{2}$，在相距一个单位的下一光栅点（1，0）上，误差项的值为原误差项加上直线的斜率，即

$$e = e + m$$

式中m为斜率。这里由于e的初值为$-\frac{1}{2}$，因而有

$$e = -\frac{1}{2} + \frac{3}{8} = -\frac{1}{8}$$

由于e为负，直线在像素中点的下方穿过该像素。用同一水平线上的像素作为直线的近似位置较好，而且y值不增加。然后，误差项再加上斜率即得下一光栅点（2，0）上的e。

$$e = -\frac{1}{8} + \frac{3}{8} = \frac{1}{4}$$

这时e为正，表明直线在中点上方穿过像素。用垂直方向上高一个单位的光栅点（2，1）能更好地逼近直线，所以y应增加一个单位。在考虑下一个像素以前，必须将误差项重新初始化，即将它减少1。这样，

$$e=\frac{1}{4}-1=-\frac{3}{4}$$

可以看到 $x=2$ 的垂直线与所求直线的交点位于直线 $y=1$下 $-1/4$ 处。由于实际误差为零时e置初值为 $-1/2$，因此可得到上述结果 $-3/4$。对于下一个光栅单元得到

$$e=-\frac{3}{4}+\frac{3}{8}=-\frac{3}{8}$$

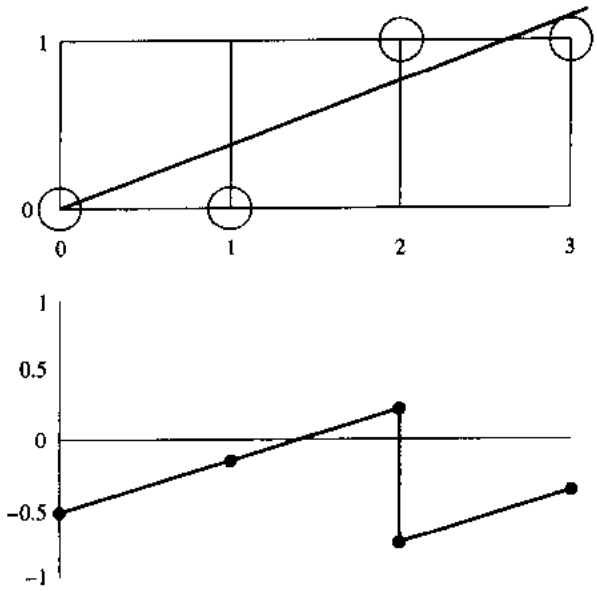


图2-5 Bresenham算法的误差项

由于e为负，y值不增加。上述讨论表明误差项等于所求直线在每一光栅元素上的y方向的截距加上 $-1/2$。

下面给出适用于第一八分圆域，即$0 \leqslant \Delta y \leqslant \Delta x$的Bresenham算法。

Bresenham's line rasterization algorithm for the first octant

the line end points are (x_1, y_1) *and* (x_2, y_2), *assumed not equal*
Integer *is the integer function*
x, y, Δx, Δy *are assumed integer;* e *is real*

initialize variables

$x = x_1$
$y = y_1$
$\Delta x = x_2 - x_1$
$\Delta y = y_2 - y_1$
$m = \Delta y / \Delta x$

initialize e *to compensate for a nonzero intercept*

$e = m - 1/2$

begin the main loop

for $i = 1$ **to** Δx
    setpixel (x, y)

```
            while (e > 0)
                y = y + 1
                e = e - 1
            end while
            x = x + 1
            e = e + m
        next i
    finish
```

算法的流程图如图示2-6所示。

例2.3　Bresenham 算法。

考虑从（0,0）到（5,5）的线段。用Bresenham算法对直线光栅化。

初值计算:

$x = 0$
$y = 0$
$\Delta x = 5$
$\Delta y = 5$
$m = 1$
$e = 1 - 1/2 = 1/2$

主循环的过程如下:

i	setpixel	e	x	y
		$\frac{1}{2}$	0	0
1	(0,0)			
		$-\frac{1}{2}$	0	1
		$\frac{1}{2}$	1	1
2	(1,1)			
		$-\frac{1}{2}$	1	2
		$\frac{1}{2}$	2	2
3	(2,2)			
		$-\frac{1}{2}$	2	3
		$\frac{1}{2}$	3	3
4	(3,3)			
		$-\frac{1}{2}$	3	4
		$\frac{1}{2}$	4	4
5	(4,4)			
		$-\frac{1}{2}$	4	5
		$\frac{1}{2}$	5	5

其结果和预期的相同（见图2-7）。由图可见位于（5,5）的光栅点未被选中。如果将**for-next**的循环变量改为从0到Δx，则该光栅点将被选中。若将**setpixel**语句移到 **next i**前，则可消除位于（0, 0）的第一个光栅点。　□

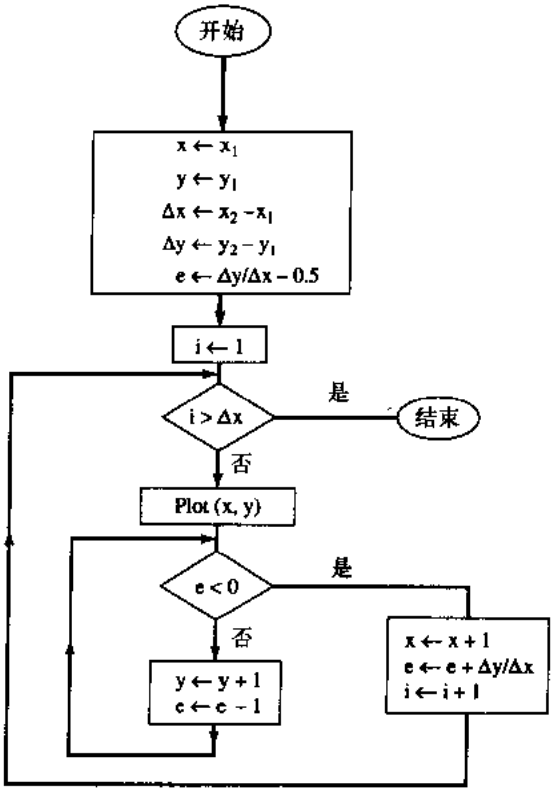


图2-6 Bresenham算法流程图

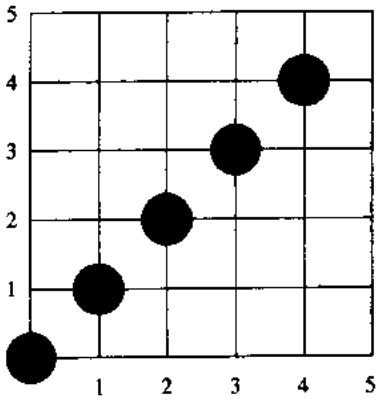


图2-7 在第一八分圆域的Bresenham算法

2.3.1 整数Bresenham算法

上述Bresenham算法在计算直线斜率和误差项时要用到浮点算术运算和除法。采用整数算术运算和避免除法可以加快算法的速度[⊖]。由于上述算法中只用到误差项的符号，因此作如下的简单变换: $\bar{e}=2e\Delta x$，即可得到整数算法（见[Spro82]）。这使本算法便于用硬件或固件实现。适用于第一八分圆域(即$0<\Delta y<\Delta x$)的修正的整数算法如下。

⊖ 根据Jack Bresenham的说法，最初提出整数算法是因为绘图仪所使用的简单计算机只能进行整数运算。

Bresenham's integer algorithm *for the first octant*
the line end points are (x_1, y_1) *and* (x_2, y_2), *assumed not equal*
all variables are assumed integer
initialize variables

$x = x_1$
$y = y_1$
$\Delta x = x_2 - x_1$
$\Delta y = y_2 - y_1$
initialize $\bar{e}$ *to compensate for a nonzero intercept*
$\bar{e} = 2 * \Delta y - \Delta x$
begin the main loop
for $i = 1$ **to** Δx
　　setpixel(x, y)
　　while $(\bar{e} > 0)$
　　$y = y + 1$
　　$\bar{e} = \bar{e} - 2 * \Delta x$
　　end while
　　$x = x + 1$
　　$\bar{e} = \bar{e} + 2 * \Delta y$
next i
finish

只要适当修改部分误差项计算，图2-6的流程图仍然适用于整数算法。

2.3.2 通用Bresenham算法

通用Bresenham算法的实现需要对其他象限的直线生成做一些修改。根据直线斜率和它所在象限很容易实现这一点。实际上，当直线斜率的绝对值大于1时，y总是增1，再用Bresenham 误差判别式以确定x变量是否需要增加1。x或y是增加1还是减去1取决于直线所在的象限（见图2-8）。通用Bresenham 算法如下。

generalized integer Bresenham's algorithm *for all quadrants*
the line end points are (x_1, y_1) *and* (x_2, y_2), *assumed not equal*
all variables are assumed integer
the **Sign** *function returns* $-1, 0, 1$ *as its argument is* $< 0, = 0,$ *or* > 0
initialize variables

$x = x_1$
$y = y_1$
$\Delta x = \text{abs}(x_2 - x_1)$
$\Delta y = \text{abs}(y_2 - y_1)$
$s_1 = \textbf{Sign}(x_2 - x_1)$
$s_2 = \textbf{Sign}(y_2 - y_1)$
interchange Δx *and* Δy, *depending on the slope of the line*
if $\Delta y > \Delta x$ **then**
　　Temp $= \Delta x$
　　$\Delta x = \Delta y$
　　$\Delta y =$ Temp
　　Interchange $= 1$

```
    else
        Interchange = 0
    end if
    initialize the error term to compensate for a nonzero intercept
    ē = 2 * Δy − Δx
    main loop
    for i = 1 to Δx
        setpixel (x, y)
        while (ē > 0)
            if Interchange = 1 then
                x = x + s1
            else
                y = y + s2
            end if
            ē = ē − 2 * Δx
        end while
        if Interchange = 1 then
            y = y + s2
        else
            x = x + s1
        end if
        ē = ē + 2 * Δy
    next i
finish
```

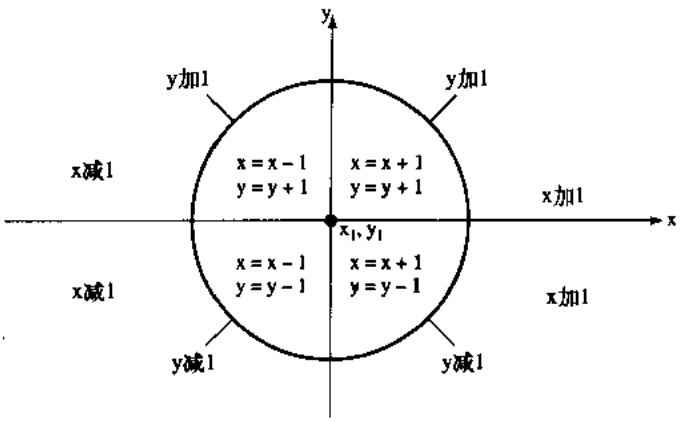


图2-8 通用Bresenham算法的判别条件

例2.4 通用Bresenham算法。

为了说明通用Bresenham算法，考虑从（0, 0）到（−8, −4）画一条直线。例2.2中曾用简单DDA算法画过这条直线。

初值计算:

$x = 0$
$y = 0$
$\Delta x = 8$
$\Delta y = 4$

$s_1 = -1$
$s_2 = -1$
Interchange $= 0$
$\bar{e} = 0$

增量执行主循环过程如下:

i	setpixel	$\bar{e}$	x	y
		0	0	0
1	(0,0)			
		8	−1	0
2	(−1,−1)			
		−8	−1	−1
		0	−2	−1
3	(−2,−1)			
		8	−3	−1
4	(−3,−1)			
		−8	−3	−2
		0	−4	−2
i	setpixel	$\bar{e}$	x	y
5	(−4,−2)			
		8	−5	−2
6	(−5,−2)			
		−8	−5	−3
		0	−6	−3
7	(−6,−3)			
		8	−7	−3
8	(−7,−4)			
		−8	−7	−4
		0	−8	−4

其结果如图2-9所示。与图2-3比较可以看到两者的差别。

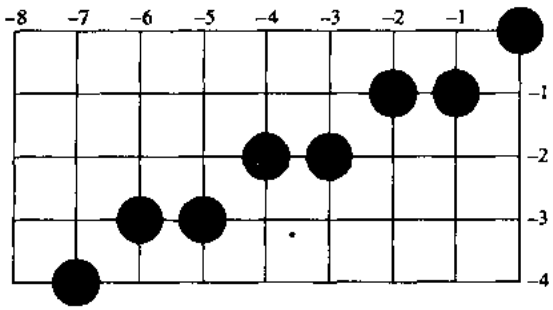


图2-9　在第三象限中通用Bresenham算法的执行结果

2.3.3 快速直线光栅化算法

虽然Bresenham算法已成为标准，但是进一步提高算法速度也是可行的。例如，可以对Bresenham算法进行修改，使之从两端同时对线进行光栅化，由此得到双倍的速度（见[Rokn90; Wyvi90]）。Wu[Wu87; Wyvi90]提出一种双步算法，有效地提高了速度。其基本原理是选择能最好地表示直线的下两个像素。分析表明，在第一条象内能够最好地表示直线的下两个像素的选择只有4种可能。和前面一样，做出合适的选择又可以归结为简单的二值判断。Wyvill[Wyvi90]从直线的两端对称地画线，进一步提高了该算法的速度。该算法的速度是原Bresenham算法的4～5倍（见[Rokn90]）。Gill[Gill94]提出了N步画线算法，并为微处

理器实现了四步算法。该算法比Bresenham算法快,而精确性类似于Bresenham算法。

2.4 圆的生成——Bresenham 算法

除直线光栅化外，其他较为复杂的函数曲线也要进行光栅化。大量工作集中在圆、椭圆、抛物线、双曲线等圆锥曲线的光栅化上（见[Pitt67; Jord73; Bels76; Ramo76; Vana84, 85; Fiel86]）。当然，研究得最多的是圆（见[Horn76; Badl77; Doro79; Suen79]），其中Bresenham 画圆算法（见[Bres77]）是一种最简单有效的方法。首先注意到只要能生成一个八分圆，那么，圆的其他部分就可通过一系列对称变换得到，如图2-10所示。实际上，如果生成的是第一八分圆（逆时针方向0°～45°），那么第二八分圆就可以通过相对$y=x$直线的对称变换得到，从而得到第一四分圆。第一四分圆相对$x=0$对称变换即得第二四分圆。将合在一起的上半圆相对$y=0$对称变换即可获得整圆。图2-10给出相应的二阶对称变换矩阵。

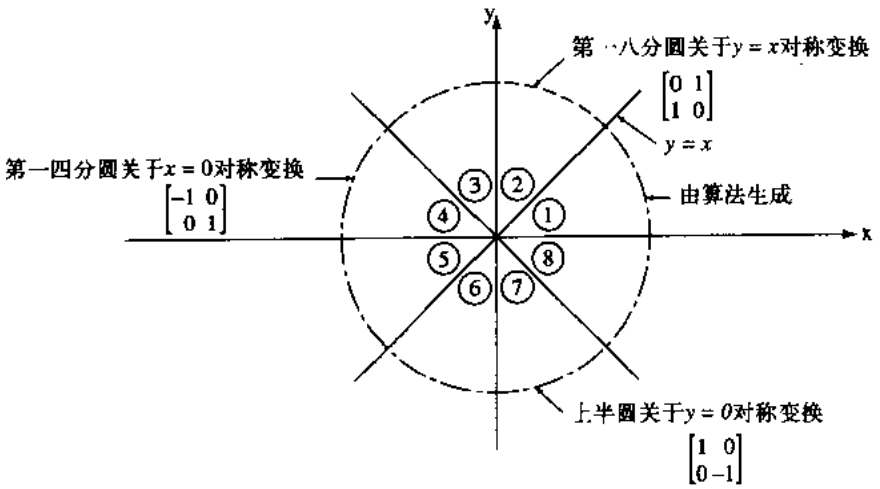


图2-10 由第一八分圆生成整圆

为了推导圆的Bresenham 生成算法，考虑以坐标原点为圆心的第一四分圆。可以看到，如果以点$x = 0$，$y = R$为起点按顺时针方向生成圆，则在第一象限内y是x的单调递减函数（见图2-11）。类似地，如果以$y = 0$，$x = R$作为起点按逆时针方向生成圆，则x是y的单调递减函数。这里取$x = 0$，$y = R$为起点按顺时针方向生成圆，假设圆心和起点均精确地落在像素点上。

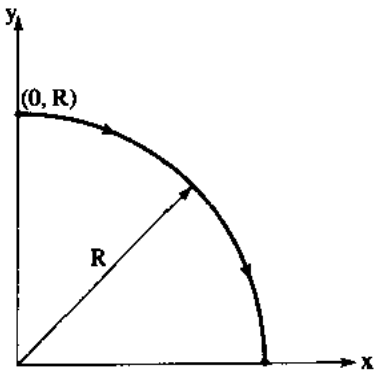


图2-11 第一四分圆

从圆上任意一点出发，按顺时针方向生成圆时，为了最佳逼近该圆，对于下一像素的取法只有三种可能的选择，即右方像素、右下角像素和下方像素，分别记为m_H、m_D和m_V，如图2-12所示。要在这三个像素中选择一个使其与真正圆的距离的平方达到最小，即下列数值中的最小者：

$$m_H = |(x_i+1)^2 + (y_i)^2 - R^2|$$
$$m_D = |(x_i+1)^2 + (y_i-1)^2 - R^2|$$
$$m_V = |(x_i)^2 + (y_i-1)^2 - R^2|$$

圆与点（x_i, y_i）附近光栅网格的相交关系只有五种可能（见图2-13），根据这五种关系可以简化计算。

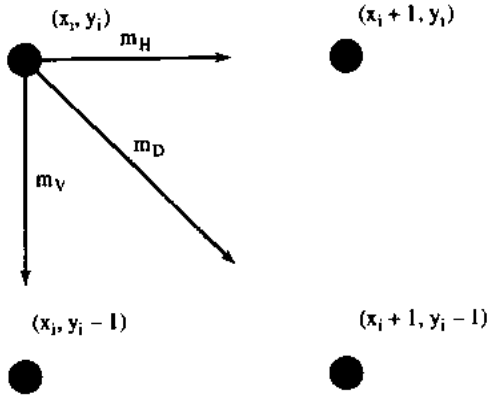


图2-12　第一象限像素的选取

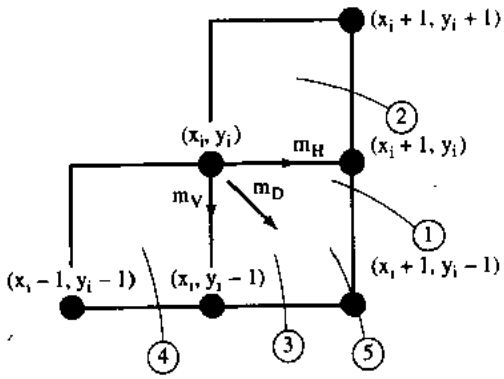


图2-13　圆和光栅网格的相交关系

从圆心到右下角像素（x_i+1，y_i−1）的距离平方与圆心到圆上点的距离平方之差等于

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

和Bresenham直线光栅化算法一样，在选取能最好表示该圆的像素时希望只利用误差项的符号，而不是它的值。

如果Δ_i<0，那么右下角点（x_i+1，y_i−1）在该圆内，即图2-13中情形1和2。显然这时只能取像素（x_i+1，y_i），即m_H；或者像素（x_i+1，y_i−1），即m_D。为了确定究竟应该选择哪一个像素，首先考察情形1，为此计算圆到像素m_H的距离平方与圆到像素m_D的距离平方之差，

即

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

如果 $\delta < 0$，那么圆到对角像素（m_D）的距离大于圆到水平像素（m_H）的距离。反之，如果 $\delta < 0$，那么圆到水平像素（m_H）的距离较大。这样

$$\text{当 } \delta < 0, \text{ 取} m_H\ (x_i+1,\ y_i)$$
$$\text{当 } \delta > 0, \text{ 取} m_D\ (x_i+1,\ y_i-1)$$

当 $\delta = 0$，即两者距离相等时，规定取右方像素。

对于情形1，右下角像素（x_i+1，y_i-1）总是位于圆内，而右方像素（x_i+1，y_i）总是位于圆外，即

$$(x_i+1)^2 + (y_i)^2 - R^2 \geqslant 0$$
$$(x_i+1)^2 + (y_i-1)^2 - R^2 < 0$$

因此 δ 的计算可简化为

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

通过加、减$-2y_i + 1$，把$(y_i)^2$配成完全平方项后得到

$$\delta = 2[(x_i+1)^2 + (y_i-1)^2 - R^2] + 2y_i - 1$$

由Δ_i定义可得更简单的关系式

$$\delta = 2(\Delta_i+y_i)-1$$

其次考虑情形2（见图2-13）。由于y是一单调递减函数，所以只能选取右方像素（x_i+1，y_i）。因为右方像素（x_i+1，y_i）和右下角像素（x_i+1，y_i-1）这时都位于圆内，故δ表达式中有如下关系：

$$(x_i + 1)^2 + (y_i)^2 - R^2 < 0$$
$$(x_i+1)^2 + (y_i-1)^2 - R^2 < 0$$

因此$\delta < 0$。根据与情形1相同的判别准则，这时应选取像素（x_i+1，y_i）。

如果$\Delta_i > 0$，则右下角点（x_i+1，y_i-1）位于圆外，即图2-13中的情形3和4。显然，这时只能选取（x_i+1，y_i-1），即m_D；或者（x_i，y_i-1），即m_V。为了导出判别准则，首先研究情形3，为此计算圆到对角像素m_D的距离平方与圆到像素m_V的距离平方之差，即

$$\delta' = |(x_i+1)^2+(y_i-1)^2-R^2| - |(x_i)^2+(y_i-1)^2-R^2|$$

如果$\delta' < 0$，即圆到下方像素（x_i，y_i-1）的距离较大，这时需选取右下角像素（x_i+1，y_i-1）。反之，如果$\delta' > 0$，即圆到右下角像素的距离较大，这时需取下方像素（x_i，y_i-1）。这样，

$$\text{当}\delta' < 0, \text{ 取} m_D\ (x_i+1,\ y_i-1)$$
$$\text{当}\delta' > 0, \text{ 取} m_V\ (x_i,\ y_i-1)$$

当距离相等时，即 $\delta' = 0$，规定选取m_D。

由于右下角像素（x_i+1，y_i-1）位于圆外，而像素（x_i，y_i-1）位于圆内，所以对于情形3，作为δ'的分量有如下关系：

$$(x_i+1)^2 + (y_i-1)^2 - R^2 > 0$$

$$(x_i)^2 + (y_i-1)^2 - R^2 < 0$$

故δ'可改写为

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

加、减$2x_i+1$，把$(x_i)^2$配成完全平方项，则有

$$\delta' = 2[(x_i+1)^2 + (y_i-1)^2 - R^2] - 2x_i - 1$$

由Δ_i定义，得到

$$\delta' = 2(\Delta_i - x_i) - 1$$

对于情形4，再次注意到由于在x单调递增时y是单调递减的，因此这时需选取下方像素(x_i, y_i-1)。这时正下方和右下角两像素均位于圆外，即δ'的分量有如下关系：

$$(x_i+1)^2 + (y_i-1)^2 - R^2 > 0$$

$$(x_i)^2 + (y_i-1)^2 - R^2 > 0$$

因此 $\delta' > 0$。根据与情形3中相同的判别准则，这时应取m_V。

最后考虑图2-13中的情形5，这时右下角像素（x_i+1，y_i-1）恰好在圆上，即$\Delta_i=0$。由于

$$(x_i+1)^2 + (y_i)^2 - R^2 > 0$$

$$(x_i+1)^2 + (y_i-1)^2 - R^2 = 0$$

从而$\delta > 0$，这时应选取右下角像素。类似地，δ'的分量有

$$(x_i+1)^2 + (y_i-1)^2 - R^2 = 0$$

$$(x_i)^2 + (y_i-1)^2 - R^2 < 0$$

由此$\delta' < 0$。这就是选取右下角像素（x_i+1，y_i-1）的条件。这样，$\Delta_i=0$时具有与$\Delta_i<0$或$\Delta_i>0$一样的判别准则。

上述结果可以归纳如下。

当$\Delta_i<0$时：

$\delta \leqslant 0$，取像素（x_i+1，y_i）　　$\rightarrow m_H$

$\delta > 0$，取像素（x_i+1，y_i-1）　　$\rightarrow m_D$

当$\Delta_i>0$时：

$\delta' \leqslant 0$，取像素（x_i+1，y_i-1）　　$\rightarrow m_D$

$\delta' > 0$，取像素（x_i，y_i-1）　　$\rightarrow m_V$

当$\Delta_i = 0$时，取像素（x_i+1，y_i-1）$\rightarrow m_D$

由此容易导出简单增量算法的递推关系。首先考虑水平移动到m_H，即像素（x_i+1，y_i）。称此为第$i+1$个像素。该新像素的坐标以及Δ_i的值为

$$x_{i+1} = x_i+1$$

$$y_{i+1} = y_i$$

$$\begin{aligned}\Delta_{i+1} &= (x_{i+1}+1)^2+(y_{i+1}-1)-R^2 \\ &= (x_{i+1})^2+2x_{i+1}+1+(y_i-1)^2-R^2 \\ &= (x_i+1)^2+(y_i-1)^2-R^2+2x_{i+1}+1 \\ &= \Delta_i+2x_{i+1}+1\end{aligned}$$

类似地，对角移动到m_D（x_i+1，y_i−1）时，新像素的坐标和Δ_i的值为

$$x_{i+1} = x_i+1$$
$$y_{i+1} = y_i-1$$
$$\Delta_{i+1} = \Delta_i+2x_{i+1}-2y_{i+1}+2$$

对于移动到m_V（x_i，y_i−1），有如下关系式：

$$x_{i+1}=x_i$$
$$y_{i+1}=y_i-1$$
$$\Delta_{i+1}=\Delta_i-2y_{i+1}+1$$

下面给出用伪代码写出的Bresenham画圆算法。

Bresenham's incremental circle algorithm *for the first quadrant*
all variables are assumed integer
initialize the variables
$x_i = 0$
$y_i = R$
$\Delta_i = 2(1-R)$
Limit = 0
while $y_i \geq$ Limit
call setpixel (x_i, y_i)
determine if case 1 or 2, 4 or 5, or 3
if $\Delta_i < 0$ **then**
$\delta = 2\Delta_i + 2y_i - 1$
determine whether case 1 or 2
if $\delta \leq 0$ **then**
call mh(x_i, y_i, Δ_i)
else
call md(x_i, y_i, Δ_i)
end if
else if $\Delta_i > 0$ **then**
$\delta' = 2\Delta_i - 2x_i - 1$
determine whether case 4 or 5
if $\delta' \leq 0$ **then**
call md(x_i, y_i, Δ_i)
else
call mv(x_i, y_i, Δ_i)
end if
else if $\Delta_i = 0$ **then**
call md(x_i, y_i, Δ_i)
end if
end while
finish
move horizontally
subroutine mh(x_i, y_i, Δ_i)
$x_i = x_i + 1$
$\Delta_i = \Delta_i + 2x_i + 1$

end sub

move diagonally

subroutine md(x_i, y_i, Δ_i)

$x_i = x_i + 1$

$y_i = y_i - 1$

$\Delta_i = \Delta_i + 2x_i - 2y_i + 2$

end sub

move vertically

subroutine mv(x_i, y_i, Δ_i)

$y_i = y_i - 1$

$\Delta_i = \Delta_i - 2y_i + 1$

end sub

置限制变量（Limit）为零使算法在水平轴上终止，从而得到第一四分圆。如果只要求生成一个八分圆，那么只要置 Limit = Integer ($R/\sqrt{2}$)即可得到第二个八分圆（见图2-10）。关于$y=x$对称变换的结果可得到第一四分圆。图2-14给出了本算法的流程图。

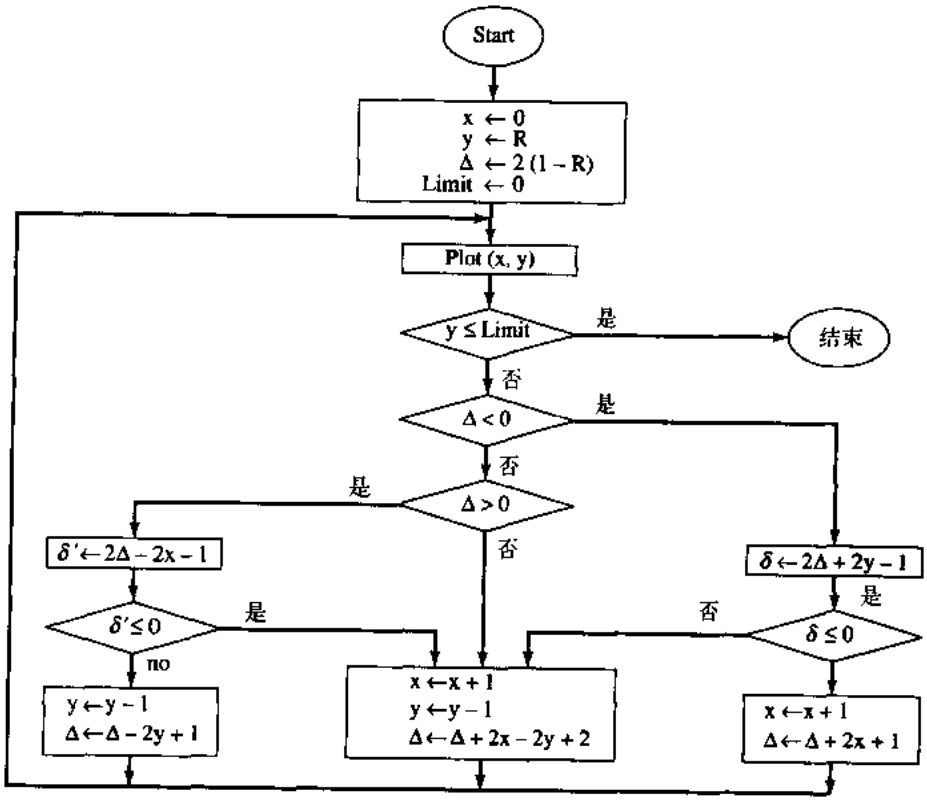


图2-14　第一象限Bresenham 增量画圆算法流程图

例2.5　Bresenham画圆算法。

以坐标原点为圆心、半径为8的圆在第一象限部分为例说明圆的生成算法。

初值计算:

$x_i = 0$

$y_i = 8$

$\Delta_i = 2(1 - 8) = -14$

Limit = 0

增量执行主循环过程

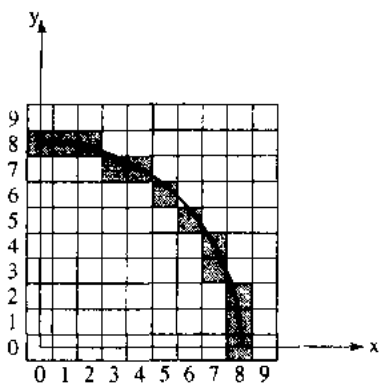
y_i > Limit
continue
setpixel (0, 8)
$\Delta_i < 0$
$\delta = 2(-14) + 2(8) - 1 = -13$
$\delta < 0$
call mh(0,8,−14)
x = 0 + 1 = 1
$\Delta_i = -14 + 2(1) + 1 = -11$
y_i > Limit
continue
setpixel (1, 8)
$\Delta_i < 0$
$\delta = 2(-11) + 2(8) - 1 = -7$
$\delta < 0$
call mh(1,8,−11)
x = 1 + 1 = 2
$\Delta_i = -11 + 2(2) + 1 = -6$
y_i > Limit
continue
setpixel (2, 8)
.
.
.
continue

算法每一次循环的结果都在下表中。算法所选的像素表为 (0, 8)、(1, 8)、(2, 8)、(3, 7)、(4, 7)、(5, 6)、(6, 5)、(7, 4)、(7, 3)、(8, 2)、(8, 1) 和 (8, 0)。

setpixel	Δ_i	δ	δ'	x	y
	−14			0	8
(0,8)					
	−11	−13		1	8
(1,8)					
	−6	−7		2	8
(2,8)					
	−12	3		3	7
(3,7)					
	−3	−11		4	7
(4,7)					
	−3	7		5	6
(5,6)					
	1	5		6	5
(6,5)					
	9		−11	7	4
(7,4)					
	4		3	7	3
(7,3)					
	18		−7	8	2
(8,2)					
	17		19	8	1
(8,1)					
	18		17	8	0

(8,0)
完成

图2-15是算法的结果和对应的圆弧。本算法很容易推广到其他象限圆或圆弧。



圆半径为8	
发亮的像素	
0	8
1	8
2	8
3	7
4	7
5	6
6	5
7	4
7	3
8	2
8	1
8	0

图2-15　Bresenham增量画圆算法的结果

2.5　椭圆的生成

一些其他的封闭的圆锥曲线(例如椭圆)也相当重要。Pitteway[Pitt67]、Maxwell和Baker[Maxw79]、Van Aken[VanA84]、Kappel[Kapp85]、Field[Fiel86]、McIlroy[McIl92]以及Fellner和Helmberg[Feil93]阐述了几种光栅化椭圆的技术，Bresenham讨论了几种重要的光栅化技术（见[Bres90]）。Da Silva（见[DaSi89]）通过使用一阶偏差分提高了中点椭圆算法的效率。他同时还解决了算法从1区切换到2区(在本节的后面部分讨论)时会遇到的像素选择问题和瘦形垂直椭圆的取整问题。最后，他还考虑了中心在原点的旋转椭圆的情况。Wu和Rokne[Wu89a]提出了一种双步算法，可以选择最佳表示椭圆的下两个像素，并证明只有四种可能的模式。而且，正确的选择模式可以归结为简单的二值问题。

圆是椭圆的特例，所以可以直接扩展Bresenham的画圆光栅化算法。但是，Van Aken[VanA84]指出将Bresenham画圆算法扩展到椭圆不能保证实际椭圆与所选像素之间的线性误差最小。Van Aken[VanA84]在Pitteway[Pitt67])和Horn[Horn76]早期工作的基础上给出了一种用于椭圆的技术，称为中点算法。Kappel[Kapp85]对该算法进行了几处改进。该算法同样也适用于直线和圆的光栅化(参见2.4节和2.6节)。Van Aken保证实际椭圆与所选像素之间的最大线性误差为二分之一。

考虑到中心在原点、轴对齐的椭圆的非参数化表示为:

$$\frac{x^2}{a^2}+\frac{y^2}{b^2}=1$$

其中a, b分别是半长轴和半短轴。传统的写法是每一项都乘以a^2b^2,因此椭圆方程为:

$$f(x,y)=b^2x^2+a^2y^2-a^2b^2=0$$

注意，该椭圆与x轴相交于 $x=a$, 与y轴相交于 $y=b$。

这里只考虑椭圆在第一象限的部分，其他象限的部分可作相应的对称变换而得。中心不在原点的椭圆可通过平移得到。和Bresenham圆光栅化算法一样，中点画椭圆算法利用了第

一象限的椭圆是随x和y的单调递增或递减函数。

椭圆上斜率为-1(即$dx/dy = -1$)的点将椭圆分成两个部分，如图2-16所示。当光栅化区域1的椭圆时($dx/dy < -1$)，当选择像素时，Δx的增量比较重要，而在区域2，Δy的增量比较重要(参见图2-17a和图2-17b)。在图2-17中，当前选中的像素(x_i, y_i)显示为阴影。而且，正如在Bresenham算法中一样，在选择下一位置时，希望使用测试函数的符号或误差值。也希望使用整数算法。注意，与Bresenham圆生成算法相比，生成的方向反向了，即从y轴开始的顺时针变成了从x轴开始的逆时针。

在Bresenham画圆算法中，椭圆与两个最近候选像素(例如点(x_i-1, y_i+1)与点(x_i, y_i+1))之间的距离用于选择适当的像素，选择最近距离的像素。在中点算法中，可以使用椭圆和两个候选像素跨度（连线）中心的距离，即，用$f(x, y)$判断单个的点，如（$x_i-\frac{1}{2}, y_i+1$）点。

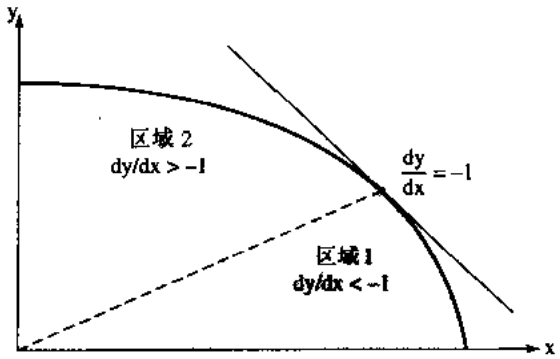


图2-16 将椭圆分成两个部分

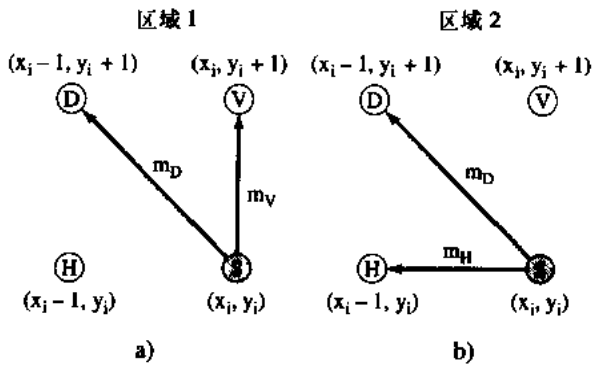


图2-17 像素选择

a) 区域1 b) 区域2

根据图2-18a，在区域1($dy/dx < -1$)，如果椭圆通过像素D和像素V中点的右部或上部，则选择像素V；如果通过像素D和像素V中点的左部或下部，则选择像素D。如图2-18b所示，在区域2($dy/dx > -1$)，如果椭圆通过像素H和像素D中点的上部，则选择像素D；如果椭圆通过像素H和像素D中点的下部，则选择像素H。从图2-18可知，最大线性误差是像素间距离的一半，即$0 < |e| < \frac{1}{2}$。而且，如果假设e为正值，则$-\frac{1}{2} < e < \frac{1}{2}$。

以下是Van Aken给出的公式，将决策变量d_i定义为$f(x,y)$在中点处的值的两倍,即在区域1为$(x_i-\frac{1}{2}, y_i+1)$，在区域2为$(x_i-1, y_i+\frac{1}{2})$，先考虑区域1:

$$
\begin{aligned}
d_{1_i} &= 2f\left(x_i - \frac{1}{2}, y_i + 1\right) \\
&= 2\left[b^2\left(x_i - \frac{1}{2}\right)^2 + a^2(y_i + 1)^2 - a^2b^2\right] \\
&= b^2\left(2x_i^2 - 2x_i + \frac{1}{2}\right) + a^2(2y_i^2 + 4y_i + 2) - 2a^2b^2
\end{aligned}
$$

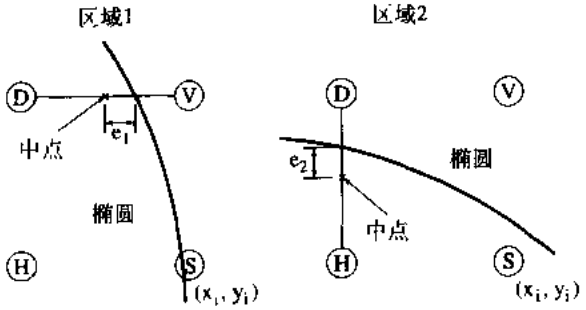


图2-18　椭圆的误差标准

a) 区域1　b) 区域2

如果椭圆通过中点,则d_{1_i}= 0。但是，很可能椭圆要通过中点$(x_i - \frac{1}{2} + e_1,\ y_i + 1)$的右部或左部。因为这一点在椭圆上，故$f(x, y)=0$。将$(x_i - \frac{1}{2} + e_1,\ y_i + 1)$代入$f(x, y)=0$，则产生以下公式:

$$
\begin{aligned}
f\left(x_i - \frac{1}{2} + e_1, y_i + 1\right) &= b^2\left(x_i - \frac{1}{2} + e_1\right) + a^2(y_i + 1) - a^2b^2 = 0 \\
&= f\left(x_i - \frac{1}{2}, y_i + 1\right) + 2b^2e_1\left(x_i - \frac{1}{2}\right) + b^2e_1^2 = 0
\end{aligned}
$$

将$d_{1_i}/2 = f(x_i - \frac{1}{2},\ y_i + 1)$代入，则:

$$
\begin{aligned}
d_{1_i} &= -4b^2e_1\left(x_i - \frac{1}{2}\right) - 2b^2e_1^2 \\
&= -2b^2e_1[(2x_i - 1) + e_1]
\end{aligned}
$$

注意，d_{1_i}的符号,它与e_1的符号正好相反。因为$-\frac{1}{2} < e_1 < \frac{1}{2}$，满足$x_i > (1 - e_1)/2$的条件是$x_i > \frac{3}{4}$。对于整型值$x_i$，$x_i > \frac{3}{4}$相当于$x_i > 0$。因此，如果$d_{1_i} < 0$，则$e_1 > 0$，而且在$(x_i,\ y_i + 1)$点的像素(即点V)被选中。如果$d_{1_i} > 0$，则$e_1 < 0$，而且$(x_i - 1,\ y_i + 1)$处的像素被选中。如果$d_{1_i} = 0$，算法选择对角线像素D。

下面讨论区域2:

$$
\begin{aligned}
d_{2_i} &= 2f\left(x_i - 1, y_i + \frac{1}{2}\right) \\
&= 2\left[b^2(x_i - 1)^2 + a^2\left(y_i + \frac{1}{2}\right)^2 - a^2b^2\right] \\
&= b^2(2x_i^2 - 4x_i + 2) + a^2\left(2y_i^2 + 2y_i + \frac{1}{2}\right) - 2a^2b^2
\end{aligned}
$$

如果$d_{2_i}=0$，则椭圆通过像素H和D的中点(参见图2-18b)。这里e_2是从中点垂直向上的正误差值，或向下的负误差值。考虑到在$(x_i-1,\ y_i+\frac{1}{2}+e_2)$点，$f(x,y)=0$。因为$d_{2_i}/2=f(x_i-1,\ y_i+\frac{1}{2}$，则:

$$\begin{aligned} d_{2_i} &= -4a^2e_2\left(y_i+\frac{1}{2}\right)-2a^2e_2^2 \\ &= -2a^2e_2[(2y_i+1)+e_2] \end{aligned}$$

因为$-\frac{1}{2}<e_2<\frac{1}{2}$，则$d_{2_i}$与$e_2$的符号正好相反，$y_i>-\frac{1}{2}$。对于整型值$y_i$，相当于$y_i>0$。因此，如果$d_{2_i}<0$，则$e_2>0$，D点被选中。如果$d_{2_i}>0$，则$e_2<0$，H像素被选中(见图2-18b)。如果$d_{2_i}=0$，算法选择水平像素D。

要完成整个算法，还应包括从区域1切换到区域2以及开始和结束算法的条件。区域1和区域2的分界点在$dy/dx=-1$处。代入椭圆方程，则为:

$$df(x,y)=d(b^2x^2+a^2y^2-a^2b^2)=2b^2xdx+2a^2ydy=0$$

或者

$$\frac{dy}{dx}=-\frac{b^2}{a^2}\frac{x}{y}$$

因为a, b, x都为正值或零，并且$y>0$，则当$b^2x<a^2y$时，$dy/dx<-1$。从区域1切换到区域2的条件是$b^2(x_i-\frac{1}{2})<a^2(y_i+1)$。

算法开始于$x=a$，$y=0$，或者

$$f(x,y)=b^2x^2+a^2y^2-a^2b^2=a^2b^2-a^2b^2=0$$

并当$x<0$时停止算法。原始算法是:

Naive midpoint ellipse algorithm for the first quadrant

```
initialize the variables
x = Integer(a + 1/2)
y = 0
while b*b*(x - 1/2) > a*a*(y + 1)      start in region 1
    call setpixel(x, y)
    d1 = b*b*(2*x*x-2*x+1/2)+a*a*(2*y*y+4*y+2)-2*a*a*b*b
    if d1 < 0 then
        y = y + 1      move vertically
    else
        x = x - 1      move diagonally
        y = y + 1
    end if
end while

initialize the decision variable in region 2
d2 = b*b*(2*x*x-4*x+2)+a*a*(2*y*y+2*y+1/2)-2*a*a*b*b

while x ≥ 0      switch to region 2
    call setpixel(x, y)
    if d2 < 0 then
        x = x - 1      move diagonally
        y = y + 1
    else
```

```
        x = x - 1        move horizontally
      end if
      d2 = b*b*(2*x*x-4*x+2)+a*a*(2*y*y+2*y+1/2)-2*a*a*b*b
    end while
  finish
```

通过增量计算d_{1_i}和d_{2_i}的新值,可以使算法更有效。在新像素位置:

$$d_{1_{i+1}} = 2f\left(x_{i+1} - \frac{1}{2}, y_{i+1} + 1\right)$$
$$= b^2\left(2x_{i+1}^2 - 2x_{i+1} + \frac{1}{2}\right) + a^2(2y_{i+1}^2 + 4y_{i+1} + 2) - 2a^2b^2$$

和

$$d_{2_{i+1}} = 2f\left(x_{i+1} - 1, y_{i+1} + \frac{1}{2}\right)$$
$$= b^2(2x_{i+1}^2 - 4x_{i+1} + 2) + a^2\left(2y_{i+1}^2 + 2y_{i+1} + \frac{1}{2}\right) - 2a^2b^2$$

对于对角线移动，x_i+1 = x_i−1，y_i+1= y_i+1，因此，代入公式得

$$d_{1_{i+1}} = b^2\left[2(x_i^2 - 2x_i + 1) - 2x_i + 2 + \frac{1}{2}\right]$$
$$+ a^2[2(y_i^2 + 2y_i + 1) + 4y_i + 4 + 2] - 2a^2b^2$$

由$f(x_i - \frac{1}{2}, y_i+1)$得:

$$d_{1_{i+1}} = 2f\left(x_i - \frac{1}{2}, y_i + 1\right) + a^2(4y_i + 6) - 4b^2(x_i - 1)$$
$$= d_{1_i} + 4a^2y_{i+1} - 4b^2x_{i+1} + 2a^2$$

同理，对于d_{2_i}+1

$$d_{2_{i+1}} = b^2\left[2(x_i^2 - 4x_i + 2) - 4x_i + 6\right]$$
$$+ a^2\left[2(y_i^2 + 2y_i + 1) + 2(y_i + 1) + \frac{1}{2}\right] - 2a^2b^2$$

由$f(x_i - \frac{1}{2}, y_i+1)$得:

$$d_{2_{i+1}} = d_{2_i} + 4a^2y_{i+1} - 4b^2x_{i+1} + 2b^2$$

对于水平移动，x_i+1=x_i−1，y_i+1 = y_i，且

$$d_{2_{i+1}} = d_{2_i} - 4b^2x_{i+1} + 2b^2$$

对于垂直移动，x_i+1 = x_i , y_i+1 = y_i+1，且

$$d_{1_{i+1}} = d_{1_i} - 4a^2y_{i+1} + 2a^2$$

使用上述结果,可以得到一个更有效的算法:

Efficient midpoint ellipse algorithm for the first quadrant

```
    initialize the variables
    x = Integer(a + 1/2)
    y = 0
    define temporary variables
    taa = a*a
    t2aa = 2*taa
    t4aa = 2*t2aa

    tbb = b*b
    t2bb = 2*tbb
    t4bb = 2*t2bb

    t2abb = a*t2bb

    t2bbx = t2bb*x
    tx = x

    initialize the decision variable in region 1
    d1 = t2bbx*(x - 1) + tbb/2 + t2aa*(1 - tbb)

    while t2bb*tx > t2aa*y        start in region 1
        call setpixel(x, y)
        if d1 < 0 then
            y = y + 1             move vertically
            d1 = d1 + t4aa*y + t2aa
            tx = x - 1
        else
            x = x - 1             move diagonally
            y = y + 1
            d1 = d1 - t4bb*x + t4aa*y + t2aa
            tx = x
        end if
    end while

    initialize the decision variable in region 2
    d2 = t2bb*(x*x + 1) - t4bb*x + t2aa*(y*y + y - tbb) + taa/2

    while x >= 0
        call setpixel(x, y)
        if d2 < 0 then
            x = x - 1             move diagonally
            y = y + 1
            d2 = d2 + t4aa*y - t4bb*x + t2bb
        else
            x = x - 1             move horizontally
            d2 = d2 - t4bb*x + t2bb
        end if
    end while
finish
```

2.6 一般函数的光栅化

下面要考虑的是光栅化一般多项式函数的算法，例如三次或四次多项式等。Jordan等

[Jord73]以及Van Aken和Novak[VanA85]讨论了这种算法。但是，对于一般多项式，这些算法都失败了，特别是对于高于二阶的曲线（见[Bels76;Ramo76]）。这些算法失败的根本原因是,除了区域有限制之外，对于三阶或高于三阶的一般多项式函数，在x方向和y方向上不是单调递增或递减，因此函数轨迹可能自交或循环，或曲线轨迹之间的距离小于一个像素。图2-19是这些的图示，箭头方向表示光栅化方向。

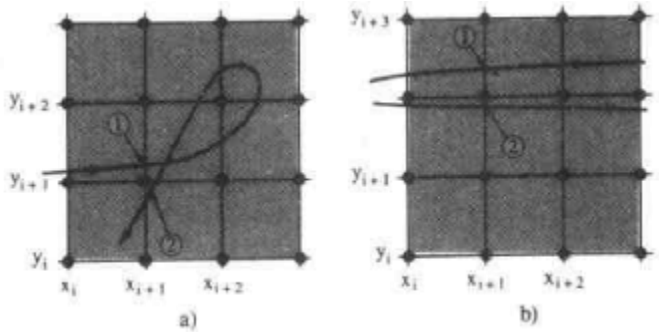


图2-19　一般多项式函数的光栅化问题

a) 自相交形成圈　b) 曲线接近

如图2-19a所示，函数处理到(x_{i+1},y_{i+1})时，必须选择点2而不是点1。根据点2进行计算，则光栅化曲线的下一点是(x_i, y_i)或(x_{i+1}, y_i)，而不是(x_{i+2}, y_{i+2})。因此，(x_{i+2}, y_{i+2})像素永远不被选中，使曲线光栅化发生错误。

如图2-19b所示，这两根曲线可能是双曲线的两支，则对于二阶一般多项式也要发生与上面相同的问题。例如,当沿着1号曲线光栅化时，在(x_{i+1},y_{i+2})点处产生的顶点在2号曲线上。这时光栅化曲线方向错误。结果导致大块区域丢失。

这里以及前面章节描述的曲线光栅化技术常称作曲线跟踪算法。另一种技术是检查光栅中的每一像素，看曲线是否通过该像素。如果曲线通过该像素，则该像素被激活。如果没有通过该像素，则该像素被忽略。显然大量的像素被忽略；因此该算法开销很大,效率不高。该类算法中的成功者是有效地限制要检查的像素数目。这类算法常称作递归空间子分算法。

递归空间子分算法的成功依赖于对二维或三维包围盒的子分[⊖]。图2-20给出了它的基本概念。只检查由对角顶点(x_i, y_i)和$(x_i+\Delta x, y_i+\Delta y)$构成的包围盒。如果曲线$f(x, y)=0$通过包围盒，且包围盒的大小大于一个像素，则对包围盒进行子分，如图中的虚线部分所示。如果曲线不通过该包围盒，则该区域用背景色生成，并忽略处理。对图2-20中的$2a$、$2b$、$2c$、$2d$四个子包围盒递归执行该算法。如图2-20所示,标记为$2c$和$2d$的子包围盒用背景色生成,并忽略处理。标记为$2a$和$2b$的子包围盒进一步子分，直到达到一个像素大小，用于给出隐函数曲线$f(x, y)=0$的颜色属性。

递归空间子分算法的有效性和精确性依赖于确定隐函数曲线$f(x, y)=0$通过包围盒的测试方法。因为曲线可以有多个值，可以有多重连接，曲线可以在一点自相交或在某一点分成多个分叉，所以必须仔细选择测试方法。其中最成功的算法（见Taubin的[Taub94a,b]）基于区间算术（interval arithmetic）原理（见[Moor79; Duff92]）。

成功的算法一般不使用确定性的“是/否”测试方法，而是使用较少计算开销的“可能/否”

⊖ Warnock隐藏面算法就是这些早期算法中的一个（参见4.4节）。

测试方法以提高算法的速度。

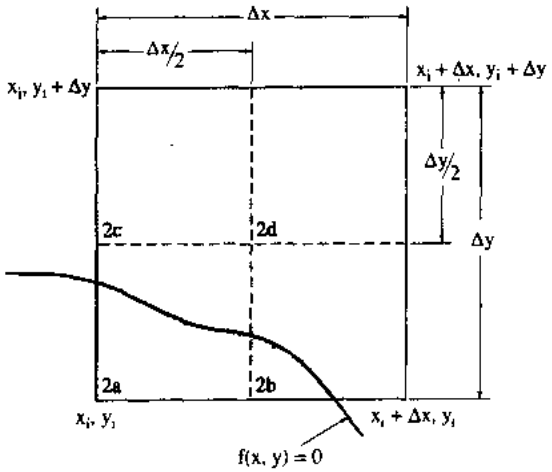


图2-20 递归空间子分

一般来说，“可能/否”测试方法可以在较少计算开销的情况下，排除大多数(不是所有)不满足某一特定条件的对象。剩下的对象必须使用计算开销更大的确定性的“是/否”测试进行筛选。结果是整个算法比较有效和快速。

递归空间子分技术可以很容易地扩展到三维，在三维情况下，该算法用于面的光栅化和判断面的相交（见[Taub94b]）。

对于在给定域上不是单调递增/递减的函数或不是单值函数的函数，最安全的光栅化技术是根据显示要求计算出函数上足够多的点，然后使用一种直线光栅化算法按合适的次序将它们连接起来。如果可能，可以用参数化形式计算函数顶点。如果参数化适当，参数化表示在函数曲率较大时可以产生更紧凑的空间点；当函数曲率较小时，可以产生空间分布更稀疏的顶点。其结果是函数的一种更好的表示。

2.7 扫描转换——显示的生成

用视频技术显示光栅图像则需要按照视频显示器（见1.4节）要求的精确模式组织图像。这个过程称为扫描转换。与随机扫描或画线式显示器的显示表只包含直线或字符信息不同，这里的显示表则需包含屏幕上所有像素的信息，且这些信息需以视频速率和扫描线的顺序，即从上到下、自左至右的方式组织和显示。有三种实现这一过程的方法，即实时扫描转换（real-time scan conversion）、行程编码(run-length encoding)、以及帧缓冲存储器(frame buffer memory)。实时扫描转换在车辆仿真领域应用最广。行程编码一般用于“存储”图像或传输图像（或正文）。帧缓冲存储器通常在具有图形功能的工作站和个人计算机中用于图像显示。

2.7.1 实时扫描转换

在实时扫描转换或即时扫描转换(on-the-fly scan conversion)过程中，图形是通过其视觉属性以及几何属性随机地表示出来的。常见的视觉属性有颜色、明暗和光强，而x, y坐标、斜率和文字则是常见的几何属性。这些几何属性按y值的次序排列。在显示每帧画面时，处理器扫描这些信息，并计算屏幕上每一像素的光强。实时扫描转换不需要大容量存储器。通常，对存储器容量的要求是能存储显示表再加一条扫描行的信息。由于图形信息是储存在随机组

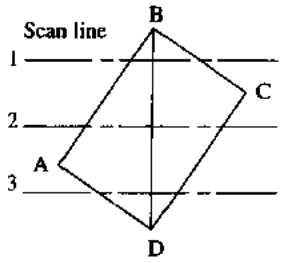
织的显示表之中，因此易于增加或删除表中信息，从而极大地便利了图形的动态显示。然而，所显示图像的复杂性受到显示处理器速度的限制。通常，这意味着图形中直线或多边形的数目、一条扫描线上交点的个数、灰度的等级或颜色的种类要受到限制。

实时扫描转换最简单的实现方法是在显示每条扫描线时处理整个显示表，求出当前扫描线与显示表中每条线的交点。对于常见的视频刷新速度，由于每显示一条扫描线只有63.5μs时间可用于处理整个显示表,所以这种技术不适用于处理复杂的图像。

为了便于说明，我们将当前的讨论限制在只包含直线的图形,其他图形可用类似方式处理，因为通常并不是图形中的每条线都和当前扫描线相交，因此使用活化边表(active edge list)方法可以减少整个工作量。活化边表包含图形中与当前扫描线相交的所有线段。有多种技术可用于建立和维护活化边表。所有技术都把图形中的线段按其最大的y值进行排序。

2.7.2 使用指针的简单活化边表

建立和维护活化边表的一个简单的方法是在这个按y排序的表中使用两个浮动的指针。其起始指针指向活化边表的表头，终止指针指向活化边表的表尾。图2-21a表示一个简单的画线图形和三条常见的扫描线。图2-21b是该图中线段的一个常见的排序表。开始时，起始指针指向表头，即BC。终止指针指向表中起始点位于所考虑扫描线之上的最后一条线段，即BD。扫描线沿图形向下移动时，其终止指针也随着向下移动，并不断把那些起始点在当前扫描线上方（包括恰好落在扫描线上）的线段包括进来。同时，起始指针也向下移动，把那些终点在当前扫描线上方的线段从活化边表中删除。图2-21b以图2-21a中的扫描线2和3为例说明了这一变化过程。图2-21c和2-21d说明了这种简单算法所存在的问题。对于那些起始于同一y值的线段，它们在表中的次序会影响活化边表的大小。例如，图2-21d中的线段BC一直包含在边表中，这样，就可能增加所需处理的信息量。



a)

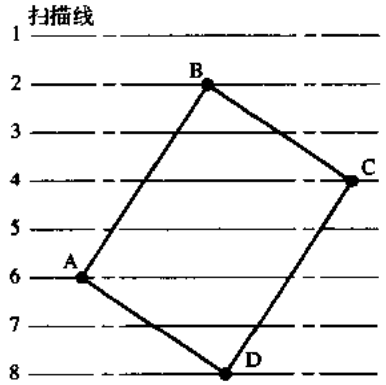
扫描线	1	2	3	1	2	3	1	2	3
	BC ← b	BC	BC	BA ← b	BA ← b	BA	BD ← b	BD ← b	BD ← b
	BA	BA ← b	BA	BC	BC	BC ← b	BA	BA	BA
	BD ← e	BD	BD ← b	BD ← e	BD	BD	BC ← e	BC	BC
	CD	CD ← e	CD	CD	CD ← e	CD	CD	CD ← e	CD
	AD	AD	AD ← e	AD	AD	AD ← e	AD	AD	AD ← e
	b)			c)			d)		

图 2-21

2.7.3 排序活化边表

这些问题和其他类似问题可以用附加数据结构加以解决。而且求图形中每条线段与各条扫

描线交点的计算方法也可以简化。首先，对图形的所有线段采用y桶排序法[⊖]。图2-22b所示的y桶排序法仅仅为每条扫描线建立一个存储单元或者称为桶。例如，对于1024条扫描线，就建立1024个桶。考察显示表的线段时，将每条线段的信息置入与该线段的最大y值相对应的桶中。对于简单的单色（黑白）线画图形，每条线段只需记录三个量，即在该桶相对应的扫描线上的x的截距，在相邻扫描线上x截距的变化量Δx以及该线段穿过的扫描线的条数Δy。对于简单图形，大



a)

y 桶

1 null
2 $x_{BA}, \Delta x_{BA}, \Delta y_{BA}, x_{BC}, \Delta x_{BC}, \Delta y_{BC}$
3 null
4 $x_{CD}, \Delta x_{CD}, \Delta y_{CD}$
5 null
6 $x_{AD}, \Delta x_{AD}, \Delta y_{AD}$
7 null
8 null

b)

活化边表

扫描线 3: $x_{BA} + \Delta x_{BA}, \Delta x_{BA}, \Delta y_{BA} - 1, x_{BC} + \Delta x_{BC}, \Delta x_{BC}, \Delta y_{BC} - 1$

扫描线 5: $x_{BA} + 3\Delta x_{BA}, \Delta x_{BA}, \Delta y_{BA} - 3, x_{CD} + \Delta x_{CD}, \Delta x_{CD}, \Delta y_{CD} - 1$

扫描线 7: $x_{CD}, 3\Delta x_{CD}, \Delta x_{CD}, \Delta y_{CD} - 3, x_{AC} + \Delta x_{AD}, \Delta x_{AD}, \Delta y_{AD} - 1$

c)

y 桶

1
2 1
3
4 8
5
6 12
7
8

索引表

1 x_{BA}
2 Δx_{BA}
3 Δy_{BA}
4 x_{BC}
5 Δx_{BC}
6 Δy_{BC}
7
8 x_{CD}
9 Δx_{CD}
10 Δy_{CD}
11
12 x_{AD}
13 Δx_{AD}
14 Δy_{AD}
15

= 结束或空。最后的扫描线。

d)

图2-22 y桶排序、活化边表以及顺序索引数据结构

⊖ 桶排序是基数等于桶的数量（扫描线数目）的基数排序（radix sort）。参见Knuth[Knut73]。

部分y桶都是空的。

当前扫描线的活化边表可从该扫描线的y桶获得信息来构成。将x截距沿扫描线排序，并对活化边表进行扫描转换。在活化边表扫描转换以后，活化边表中每条线段的Δy值减1，如果$\Delta y < 0$，从活化边表中剔除该线段。最后，活化边表中每条线段在新的扫描线上的x截距等于其原来截距值与Δx之和。对所有扫描线重复上述过程。图2-22c是图2-22a中简单线画图形在扫描线3、5、7上的活化边表。

如果y桶的大小是固定的，那么可用于储存每条扫描线上交点的存储量也是固定的。这就相当于预先限定任一扫描线上的最大交点数。这样就限制了图形的复杂性。避免出现这种局限的一种技术是在数据结构中采用顺序索引表。这时，每个y桶只包含一个指针，它指向数据结构中始于该扫描线的第一条线段的那个单元。图2-22a的顺序索引表及其数据结构如图2-22d所示。对于所示的特定数据结构，给定扫描线的数据以三个为一组成组地存取，直到出现空行或结束符号为止。

对于垂直或近于垂直的线段用确定线段与各扫描线交点的方法是可行的。然而对于接近水平的线段这样只能得到很少几个交点。这样表示的线段是无法令人接受的。一种简单的解决方法是求出线段在两条相邻扫描线上的交点，并激活交点之间的所有像素（见图2-23）。对于水平线段的扫描转换则只用到两个端点的数据。

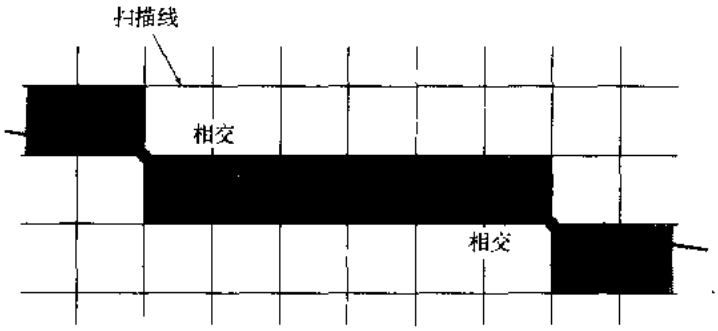


图2-23　处理接近水平线段的简单扫描转换方法

2.7.4　使用链表的活化边表

由于在每一帧时间内都要将整个图形处理一遍，因此实时扫描转换适用于高度交互性的图形。当采用y桶排序法时，若要增添或删除显示表中的线段，只需在相应的y桶及相关的数据结构中增添或删除它们即可。对于长度固定的y桶，这很容易实现（参见图2-22b）。为了在显示过程中方便地增添或删除线段，通常采用链表数据结构，如图2-24所示。由图2-24b可以看到，链表要求说明每组数据的终点，该扫描线上下一组数据的位置（例如表中第4项）和链的终结。如果给图形增添一条BD线段，那么链表修改的结果如图2-24d所示。有关线段BD的信息增添在数据表的尾部。显示处理器根据第8项上修改后的链接指令指向这一单元。如果从图形中删除线段BC，则链表按图2-24f进行修改。注意到这时第4项中的链接指令已被修改，包含线段BC信息的那些单元将被跳过。

2.7.5　修改链表

这一简单例子说明了交互式图形应用中链表修改的基本概念。然而它并未给出所有细节。

例如，由上所述，除非那些“丢失”的单元（图2-24f的表中5－8项）可以重新利用或对表进行信息压缩，否则表的长度一般会不断地增长。

有时把这种处理称为“垃圾回收”。“垃圾回收”技术使用下压式堆栈,即先入后出(FILO)。简而言之，通过将链表的第一个地址压入堆栈来启动堆栈。将扫描线信息放入链表之前，从栈顶弹出(获得)链表的下一可用位置。所有的y桶信息插入链表之后，在下一个位置上放置终止标志，并将接下来的位置的地址压入堆栈中。

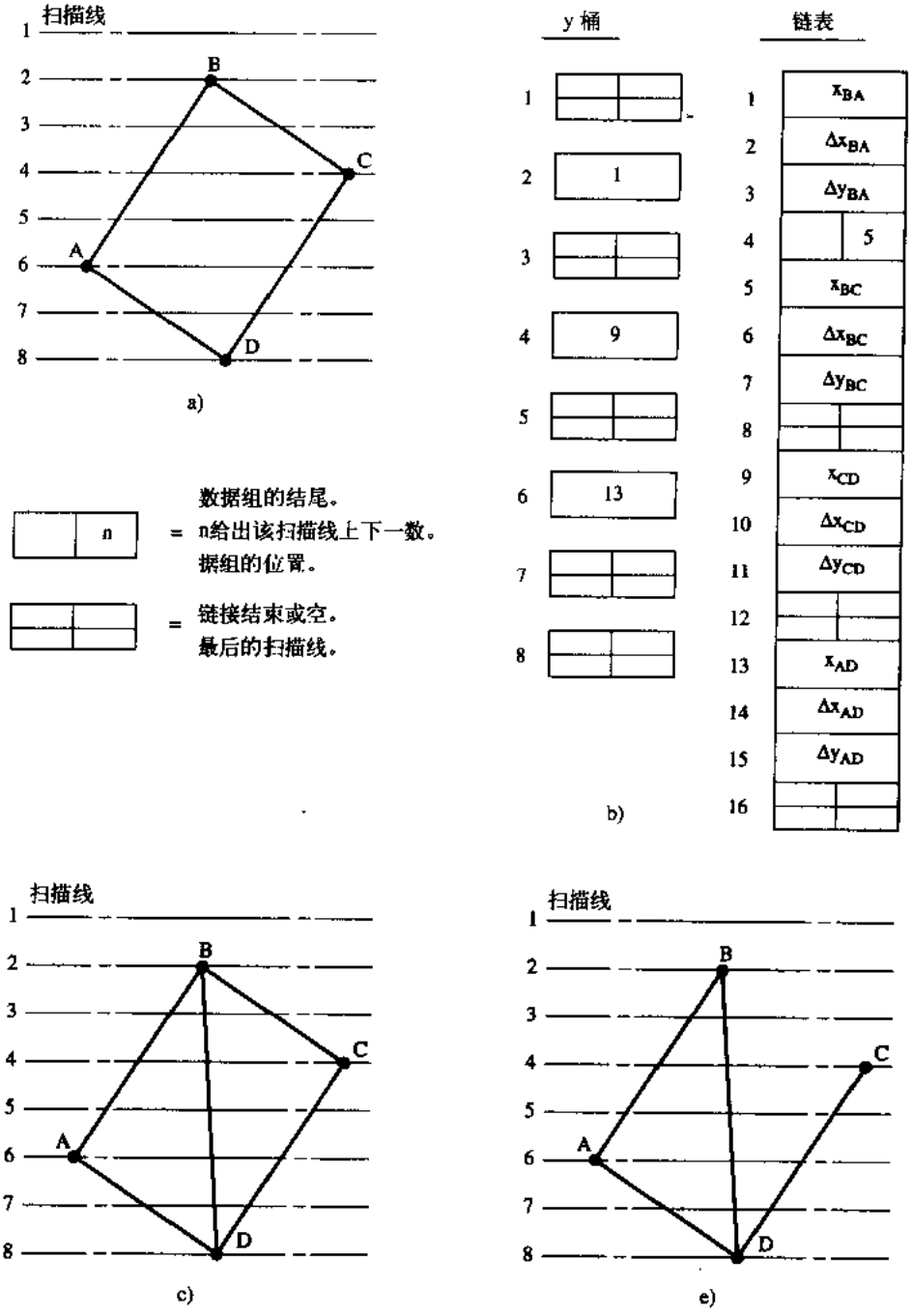


图2-24 应用于交互式图形的y桶排序和链表

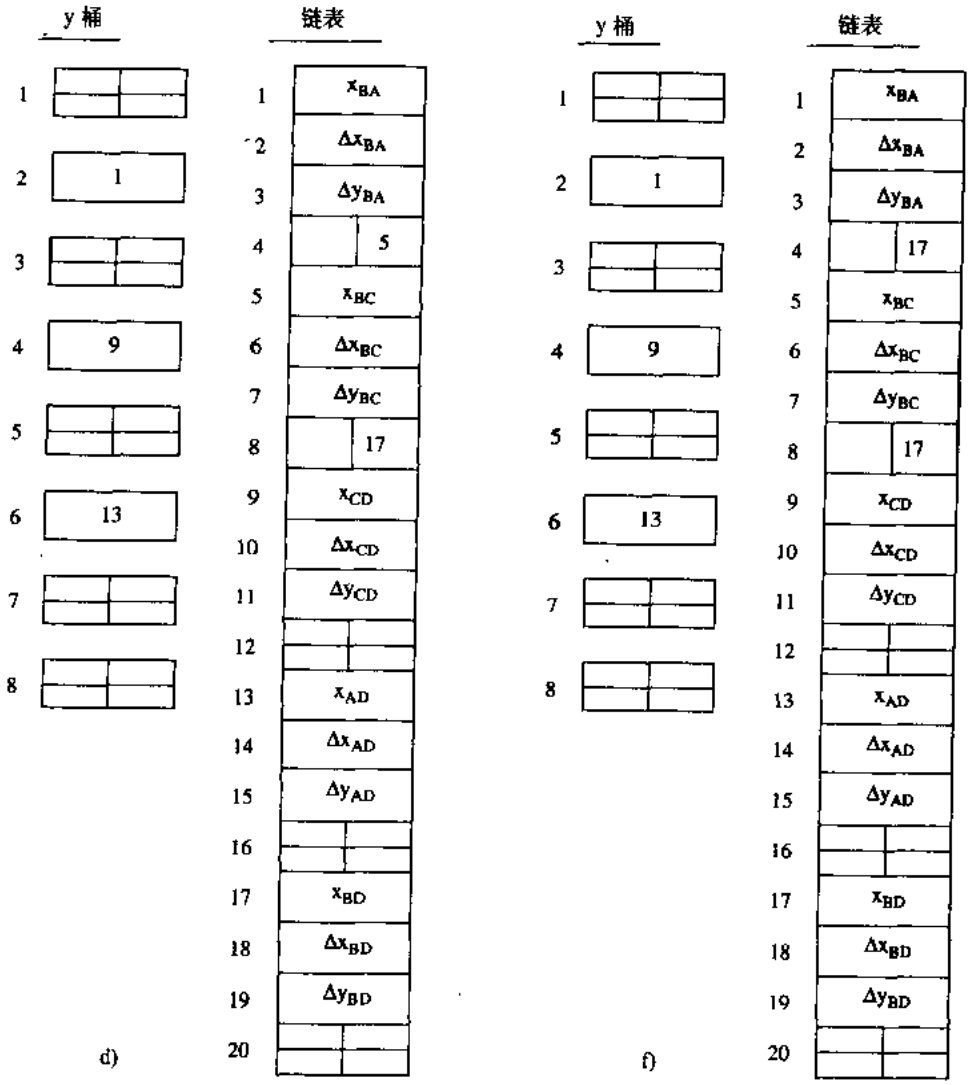


图 2-24（续）

向链表添加元素的部分伪代码如下:

get pointer into the liked list from the scan line *y* bucket
using pointer, follow linked list to the terminator flag
Pop next available link address from the stack
change terminator flag to the new link address
insert the new data
add terminator flag
then
if the stack is empty
　　Push the next address onto the stack
end if

从链表中删除内容要稍微复杂一些。基本方法是从相应的*y*桶获得链表的初始指针，接着

找到要删除的信息。如果要删除的数据不是y桶的最后一条内容，则前一条内容的链接地址必须代之以已删除内容的链接地址；否则置以终止标志。

从链表中删除内容的部分伪代码如下：

```
get pointer into the linked list from the scan line y bucket
if (data to be deleted is the first item) then
    Push y bucket pointer onto stack
    if (forward link address) <> terminator flag then
        change y bucket pointer to the forward link address
    end if
else
    Push previous link address onto stack
    if next link item is <> terminator flag then
        change previous link address to the next link address
    else
        change link address to terminator flag
    end if
    end if
```

关于链表以及数据结构的详细讨论可参阅文献[Knut73]或[Stan80]。

2.8 图像压缩

图像压缩的两项最常用的技术是行程编码和区域图像压缩。两者都利用了图像的相关性。

2.8.1 行程编码

行程编码技术试图利用这样一个事实，即图形中大块区域往往具有相同的光强或色彩。最简单的行程编码只提供光强和在给定扫描线上具有该光强的连续像素的数目这两个信息。图2-25a是一个30 × 30光栅网格上的简单黑白线画图形及对于扫描线1、15和30的相应编码。编码数据每两个一组。第一个数表示光强，第二个数表示在同一扫描线上具有该光强的连续像素的数目，即

光　强	程　长

这样，图2-25a中扫描线1有30个像素的光强为零，即黑色或背景颜色。整个画面可编码成208个数。若采用像素存储，即每一像素一个信息（称为位图），对如图2-25a所示的30 × 30光栅，则需要900个光强值。这时，行程编码技术所达到的信息压缩比为1:4.33(0.231)。

行程编码技术易于处理实区域图形。图2-25b是该方法的图示，并给出扫描线1、15和30的编码。特别注意扫描线15的编码。图2-25b的整幅图形可用136个数来编码，数据压缩比达到1:6.62(0.151)。实区域图形编码所需要的信息量少于线框图形的编码信息量，这是因为两条边线只需要一对光强-程长值。

这种简单的行程编码技术很容易推广到彩色图形的处理。对于彩色图形，首先给出同一扫描线上具有相同颜色的相邻像素的个数，然后给出红、绿、蓝色彩枪的强度，例如

红色强度	绿色强度	蓝色强度	程　长

对于简单的彩色显示器，它的每支色彩枪只有两种状态，即完全关断（0）或完全接通（1），对于图2-25b所示的蓝底黄色三角形，扫描线15的编码如下（见表1-1）：

15	0	0	1	7	1	1	0	8	0	0	1

行程编码图形的数据压缩比可以达到1:10。其意义不仅在于节省了存储，而且在于节省了储存计算机生成的动画的空间。这也节省了照片和文件传真的传送时间，行程编码技术在这两个领域里应用十分广泛。现以储存30秒动画为例说明对存储空间的要求，设画面的分辨率为1024 × 1280 × 24，视频刷新频率为每秒30帧，所要求的存储空间为即使 1 : 4 的数据压缩比也可以极大地减少对存储的要求。

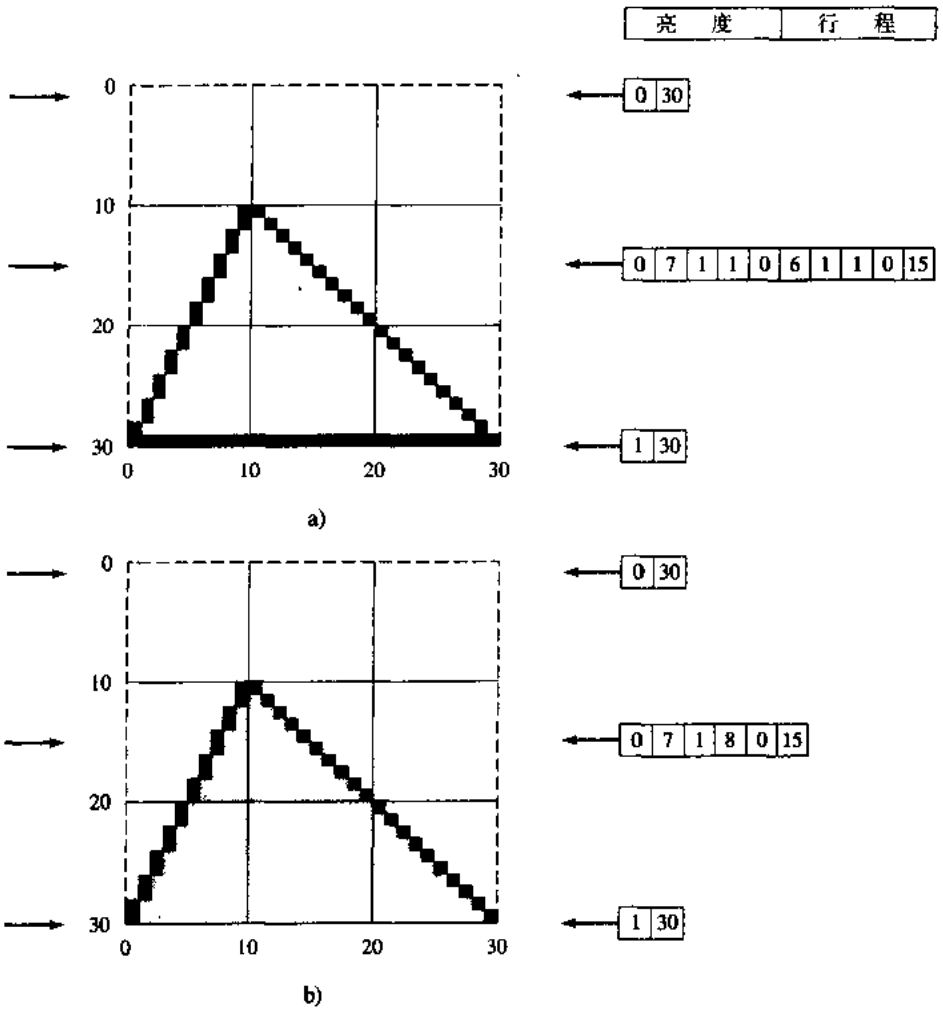


图2-25　行程编码的图形

$$(1024 \times 1280 \times 24 \times 30 \times 30) / (8\text{位/字节}) = 3456\ MB$$

行程编码技术也有它的不足之处。由于行程是顺序存储的，因此图形中线段或文字的增

删不但困难而且费时，要包含大量的编码和解码的操作。最后，短程长图形所要求的存储量可能达到按像素存储的两倍（见图2-26）。图2-26是一幅黑白相间的垂直条纹图，条纹宽度为1个像素。其行程编码是

1	1	1	0

并重复15次。这样，对于这幅图形采用行程编码需储存1800个值，而按像素存储只需要900个数据。这是数据压缩比为2的负的数据压缩。

好的行程压缩技术可以发现会导致负数据压缩的位置，自动切换到原像素存储。参考文献[Glas91]中提出的一种简单的自适应行程编码技术使用编码像素符号，使之能够从行程编码与原像素存储之间切换。特别地，如果编码像素符号为负，则使用原像素存储，否则,使用行程编码。这两种情况下，写在负号后的数字都表示像素的亮度，或者RGB值的强度，例如:

-3	1	0	0	1	0	1	0	0	1	6	1	2	6

定义了三个用RGB值(1,0,0)、(0,1,0)、(1,0,0)表示的像素,然后是六个用RGB值(1, 2, 6)表示的像素。

Laws（见[Laws75]）、Hartke(见[Hart78])，以及Peterson等人（见[Pete86]）研究了实现行程编码原理的有效方法。

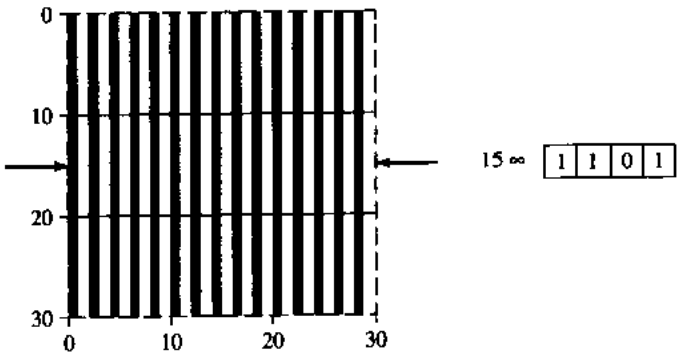


图2-26 行程编码技术对于短程长的局限性

2.8.2 区域图像压缩

行程编码想要利用一幅图像扫描线的相关性,即扫描线中下一像素是否与当前像素有相同的特征。扫描线相关性将整幅图像当作一维数组进行处理。

区域图像压缩技术想要利用图像的区域相关性,即相邻区域的像素具有相同的特征。区域图像压缩技术将图像当作二维区域进行处理。

虽然Warnock风格的不确定分而治之(divide & conquer)算法(参见4.4节)也是可行的。但把图像当作按扫描线次序排列的固定大小像素块更为方便。虽然像素块可以是任意大小，2×2，4×4，8×8，16×16，32×32，等等，这里我们只讨论8×8像素块。

如图2-27所示，两个8×8像素块的强度值来自于两个不同的图像。图2-27a的像素块来自于一幅单色连续色调、256级灰度(强度)的自然图像。图2-27b的像素块来自于一幅计算机生成的简单的256级灰度的单色图像。

首先将每一块8×8像素块的像素亮度值相加，除以64并取整,该值是像素块的平均像素强

度。对于图2-27a所示的连续色调图像，它的平均强度是157。然后，每一像素强度减去平均像素强度获得差分图像，结果如图2-28所示，它的第一个元素是平均强度。存储平均强度会增加存储量，这样做能提供额外功能。

139	144	149	153	155	155	155	155
144	151	153	156	159	156	156	156
150	155	160	163	158	155	156	156
159	161	162	160	160	159	159	159
159	160	161	162	162	155	155	155
161	161	161	161	160	157	157	157
162	162	161	163	162	157	157	157
162	162	161	161	163	158	158	158

a)

160	160	160	160	160	255	255	255
160	160	160	160	160	255	255	255
160	160	160	160	160	255	255	255
255	255	255	255	255	2	2	2
255	255	255	255	255	2	2	2
255	255	255	255	255	2	2	2
255	255	255	255	255	2	2	2
255	255	255	255	255	2	2	2

b)

图2-27　8 × 8像素块

a) 单色连续色调自然图像　b) 计算机生成的单色简单图像

以前章节讨论过的行程编码技术在这里用于差分图像的每一行。对于图2-28a所示的差分图像，结果如图2-29a所示；对于图2-28b所示的差分图像，结果如图2-29b所示。

157							
−18	−13	−8	−4	−2	−2	−2	−2
−13	−6	−4	−1	2	−1	−1	−1
−7	−2	3	6	1	−2	−1	−1
2	4	5	3	3	2	2	2
2	3	4	5	5	−2	−2	−2
4	4	4	4	3	0	0	0
5	5	4	6	5	0	0	0
5	5	4	4	6	1	1	1

a)

173							
−13	−13	−13	−13	−13	82	82	82
−13	−13	−13	−13	−13	82	82	82
−13	−13	−13	−13	−13	82	82	82
82	82	82	82	82	−171	−171	−171
82	82	82	82	82	−171	−171	−171
82	82	82	82	82	−171	−171	−171
82	82	82	82	82	−171	−171	−171
82	82	82	82	82	−171	−171	−171

b)

图2-28　8 × 8像素块的差分图像

a) 图2-27a的单色连续色调自然图像　b) 图2-27b的单色计算机生成图像

157								
−4	−18	−13	−8	−4	4	−2		
−5	−13	−6	−4	−1	2	3	−1	
−8	−7	−2	3	6	1	−2	−1	−1
−5	2	4	5	3	3	3	2	
−5	2	3	4	5	5	3	−2	−2
4	4	−1	3	3	0			
−5	5	5	4	6	5	3	0	
−5	5	5	4	4	6	3	1	

a)

173			
5	−13	3	82
5	−13	3	82
5	−13	3	82
5	82	3	−171
5	82	3	−171
5	82	3	−171
5	82	3	−171
5	82	3	−171

b)

图2-29　行程编码数据

a) 图2-28a的单色连续色调自然差分图像　b) 图2-28b计算机生成的单色简单差分图像

检查一下图2-29a会发现，存储8 × 8像素块需要64条信息。因此并没有压缩数据。但是，对

于图2-28b的差分图像，行程编码将所需的信息条数减少到33，数据压缩比例为0.515，如图2-29b所示。

还有其它行程编码技术可用于基本8×8像素块。例如，像素块可以子分为2×2子像素块，进一步利用区域的相关性，如图2-30a所示。每一2×2子像素块以Z字形进行行程压缩，如图2-30a左上角于像素块所示。当用于图2-28a所示的差分图像时，需要78条信息，压缩因子为1.22，即压缩后信息量比原信息量还大!对图2-28b所示的差分图像采用同一算法，压缩因子为0.843。

157

−18	−23	−8	−4	−2	−2	−2	−2
−16	−6	−4	−1	2	−1	−1	−1
−7	−2	3	6	1	−2	−1	−1
2	4	5	3	3	2	2	2
2	3	4	5	5	−2	−2	−2
4	4	4	4	3	0	0	0
5	5	4	6	5	0	0	0
5	5	4	4	6	1	1	1

a)

173

−13	13	−13	13	−13	82	82	82
−13	−13	−13	−13	−13	82	82	82
−13	−13	−13	−13	−13	82	82	82
82	82	82	82	82	−171	−171	−171
82	82	82	82	82	−171	−171	−171
82	82	82	82	82	−171	−171	−171
82	82	82	82	82	−171	−171	−171
82	82	82	82	82	−171	−171	−171

b)

图2-30 Z字形行程编码技术

a) 2×2 b) 8×8

还有一种Z字形行程压缩模式可以压缩整个8×8像素块，如图2-30b所示。对图2-28a的

差分图像采用这种Z字形模式压缩，需要67条信息，压缩因子为1.05，压缩率为负值。对图2-28b采用该算法，需要32条信息，压缩因子为0.5。目前这种Z字形行程编码模式用于JPEG(Joint Photographic Expert Group)标准(参见[Nels92])。显然，能够压缩的数据量依赖于图像的特性和编码技术。

如果用户在一大型图像库中查找，想要找到一幅特定的图像；或者如果用户在远程观察一幅图像，图像的累进传输或生成就很重要。通常用户只需看图像的粗略大概，以此决定是否要观察该图像的细节部分。此外，对于复杂图像，用户还可能只对图像的一部分感兴趣，只需一部分图像的细节。这就是我们要保存8×8像素块的平均强度和创建差分图像的原因。

对于8×8像素块，我们还提出了图像的层次表示。在确定和存储了8×8像素块的平均强度之后，像素块子分为四个4×4像素块。计算出每一4×4像素块的平均强度，并以顺时针次序存储，从8×8像素块的左上角开始。将每一4×4像素块进一步子分为2×2像素块，确定平均强度，并从4×4像素块的左上角开始以顺时针次序存储。最后，根据8×8像素块的平均强度值(如图2-28所示)，一个像素一个像素地行程编码差分图像。该技术类似于纹理生成(参见5-12)中的mipmap。结果如图2-31a和图2-31b所示。分析一下图2-31a可知，自然图像2-27a的压缩因子为1.31。对于图2-27b的计算机生成图像，现在的数据压缩因子为0.828，而不是0.515。但是，如果图像用8×8像素块的平均强度显示，而不是用每一像素的强度进行显示，则1024×1024图像的传输压缩因子是0.0152(16k/1024k)，如果用2×2像素块的平均强度显示，则压缩因于为0.25(256k/1024k)。

157								
153	156	161	158					
144	152	156	155					
156	161	158	157					
160	161	158	156					
162	161	160	157					
−4	−18	−13	−8	−4	4	−2		
−5	−13	−6	−4	−1	2	3	−1	
−8	−7	−2	3	6	1	−2	−1	−1
−5	2	4	5	3	3	3	2	
−5	2	3	4	5	5	3	−2	−2
4	4	−1	3	3	0			
−5	5	5	4	6	5	3	0	
−5	5	5	4	4	6	3	1	

a)

173			
183	189	255	65
160	160	207	255
207	207	168	128
255	255	128	2
255	255	128	2
5	−13	3	82
5	−13	3	82
5	−13	3	82
5	82	3	−171
5	82	3	−171
5	82	3	−171
5	82	3	−171
5	82	3	−171

b)

图2-31　用于累进传输的8×8像素块编码

a) 单色连续色调自然图像　b)计算机生成的单色简单图像

2.9　显示直线、字符和多边形

光栅图形显示中的基本元素是显示直线、字符和多边形。

2.9.1　线段显示

回顾第1章(见1.2节)关于帧缓冲存储器的讨论，可以在概念上将帧缓冲看作一个简单的绘制平面，假设帧缓冲存储器是一个简单器画线显示器。首先清除帧缓冲存储器或者把它置成

背景光强或颜色。接着根据Bresenham算法或者DDA算法对直线光栅化，然后将相应像素写入帧缓冲存储器。当一帧图形完成时，显示控制器按扫描线顺序读出帧缓冲存储器，并把结果传送到视频监示器上。

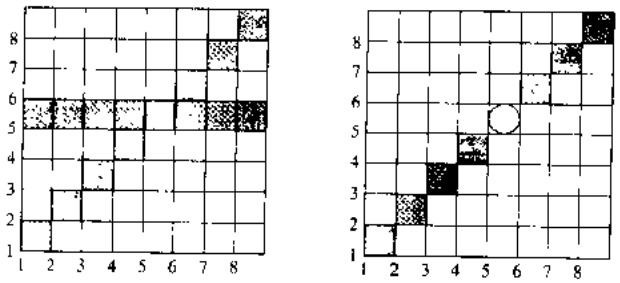


图2-32 帧缓冲存储器中线段的擦除

若需要有选择地擦除线段，那么可采用光栅化算法将缓冲器中相应的像素写成背景光强或颜色，这样线段就被擦除了。但如图2-32所示，这种技术也存在一个问题。即若被擦除直线与另一直线相交，那么在另一直线上将留下一个小空隙。图2-32a上给出两条相交直线。如果将$y = 5$水平线上的像素显成背景光强或颜色来擦除这条线，那么另一条直线上像素(5, 5)处将留下一个小空洞。查出和填上这些小空洞并不困难。实际上只要求出被删除直线与图形中其他直线的交点即可。但是对于复杂图形来说，这是一件费时的工作。

可以利用包围盒检查（boxing test）或最小最大检查（minimax test）来减少所需工作量。这一方法如图2-33所示。根据线段ab的x和y的最大和最小值画一虚线表示的盒，只有穿过这个盒的直线才有可能与ab相交。然后用如下算法对每一线段进行检查。

```
minimax or boxing test
    if Xlinemax < Xboxmin or
      Xlinemin > Xboxmax or
      Ylinemax < Yboxmin or
      Ylinemin > Yboxmax
    then
        no intersection
    else
        calculate intersection
    end if
finish
```

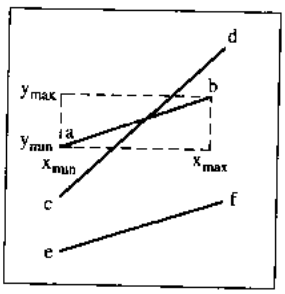


图2-33 包围盒检查或最大最小检查

2.9.2 字符显示

字母数字字符是利用掩码写入帧缓冲存储器的。字符掩码是包含表示该字符的像素图案的一小块光栅（见图2-34a）。某些应用问题的特定符号如电阻、电容或数学符号等也可用字符掩码表示。掩码本身仅含一些二值数，以指出掩码中的像素是否用于表示字符或符号。在简单的黑白显示器中，当像素用于表示字符或符号时，对应的二进制值通常取为1，否则为0。在彩色显示器中则用更多像素位数以表示色彩的浓淡和查色表的指针。

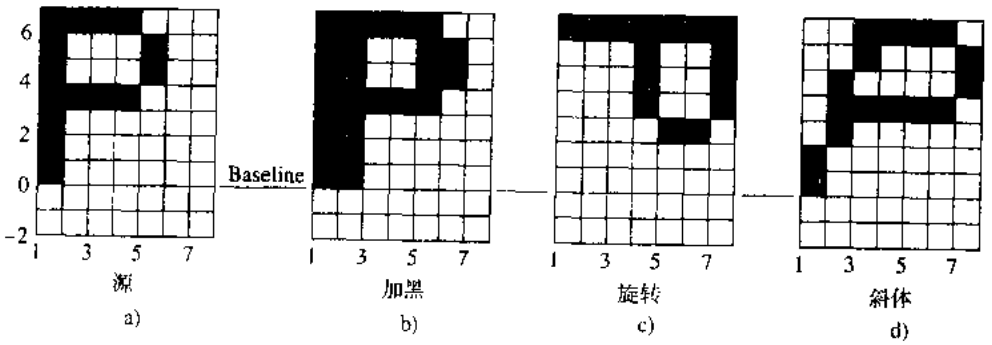


图2-34 变换后的字符掩码

指定掩码的原点在帧缓冲存储器中的坐标（x_0, y_0）就可以将字符掩码中每像素相对x_0, y_0平移后的值写入帧缓冲存储器。对于二值掩码，实现该过程的简单算法如下：

Mask insertion *into the frame buffer*
Xmin, Xmax, Ymin, Ymax *are the limits of the mask*
x_0, y_0 *is the location in the frame buffer*

```
for j = Ymin to Ymax − 1
    for i = Xmin to Xmax − 1
        if Mask(i, j) <> 0 then
            write Mask(i,j) to the frame buffer at (x0 + i, y0 + j)
        end if
    next i
next j
finish
```

将帧缓冲存储器中相应像素置成背景光强或颜色就可擦除帧缓冲存储器中的字符。

当字符写入帧缓冲存储器后，可对字符掩码进行修改以获得不同字体或方向。图2-34是几个简单的修改例子。图2-34a是原来的字符掩码。将掩码写入帧缓冲存储器中两个相邻单元x_0和x_0+1，可以得到粗体字（见图2-34b）。字符可以旋转（见图2-34c）或倾斜，使字符成为斜体（见图2-34d）。

2.9.3 实区域扫描转换

前面只讨论了光栅扫描设备上线段的表示问题。然而光栅扫描的特点是它具有表示实区域的能力。根据边或顶点的简单描述生成实区域的过程称为实区域扫描转换或多边形填充或轮廓线填充。有许多种填充轮廓线的方法，一般可以把它们分成两大类：扫描转换和种子填充。

扫描转换技术是按扫描线的顺序确定某一点是否位于多边形或轮廓线范围之内。这些算法一般从多边形或轮廓线的“顶部”开始进行到它的“底部”。扫描转换方法同样适用于画线

显示器。在画线显示器中用于画剖面线或轮廓的阴影线（见图2-35）。

种子填充方法首先假定封闭轮廓线内某点是已知的。然后算法开始搜索与种子相邻且位于轮廓线内的点。如果相邻点不位于轮廓线内，那么就已经到达轮廓线的边界。如果相邻点位于轮廓线之内，那么这一点就成为新的种子点，然后继续递归地搜索下去。种子填充算法只适用于光栅扫描设备。

图2-35 剖面线或阴影线

2.10 多边形填充

一般的封闭轮廓线都是简单的多边形。若轮廓线由曲线构成，则可用适当的多边形逼近之。填充多边形的最简单方法是检查光栅的每一像素是否位于多边形内。由于大多数像素不在多边形内，因此这种方法的效率很低。计算多边形的包围盒可以减少计算量。所谓包围盒就是包含该多边形的最小矩形。只有在包围盒内的那些点需要检查（见图2-36）。图2-36a中的多边形采用包围盒后，可极大地减少所需要检查的像素个数。然而，对图2-36b所示的多边形，可以减少的检查工作量却要少得多。

多边形的扫描转换

除边界线外，相邻的像素几乎都具有相同的特性，这种性质称为空间连贯性。根据这一事实提出了一种比内部检查法更为有效的方法。光栅扫描图形显示的扫描线上的相邻像素几乎也都具有相同的特性。这就是所谓扫描线的连贯性。

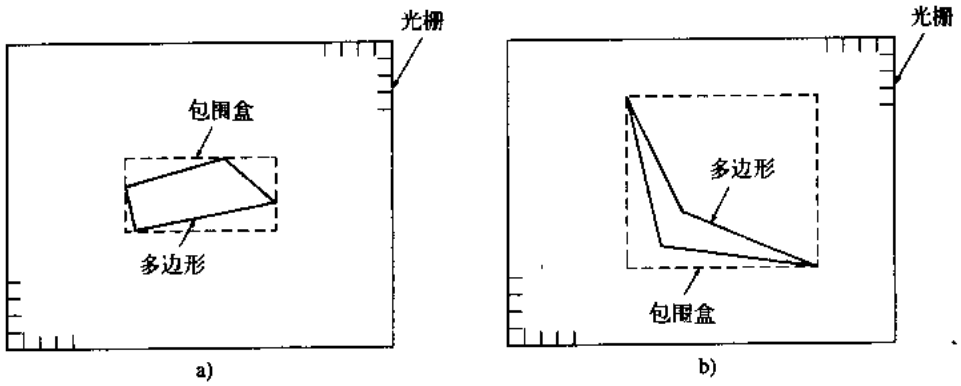


图2-36 多边形包围盒

在给定的扫描线上，像素的这种特性只有在多边形的边和该扫描线交点处才会发生变化。

这些交点把扫描线划分成一些段。对于图2-37所示的简单多边形，扫描线2与多边形交于x = 1和x = 8。这两个交点把扫描线分成如下三段：

$x < 1$　　多边形外

$1 \leqslant x \leqslant 8$　　多边形内

$x > 8$　　多边形外

类似地，扫描线4被分成五段

$x < 1$　　多边形外

$1 \leqslant x \leqslant 4$　　多边形内

$4 < x < 6$　　多边形外

$6 \leqslant x \leqslant 8$　　多边形内

$x > 8$　　多边形外

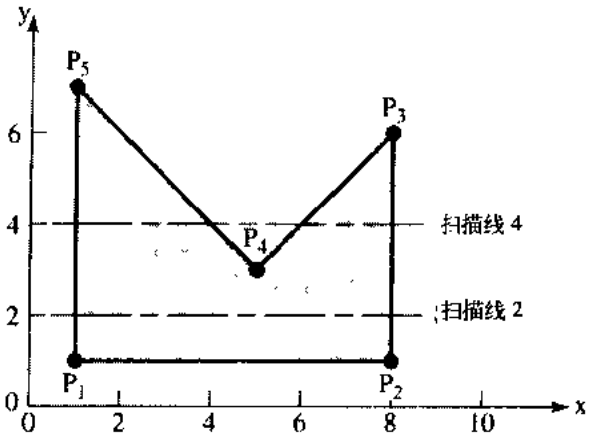


图2-37　实区域扫描转换

扫描线4上的交点并不一定要按自左至右的顺序求出。例如，若多边形由顶点$P_1P_2P_3P_4P_5$表示，其相邻顶点对组成的边为P_1P_2、P_2P_3、P_3P_4、P_4P_5、P_5P_1，那么该多边形的边与扫描线4的交点就按8、6、4、1顺序求出。然后按x的递增顺序对其排序，即为1、4、6、8。

在确定扫描线上像素的光强、色彩或明暗时，需将经过排序的交点配对。每对交点所确定的区间均取多边形的光强或色彩。交点对之间的区间则取背景光强或色彩。当然，从扫描线的起点到第一交点之间以及从最后一个交点到扫描终点之间应置成背景光强或色彩。对于图2-37所示的多边形，在扫描线4上从0到1、从4到6以及从8到10之间的像素被置成背景色，而从1到4以及从6到8之间的像素被置成多边形的光强或色彩。

要精确决定哪些像素被激活，则更需慎重处理，现以图2-38所示的简单矩形为例说明之。矩形顶点的坐标为(1,1)、(5,1)、(5,4)和(1,4)。扫描线1到4与该矩形交于x=1和5。像素仍如前所述由其中心的坐标表示，那么在每一条扫描线上的x坐标为1、2、3、4和5的像素将激活。其结果如图2-38a所示。由此可见，被激活像素所覆盖的面积等于20个单位，而矩形实际面积却等于12个单位。还有，假设一个矩形的端点坐标是(5,1)、(10,1)、(10,4)和(5,4)，那么它与原来矩形邻接，表示这个矩形的左边的像素就会和原来矩形的右边重合。如果这两个矩形有不同的颜色，写像素又是采用异或的形式，那么将会导致奇怪的颜色。

只要对扫描线坐标系统和激活测试方法进行适当修改就可解决这一问题（见图2-38b）。

设扫描线通过每行像素的上边缘，即如图2-38b所示的位于每行像素的半宽线。对激活检查可作如下修改，即考察位于交点右边的像素的中心是否落在区间之内。这里像素的地址仍然由其中心坐标表示。图2-38b给出了该方法所产生的正确结果。

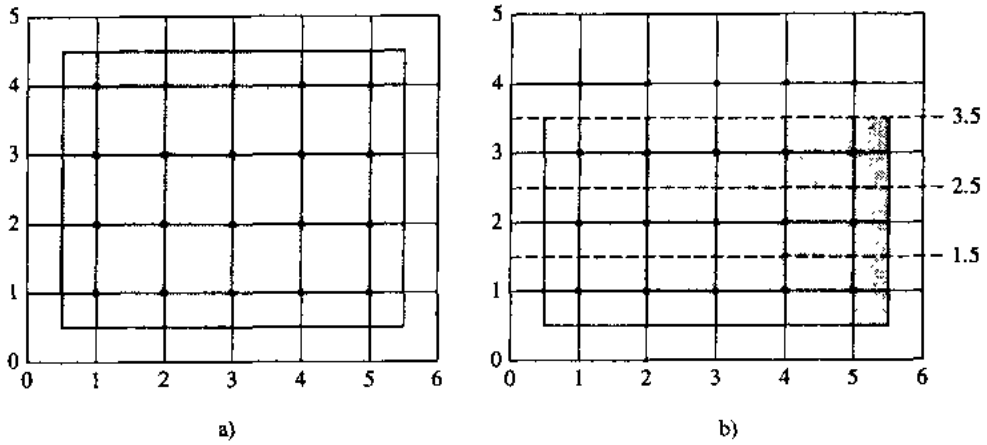


图2-38 扫描线坐标系

实际上，为了正确处理邻接的多边形，中心扫描线右边技术不考虑多边形顶部和右部的边。这偶尔也会产生瑕疵，比如，在相邻多边形中没有被激发的像素会产生空洞，这些瑕疵是数字化具有无限频率的连续模拟信号时出现的走样现象（见2.16节）。

水平边不与扫描线相交，因此不予考虑。但这并不意味着无法产生水平边。由图2-35可知，它们可由顶部以及底部的像素行构成。图2-38b显示出经修改的扫描线坐标系产生的正确的多边形的顶边和底边。

当扫描线恰好交于多边形的顶点时（见图2-39）会出现新的问题。采用中心扫描线转换时，例如 $y = 3.5$的扫描线与多边形的交点为奇数个，其交点的x坐标分别为2、2和8。因此按交点对划分像素会导致错误的结果，即位于(0, 3)和(1, 3)的像素取背景颜色，位于(2, 3)的像素取多边形颜色，位于(3, 3)到(7, 3)之间的像素取背景颜色以及位于(8, 3)和(9, 3)的像素取多边形颜色。由观察可知，当扫描线与多边形顶点相交时该交点只计为一个。这时，对于$y = 3.5$的扫描线仍可得到正确的结果。然而，扫描线$y = 1.5$在(5, 1)处应计为两个交点，这表明该方法是不正确的。对于这条扫描线，按交点对像素划分的结果是正确的，即只有像素(5, 1)

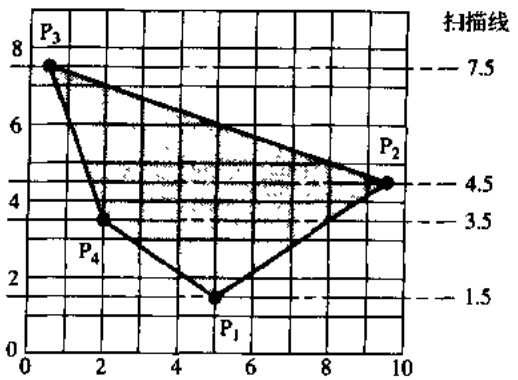


图2-39 扫描线交点的奇偶性

取多边形颜色。如果在顶点处只计一个交点，那么从(0, 1)到(4, 1)之间的像素取背景色，而从(5, 1)到(9, 1)之间的像素取多边形色。

正确的方法是当扫描线与多边形顶点交于多边形的局部最高点或局部最低点时，交点应计两次，否则只计一次。确定多边形某顶点是否是局部最高点或最低点只需检查交于该顶点的两条边的另两个端点，如果它们的y值都大于顶点y值，则该顶点是局部最低点。如果它们的y值都小于顶点的y值，则该顶点是局部最高点。如果一条边大于顶点的y值，而另一条边小于顶点的y值，则该顶点既不是局部最高点也不是局部最低点。在图2-39中，P_1是局部最低点，P_3是局部最高点，P_2和P_4既不是局部最高点也不是局部最低点。故扫描线在P_1和P_3处的交点应计两次，而在P_2和P_4处只计一次。

2.11 简单的奇偶扫描转换算法

简单的多边形扫描转换算法利用一个事实,即扫描线多边形相交常常一对一对地发生，因此使用一个称为奇偶位的标志来确定扫描线上的某一像素在多边形内部还是在外部。在扫描线的开始处，奇偶位设置为0，表示扫描线在多边形的外部。扫描线与多边形第一次相交时，奇偶位设置为1，表示扫描线现在在多边形的内部。到下一个交点时，奇偶位设置为0，表示扫描线已通过多边形，又在多边形的外部。当奇偶位为0时，像素设置为多边形背景色。当奇偶位为1时，像素设置为多边形色。图2-40给出了一个简单凹多边形中两条扫描线的奇偶位值。使用中心扫描线的约定，在 $y = 2.5$ 处计算出扫描线2的交点为$x = 1,8$。对扫描线4，在 $y = 4.5$ 处计算出交点的x坐标为1、3.5、6.5和8。在每一交点处奇偶位都发生变化，如图2-40所示。 下面给出实现简单奇偶扫描转换算法的伪代码。

A simple parity polygon scan conversion algorithm

sub Paritysc(y, left, right, numint, stack)

prior to calling paritysc, the intersections on a scan line are sorted into scanline order and pushed onto a stack. paritysc is then called for each scan line

y	=	*the current scan line value*
left	=	*the starting value for the left most pixel*
right	=	*the ending value for the right most pixel*
numint	=	*the number of intersections on the scan line*
stack	=	*contains the intersections sorted in scan line order*
x	=	*the current pixel on the scan line*

Pop *is a function that removes intersection values from a stack*

initialize the variables

```
i = 1
parity = 0
oldparity = parity
oldxi = left − 1
x = left
Pop stack(xi)      get the first intersection value
for x in the range  left ≤ x ≤  right on the scan line
while x ≤ right
    for each intersection on the scan line
```

```
        while i ≤ numint
            notice the rounding so that the right side of the pixel is checked
            xi = oldxi      takes care of duplicate intersection values
            if x ≥ int(xi + 0.5) or xi = oldxi then
                oldparity = parity
                parity = parity + 1
                if parity > 1 then parity = 0
                oldxi = xi
                i = i + 1
                Pop stack(xi)      get the next intersection value
            end if
            if parity = 1 and x + 0.5 ≤ xi then
                call setpixel (x, y, red, green, blue)
            end if
            x = x + 1
            if x ≥ right then i = i + 1
        end while
        x = x + 1
    end while
end sub
```

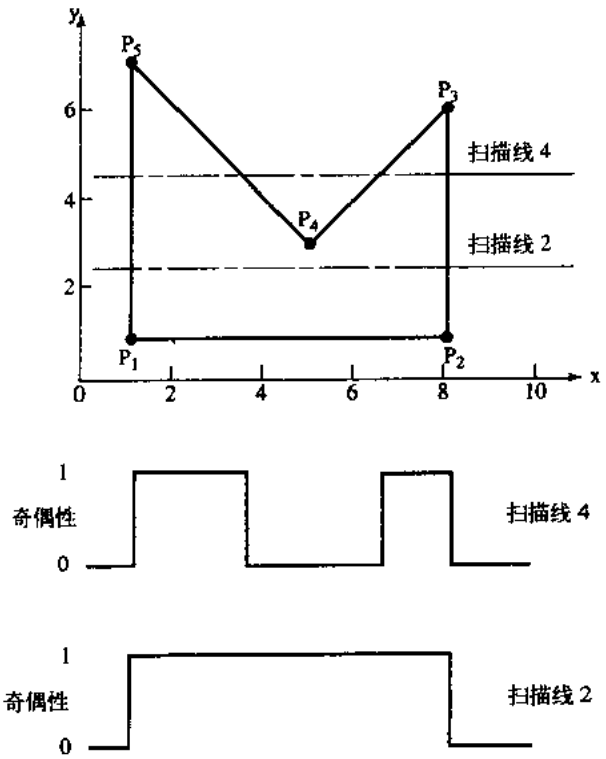


图2-40 奇偶实体区域扫描转换

图2-27和图2-39所示的多边形的扫描转换结果分别如图2-41a和图2-41b所示。注意，正如所期望的那样，扫描转换的多边形位于实际多边形的左下边，这是采用中心像素寻址、并且忽略多边形的右上边界的结果。

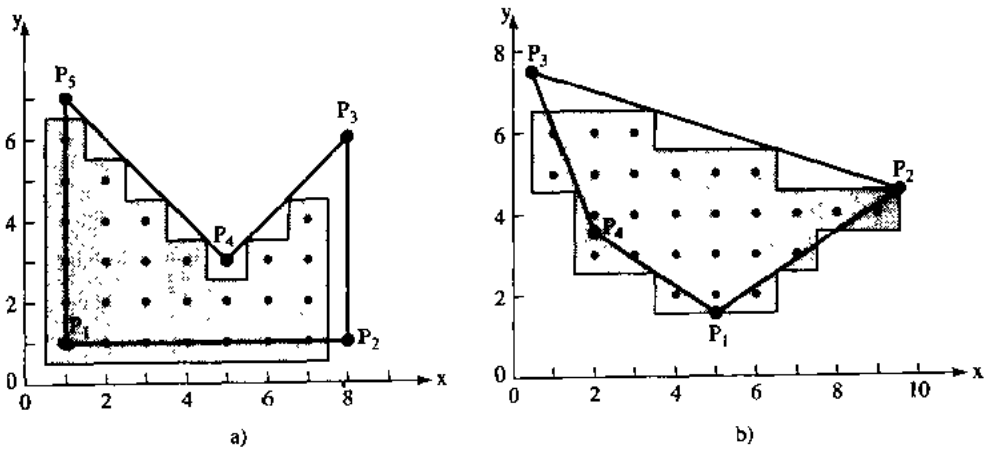


图2-41　奇偶扫描转换多边形

但是，根据代数几何知识(参见O'Rourke的[ORou94])，具有n个顶点（按反时针方向计算）的任意凹多边形和凸多边形的面积是

$$A_p = \frac{1}{2}\sum_{i=1}^{n}(x_i y_{i+1} - y_i x_{i+1})$$

这表示扫描转换多边形的面积误差是可以接受的。作为特例,图2-41a所示的多边形的面积是26.5。而扫描转换后的多边形面积是27，误差为1.9%。同样，图2-41b所示的多边形的实际面积是24.75，而扫描转换后的多边形面积是26，误差为5%。显示分辨率增加，则误差减少。

所有的扫描转换算法对算符 <、> 或 = 的使用都很敏感，例如，在同一简单奇偶扫描转换算法中，用>替代≥将显著地改变结果。例如，如果将命令行:

if x ≥ int(xi + 0.5) **or** xi = oldxi **then**

替换为:

if x > int(xi + 0.5) **or** xi = oldxi **then**

则图2-41a中的像素(1, 1)、(1, 2)、(1, 3)、(1, 4)、(1, 5)、(1, 6)和(6, 3)都被忽略。现在扫描转换后的面积为20，而不是27，误差为24.5%。同理,对图2-41b，则像素(1, 6)、(1, 5)、(4, 2)和(2, 3)被忽略，面积误差将增加到11%。

2.12　有序边表多边形扫描转换

另一种可有效地扫描转换实体区域多边形的方法是所谓的有序边表算法。它的基本思想是将多边形边与扫描线的交点进行排序。该算法的效率依赖于排序算法的效率。

2.12.1　简单的有序边表算法

下面给出一个特别简单的有序边表算法。

简单的有序边表算法

数据准备:

求出多边形每条边与中心扫描线的交点。为此可采用Bresenham或DDA算法。

水平边不考虑。把各个交点的坐标$(x, y+\frac{1}{2})$存储在表中。按扫描线以及扫描线上交点x值的递增顺序对该表进行排序，即当$y_1>y_2$或$y_1=y_2$和$x_1<x_2$时，(x_1, y_1)将位于(x_2, y_2)的前面。

对数据进行扫描转换:

按(x_1, y_1)和(x_2, y_2)形式成对提取已排序表的元素。这时表的构造保证有$y=y_1=y_2$以及$x_1<x_2$。在扫描线y上激活那些x整数值满足$x_1<x+\frac{1}{2}<x_2$关系的像素。

例2.8 简单的有序边表。

以图2-37所示的多边形为例。多边形的顶点为$P_1(1, 1)$、$P_2(8, 1)$、$P_3(8, 6)$, $P_4(5, 3)$和$P_5(1, 7)$。它与中心扫描线的交点为

扫描线1.5: (8, 1.5), (1, 1.5)

扫描线2.5: (8, 2.5), (1, 2.5)

扫描线3.5: (8, 3.5), (5.5, 3.5), (4.5, 3.5), (1, 3.5)

扫描线4.5: (8, 4.5), (6.5, 4.5), (3.5, 4.5), (1, 4.5)

扫描线5.5: (8, 5.5), (7.5, 5.5), (2.5, 5.5), (1, 5.5)

扫描线6.5: (1.5, 6.5), (1, 6.5)

扫描线7.5: 无

按从上到下,自左至右扫描顺序排列后的整个表如下:

(1, 6.5), (1.5, 6.5), (1, 5.5), (2.5, 5.5), (7.5, 5.5), (8, 5.5), (1, 4.5), (3.5, 4.5), (6.5, 4.5), (8, 4.5), (1, 3.5), (4.5, 3.5), (5.5, 3.5), (8, 3.5), (1, 2.5), (8, 2.5), (1, 1.5), (8, 1.5)

从表中成对地提取交点,应用上述算法得到活化像素表为

(1, 6)

(1, 5), (2, 5), (7, 5)

(1, 4), (2, 4), (3, 4), (6, 4), (7, 4)

(1, 3), (2, 3), (3, 3), (4, 3), (5, 3), (6, 3), (7, 3)

(1, 2), (2, 2), (3, 2), (4, 2), (5, 2), (6, 2), (7, 2)

(1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1)

其结果如图2-42所示。注意，垂直和水平边都要正确给出，正如所料，扫描转换区域倾向于左边和下边。同时还要注意扫描转换不同于简单的奇偶算法(参见图2-41a)。这并不意味

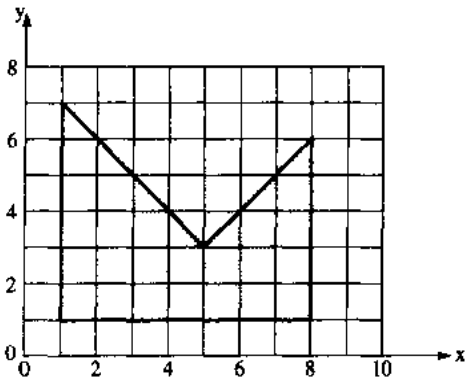


图2-42 图2-37的实区域扫描转换结果

着一种算法比另一种算法好，只是表示两种算法是不同的。　□

2.12.2 更有效的有序边表算法

前节给出的简单算法生成一张很大的表，并且需要对整个表排序。提高排序的效率可改进上述算法。为此应用2.7节中所述的y桶排序法将扫描线按垂直方向的y排序和沿水平方向的x排序分开进行。算法如下。

更有效的有序边表算法：

数据准备：

求出多边形每条边与位于$y+\frac{1}{2}$的中心扫描线的交点。为此可采用Bresenham算法或DDA算法计算这些交点。水平边不考虑。把交点的x坐标存入对应的y桶内。

在处理每条扫描线时，即对于每一y桶，把x交点表按x递增顺序进行排序，即当$x_1 \leqslant x_2$时，x_1位于x_2之前。

数据的扫描转换：

对于每条扫描线从x排序表中成对地提取交点。在对应y桶的扫描线上激活那些x整数值满足关系式$x_1 \leqslant x+\frac{1}{2} \leqslant x_2$的像素。

上述算法首先采用y桶排序对扫描线进行排序，然后沿扫描线按x进行排序。这样，在排序过程全部完成之前就已开始进行扫描转换。而且该算法易于从显示表中增加或删除信息，因为只要对相应的y桶增删信息即可。因此只有那些改动的扫描线需要重新排序。通过下例进一步说明这一算法。

例2.9　更有效的有序边表。

重新考虑例2.8中讨论的多边形（见图2-37）。首先，建立扫描线0－8的y桶（见图2-43），从P_1开始按逆时针方向依次求出多边形各边与扫描线的交点，并存于一张按桶分类的表中，如图2-43a所示，显然该表未经x方向排序。这些交点是根据中心扫描线方法求得的。为了说明的需要，在图2-43b中它们是按序显示的。实现时可采用一个小的扫描线缓冲器（见图2-43c）来存储对x排序的交点值以提高交点表的增删效率。由于每条扫描线在移入扫描线缓冲器之前是

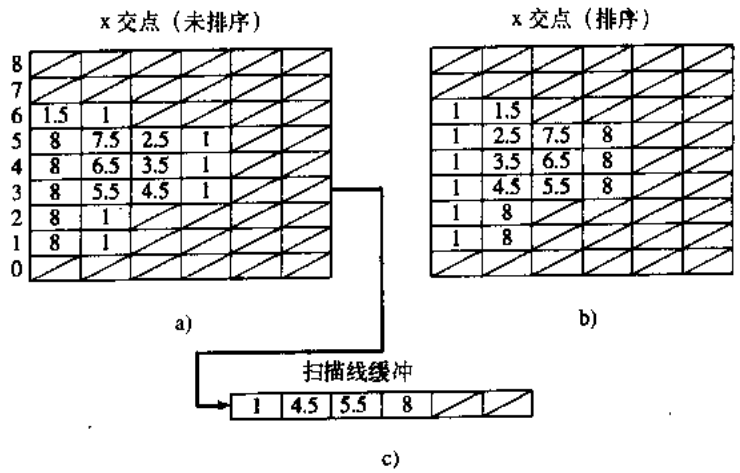


图2-43　图2-37中多边形扫描线的y桶

不对x排序的，因此只需要把它们加到每个y桶的尾部即可，这样无需保持完全排序的y桶表。

从x排序表中成对地提取交点，应用上述算法得到每条扫描线的活化像素表。其结果与例2.8相同。如图2-42所示。 □

虽然第二个算法简化了排序，但它限制了在给定扫描线上的交点个数，或者要求大量存储空间。实际上，其中很多空间并没有利用。采用链表的附加数据结构就可以解决这一问题。预先计算每条扫描线与多边形每条边的交点很费时。同时也需要具有存储大量数据的空间，如同前面介绍实时扫描转换时所述，引入活化边表（见2.7节）可进一步压缩数据存储空间，并且可采用增量法计算扫描线的交点。算法如下：

采用活化边表的有序边表算法：

数据准备：

采用位于$y+\frac{1}{2}$的中心扫描线，对于多边形的每条边求出与其相交的最高扫描线。

把该多边形的边存入与这一扫描线相对应的y桶。

把x交点的初值、多边形穿过的扫描线条数Δy以及相邻扫描线之间的x增量Δx存入链表。

数据的扫描转换：

对每条扫描线，检查相应的y桶是否有新的边。把新的边移入活化边表。

在活化边表中对交点按x递增顺序排序，即当$x_1 \leqslant x_2$时，x_1位于x_2之前。

从已对x排序的表中成对地取出交点。在扫描线y上激活x整数值满足$x_1 \leqslant x+\frac{1}{2} \leqslant x_2$的像素。活化边表中每条边的$\Delta y$减1。若$\Delta y < 0$，则从活化边表中删除这条边。计算新的$x$交点$x_{新}=x_{旧}+\Delta x$。

对下一条扫描线重复上述过程。

本算法假设所有数据已预先转换成多边形表示。Heckbert（见[Heck90a]）给出了类似算法的代码。Whitted（见[Whit81]）给出了一个不受这个假设限制的更通用的算法。

例2.10 采用活化边表的有序边表。

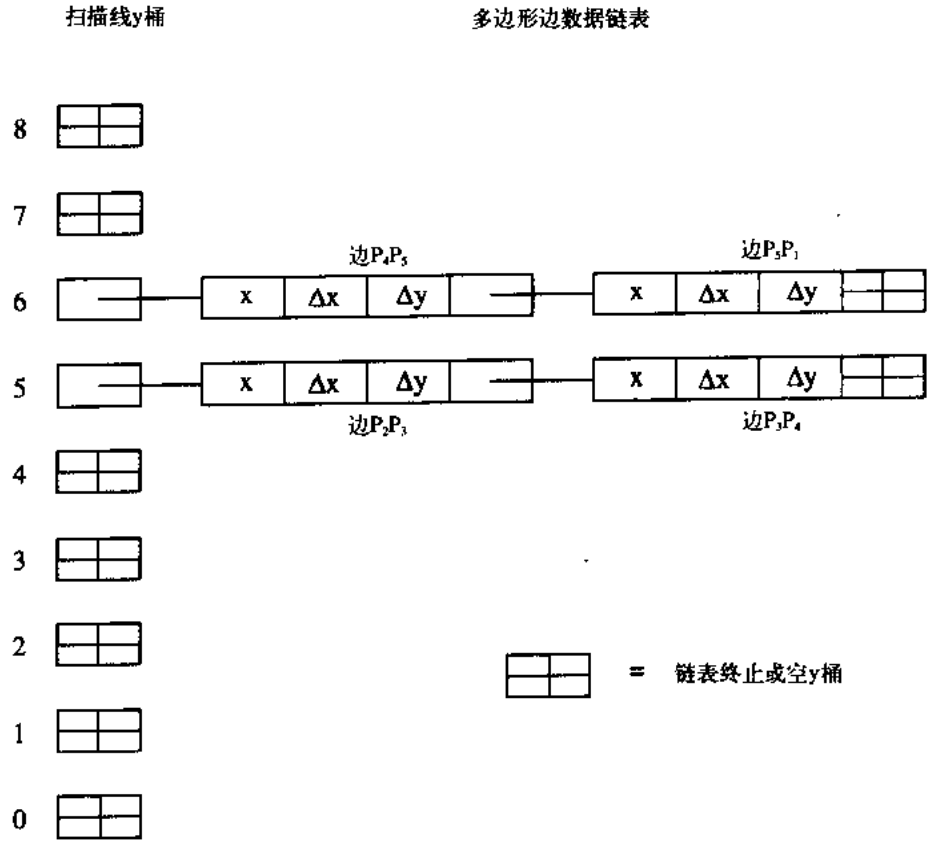
再次考虑如图2-37所示的简单多边形。检查多边形边表可看出，扫描线5是与边P_2P_3以及P_3P_4相交的最高扫描线，扫描线6是与边P_4P_5以及P_5P_1相交的最高扫描线。图2-44a是与图2-37中九条扫描线(0－8)相对应的九个y桶的链表结构的原理图。

可以看到大多数 y 桶是空的。实现过程如图2-44b所示。这里的y桶表是一维数组，每一元素对应一条扫描线。在每条扫描线桶的数组元素中只含一个简单指针，它指向相应的链表数组，如图2-44b所示。

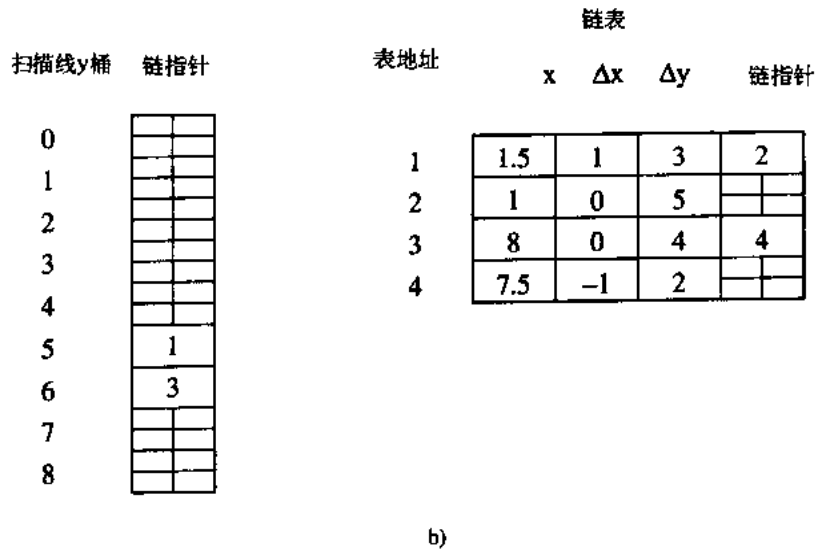
链表采用$n \times 4$数组实现。每一数组索引n对应4个元素：多边形边与它所通过的最高扫描线交点的x值；该多边形在相邻扫描线之间的x增量Δx；该多边形的边穿过扫描线的条数Δy；以及指向始于这条扫描线上的下一条多边形边的链接指针，如图2-44b所示。由图可见，扫描线5的y桶包含链接指针1，它指向链接数据表的首地址。链接数据表中前三项是与边P_2P_3有关的数据。第四项为指针，它指向下一个数据地址。

活化边表用$n \times 3$堆栈数组实现。图2-44c给出了九条扫描线的活化边表的全部内容。从图形顶部最高扫描线8开始顺序检查全部扫描线 (y桶)。由于扫描线8和7的y桶为空，故活化边表也为空。扫描线6增加两个元素到活化边表，扫描线5再增加两个元素。在扫描线2上，由于P_3P_4和

P_4P_5两条边的Δy变为负值，因此从活化边表中删除它们。类似地，P_2P_3和P_2P_1两条边在扫描线0上被删除。最后，在扫描线0上y桶为空，活化边表为空，此时已无其他y桶，故算法结束。



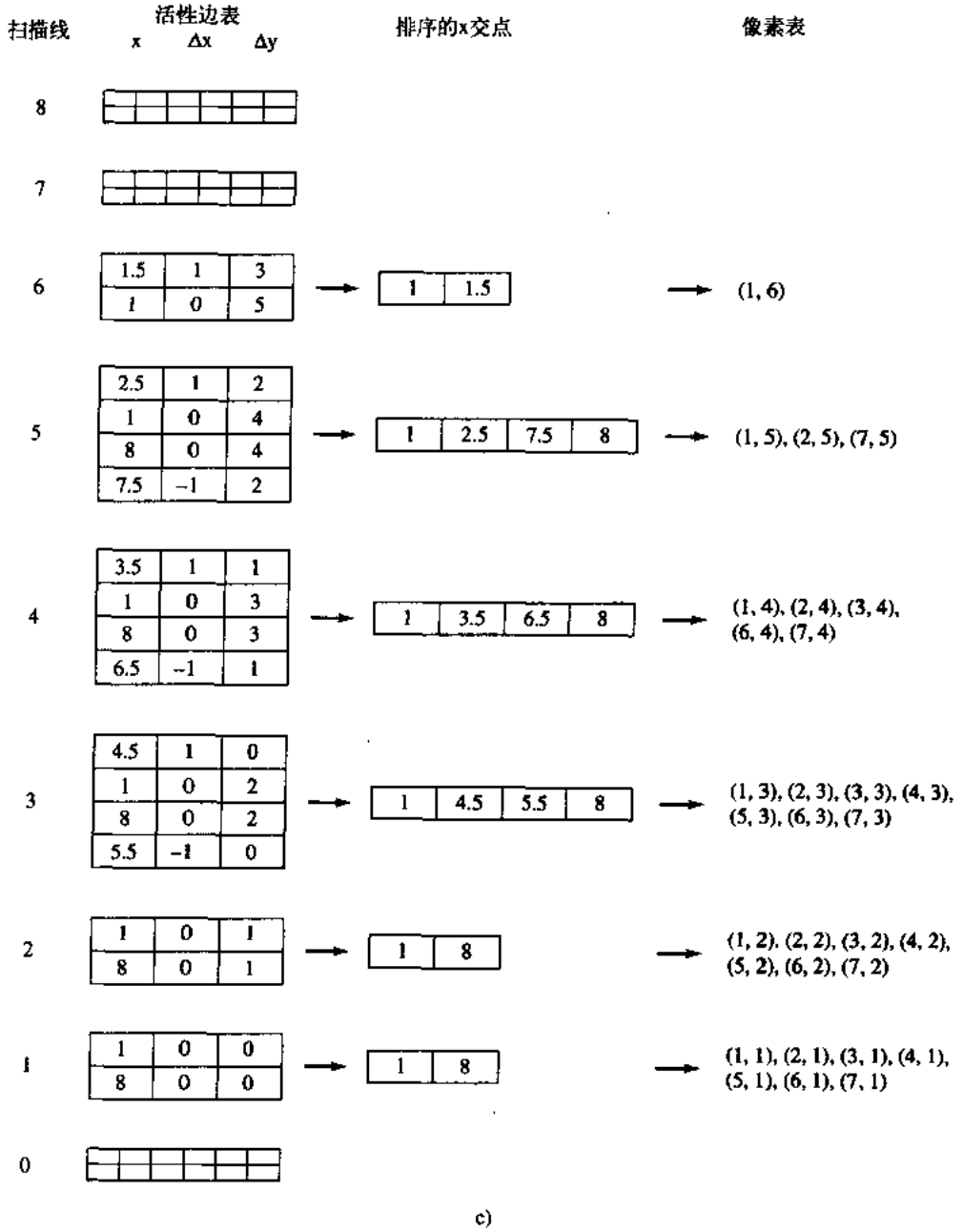
a)



b)

图2-44　图2-37所示多边形的链表原理图

a) 基本结构　b) 以一维数组方式实现　c)活化边表



c)

图2-44 （续）

对于每一条扫描线，从活化边表中取出活化边与该扫描线的交点并按x递增顺序将其排序，把它们存放到由$1 \times n$数组实现的区段(span)缓冲器中。由区段缓冲器中成对地取出交点。然后根据上述算法生成活化像素表。所有扫描线的这些像素表的组合与前例结果相同，如图2-37所示。 □

2.13 边填充算法

有序边表算法是一种非常有效的算法，它使所显示的每个像素只访问一次。这样，输入/

输出的要求可降为最少。每组（或区段）激活像素的端点在输出前就已算出。因此对区段可应用明暗算法以得到连续色调的图形。由于该算法与输入/输出的具体操作无关，因而它也与设备无关。本算法的主要缺点在于对各种表的维持和排序的开销太大。另一种实区域扫描转换方法即所谓边填充算法（[Ackl80]）可极大地压缩所需建立的表的数目。下面介绍一种简单的边填充算法。

边填充算法：

对于每一条与多边形边相交的扫描线，设其交点为(x_1, y_1)，将像素中心位于(x_1, y_1)右边，即$x + \frac{1}{2} > x_1$的全部像素(x, y_1)取补。

应用中心扫描线转换方法计算扫描线与边的交点。对多边形的每条边分别应用上述算法。多边形边的次序无关紧要。图2-45所示为对图2-37所示多边形进行实区域扫描转换的各个步骤。注意，活化像素与简单奇偶算法(参见2.11节)相同，但与有序边表算法有所不同。特别地，边填充算法没有激活位于(5, 3)、(6, 4)和(7, 5)的像素，即边P_3P_4以不同的方式光栅化。差别在于如何处理那些恰好一半在多边形内、一半在多边形外的像素。在有序边表算法中这些像素总是被激活。而在边填充算法中，只有位于多边形内部并处在像素中心左侧的像素才被激活。

本算法最适用于具有帧缓冲存储器的显示器。这就使得可以按任意的顺序处理多边形的边。在处理每一条边时，帧缓冲存储器中被访问的是那些扫描线与该边交点相对应的像素。当所有的边处理完毕后，按扫描线顺序读出帧缓冲存储器的内容，并送入显示设备。图2-45所示为该算法的主要不足之处，即对于复杂的图形每一像素可能被访问多次。因此这一算法受到输入/输出条件的限制。

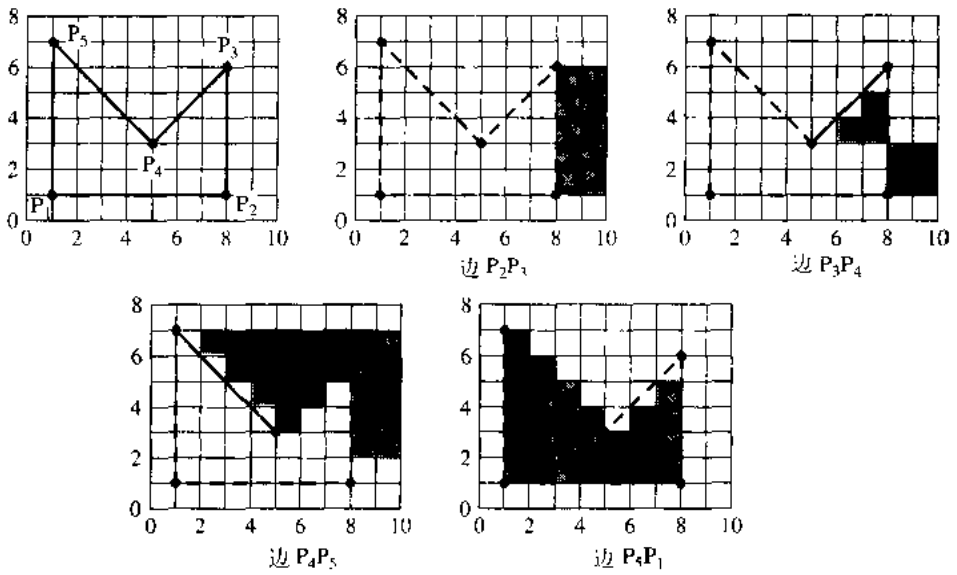


图2-45　边填充算法

引入栅栏（fence）可使边填充算法访问像素的次数减少（[Dunl83]）。这就是所谓栅栏填充算法。再次以图2-37所示的多边形为例。在图2-46中给出栅栏填充算法的基本思想。该算法描述如下。

栅栏填充算法:

对于每一条与多边形边相交的扫描线:

如果交点位于栅栏之左，则将所有中心位于扫描线与边界交点之右和栅栏之左的像素取补。

如果交点位于栅栏之右，则将所有中心位于扫描线与边交点之左或交点之上以及栅栏之右的像素取补。

本算法采用中心扫描线转换。栅栏位置通常取多边形顶点之一较为方便。同样，本算法最适用于帧缓冲存储器。边填充和栅栏填充两者的不足之处在于像素可能不止一次地被访问。对算法稍作修改即所谓边标志算法（见[Ack181]）就可克服这一缺点。边填充、栅栏填充以及边标志算法等并不限于简单多边形。

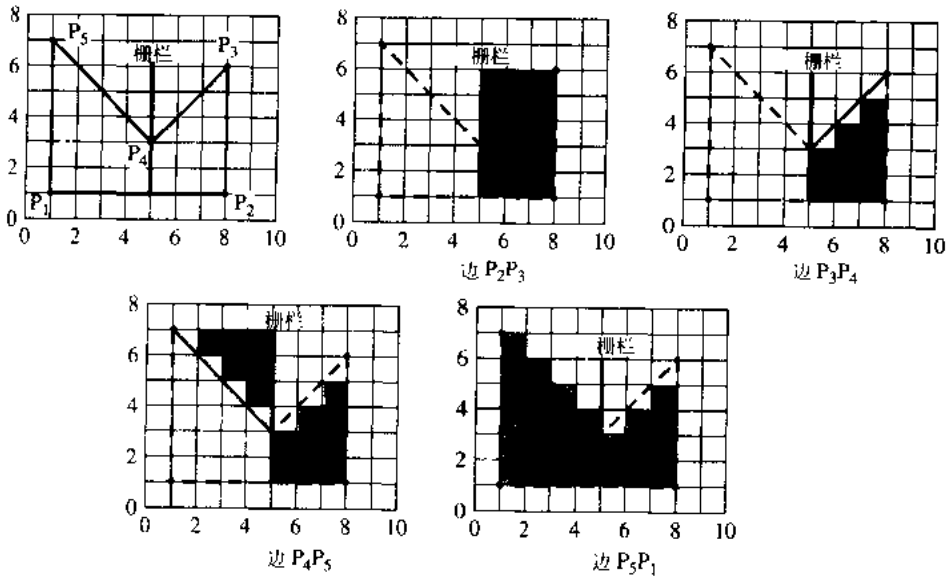


图2-46 栅栏填充算法

2.14 边标志算法

边标志算法（见[Ack181]）的实现过程可分为两步。第一步勾画轮廓线。在每条扫描线上建立各区段的边界像素对。第二步填充这些边界像素之间的全部像素。算法可更清楚地描述如下。

边标志算法:

勾画轮廓线:

按中心扫描线约定，对每条与扫描线相交的多边形边，将中心位于交点之右、即$x + \frac{1}{2} > x_{交点}$的最左像素置为边界值。

填充:

```
For each scan line intersecting the polygon
    Inside = FALSE
    for x = 0 (left) to x = xmax (right)
```

```
            if the pixel at x is set to the boundary value then
                negate Inside
            end if
            if Inside = TRUE then
                set the pixel at x to the polygon value
            else
                reset the pixel at x to the background value
            end if
        next x
```

例2.11　边标志算法。

将边标志算法应用于图2-34所示的多边形。首先勾画出轮廓线。结果如图2-47a所示。像素（1,1）、（1,2）、（1,3）、（1,4）、（1,5）、（1,6）、（2,6）、（3,5）、（4,4）、（5,3）、（6,3）、（7,4）、（8,4）、（8,3）、（8,2）和（8,1）被激活。

然后填充多边形。为了说明这一过程，举扫描线3为例，如图2-47b所示。该扫描线上位于x = 1、5、6和8的像素在勾画轮廓线时被激活。应用填充算法得到。

开始:

Inside=FALSE

对于 $x = 0$	该像素未置成边界值，Inside = FALSE，因此无动作。
对于$x = 1$	该像素已置成边界值，Inside取反置成TRUE。Inside = TRUE，因此该像素取多边形值。
对于$x = 2, 3, 4$	像素未置成边界值。Inside = TRUE，所以像素取多边形值。
对于$x = 5$	该像素已置成边界值，Inside取反置成FALSE。Inside = FALSE，所以像素取背景值。
对于$x = 6$	该像素已置成边界值，Inside取反置成TRUE。Inside = TRUE，所以像素取多边形值。
对于$x = 7$	该像素未置成边界值。Inside = TRUE，所以像素被置成多边形值。
对于$x = 8$	该像素已置成边界值：Inside取反置成FALSE。Inside = FALSE，所以像素取背景值。

其结果如图2-47c所示。填充整个多边形的最终结果与图2-45的边填充算法的结果相同。　□

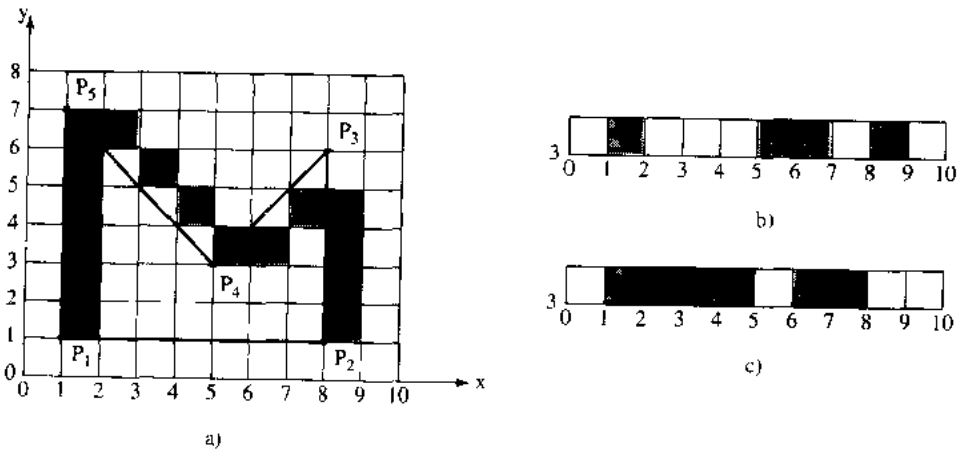


图2-47　边标志算法

边标志算法只访问每个像素一次。因此对输入/输出的要求较边填充或栅栏填充等算法要低得多。在帧缓冲存储器中应用这些算法时，都无需建立、维持边表以及对它进行排序。用软件实现时，有序边表和边标志算法有相同的执行速度（[Whit81]）。然而，边标志算法更适合于硬件或固件实现，这时它的执行速度比有序边表算法快一到两个数量级（见 [Ackl81]）。

2.15 种子填充算法

以上讨论的填充多边形的算法都是按扫描线顺序进行的。种子填充算法则采用完全不同的方法。种子填充算法假设在多边形或区域内部至少有一个像素是已知的。然后设法找到区域内所有其他像素，并对它们进行填充。区域可以用其内部定义或边界定义。如果区域是用其内部定义的，那么，区域内部所有像素具有同一种颜色或值，而区域外的所有像素具有另一种颜色或值（见图2-48）。如果区域是采用边界定义，那么区域边界上所有像素均具有特定的值或颜色（见图2-49）。区域内部所有的像素均不取这一特定值。然而，边界外的像素则可具有与边界相同的值。填充内部定义区域的算法称为泛填充算法（flood fill algorithm）。填充边界定义的区域的算法称为边界填充算法。以下将集中讨论边界填充算法。然而，用类似方法可得到泛填充算法。

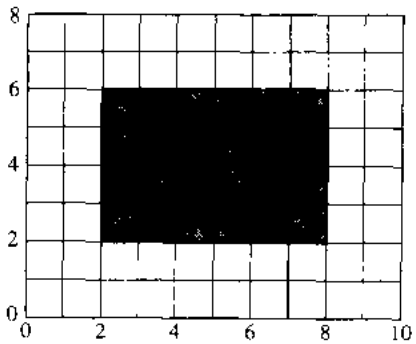


图2-48 内部定义的区域

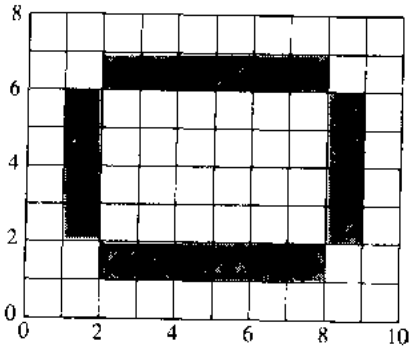


图2-49 边界定义的区域

内部定义的或边界定义的区域可分为4连通或8连通两种。如果区域是4连通的，那么区域内每一像素可通过四个方向，即上、下、左、右的移动组合到达。对于8连通区域，区域内的每一像素可通过两个水平方向、两个垂直方向和四个对角线方向的移动组合到达。8连通算法可以填充4连通区域，但是4连通算法不能填充8连通区域。图2-50是4连通和8连通内部定义

区域的简单例子。显然，图2-50b上的8连通区域的子区域是4连通的，然而从一个子区域到达另一个子区域则需8连通算法实现。若每个区域是一分离的4连通区域，并且每一个需填成不同的颜色或值，那么用8连通算法会使两个子区域被错误地填上同一颜色。

图2-51显示出图2-50上8连通区域被重新定义成边界定义区域后的结果。由图2-51可看出，当区域是8连通的、即两个子区域在顶角相接时，则其边界是4连通的。同时也可以看到，对于该 4连通区域、其边界是8连通的。2.24节和2.25节将集中讨论4连通算法。只要把四个方向的填充改为8个方向填充即可得到等价的8连通算法。

a) 4连通　　b) 8连通

图2-50　4连通和8连通内部定义区域

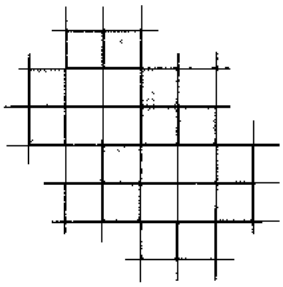


图2-51　4连通和8连通边界定义区域

2.15.1　简单的种子填充算法

对于边界定义区域，可使用堆栈建立简单的种子填充算法。堆栈就是一个数组或存储空间，数值可依次存入堆栈，也可依次移出。当新的数值存入或压入堆栈时，所有以前存入的数值被向下压一层；当数值从堆栈中移出或弹出时，所有以前存入的数值向上浮动或上推一层。这种堆栈称为先进后出（FILO）堆栈或下推堆栈。简单的种子填充算法如下。

采用堆栈的简单种子填充算法：

种子像素压入堆栈

当堆栈非空时

从堆栈中弹出一个像素

将该像素置成所要求的值

检查每个与当前像素邻接的4连通像素是否是边界像素或者是否已置成所要求的值。若是上述两种情况之一，则略而不计。否则把该像素压入堆栈。

若对上述算法稍作修改，即若考察8连通像素而不仅是4连通像素，那么上述方法同样适用于8连通区域。若种子像素和边界定义区域为已知，那么算法可更正式地描述如下。

simple seed fill algorithm *for 4-connected boundary-defined regions*

Seed(x, y) *is the seed pixel*
Push *is a function for placing a pixel on the stack*
Pop *is a function for removing a pixel from the stack*
Pixel(x, y) = Seed(x, y)
initialize stack
Push Pixel(x, y)
while (stack not empty)

```
get a pixel from the stack
Pop Pixel(x, y)
if Pixel(x, y) < > New value then
    Pixel(x, y) = New value
end if
examine the surrounding pixels to see if they should be placed
  onto the stack
if (Pixel(x + 1, y) < > New value and
    Pixel(x + 1, y) < > Boundary value) then
    Push Pixel(x + 1, y)
end if
if (Pixel(x, y + 1) < > New value and
    Pixel(x, y + 1) < > Boundary value) then
    Push Pixel(x, y + 1)
end if
if (Pixel(x − 1, y) < > New value and
    Pixel(x − 1, y) < > Boundary value) then
    Push Pixel(x − 1, y)
end if
if (Pixel(x, y − 1) < > New value and
    Pixel(x, y − 1) < > Boundary value) then
    Push Pixel(x, y − 1)
end if
end while
```

本算法检查4连通像素，并从当前像素的右侧像素开始按逆时针方向把相应的像素压入堆栈。

例2.12 简单的种子填充算法。

作为本算法的一个实例，考察由顶点（1,1）、(8,1)、(8,4)、(6,6）以及（1,6）所决定的边界定义多边形区域（见图2-52），取种子像素为（4,3)。算法逐个地用像素填充多边形,如图2-52中箭头所示。每个像素中的数字是该像素在算法进行过程中在堆栈中的位置。注意到某些像素有几个数字。这说明该像素曾几次被压入堆栈。当算法进行到像素（5,5）时,堆栈深度为23层,它包含下列像素（7,4)、(7,3)、(7,2)、(7,1)、(6,2)、(6,3)、(5,6)、(6,4)、(5,5)、(4,4)、(3,4)、(3,5)、(2,4)、(2,3)、(2,2)、(3,2)、(5,1)、(3,2)、(5,2)、(3,3)、(4,4)和(5,3)。

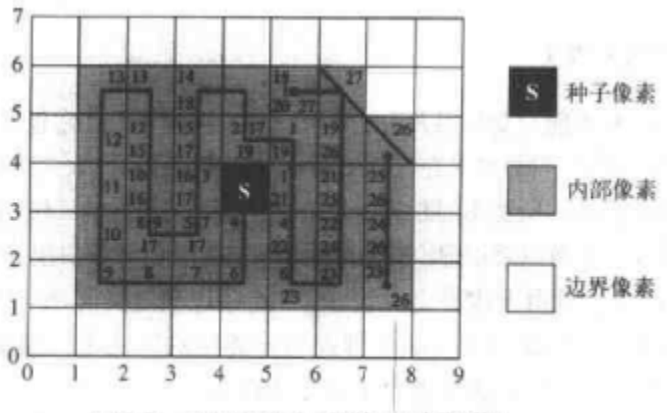


图2-52 采用简单堆栈的种子填充算法

由于像素（5,5）周围的所有像素具有边界值或新值，这时没有新像素被压入堆栈。因此像

素（7, 4）就被推出堆栈,算法接着填充（7, 4）、(7, 3)、(7, 2）和（7, 1）这一系列。当到达像素（7, 1）时，周围的像素或已置成新值或为边界像素。至此多边形已全部填充完毕，因此只从堆栈中弹出像素而不再填充新的像素，直至堆栈为空时停止。当堆栈为空时，算法终止。 □

例2.12中的多边形是一个简单的开区域。但本算法也可用于填充有孔区域，详见下例。

例2.13　有孔多边形的简单种子填充算法。

作为本算法应用于边界定义的有孔多边形区域的例子，见图2-53所示的图形。其中多边形的顶点同上例，为（1,1）、(8,1)、(8,4)、(6,6）和（1,6)。其内孔由顶点（3,2)、(6,2)、(6,4）和（3,4）决定。种子像素是（4,4)。多边形的内孔使算法将沿着与例2.12完全不同的路径填充多边形。图2-53中用箭头表示其路径。每个像素上所标记的数字表示算法进行过程中它在堆栈的位置。当算法进行到像素（3,1）时，周围所有4连通像素或具有新值或为边界像素。因此无像素可压入堆栈。这时，堆栈深度为14层。堆栈内的像素为（7,1)、(7,2)、(7,3)、(6,5)、(7,4)、(6,5)、(3,1)、(1,2)、(1,3)、(1,4)、(2,5)、(3,5)、(4,5）和（5,4)。

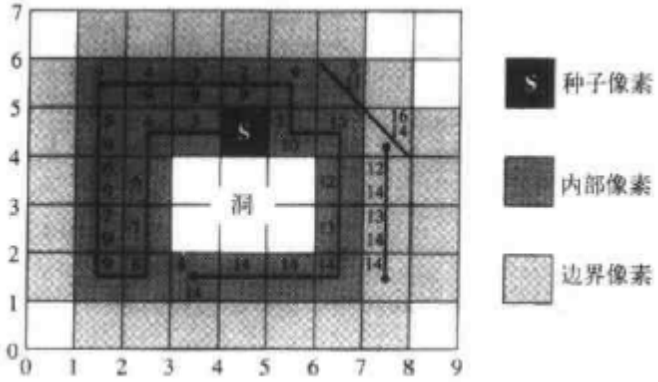


图2-53　采用简单堆栈对有孔区域进行种子填充算法

从堆栈推出像素（7, 1）以后，算法将填充（7, 1)、(7, 2)、(7, 3）和（7, 4）这一系列像素，而不再向堆栈压入新的像素。同样，在像素（7, 4）周围，所有4连通像素或具有新值或为边界像素。继续检查堆栈，在像素（6, 5）之前查不到新的像素。填好像素（6, 5）后,多边形填充结束。算法继续处理堆栈,直到结束，而不再填充像素。当堆栈为空时，算法结束。 □

2..15.2　扫描线种子填充算法

上述两例都说明堆栈可能会变得很大，还可进一步发现堆栈内经常包含一些重复的和不必要的信息。使堆栈尺寸极小化的一种算法是在任意不间断扫描线区段中只取一个种子像素（见[Smit79]），称此为扫描线种子填充算法。所谓不间断区段是指在一条扫描线上的一组相邻的像素。下面采用启发式方法导出这一算法。也可以采用理论较强的以图论为基础的方法导出（见[Shan80]）。

扫描线种子填充算法适用于边界定义区域。4连通边界定义区域可以为凸也可以为凹，还可以包含一到多个孔。在边界定义区域外部或与其邻接的区域中，像素的值和颜色不同于填充区域或多边形的像素的值。算法从概念上可分成四步。

扫描线种子填充算法:

从包含种子像素的堆栈中弹出区段的种子像素。

沿着扫描线对包含种子像素的区段左右像素进行填充，直至遇到边界像素为止，从而填满包含种子像素的区段。

区段内最左的和最右的像素记为X_{left}和X_{right}。

在$X_{left} \leqslant x \leqslant X_{right}$范围内，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或者前面已经填充过的像素。如果这些扫描线既不包含边界像素，也不包含已填充的像素，那么在$X_{left} \leqslant x \leqslant X_{right}$中把每一区段的最右像素取作为种子像素并压入堆栈。

算法在初始化时向堆栈压入一个种子像素，并在堆栈为空时结束。算法将跳过孔和区域边界的凹陷处（见图2-54和例2.14）。更为完整的伪代码算法如下。

```
scan line seed fill algorithm
Seed(x, y) is the seed pixel
Pop is a function for removing a pixel from the stack
Push is a function for placing a pixel on the stack
    initialize stack
    Push Seed(x, y)
    while (stack not empty)
        get the seed pixel and set it to the new value
        Pop Pixel(x, y)
        Pixel(x, y) = Fill value
        save the x coordinate of the seed pixel
        Savex = x
        fill the span to the right of the seed pixel
        x = x + 1
    while Pixel(x, y) < > Boundary value
        Pixel(x, y) = Fill value
        x = x + 1
    end while
    save the extreme right pixel
    Xright = x - 1
    reset the x coordinate to the value for the seed pixel
    x = Savex
    fill the span to the left of the seed pixel
    x = x - 1
    while Pixel(x, y) < > Boundary value
        Pixel(x, y) = Fill value
        x = x - 1
    end while
    save the extreme left pixel
    Xleft = x + 1
    reset the x coordinate to the value for the seed pixel
    x = Savex
check that the scan line above is neither a polygon boundary nor
  has been previously completely filled; if not, seed the scan line
start at the left edge of the scan line subspan
```

```
x = Xleft
y = y + 1
while x ≤ Xright
    seed the scan line above
    Pflag = 0
    while (Pixel(x, y) <> Boundary value and
      Pixel(x, y) <> Fill value and x < Xright)
        if Pflag = 0 then Pflag = 1
        x = x + 1
    end while
    push the extreme right pixel onto the stack
    if Pflag = 1 then
        if (x = Xright and Pixel(x, y) <> Boundary value
          and Pixel(x, y) <> Fill value then
            Push Pixel(x, y)
        else
            Push Pixel(x − 1, y)
        end if
        Pflag = 0
    end if
        continue checking in case the span is interrupted
        Xenter = x
        while ((Pixel(x, y) = Boundary value or Pixel(x, y)
              = Fill value) and x < Xright)
            x = x + 1
        end while
        make sure that the pixel coordinate is incremented
        if x = Xenter then x = x + 1
    end while
    Check that the scan line below is not a polygon boundary
      nor has been previously completely filled.
        this algorithm is exactly the same as that for checking the scan
          line above except that y = y − 1 is subsitituted for y = y + 1
    end while
finish
```

其中**Pop**的功能是从堆栈中弹出像素的x, y坐标，**Push**的功能是将它们压入堆栈。

例2.14　扫描线种子填充。

将上述算法应用于图2-54所示的边界定义的多边形区域。算法初始化时将多边形种子像素（图2-54a上标记的种子（5, 7））压入堆栈。算法一开始，就将这个像素作为区段种子从堆栈中推出。然后向左和向右填充种子所在区段。找出区段的端点是$X_{right} = 9$和$X_{left} = 1$。然后检查上面一条扫描线，它不是边界线，且尚未填充。在$1 \leqslant x \leqslant 9$范围内最右的像素是（8, 8）。这一像素（在图2-54a中标记为1），被压入堆栈。接着检查下面一条扫描线，它既非边界线，也未被填充。在$X_{left} \leqslant x \leqslant X_{right}$范围内有两个子区段。左边子区段取像素（3, 6）为种子，用2来标记它（见图2-54a），然后把它压入堆栈。右边区段取像素（9, 6）为种子，也把它压入堆栈。注意，像素（9, 6）并非是该区段的最右端像素。然而，在$X_{left} \leqslant x \leqslant X_{right}$范围内，即

在$1 \leqslant x \leqslant 9$范围内，它是最右端像素。现在，算法一遍就结束了。

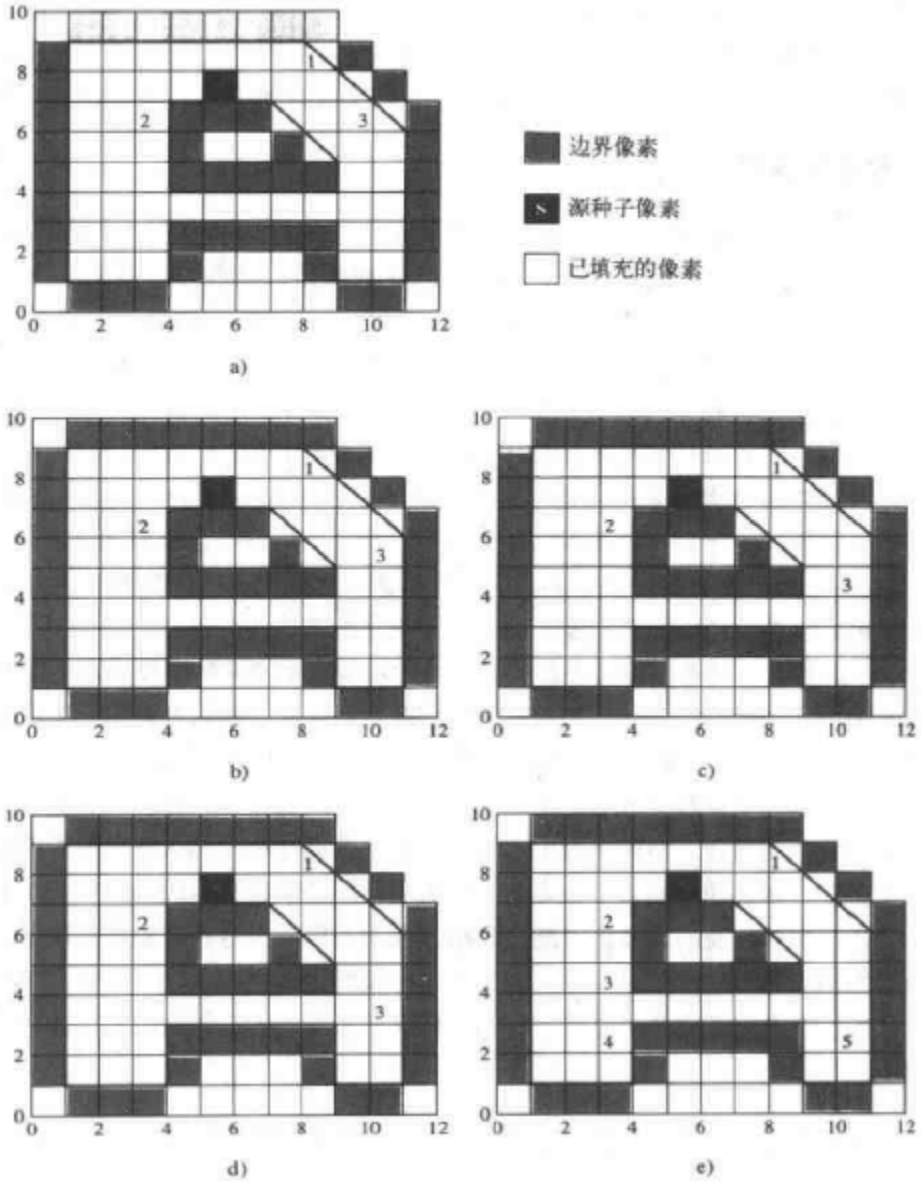


图2-54 面向扫描线的多边形种子填充算法

算法继续从堆栈中推出顶部像素。这里，算法逐条往下填充多边形右边扫描线区段。图2-54b到图2-54d给出了这一结果。图2-54d中扫描线3的种子像素是（10,3），向左以及向右填充该区段得到$X_{left}=1$和$X_{right}=10$。检查上面的扫描线，得到左边子区段的种子像素（3,4），把它压入堆栈。右边子区段已被填充。检查下面的扫描线，得到左边子区段的种子像素（3,2）以及右边子区段的种子像素（10, 2）。这两个像素也被压入堆栈。堆栈深度在这条扫描线上达到最大。

由此至算法结束只剩下一点需要说明。在填充了图2-54e中以像素5、4、3为种子的4连通多边形区域以后，标记为2的像素被从堆栈中推出。这里，种子所在的扫描线以及在其上下两条扫描线上的所有像素均已填充。这样，没有新的像素被压入堆栈。然后，算法将标记为1

的像素作为种子像素从堆栈中推出，并填充这条扫描线。这时仍然没有新像素压入堆栈。此时堆栈已空，多边形已填满，算法结束。 □

与2.15节的种子填充算法相比，在上例中堆栈的最大深度为5。有关多边形以及区域的种子填充的其他方法可参阅Pavlidis[Pavl82]、Heckbert[Heck90b]和Fiskin[Fisk85, 90]。

2.16 图形反走样基础

为了提出有效的反走样方法，有必要弄清楚造成走样的原因。从根本上讲，走样的出现是因为直线、多边形边、色彩边界等是连续的，而光栅则是由离散的点组成。为了在光栅显示设备上表现直线、多边形等，必须在离散位置上采样。这可能导致意外的结果。例如，考虑如图2-55a所示的信号。频率较低的第二个信号示于图2-55c。如果对这两个信号用同一频率（用小×字符表示）进行采样或光栅化，那么重建后的信号完全相同（见图2-55b和图2-55d）。图2-55d称作图2-55b样本的走样，因此它也是图2-55a信号的走样。图2-55a上高频信号的采样频率过低而造成欠采样。为了避免图形走样，采样频率至少应是信号最高频率的两倍。在信号分析中，这个频率称为Nyquist限定(频率)。如果就信号周期而言（周期∝1/频率），采样间隔必须小于信号周期的一半。欠采样会造成高频图像绘制失真。例如，它使得栅栏或百叶窗看起来似乎由几块阔条构成，而不是由很多细条构成。

前面介绍了计算机生成图像时通常存在的三种走样现象中的两种：锯齿形边以及图形细节或纹理绘制失真。第三种现象出现在显示非常微小对象的场合。如果对象小于像素的尺寸，或者它未能覆盖住像素中用于计算其属性的那一点，那么这一对象将不会在图上出现。另一方面，如果这个很小的对象覆盖像素中用于计算其属性的那一点，它将不恰当地代表整个像素的属性。如图2-56中所示的左面像素，当像素的中心点用于确定其属性时，那么整个像素的属性由这个微小对象决定。图2-56表明右面几个像素中的对象将被忽略或丢失，因为没有一个对象覆盖像素的中心。注意，那些细长的对象也可能被忽略。在动画序列中这些现象特别明显。图2-57是关于小三角形的动画序列中的三帧画面。如果像素的属性由其中心决定，那么在第一帧，这个对象是不可见的。在第二帧它是可见的；在第三帧又是不可见的。因此在动画序列中微小对象将会忽隐忽现。

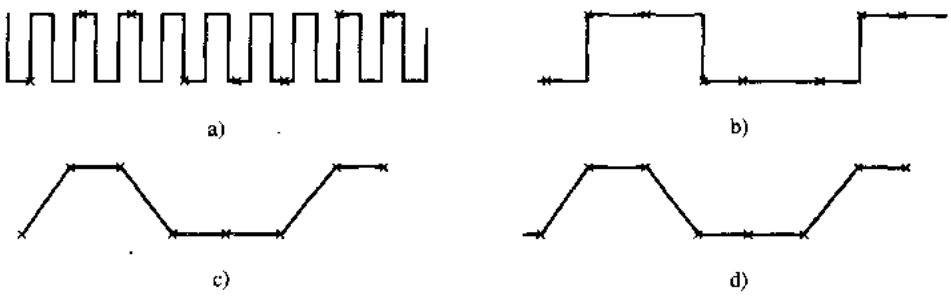


图2-55 采样和走样现象

基本上有两类反走样方法。第一类方法是提高采样频率。这可由提高光栅分辨率来实现。这时，可显示出图形的一些细节。然而，CRT光栅扫描设备显示非常精细光栅的能力是有限的。由于这一限制使人们想到在较高分辨率上对光栅进行计算，然后采用某种平均算法得到较低分辨率的像素的属性，并显示在分辨率较低的显示器上（见[Crow81]）。这种方法称为

超采样或后置滤波。

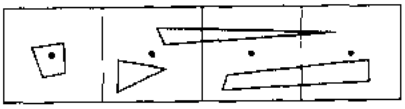


图2-56 显示细小对象时的走样现象

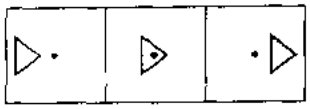


图2-57 动画的走样现象

第二类反走样方法是把像素作为一个有限区域，而不是作为一个点来处理。首先讨论一些启发性的技术，然后介绍一些数学基础。把像素作为一个有限区域来处理等价于图像的前置滤波（见[Crow81]）。

2.16.1 超采样

在超采样(或称后置滤波)中，对物理光栅设备施加分辨率超过物理光栅设备的伪光栅化。图像在较高分辨率的伪光栅设备上光栅化，对子像素组取平均以获得每个物理像素的属性。一般地讲，人们感兴趣的是以下三种实体：直线、多边形或面的边以及多边形或面的内部。

2.16.2 直线

首先考虑直线的情况。将每一像素分成$n \times n$的子像素组，在物理像素组内对直线进行光栅化，例如使用Bresenham算法(见2.3～2.5节)进行光栅化，因此可以计算出激活的子像素数。图2-58所示为它的基本概念，其中虚线表示子像素。物理像素的强度是与物理像素内被激活的子像素数与可能被激活的子像素的比值成比例。这里，在$n \times n$伪光栅上，可光栅化的最多子像素数是n。因此，该直线的最大强度值范围是$0 \sim n$，共$n+1$个数。反走样直线中物理像素的强度是

$$I_{line} = \frac{s}{n} I_{max}$$

其中，s是被激活子像素数，I_{max}是系统所支持的最大强度值。

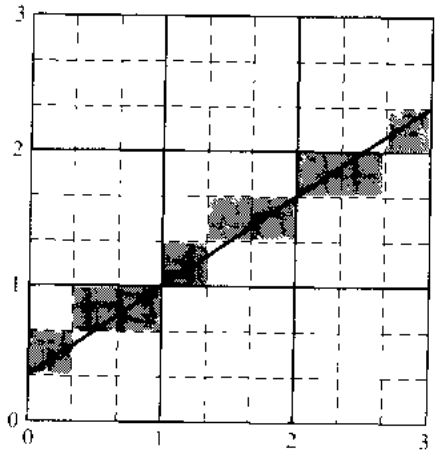


图2-58 走样细直线

另一种反走样直线的技术是考虑直线具有固定的宽度，如图2-59所示。这里，将直线当作具有一个物理像素宽的矩形处理。计算中心在矩形内的子像素的数目，对于每一物理像素，将上面计算而得的子像素数与n^2+1个可能的的强度数比较，决定物理像素的强度为

$$I_{\text{line}} = \frac{s}{n^2} I_{\max}$$

注意这里讨论的情况与以前讨论的情况不同,前面讨论的强度只有n+1个可能,这里有n^2+1个可能。

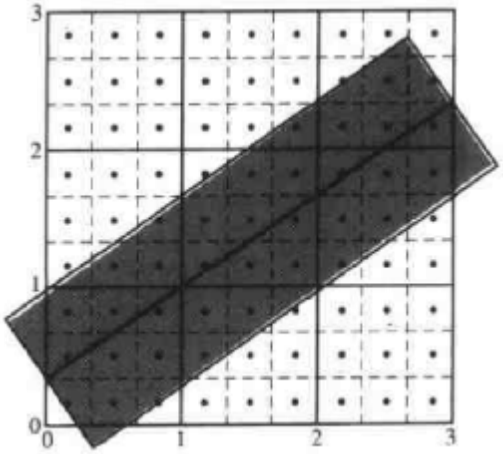


图2-59　反走样固定宽度的直线

对于彩色的情况,反走样必须考虑背景色。一般使用线性插值。如果(r,g,b)表示颜色的三元色组,则反走样像素的颜色为:

$$(r, g, b)_{\text{pixel}} = (I)(r, g, b)_{\text{line}} + (1-I)(r, g, b)_{\text{background}}$$

为了确定s的值,必须计算出位于表示直线的固定宽度矩形内的子像素数。这是所谓点在多边形内问题的一个例子(参见4.4节)。这里使用特别简单的技术，称为替换测试法。

图2-60显示了直线上的一个内部物理像素,即不在直线端点上的像素。矩形条表示直线只有一个物理像素宽，直线通过矩形条的中心。直线位于第一象限，斜率m在0 ~ 1之间。正方形的物理像素子分成9个正方形子像素。子像素的中心用+符号表示。

因为这是一个内部物理像素,直线通过该像素。因此,只需确定子像素的中心是否位于矩形条的两条边之间，即在上边与下边之间(如果是直线的端点，则还要检查矩形的端边)。对于内部像素,确定子像素的中心是否位于矩形条之中可以通过由边的直线方程产生的测试函数完成。

直线方程为:

$$y = mx + y_1$$

其中y_1是直线与物理像素(x_i,y_i)的左边交点，因为上边和下边与直线平行，所以它们的斜率相同。根据图2-60中的虚线直角三角形可获得y截距。从直线到边的垂直长度为1/2。从物理像素的左边到垂直相交点的距离为$m/2$，因此:

$$l = \frac{1}{2}\sqrt{m^2+1}$$

底边的方程为:

$$y = mx + y_1 - l = mx + y_1 - \frac{1}{2}\sqrt{m^2-1}$$

相对称的上边方程为

$$y = mx + y_1 - l = mx + y_1 + \frac{1}{2}\sqrt{m^2 - 1}$$

测试函数为

$$\mathrm{TF}_L = y - y_1 - mx + \frac{1}{2}\sqrt{m^2 - 1}$$

$$\mathrm{TF}_U = y - y_1 - mx - \frac{1}{2}\sqrt{m^2 - 1}$$

其中，(x, y)是要测试的点的坐标。

如图2-60所示，从左至右沿着边检查，如果测试函数为正，则点位于直线的左边；如果为负，则点位于直线的右边。如果测试函数为零，则点在直线上。因此，如果点在矩形条内，则$\mathrm{TF}_L \leqslant 0$，且$\mathrm{TF}_U \geqslant 0$。注意，点在直线上时认为其在矩形条内。

简单的伪代码算法pxlcnt计算中心在矩形内的子像素数。注意，它利用已知的子像素中心的空间位置值。

Algorithm to count subpixel centers within a physical pixel for antialiasing lines

assumes the pixel is square and divided into n^2 subpixels
assumes the slope of the line is $0 \leq m \leq 1$

```
n is the number of divisions on each edge of the physical pixel
m is the slope of the line
x is the x coordinate of the subpixel center
y is the y coordinate of the subpixel center
y1 is the intercept at xi relative to yi
sub pxlcnt(y1,n,m,hits)
    these next four quantities, included here for completeness,
      should be calculated in the calling routine
      outside the calling loop for efficiency
    constant = (sqr(m*m + 1))/2
    increment = 1/n
    xstart = increment/2
    xstop = 1 - xstart
    ystart = y1 - 3*increment
    ystop = y1 + 1 - increment
    hits = 0
    count the interior subpixel centers
    for y = ystart to ystop step increment
        yconst = y - y1
        for x = xstart to xstop step increment
            temp = yconst - m*x
            tfu = temp - constant
            tfl = temp + constant
            change this next statement to  if tfu <> 0 and tfl <> 0
                and Sign(tfu) <> Sign(tfl) then
            if a subpixel center lying on an edge is to be excluded
            if Sign(tfu) <> Sign(tfl) then
```

```
                hits = hits + 1
            end if
        next x
    next y
end sub
```

水平或近水平(垂直或近垂直)直线会产生一些问题。如果要考虑一个点是否在矩形条边外,即用$TF_U < 0$和$TF_L > 0$作测试。对于$y_1 = 0$和$m = 0$，有三个子像素中心在多边形条内部；而对于$y_1 = 0^+$和$m = 0$，则有6个子像素在多边形条内部；对$y_1 = 1/3^+$和$m = 0$，有9个子像素中心在多边形条内。因此，直线上y截距的微小变化都会导致直线中像素的强度的剧烈变化。对于近水平线和动画序列会产生一些特别问题，Klassen（见[Klas93]）和Schilling（见[Schi91]）给出了有效的技术来处理这些问题。

从视觉上考虑,反走样模糊了多边形边或直线，利用人眼和人脑的合成特性给人以光滑边或直线的印象。视觉系统的特征要求使用更宽的矩形来表示一条直线。但是，简单计算中心在较宽直线内的子像素数是不够的。必须根据某些度量参数对每一子像素的贡献进行加权，例如，根据子像素中心与直线中心线的距离确定权重。根据这种“度量”结果可以产生三角权重函数。

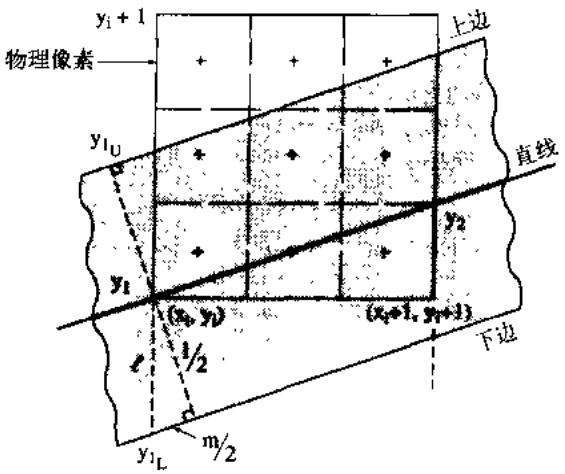


图2-60　反走样固定宽度直线时计算子像素

图2-61给出了一条用两个物理像素宽度矩形条表示的直线。直线位于第一象限内，斜率为$0 \leqslant m \leqslant 1$。除了计算中心位于矩形条内的子像素数，还必须确定直线与子像素中心的垂直距离。

对于两个像素宽的矩形条,要扩展搜索的区域还包括:

$$\frac{1}{2n} < x < 1-\frac{1}{2n}$$

$$y_1 - \left(1-\frac{1}{2n}\right) < y < y_1 + \left(2+\frac{1}{2n}\right)$$

长度l为:

$$l = \sqrt{m^2+1}$$

上边和下边的方程为:

$$y_U = mx + y_1 + l = mx + y_1 + \sqrt{m^2+1}$$

$$y_L = mx + y_1 - l = mx + y_1 - \sqrt{m^2+1}$$

测试函数为:

$$tfu = (y_U - y_1) - mx - l = (y - y_1) - mx - \sqrt{m^2+1}$$

$$tfl = (y_L - y_1) - mx + l = (y - y_1) - mx + \sqrt{m^2+1}$$

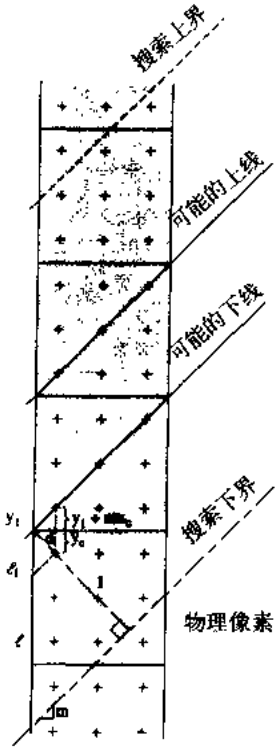


图2-61 两个物理像素宽的固定宽度直线的几何模型

参见图2-61，根据相似的三角形可产生从直线到子像素中心的垂直距离d:

$$d = \frac{l}{l} = \frac{y_1 - y_c + mx_c}{\sqrt{m^2+1}}$$

其中x_c、y_c是子像素中心的坐标。

三角权重函数为:

$$w = \frac{1 - fd}{3n^2} \qquad 0 \leqslant f \leqslant 1$$

其中f 可用来调节一些特定子像素的权重。例如，如果$f = 0$，则所有子像素的权重都相同；但是如果$f = 1$，则与矩阵条边界相近的像素权重较小。这种权重函数还有利于减小水平（和垂直）直线的相对强度(相对于对角线的强度)。

可以很容易地修改以前讨论过的子像素计算算法，使它可以计算较多子像素的数目，并计算权重。算法如下。

Algorithm to count subpixel centers within a two physical pixel wide rectangular strip for antialiasing lines

assumes pixels are square and divided into n^2 subpixels
assumes the slope of the line is $0 \le m \le 1$
n *is the number of divisions on each edge of the physical pixel*
m is the slope of the line
x is the x coordinate of the subpixel center
y is the y coordinate of the subpixel center
y_1 is the intercept at x_i relative to y_i

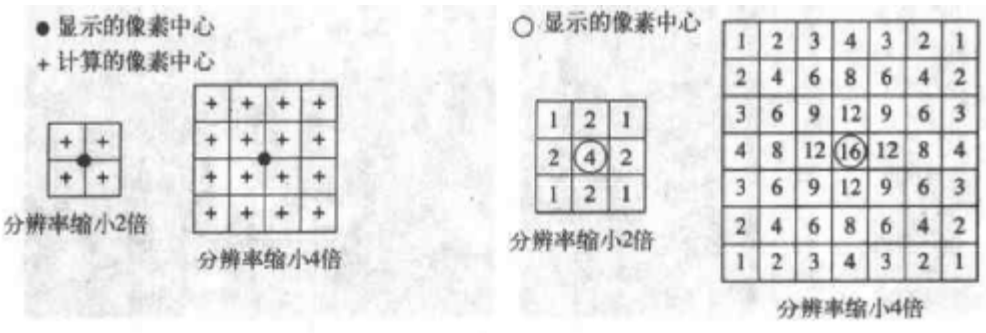
```
sub bpxlcnt(y1,n,m,hits)
    these next four quantities, included here for completeness,
      should be calculated in the calling routine
      outside the calling loop for efficiency
    constant = sqr(m*m + 1))
    rconstant = 1/constant
    increment = 1/n
    halfincrement = increment/2
    ystart = y1 - 1 - halfincrement
    ystop = y1 + 2 + halfincrement
    xstart = halfincrement
    xstop = 1 - start
    hits = 0
    count the interior subpixel centers
    for y = ystart to ystop + 1 step increment
        yconst = y - y1
        for x = xstart to xstop step increment
            temp = yconst - m*x
            tfu = temp - constant
            tfl = temp + constant
            change this next statement to  if tfu <> 0 and tfl <> 0
                and Sign(tfu) <> Sign(tfl) then
            if a subpixel center lying on an edge is to be excluded
            if Sign(tfu) <> Sign(tfl) then
                d = (y1 - y + m*x)*rconstant
                if d < 0 then
                    hits = hits + (1 + fd)
                else
                    hits = hits + (1 - fd)
                end if
            end if
        next x
    next y
end sub
```

对于彩色，也可使用前面给出的线性混合函数。进一步改进是使用相邻物理像素的子像素数目来将它们的颜色或亮度与背景相混合。

2.16.3 多边形内部

现在考虑多边形内部的超采样问题。一般使用两种求平均强度方法：即均匀平均和加权平均。图2-62a给出了一种均匀平均方式，即建立是物理分辨率2倍和4倍的伪分辨率，再对伪分辨率的周围像素进行均匀平均。在这个过程中对每一显示像素子分为子像素以构成较高分辨率的伪光栅。像素属性由每一子像素的中心确定，并对它进行均匀平均以获得显示像素的属性。

如果计算显示像素属性时考虑的子像素数目很多，并对它们的贡献进行加权，则产生的效果会更好。图2-62b表示由Crow（见[Crow81]）提出的2倍和4倍于物理分辨率的伪分辨率下的加权平均。对于这些加权平均，当计算物理像素的属性时，伪分辨率提高2倍，要考虑9个子像素的加权平均。伪分辨率提高4倍，要考虑49个子像素的加权平均。注意，对于这些加权平均(又称为Bartlett滤波)，像素的贡献在x和y方向上线性减少，而在对角线方向上则按抛物线减少。

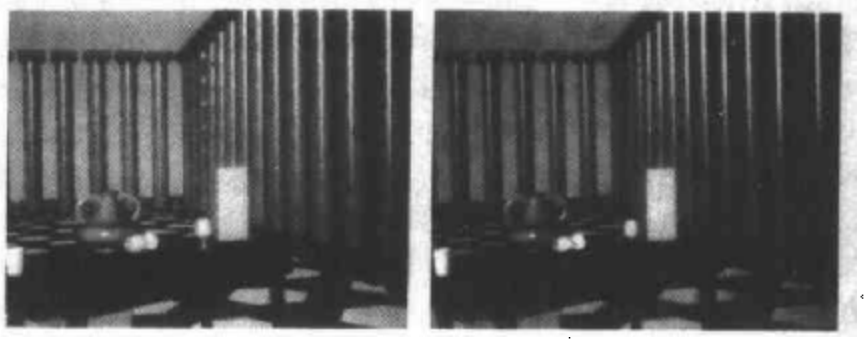


a) b)

图2-62 像素平均

a) 均匀平均 b) 加权平均(数字表示相应的权重)

图2-63是一幅以256 × 256的分辨率显示的复杂景象。图2-63a是在512 × 512分辨率上计算的，图2-63b是在1024 × 1024的分辨率上计算的。采用均匀平均法获得显示分辨率为256 × 256的像素。图2-64a和图2-64b是两个相同的场景，分别在分辨率为512 × 512和1024 × 1024上算出，并采用图2-62b所示的加权平均法计算的以256 × 256分辨率显示的图像。



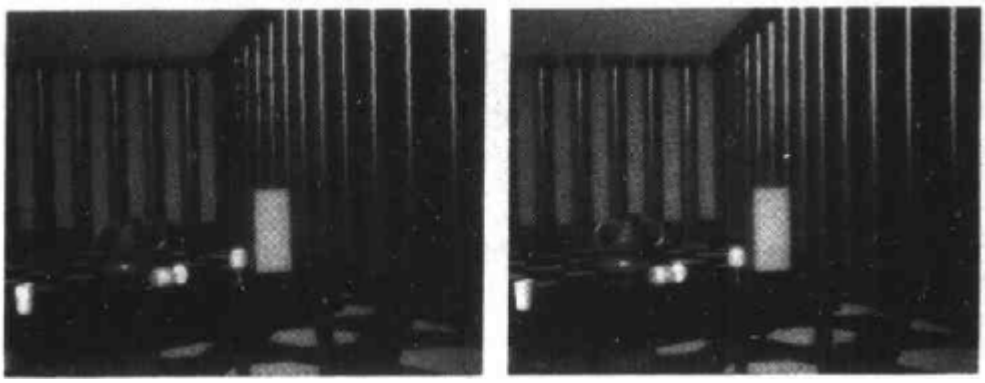
a) b)

图2-63 以256 × 256像素分辨率显示的高分辨率图像。使用均匀平均法

a) 按512 × 512计算图形 b)按1024 × 1024计算图形(由Frank Crow提供)

2.16.4 简单区域反走样

在前面讨论直线光栅化和多边形填充算法时，像素的光强或颜色是由像素区域内一点的光强或颜色决定的。这些方法假定像素是数学上的一个点，而不是一个有限区域。例如，在图2-4和Bresenham算法中，像素的光强由直线与像素边界的交点位置决定。在前面讨论多边形实区域扫描转换方法时，确定像素是否位于多边形之内或之外，是以像素中心位置来决定的。如果像素的中心位置位于多边形之内，则整个像素被激活。如果位于多边形之外，则整个像素被忽略。在简单的两级灰度显示器中，即只有黑或白、多边形颜色或背景颜色，那么这一方法是适用的。其结果是产生阶梯状或锯齿状的多边形的边或线段。如前面讨论过的那样，根本上讲阶梯效应是由于直线和多边形的边的欠采样造成的，阶梯现象与显示器的离散像素相对应。在前面讨论的反走样技术中，子像素对物理像素属性的贡献是基于它的中心位置的。这将反走样现象降到了一个较低级别。



a)　　b)

图2-64　以256 × 256像素分辨率显示的高分辨率图像。采用加权平均法

a) 按512 × 512计算图形　b) 按1024 × 1024计算图形 (由Frank Crow提供)

对于多灰度等级或多色彩显示器，边或直线的外观可通过模糊技术而得到改善。简单的启发式方法是使多边形上像素的光强与该像素在多边形内的面积成正比。图2-65说明了这种简单的反走样方法。图上是一条斜率为5/8的多边形的边，多边形位于该边的右侧。在图2-65a中，多边形的边是在只有两级灰度的情况下按标准Bresenham算法光栅化的结果。该边显示出锯齿形或阶梯形的特征图案。在图2-65b中，用位于多边形内的像素面积来确定像素的光强等级，它们可以等于八级（0 ~ 7）灰度中的一级。可以看到，图2-65a中某些全黑的像素在图2-65b中是白色的，因为这些像素位于多边形内的面积小于像素面积的1/8。

Pitteway和Winkins对Bresenham算法稍作修改后得到像素在多边形内的面积近似值（见[Pitt80]）。可用这一近似值调制像素的光强。当斜率为m（$0 < m < 1$）的直线穿过像素时，如图2-66所示，可能穿过一个或两个像素。如果只穿过一个像素（见图2-66a），那么直线右下方的面积等于$y_i + m/2$。如果必须考虑两个像素（见图2-66b），那么下面像素的面积$Area_1$等于$1-(1-y_i)^2/2m$，上面像素的面积$Area_2$等于$(y_i-1+m)^2/2m$。对于位于第一八分圆域内、斜率为$0 < m < 1$的直线，上面像素的面积可能非常小，在采用上述简单启发式方法时可予以忽略，如图2-65b中的像素（1, 1）。然而，这一面积加上下面像素的面积能更真实地表示多边形的边。这两部分面积之和等于$y_i+m/2$。

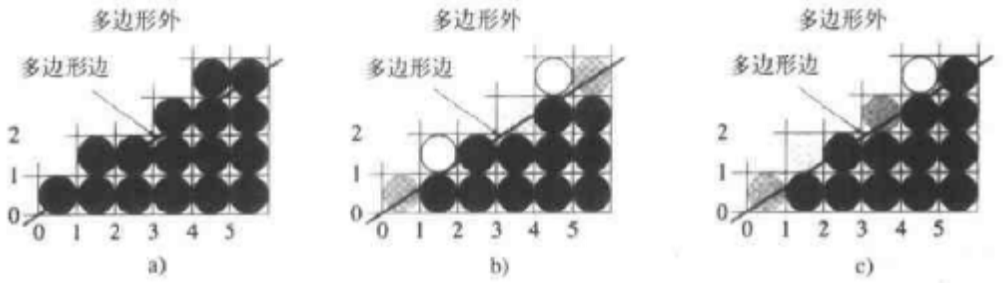


图2-65 简单的反走样算法生成的多边形

a)未经反走样处理 b) 光强与像素在多边形内的面积成正比 c) 采用修正的Bresenham算法

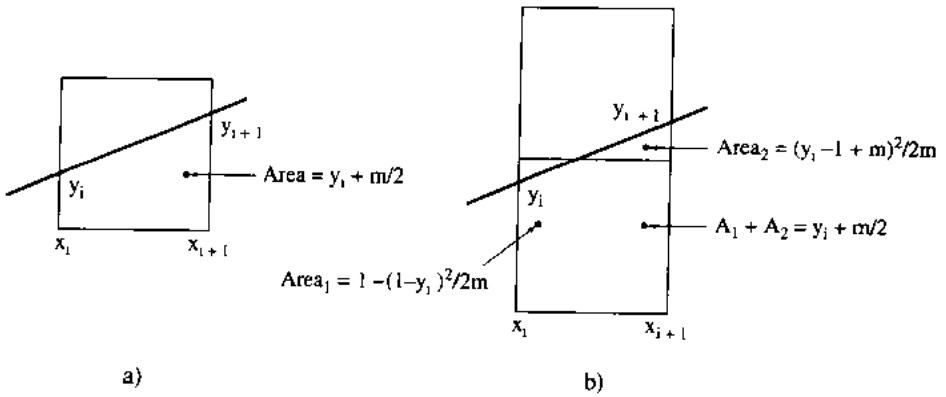


图2-66 采用区域反走样方法的Bresenham算法

将w = 1-m加到原Bresenham算法的误差项中，即引入变换e - = e+w，那么0 ≤ e - ≤ 1。现在，误差项ē表示像素在多边形内面积的大小，即yi+m/2。考虑到这些修正，误差项的初值取1/2。因此，根据图2-6给出的算法，第一个像素的光强总是等于最大值的一半。经过在程序中适当调整setpixel语句，第一个像素显得更为真实。而且，光强值可由斜率（m）、权因子（w）和误差（ē）中乘入光强最大值I而直接得到，而无需逐次计算光强最大值的比例。修正后的算法如下。

modified Bresenham algorithm with antialiasing
the line is from $((x_1, y_1))$ *to* (x_2, y_2)
I *is the number of available intensity levels*
all variables are assumed integer

initialize the variables

$x = x_1$
$y = y_1$
$\Delta x = x_2 - x_1$
$\Delta y = y_2 - y_1$
$m = (I * \Delta y)/\Delta x$
$w = I - m$
$\bar{e} = I/2$

setpixel(x, y, m/2)
while $(x < x_2)$
if $\bar{e} < w$ **then**

```
            x = x + 1
            ē = ē + m
        else
            x = x + 1
            y = y + 1
            ē = ē − w
        end if
    setpixel (x, y, ē)
    end while
finish
```

在计算第一像素的光强时，假设直线的起始坐标地址正好等于像素坐标。算法流程图如图2-67所示。图2-65c给出了应用上述算法画出的斜率为m = 5/8、光强为八级的直线。采用在基本Bresenham算法中用到的类似方法可将这一算法推广到其他八分圆域（见2.3节）。

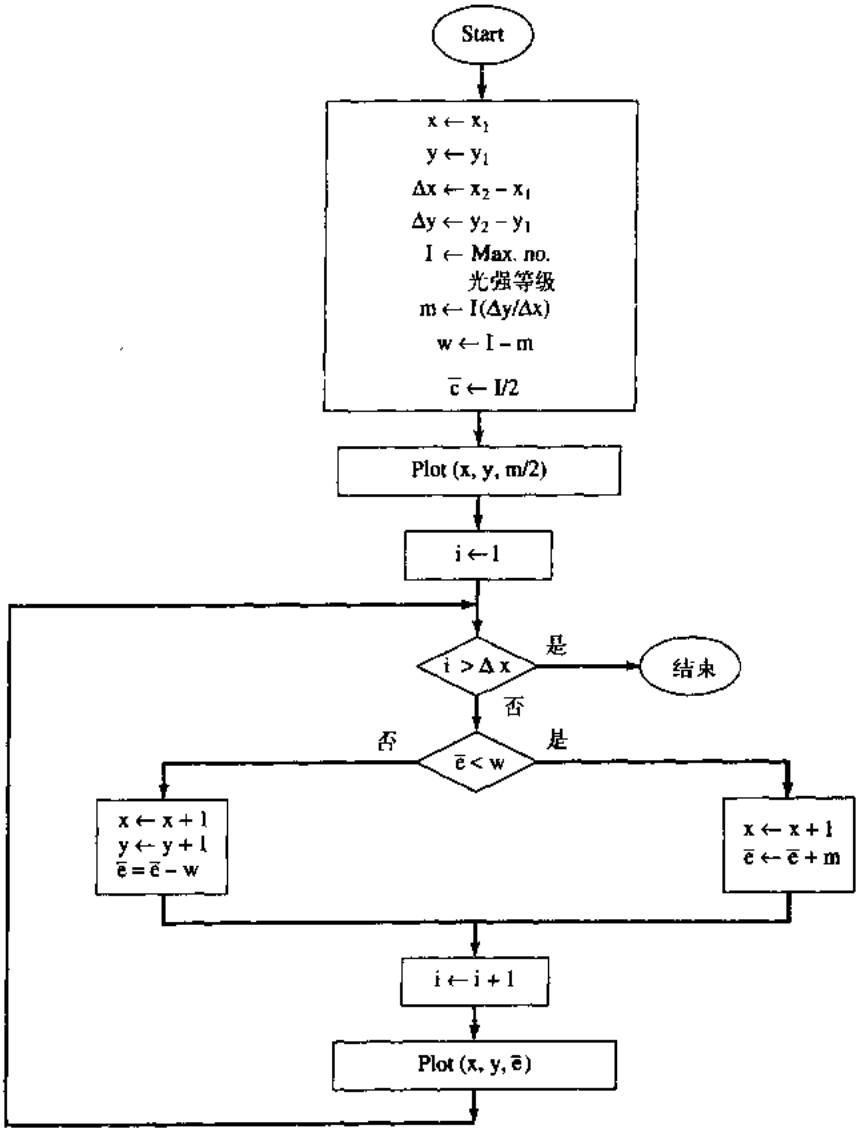


图2-67　使用Bresenham算法的区域反走样

另一种方法是通过子分与多边形边界相交的像素并计算中心在多边形内部的子像素数目来估算像素的面积。最优情况下，子像素数等于强度值。对于彩色，子像素数等每一红、绿、蓝(或粉红、青色、黄色)分量的强度值。通过修正前面给出的pxlcnt子程序可以确定中心在多边形内部的子像素数。对于单色显示，物理像素的强度值为:

$$I=\frac{s}{n^2}I_{\max}$$

其中s是中心在多边形内的子像素数，n^2是子像素数。对于彩色:

$$I(r,g,b)=\frac{s}{n^2}I(r,g,b)_{\text{polygon}}+(1-\frac{s}{n^2})I(r,g,b)_{\text{background}}$$

该技术可以通过对边的盒式滤波卷积来近似模拟边的前置滤波。

图2-68显示了一个斜率为m = 5/8的多边形边。多边形内部在右侧。与边相交的每个物理像素子分为子像素(4 × 4网格)。标记为a的像素有9个子像素，中心在多边形内；而标记为b的像素所有的子像素都在多边形内。像素a用9/16全强度生成，像素b用全强度生成。像素e和像素f在多边形内有4个和13个子像素，分别用1/4和13/16的强度生成。

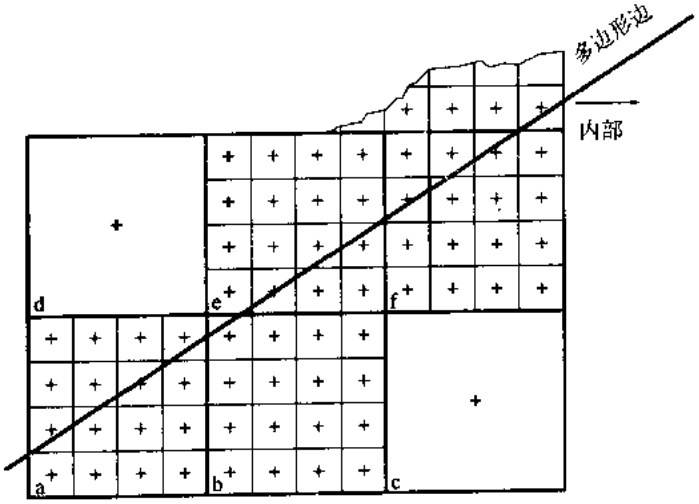


图2-68 使用子分法的近似面积反走样

2.16.5 卷积积分与反走样算法

上节讨论的简单反走样方法的推广需要采用所谓卷积积分的数学方法。为了实现反走样，要用卷积核对图形信号作卷积。其结果可用于确定像素的属性。卷积积分为

$$c(\xi)=\int_{-\infty}^{\infty}h(\xi-x)y(x)dx$$

式中$h(\xi-x)$是卷积核(卷积函数)；$y(x)$是被卷积的函数；c(ξ)是$h(\xi-x)$和$y(x)$的卷积。

从数学定义出发很难直观地理解卷积积分的物理意义。然而通过简单的图形分析，它的意义就清楚了（见[Brig74]）。

用简单的盒形或正方形卷积核（$h(x)$ = 1，$0<x<1$），对函数$y(x)=x$（$0<x<1$）即一条45°线段进行卷积。卷积核如图2-69a所示。卷积核相对纵坐标的反射为$h(-x)$，如图2-69b

所示。然后将反射核向右平移ξ，得到$h(\xi-x)$，见图2-69c。然后将这一经过对称、平移变换后的函数和被卷积函数$y(x)$（见图2-69d）相乘，将不同的ξ值所得的结果示于图2-69e。组合曲线（函数）下的面积就是卷积积分$c(\xi)$的值（见图2-69e）。注意到这时卷积积分只有在$0 \leqslant x \leqslant 2$范围内不等于零。这样，求卷积积分等价于将卷积核作对称、平移变换，并将所得函数与被卷积函数相乘，然后求所得组合曲线下面的面积。卷积核的数学表达式是

$$h(x) = 1,\ 0 \leqslant x \leqslant 1$$

反射变换得到

$$h(-x) = 1,\ -1 \leqslant x \leqslant 0$$

平移ξ得到

$$h(\xi - x) = 1,\ \xi - 1 \leqslant x \leqslant \xi$$

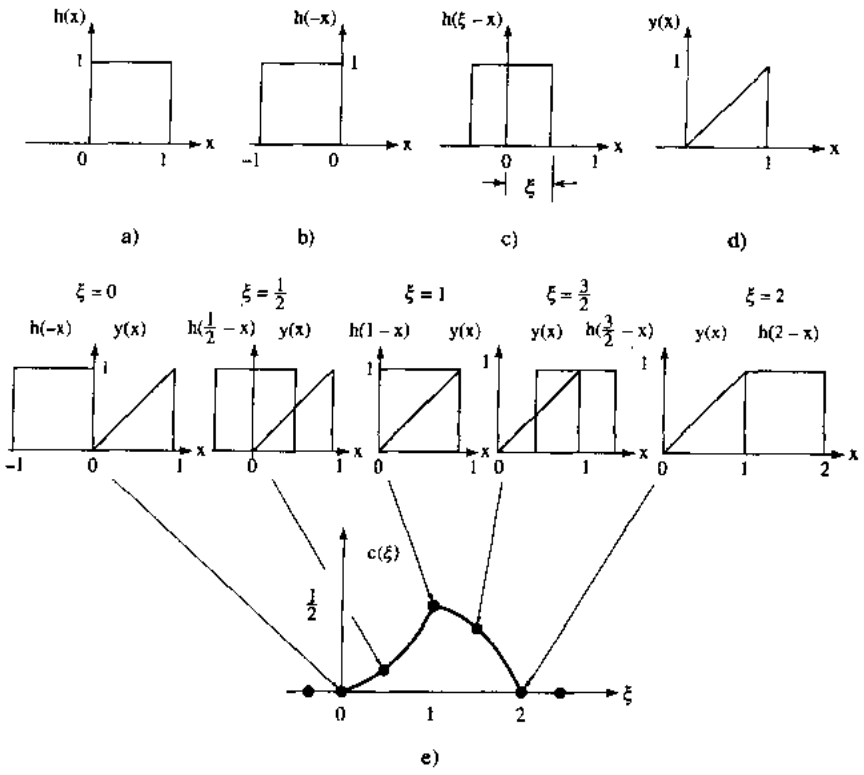


图2-69　卷积

由于卷积核和卷积函数$y(x)$只在有限区间内不为零，所以卷积积分的极限也是有限的。如何求出这些极限值？图2-69清楚地说明，其下限为两个函数不为零时最小值中的最大值，其上限为两个函数不为零时最大值中的最小值。这样，

$$c(\xi) = \int_{-\infty}^{\infty} h(\xi - x)y(x)dx = \int_{0}^{\xi} h(\xi - x)y(x)dx \qquad 0 \leq \xi \leq 1$$

$$= \int_{\xi-1}^{2} h(\xi - x)y(x)dx \qquad 1 \leq \xi \leq 2$$

将$h(\xi-x)$和$y(x)$代入上式得到

$$c(\xi)=\int_0^{\xi}(1)(x)dx=\frac{x^2}{2}\Big]_0^{\xi}=\frac{\xi^2}{2}\qquad 0\le\xi\le 1$$

$$=\int_{\xi-1}^{2}(1)(x)dx=\frac{x^2}{x}\Big]_{\xi-1}^{2}=\frac{\xi}{2}(2-\xi)\qquad 1\le\xi\le 2$$

这两个都是抛物线函数，如图2-69e所示。如果该线的斜率不是1而是m，那么其一般结果为$m\xi^2/2$和$m\xi/2(2-\xi)$。

为了考察这一方法与反走样的关系，回顾一下由像素位于多边形内的面积计算像素光强的方法。从上式给出的卷积函数$c(\xi)$可以看出，当$m<1$时，在像素右边界，即$x=\xi=1$处，卷积函数值等于像素位于多边形内的面积，即为$m/2$（见图2-66a，$y_i=0$）。当$m>1$时，卷积积分的值等于直线穿过的两个像素位于多边形内的面积之和（见图2-66b，$y_i=0$）。这一结果很容易推广到$y_i\neq 0$的情况，这样，前面两个算法（启发式面积调制算法和修正的Bresenham算法）等价于边函数即直线$y=mx+b$在像素右边界处用盒函数（卷积核）的卷积运算。

卷积运算常称为滤波函数，这时卷积核称为滤波函数。上面讨论的简单的面积方法是对图像进行前置滤波。在图像被显示之前由前置滤波调整计算分辨率上的像素属性。

2.16.6 滤波函数

以前讨论的卷积是一维的，而计算机图形图像是二维的。上述理论可以很容易地扩展到二维，但是计算开销较大。一般地讲，可以使用理论结果的近似。

虽然简单的二维盒式滤波器(卷积核)能产生令人满意的结果，但是三角形滤波器、高斯滤波器和B样条滤波器将产生更好的结果。已经研究了各种二维滤波器，如简单盒式、棱锥形、圆锥形以及二维高斯卷积核(滤波函数)等（见[Crow81; Feib80; Warn80; Gupt81]）。图2-70a,b,c给出了几个理论二维滤波器（卷积核）。虽然这里显示的滤波器只覆盖3×3像素的图像区域,但是也有覆盖5×5、7×7、9×9等的像素区域的滤波器。使用滤波器覆盖大片区域的效果是进一步雾化图像。图2-70b,d,e,f所示为理论滤波器的相应离散化近似。可以通过确定每一像素中心处的滤波器的值来获得这些值。即滤波器内的每一像素的强度乘以像素的权重；将这些值加起来，除以滤波器覆盖的像素数乘最大权重。通过归一化权重可以避免除法,这时权重的最大值为1。该加权平均强度可以用于生成滤波器中心的像素。

图2-71的场景与图2-63和图2-64相同，是在256×256像素的分辨率下计算的，采用简单的盒式滤波器进行前置滤波，显示分辨率为256×256。注意它与图2-63和图2-64相比，走样现象减少了。

对于面积小于像素面积的小多边形以及细长多边形，简单的卷积滤波器并不总是有效的。然而，可以采用裁剪方法实现图形反走样（见第3章和[Crow81]）。裁剪窗口由像素区域的边界构成，每个多边形对窗口的边界进行裁剪。裁剪后的多边形面积和像素面积之比，可用来调制像素的光强。如果在像素内有多个小多边形，则用它们属性的平均值或者加权平均值去调制像素的属性。图2-72就是一个这样的例子。

在本书的后面章节还要介绍一些其他的反走样技术，如阴影(5.11节)、纹理(5.12节)、光线跟踪(4.20节和5.16节)和辐射度(5.17节)。

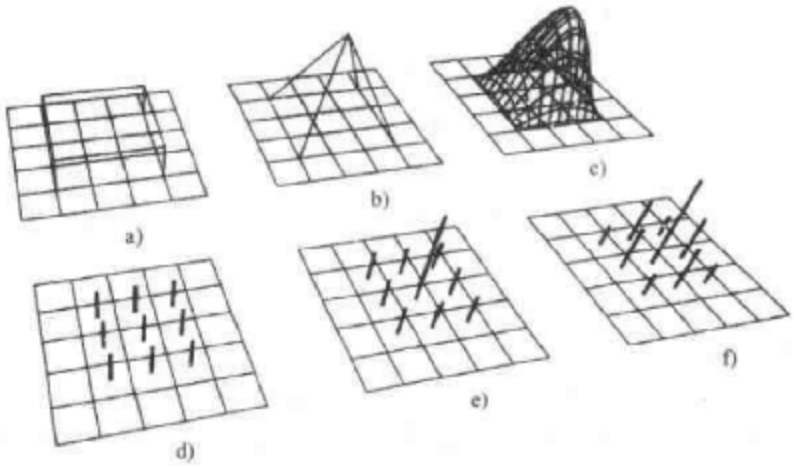


图2-70　二维滤波器函数

a) 简单盒式滤波器　b) 棱锥滤波器　c) 二次B样条滤波器　d) 离散化盒式滤波器

e) 离散化盒式棱锥 滤波器　f) 离散化B样条滤波器

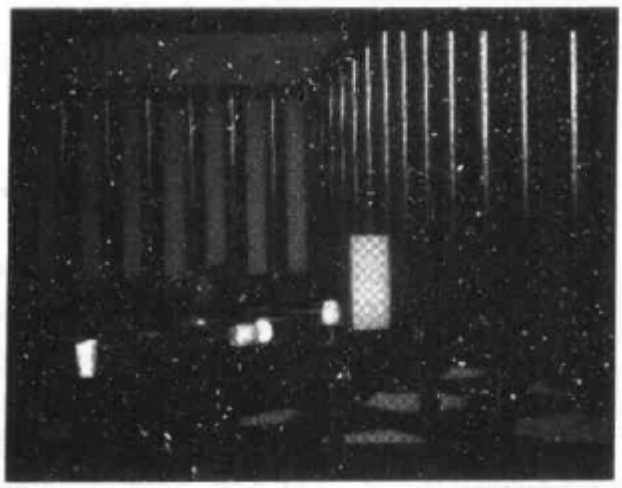


图2-71　分辨率为256 × 256像素，经过前置滤波反走样处理的图像

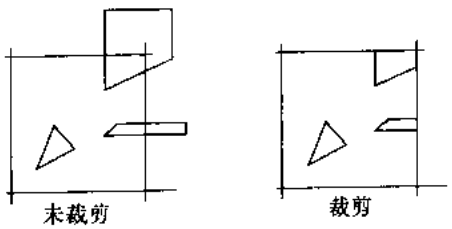


图2-72　使用裁剪的反走样

2.17　半色调技术

反走样是一种采用多级光强提高视觉分辨率的技术。另一方面，半色调技术（halftoning）则是一种采用最少的光强级别，一般为黑和白，来提高视觉分辨率，即获得多

级灰度或多级光强的技术。半色调技术是一门古老的技术。最初用于绘制丝织图案和其他纺织品图案。近代半色调印刷技术是由Stephen Hargon于1880年发明的。运用这一技术在严格两级灰度显示介质（即白纸和黑墨水）上可以产生范围很宽的照相灰度等级。半色调印刷技术是一种丝网或网格印刷过程（见[Koda82]）。印刷点的尺寸与丝网的精细度以及曝光时间的长短有关。报纸上的照片由于纸张质量较差一般采用每英寸50～90点的丝网。书籍和杂志所用的纸张质量较高，因此可以采用每英寸100～300点的丝网。这是半色调技术成功的基础,由于人的视觉系统相当于一个积分器，它可以调和并平滑离散信息。

2.17.1 模版化

计算机生成的图像的视觉分辨率可以通过所谓的模版化（patterning）技术得到提高。半色调技术采用不同尺寸的印点，而模版化技术一般采用固定尺寸的印点。对于一定分辨率的显示器，几个像素被组合成为一个模版单元。因此采用模版化技术改善视觉分辨率的过程是以牺牲空间分辨率为代价的。图2-73a是两级黑白显示器可能采用的一组模版。每个单元由4个像素组成。这种排列产生5种可能的光强或灰度等级（0～4）。一般地讲，对于两级灰度显示器，可能构成的光强等级数等于单元中像素个数加1。对于边长为n的正方形，可以获得2^n+1种亮度。

在选择模版时必须细心，否则会产生不希望有的细微结构图。例如不应该采用图2-73b和图2-73c所示的模版。在光强为常数的大面积区域，图2-73b所示的模版会在图像上产生不希望出现的水平线条，而采用图2-73c时则会出现垂直线条。增大单元的尺寸可以增加光强等级数。图2-74是3×3像素单元的模版图。这些模版产生10（0～9）级光强。模版单元不必为正方形。图2-75是3×2像素单元的模版图，它产生7（0～6）级光强。

为了避免将不必要的瑕疵引入图像，一般不使用对称模版。模版应当是一个不断增长的序列，即在低亮度模版上“打开”的任何一个像素必须出现在高亮度的模版中。模版应当从单元的中心向外增长，这样在亮度增加时，产生点的大小增加的效果。由于许多输出设备，特别是激光打印机和胶片，不能很好地绘制隔开的点（像素），所以像素必须是相邻的。注意如图2-73a和图2-74所示的模版遵循了这个规则；同时还要注意，这些模版与水平方向成45°角。这个角度与传统照相半色调中的“画面角”(screen angle)相对应。Postscript(见[Adob85])的半色调模版使用了可变画面角。Knuth（见[Knut87b]）对用数字半色调生成字体进行了实验。

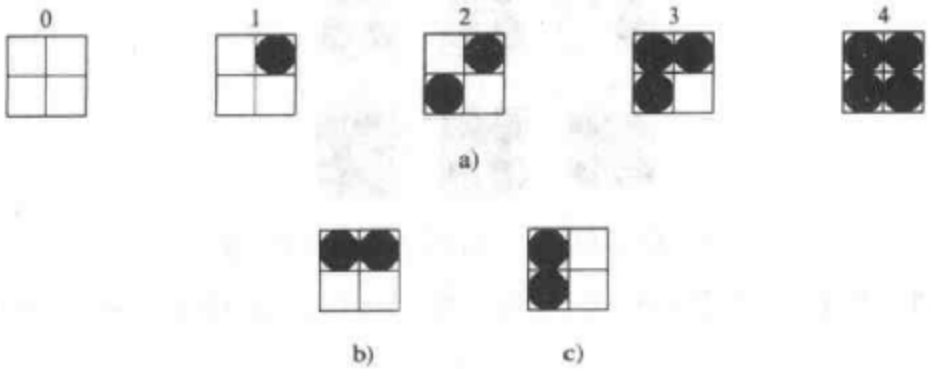


图2-73 2×2二级模式单元

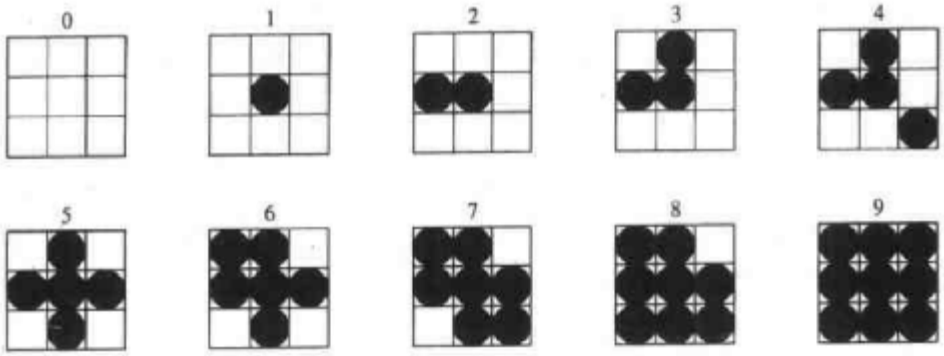


图2-74　3×3二级模式单元

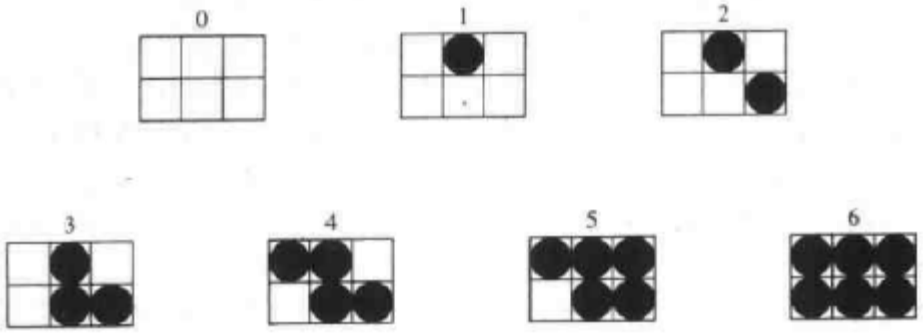


图2-75　3×2二级模式单元

如果点的尺寸可以变化，则能获得更多的光强等级。图2-76是2×2像素单元的模版图，它有两种尺寸的点，可以形成9级光强。一个类似的具有两种尺寸点的3×3像素单元可以构成27种光强等级。如果每一像素用多个二进制位表示，那么也可以得到更多的光强等级。每个像素用两个二进制位表示的2×2像素模版将构成13级光强，见图2-77。每个像素所对应的二进制位数越多，模版单元越大，则产生的光强等级数就越多（见[Pirs83]）。

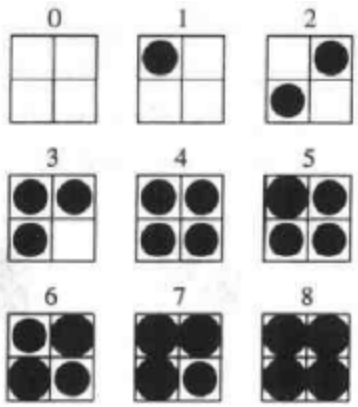


图2-76　具有多种尺寸的点的2×2两级灰度模式单元

可以将半色调单元模版看作是矩阵。例如，图2-73所示的简单的2×2模版可以用以下矩阵表示：

$$[P_2]=\begin{bmatrix}3 & 1\\ 2 & 4\end{bmatrix}$$

同理，图2-74所示的3 × 3模式可以用以下矩阵表示:

$$[P_3]=\begin{bmatrix}6 & 3 & 9\\ 2 & 1 & 5\\ 8 & 7 & 4\end{bmatrix}$$

如果要显示的像素的光强I等于或大于模版矩阵中的值，则激活模版单元中该位置的相应像素。

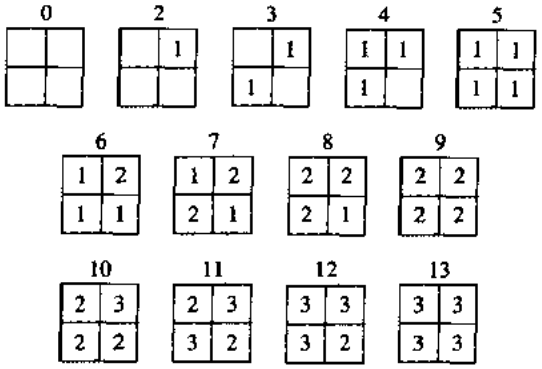


图2-77 每个像素用两个二进制位表示的2 × 2模式单元

要显示的真实图像很少存储为模版矩阵中的最大光强，因此，将模版矩阵的入口值标准化为0 ~ 1之间的值是很有利的。对于模版方阵，标准化矩阵 $[\hat{P}_n]$ 为:

$$[\hat{P}_n]=\frac{1}{n^2+1}[P_n]$$

例如，归一化后的 $[\hat{P}_2]$ 矩阵为

$$[\hat{P}_2]=\frac{1}{n^2+1}[P_2]=\frac{1}{5}[P_2]=\begin{bmatrix}0.6 & 0.2\\ 0.4 & 0.8\end{bmatrix}$$

归一化后的$[P_3]$矩阵为:

$$[\hat{P}_3]=\frac{1}{n^2+1}[P_3]=\frac{1}{10}[P_3]=\begin{bmatrix}0.6 & 0.3 & 0.9\\ 0.2 & 0.1 & 0.5\\ 0.8 & 0.7 & 0.4\end{bmatrix}$$

如果要显示的图像的最大光强为$I_{\max}$，则通过将标准化模式矩阵 $[\hat{P}_n]$ 的每一入口与$I_{\max}$相乘并取整而转换成显示模式矩阵$[P_{n_D}]$，即:

$$[P_{n_D}]=\mathrm{Int}(I_{\max}[\hat{P}_n])$$

例如，如果$I_{\max}=255$，则:

$$[P_{2_D}]=\mathrm{Int}\left(255\begin{bmatrix}0.6 & 0.2\\ 0.4 & 0.8\end{bmatrix}\right)=\begin{bmatrix}153 & 51\\ 102 & 204\end{bmatrix}$$

且:

$$[P_{3_D}]=\mathrm{Int}\left(255\begin{bmatrix}0.6 & 0.3 & 0.9\\ 0.2 & 0.1 & 0.5\\ 0.8 & 0.7 & 0.4\end{bmatrix}\right)=\begin{bmatrix}153 & 76 & 229\\ 51 & 25 & 127\\ 204 & 178 & 102\end{bmatrix}$$

一个有趣的问题是需要有多少光强等级来表示连续色调(照片的)图像（类似于本书的图像）？对于4、8和16的光强等级，光强轮廓还可能很清楚，但是对64级(n=6)光强等级，轮

廓可能完全消失。为了消去光强轮廓的剩余效果，需要n=8，即256级强度级。为了以300 dpi（每英寸点数）的分辨率、256级光强等级、使用8×8模版来显示图像，需要物理设备的分辨率为2400 dpi。将光强等级减小到64(n=6)需要物理设备的分辨率为1800 dpi。

2.17.2 阈值和误差分布

模版化使得空间分辨率降低。如果图像分辨率低于显示分辨率，那么这种结果是可以接受的。一些既能改善视觉分辨率又能保证空间分辨率不降低的技术也已提出来了（见[Jerv76]）。最简单的一种方法是对于每个像素取固定的阈值。在该像素上图像的光强超过某阈值时，置成白色；否则为黑色，即

if I(x, y) > T **then** White **else** Black

式中$I(x, y)$是图像在(x, y)像素上的光强。White对应于最高显示光强，Black对应于最低显示光强。通常阈值取为最高光强的一半。图2-78b是图2-78a所示照片在T = 150时的结果。原始照片的每一个像素的光强按0 - 255级即8比特数字化。简单阈值的矩阵形式是一个1×1矩阵[T]，T的取值范围对于归一化矩阵 $\hat{T}$ 为$0 \leqslant \hat{T} \leqslant 1$，对于显示矩阵为$0 \leqslant T \leqslant I_{max}$。

图2-78b表明简单的阈值方法会导至大量细节的丢失。这在头发和脸部特别明显。细节丢失是由于每个像素所显示光强的相对误差较大所造成的。

Floyd和steinberg（见[Floy75]）提出了一种方法将这一误差分散到周围像素之中。而且算法很巧妙，它是将误差向下（上）和向右分散。因此，如果是按扫描线顺序计算图像，那么无需进行反向跟踪。特别地，误差的7/16传给当前扫描线右侧的像素，3/16传给扫描线下部(或上部)的左侧像素，5/16传给正下侧(或正上侧)的像素，1/16传给当前像素下(或上)部的扫描线的右侧像素（见图2-79a）。根据从左到右、从上到下的扫描线次序，相应的误差分布矩阵为:

$$\frac{1}{16}\begin{bmatrix} 0 & 0 & 0 \\ 0 & * & 7 \\ 3 & 5 & 1 \end{bmatrix}$$

其中，*表示当前像素，0表示没有误差传给扫描线上部和当前像素左侧的像素。

令阈值等于显示光强的最高和最低的一半，即T=(Black+White)/2，则算法为

Floyd–Steinberg error distribution algorithm

```
Xmin, Xmax, Ymin, Ymax are the raster limits for each scan line –
  top to bottom
    T = (Black + White)/2
    for y = Ymax to Ymin step – 1
        for each pixel on a scan line–left to right
        for x = Xmin to Xmax
            determine pixel display value for threshold T
              and calculate error
            if I(x, y) < T then
                Pixel(x, y) = Black
                Error = I(x, y) - Black
            else
                Pixel(x, y) = White
```

```
            Error = I(x, y) − White
        end if
        display pixel
        setpixel (x, y, Pixel(x, y))
        distribute error to neighboring pixels
        I(x + 1, y) = I(x + 1, y) + 7 * Error/16
        I(x + 1, y − 1) = I(x + 1, y − 1) + Error/16
        I(x, y − 1) = I(x, y − 1) + 5 * Error/16
        I(x − 1, y − 1) = I(x − 1, y − 1) + 3 * Error/16
    next x
next y
finish
```

可以很巧妙地构建该算法的简化方法，使得误差可以向下和向右传播。因此，如果图像以扫描线次序计算，则不需进行反向跟踪。Floyd-Steinberg算法将3/8误差分散到右边像素，3/8分散到下面像素，1/4分散到对角线像素上(见图2-79b)。可以很简单地修改前面给出的伪代码，完成简化算法。将误差分散到相邻像素可以改进图像的细节,因为它保留了图像中的信息相关性。

除了Floyd-Steinberg算法之外，Jarvis(见[Jarv76])、Stucki（见[Stuc81]）和Knuth（见[Knut87a]）都给出了单遍扫描误差分散算法。与数字半色调技术(见[Schu91; Gots93])相比，Jarvis等给出的最小化平均误差的分散误差算法效果很好，特别对于图像的动画序列。Jarvis等的误差分散矩阵为:

$$\frac{1}{48}\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & * & 7 & 5 \\ 3 & 5 & 7 & 5 & 3 \\ 1 & 3 & 5 & 3 & 1 \end{bmatrix}$$

其中，*表示当前像素，0表示没有得到分散误差的像素。Jarvis等的矩阵力求使源图像与显示图像之间的误差最小化。它还进行了边界增强处理。边界像素需要特殊处理。Jarvis和他的合作者们将顶部两个像素宽的区域的误差分散值设置为0。另外还可使用多次扫描的巡回误差分散算法(见[Wu94])。

因为光栅扫描的固定次序为从左到右，误差分散算法有在图像中引入重复模版的趋势。可以通过打破严格扫描线次序的方式减少这些重复模版造成的效果。Ulichney（见[Ulic87]）建议使用来回式模版，即从左至右生成一扫描线，再从右至左生成另一扫描线。另一种图像生成模式，例如Peano曲线（即一种分形曲线，参见[Witt82])也可产生较好的结果。Velho和Gomes（见[Velh91]）将该技术扩展到Hilbert曲线。

Zhang（见[Zhan96]）开发的线分散算法沿着垂直于图像对角线的方向处理图像，即沿着与图像的左上角到图像右下角的对角线相垂直的方向处理图像。该算法随机地分散误差，使用由Knuth在点分散算法[Knut87a]中引入的像素类概念。误差分散到右边的像素、下面的像素、对角线下的像素，再右边的像素，和简化的Floyd-Steinberg算法类似（参见图2-78b)。线分散算法很容易并行进行，并行算法可以利用神经网(见[Lipp87])。算法也可使用有效的串行实现。图2-80给出了线分散算法与Floyd-Steinberg误差分散算法的比较。

图2-78　两级灰度显示技术

a) 原始照片　b) 简单阈值方法　c) 带有8 × 8抖动矩阵的有序抖动

（贝尔实验室J.F.Jarvis提供）

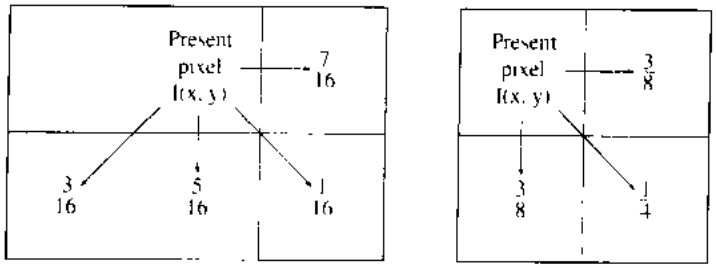


图2-79 Floyd-Steinberg算法的误差分布

a) 原始算法 b) 简化算法

a)

b)

图2-80 半色调图像Zelda

a) 边加强分散算法 b)带边加强的Floyd-Steinberg误差分散算法

(yuefeng zhang提供)

2.17.3 有序抖动

有序抖动法（dither）是另一种适用于两级灰度显示器、能提高视觉分辨率而又不降低空间分辨率的方法。这个方法在图像中引入一个随机误差。在每一像素的光强与所选的阈值进行比较之前，将这一误差加到该像素的光强上。引入一个完全随机误差并不产生最佳效果。但是存在着一种最佳添加误差，它能使模版纹理效应降到最低程度（见[Baye73]）。由于误差模版比图像小，所以它是平铺到整个图像中，即以国际象棋棋盘模式加到图像中去，该技术是分散式有序点抖动的一种形式。与激光打印机或胶片这样的输出设备不同，CRT可以绘制单个的像素。因此可以选择分散式有序点抖动技术。

最小的有序抖动模版或矩阵是2 × 2。最优的2 × 2矩阵最早由Limb给出（见[Limb69]），见下式

$$[D_2]=\begin{bmatrix}0 & 2\\ 3 & 1\end{bmatrix}$$

用递归关系可以得到更大的抖动模版，如4 × 4、8 × 8等(见[Jarv76])

$$[D_n]=\begin{bmatrix}4D_{n/2} & 4D_{n/2}+2U_{n/2}\\ 4D_{n/2}+3U_{n/2} & 4D_{n/2}+U_{n/2}\end{bmatrix} \qquad n \geqslant 4$$

式中n是矩阵的大小，且

$$[U_n]=\begin{bmatrix}1 & 1 & \cdots & 1\\ 1 & 1 & & \\ \vdots & & & \\ 1 & & & \end{bmatrix}$$

例如4 × 4的抖动矩阵是

$$[D_4]=\begin{bmatrix}0 & 8 & 2 & 10\\ 12 & 4 & 14 & 6\\ 3 & 11 & 1 & 9\\ 15 & 7 & 13 & 5\end{bmatrix}$$

这两个例子说明抖动矩阵D_n将产生n^2个光强。而且随着n增加，图像并不会丢失它的空间分辨率。规范化有序抖动矩阵$[\hat{D}_n]$可以通过除以n^2-1取得。对于给定的最大图像强度I_{max}，显示抖动矩阵为$I_{max}[\hat{D}_n]$。有序抖动算法可描述如下。

```
Ordered dither algorithm
Xmin, Xmax, Ymin, Ymax are the raster limits for each scan line —
  top to bottom
Mod is a function that returns the modulo value of its arguments
    for y = Ymax to Ymin step −1
        for each pixel on a scan line-left to right
        for x = Xmin to Xmax
            determine position in dither matrix
            i = (x Mod n) + 1
            j = (y Mod n) + 1
            determine pixel display value
            if I(x, y) < D(i, j) then
```

```
            Pixel(x, y) = Black
        else
            Pixel(x, y) = White
        end if
        display pixel
        setpixel(x, y, Pixel(x, y))
    next x
next y
finish
```

图2-78c是图2-78a的原始照片经8 × 8有序抖动矩阵处理后的图像。8 × 8抖动矩阵有效地产生了64级光强。在图2-78c上，图像的大量细节得到恢复。同时要注意,有序抖动会减少对比度。

如果显示分辨率大于图像的分辨率,则可以通过插值来增加图像的分辨率，使之达到显示分辨率，并在结果上使用有序抖动的方法来改进显示图像的质量。Jarvis等[Jarv76]建议的插值机制为:

$$I_{x,y} = S_{x',y'}$$

$$I_{x+1,y} = \frac{1}{2}(S_{x',y'} + S_{x'+1,y'})$$

$$I_{x,y+1} = \frac{1}{2}(S_{x',y'} + S_{x',y'+1})$$

$$I_{x+1,y+1} = \frac{1}{4}(S_{x',y'} + S_{x'+1,y'} + S_{x',y'+1} + S_{x'+1,y'+1})$$

其中(x', y')和(x, y)分别是图像的像素位置和显示的像素位置，$x=2x'$，$y=2y'$，$S_{x',y'}$和$I_{x,y}$分别表示图像的像素和要显示的像素的光强。

Ulichney[Ulic87]的数字半色调标准参考书中给出了分散式点有序抖动的额外细节。Ulichney还讨论了旋转的抖动模版，在抖动算法中加入任意噪声（蓝噪声和白噪声），并使用锐化过滤器。Ostromoukhov等[Ostr94]也讨论了旋转的分散式点抖动。

以前讨论的半色调假设是两级（黑色和白色）显示介质。对于多级光强单色(灰度级)或彩色显示图像和介质也可使用半色调。基本原理不变。在彩色图像和显示介质的情况下，对每一色彩分量分别和同时

$$\text{粉红} = \begin{bmatrix} 6 & 13 & 7 & 14 \\ 10 & 1 & 9 & 3 \\ 5 & 15 & 4 & 12 \\ 11 & 2 & 8 & 0 \end{bmatrix} \qquad \text{黄色} = \begin{bmatrix} 11 & 5 & 10 & 6 \\ 2 & 15 & 1 & 13 \\ 8 & 4 & 9 & 7 \\ 0 & 12 & 3 & 14 \end{bmatrix}$$

Kubo也用预置的“随机”模版进行了实验。

Heckbert[Heck82]使用简化的Floyd-Steinberg误差分散算法来改进量化彩色图像质量。Thomas和Bogart[Thom91]给出了Floyd-Steinberg算法的多级光强版本。对于多级光强显示，重要的是在任意位置(x, y)的图像光强不能超过最大允许显示光强。如果超过最大允许显示光强，则该值置为最大允许值。

Thomas和Bogart[Thom91]还给出了有序抖动算法的多级光强版本。如果多级光强显示介质有$l+1$个可能光强，则有序抖动算法的内循环变为：

determine pixel display value

$$\textbf{if } \mathrm{I(x,y)} - \textbf{Integer}\left(\frac{\mathrm{I_{max}}}{\ell} \textbf{Integer}\left(\frac{\mathrm{I(x,y)}}{\mathrm{I_{max}}} \ell \right) \right) < \mathrm{D(i,j)} \textbf{ then}$$

$$\quad \mathrm{Pixel(x,y)} = \textbf{Integer}\left(\frac{\mathrm{I(x,y)}}{\mathrm{I_{max}}} \ell \right)$$

else

$$\quad \mathrm{Pixel(x,y)} = \textbf{Integer}\left(\frac{\mathrm{I(x,y)}}{\mathrm{I_{max}}} \ell \right) + 1$$

end if

其中$I(x, y)$是(x, y)处图像像素的强度，I_{max}是显示介质的最大可用光强，而$P(x,y)$是显示像素的光强。

Well等[Well91]使用基于人的视觉系统特征的非线性色彩抖动技术来改进图像质量。他特别将24位图像(8位红色、8位绿色、8位蓝色)抖动到12位（红色、绿色和蓝色各4位）。对于矢量的情况，他们发现将抖动矩阵与矢量方向对齐可以改进结果，特别是反走样矢量的情况(参见2.27和2.28节)。