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）。

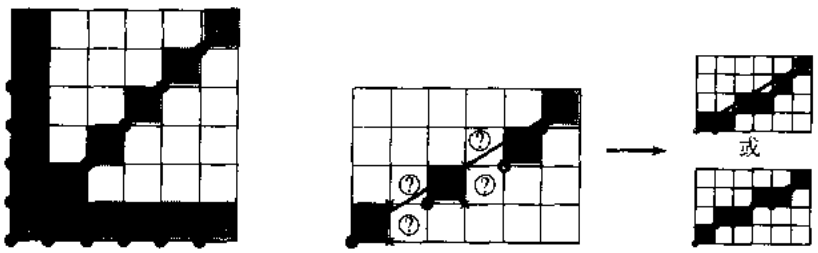


图2-1　直线的光栅化

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。

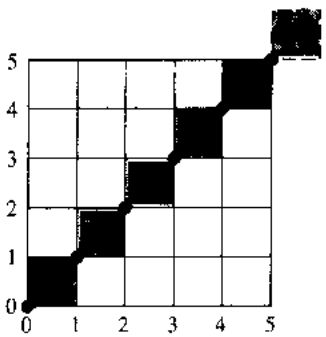


图2-2 在第一象限简单DDA算法产生的结果

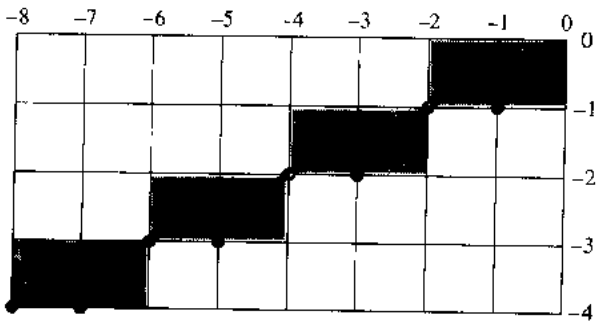


图2-3 在第三象限简单DDA算法生成的结果

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）。

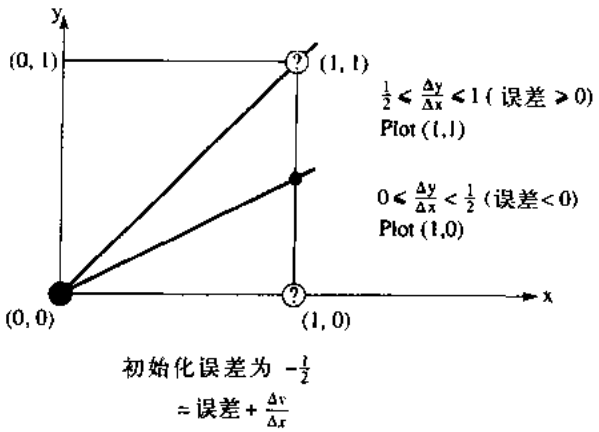


图2-4 Bresenham算法基础

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

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

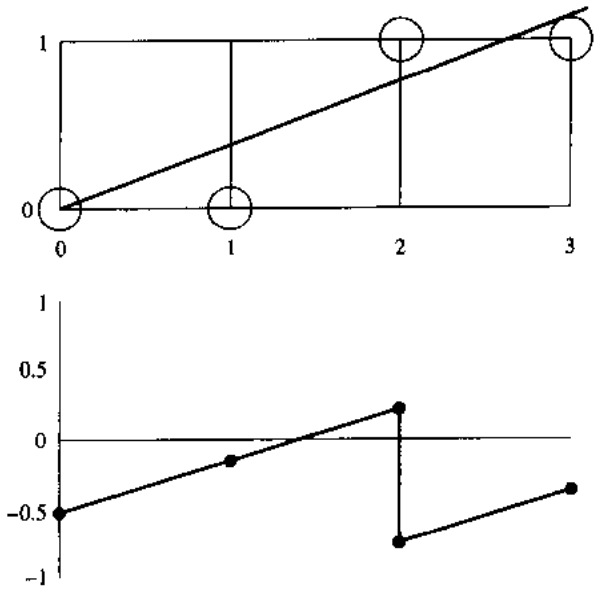


图2-5 Bresenham算法的误差项

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。　□

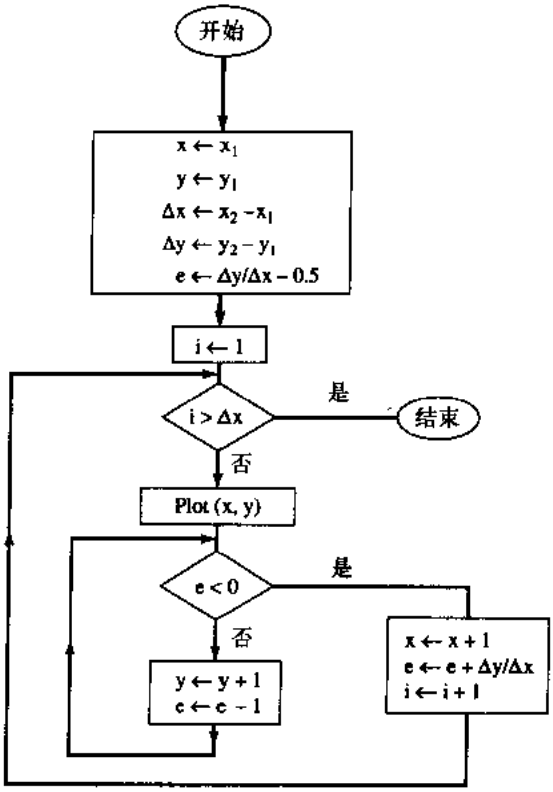


图2-6 Bresenham算法流程图

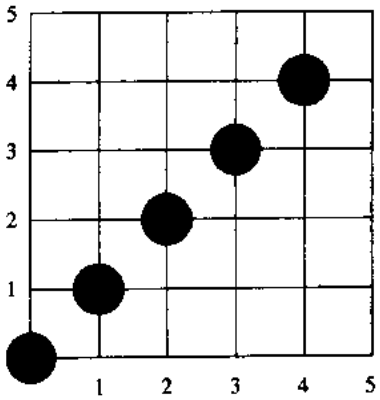


图2-7 在第一八分圆域的Bresenham算法

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
```

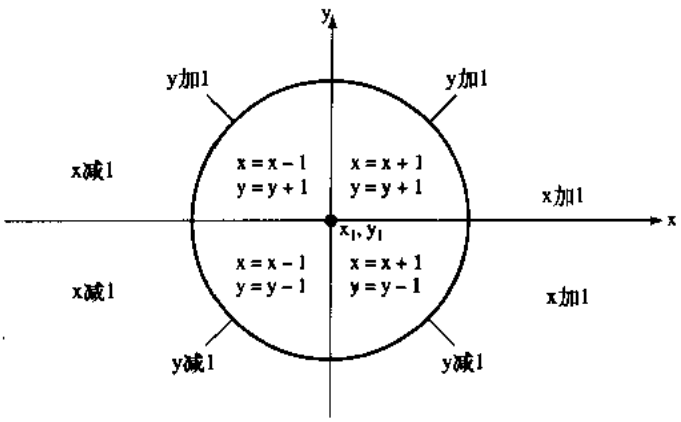


图2-8 通用Bresenham算法的判别条件

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。

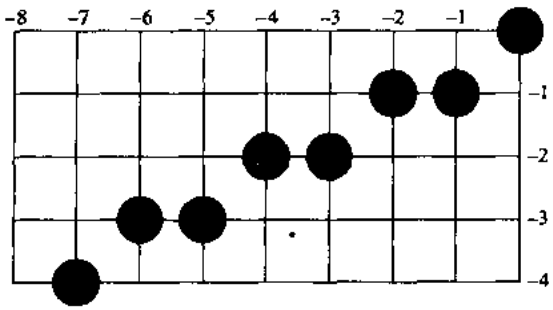


图2-9　在第三象限中通用Bresenham算法的执行结果

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。

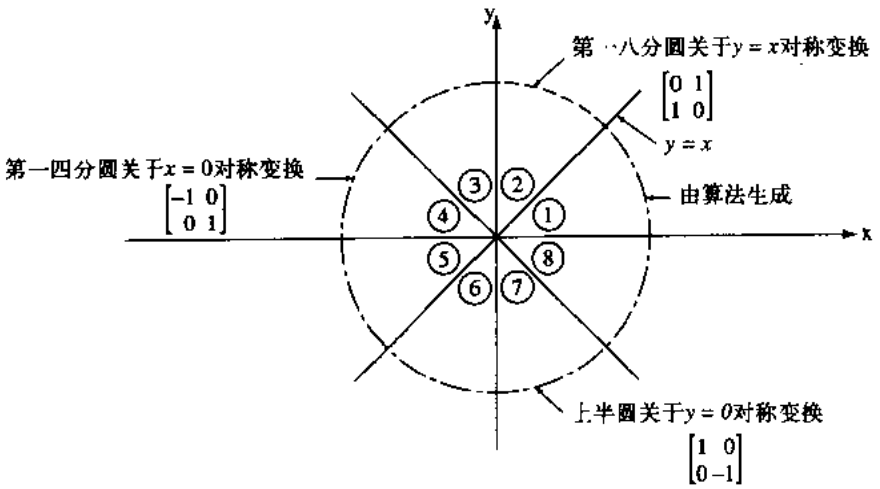


图2-10 由第一八分圆生成整圆

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。

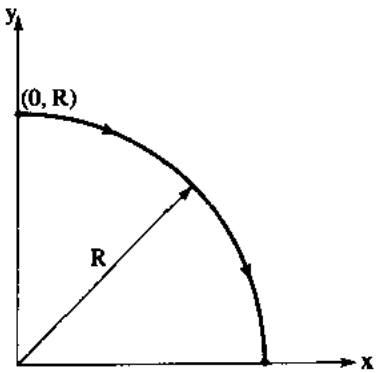


图2-11 第一四分圆

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。

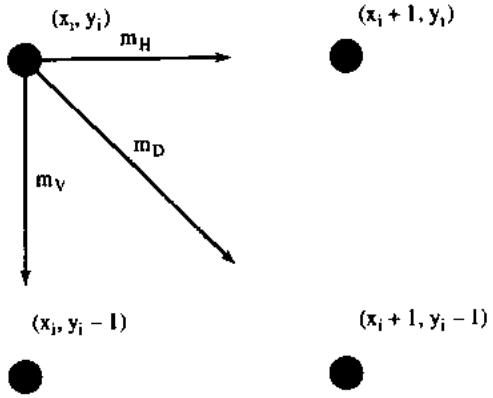


图2-12　第一象限像素的选取

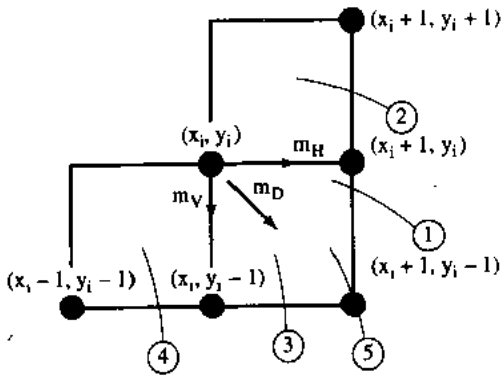


图2-13　圆和光栅网格的相交关系

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。

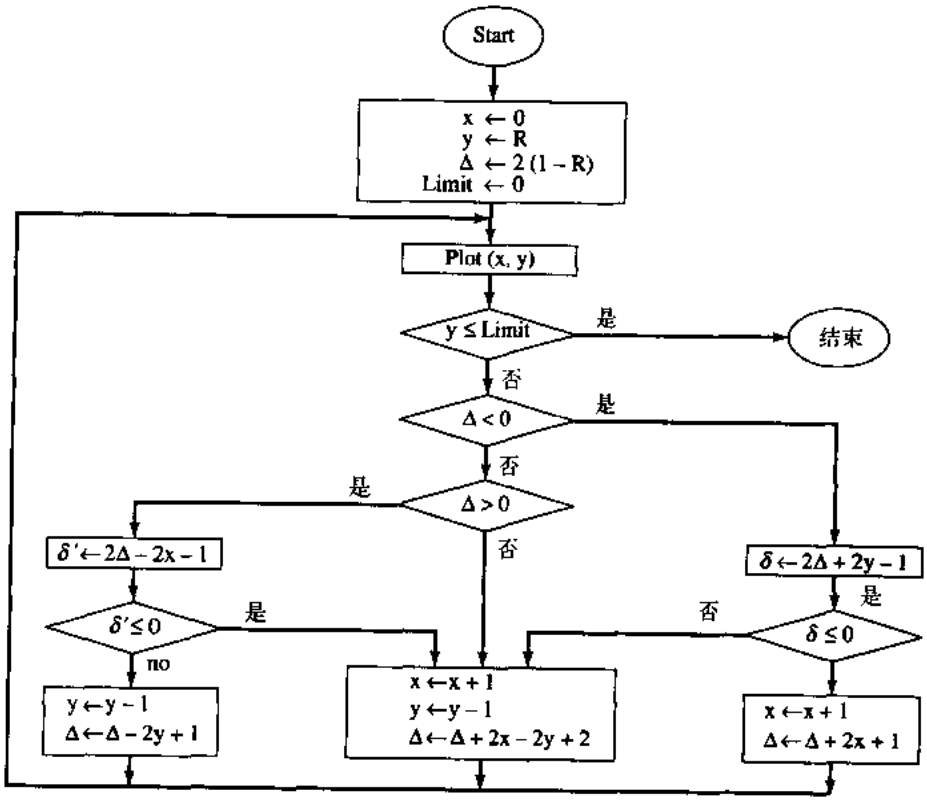


图2-14　第一象限Bresenham 增量画圆算法流程图

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

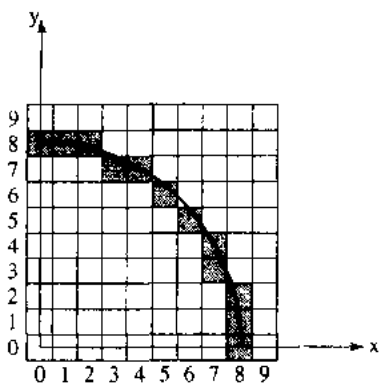
y_i > Limit
continue
setpixel (0, 8)
$\Delta_i < 0$
$\delta = 2(-14) + 2(8) - 1 = -13$
$\delta < 0$
call mh(0,8,−14)
x = 0 + 1 = 1
$\Delta_i = -14 + 2(1) + 1 = -11$
y_i > Limit
continue
setpixel (1, 8)
$\Delta_i < 0$
$\delta = 2(-11) + 2(8) - 1 = -7$
$\delta < 0$
call mh(1,8,−11)
x = 1 + 1 = 2
$\Delta_i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

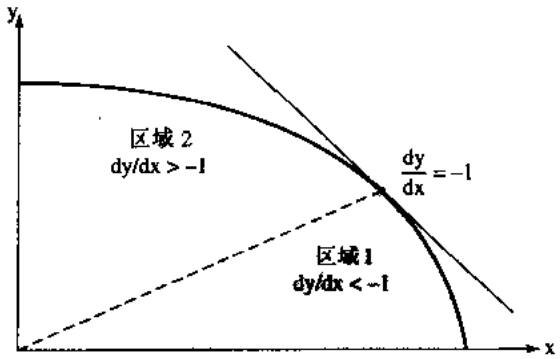


图2-16 将椭圆分成两个部分

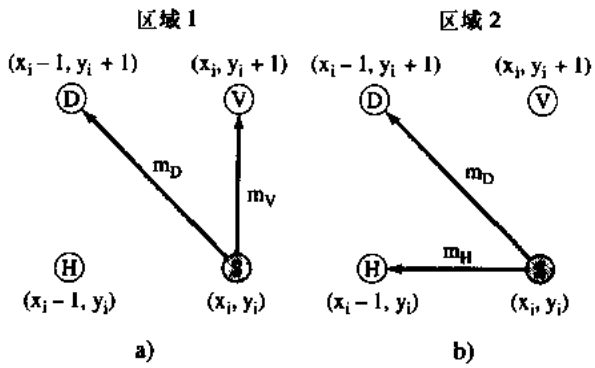


图2-17 像素选择

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:

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

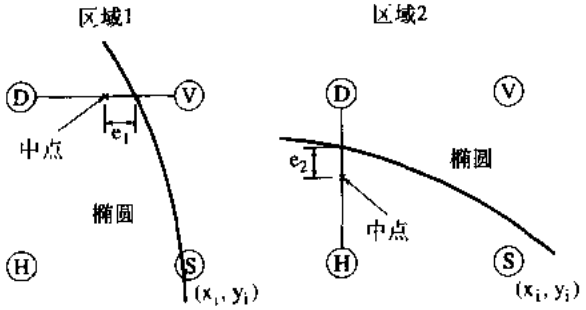


图2-18　椭圆的误差标准

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。

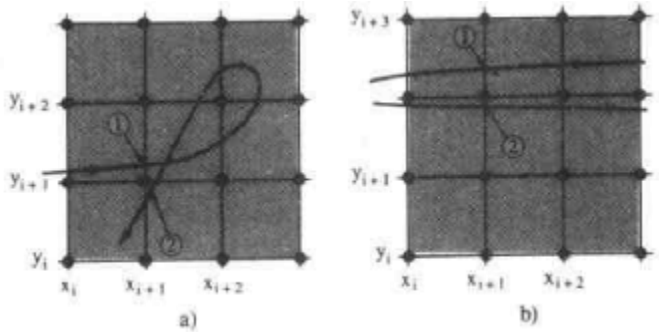


图2-19　一般多项式函数的光栅化问题

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。

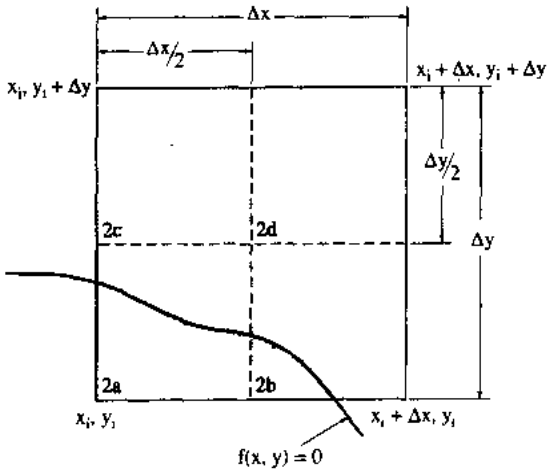


图2-20 递归空间子分

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