计算机图形学的基础算法

刘勇奎 著

第2版

科学出版社
www.sciencep.com

计算机图形学的基础算法

（第 2 版）

刘勇奎　著

科 学 出 版 社

北 京

内 容 简 介

本书是作者总结十多年来对计算机图形学基础算法研究成果的一部专著。书中大部分内容为作者已发表或尚未发表的研究成果。主要包括：图形的生成、裁剪，六角网格上的图形算法、三维图形算法，以及与图形相关的图像处理与识别算法等。书中内容主要侧重于较新的像素级算法和三维图形算法。

本书适于计算机图形学的专业研究人员及大专院校师生阅读参考。

图书在版编目(CIP)数据

计算机图形学的基础算法/刘勇奎著. —2版. —北京：科学出版社，2007
ISBN 978-7-03-018311-8

Ⅰ.计… Ⅱ.刘… Ⅲ.计算机图形学—算法理论—研究 Ⅳ.TP391.41

中国版本图书馆CIP数据核字(2006)第156141号

责任编辑：张 敏 / 责任校对：郑金红
责任印制：安春生 / 封面设计：嘉华永盛

科 学 出 版 社 出版
北京东黄城根北街16号
邮政编码：100717
http://www.sciencep.com
西 源 印 刷 厂 印刷
科学出版社发行 各地新华书店经销
*
2007年1月第 一 版 开本：B5(720×1000)
2007年1月第一次印刷 印张：12 1/4
印数：1—2 500 字数：232 000

定价：36.00元

(如有印装质量问题，我社负责调换〈路通〉)

前　言

本书是作者多年来关于计算机图形学及相关的图像处理与模式识别研究成果的系统归纳和总结。书中包括了图形的生成、裁剪及图像显示与识别方面的多种算法。其中除了 Bresenham 算法等少数几个算法之外，均为作者提出的算法。这些算法有的已在国内外发表，有的还尚未发表。除此之外，本书还简要介绍了这些领域目前国际上最新的成果。

本书第二版在第一版的基础上，补充了作者近五年的研究成果，增加了“三维图形算法”和“压缩链码”两章。

书中首先介绍了直线和曲线生成的几个算法。其中主要是像素级整数型生成算法。例如对于直线生成，目前国际上的研究热点是一步生成多点的算法，本书也讨论了这样的算法。另一个重点内容是介绍了一个可生成任意曲线的像素级整数算法。目前出现的像素级生成算法只有直线、圆及其他圆锥曲线，而该算法可生成任何常用曲线。然后是图形的裁剪算法，包括在其他文献中较难见到的凹多边形窗口的裁剪算法及曲线窗口和曲线裁剪算法等。其次是与图形相关的图像处理及模式识别算法。最后介绍了六角网格的特点及在其上使用的图形及图像处理算法。

本书的研究得到了国家自然科学基金(项目号:69403004,60473108,60675008)的资助。在本书出版之际，首先要感谢浙江大学石教英教授，他为作者提供了许多学术上的帮助，还要感谢诸位同事及学生李笑牛、王晓强、王鹏杰、何丽君、云健等，他们对书中的大部分算法进行了实现和验证。

目　　录

前言

第1章　直线与曲线的生成 …… 1

1.1　圆及椭圆的多边形逼近及线式生成 …… 1

1.2　直线的像素级生成算法 …… 5

1.2.1　Bresenham 直线生成算法 …… 6

1.2.2　单点直线生成算法已无优化的余地 …… 8

1.2.3　一个双点 Bresenham 直线生成算法 …… 9

1.2.4　直线的对称生成 …… 12

1.2.5　多点直线生成算法 …… 14

1.2.6　多点直线生成算法所存在的问题 …… 17

1.2.7　多灰度级直线 …… 17

1.3　圆的像素级生成算法 …… 19

1.3.1　圆的像素级生成算法概述 …… 19

1.3.2　最快的像素级圆生成单点算法 …… 21

1.3.3　圆的双步(双点)生成算法 …… 23

1.3.4　圆生成算法的比较 …… 25

1.4　抛物线的像素级生成算法 …… 27

1.5　一个通用的隐函数曲线逐点生成算法 …… 31

1.6　等值线的抽取与绘制 …… 33

1.7　将参数曲线转换成隐函数曲线后生成 …… 37

1.7.1　二次 Bezier 曲线的生成 …… 37

1.7.2　三次 Bezier 曲线的生成 …… 41

1.7.3　算法的进一步优化 …… 43

1.7.4　二次和三次 B 样条曲线的生成 …… 45

1.8　参数曲线的像素级生成算法 …… 47

1.8.1　现有算法介绍 …… 47

1.8.2　最佳的 n 值 …… 49

试读结束：需要全本请在线购买： www.ertongbook.com

1.8.3 曲线的双步生成算法 …… 50
第2章 图形裁剪 …… 60
2.1 矩形窗口的裁剪算法 …… 60
2.1.1 矩形窗口的直线裁剪 …… 60
2.1.2 矩形窗口的圆及椭圆裁剪 …… 63
2.1.3 参数曲线裁剪 …… 69
2.2 一般多边形窗口的直线裁剪算法 …… 72
2.2.1 算法概述 …… 73
2.2.2 交点计算 …… 74
2.2.3 直线通过多边形的一个顶点或与其一边重合情况的处理 …… 75
2.2.4 算法实现 …… 76
2.3 圆形和椭圆形窗口裁剪算法 …… 78
2.3.1 圆形窗口的线裁剪 …… 78
2.3.2 椭圆形裁剪窗口 …… 80
2.4 多边形窗口的多边形裁剪算法 …… 80
2.4.1 基本概念与定义 …… 81
2.4.2 新算法的数据结构 …… 81
2.4.3 新算法 …… 84
2.4.4 交点的判断与计算 …… 88
2.4.5 两多边形的边重合或者两多边形在顶点处相交的特殊情况的处理 …… 91
2.4.6 算法比较 …… 92
2.4.7 小结 …… 93
2.5 区域的“交”、“差”、“并”操作 …… 94
第3章 三维图形算法 …… 99
3.1 沿三维直线的体素遍历多步整数算法 …… 99
3.1.1 二维空间中像素的直线遍历算法 …… 100
3.1.2 三维空间中体素的直线遍历 …… 103
3.1.3 算法分析与比较 …… 107
3.2 曲线和曲面的相交线(面)逼近 …… 109
3.2.1 一般曲线的相交折线逼近 …… 109
3.2.2 三维球体表面的逼近表示与数据压缩 …… 112

3.2.3 一般三维物体表面的逼近表示与数据压缩 …… 114
3.3 网格模型的数据压缩 …… 116
3.3.1 表示点的三维坐标的一维化 …… 116
3.3.2 网格模型点表的压缩 …… 117
3.3.3 网格模型面表的压缩 …… 118
3.3.4 实验结果及分析 …… 120
第4章 有关图像显示与识别的几个问题 …… 123
4.1 图像与图形的树表示及搜索 …… 123
4.2 多面体的隐藏线消除 …… 125
4.2.1 解决问题的方法 …… 126
4.2.2 求凸多面体的一个可见面 …… 127
4.2.3 消隐线算法 …… 129
4.3 反走样技术 …… 130
4.3.1 反走样直线算法 …… 131
4.3.2 反走样圆算法 …… 134
4.4 多灰度级图像的二值显示问题 …… 136
4.4.1 误差分散方法及分析 …… 137
4.4.2 误差分散方法的改进 …… 138
4.5 噪声的模拟产生方法 …… 139
4.6 借助曲线生成方法进行曲线识别 …… 140
4.6.1 直线的识别 …… 140
4.6.2 圆及椭圆链码的识别 …… 142
4.7 边界曲线的特征点抽取 …… 146
4.7.1 综合方法的基本原理 …… 146
4.7.2 算法实现 …… 149
第5章 压缩链码 …… 151
5.1 角度差编码的压缩链码 …… 151
5.1.1 角度差编码压缩链码原理 …… 152
5.1.2 角度差编码压缩链码与 Freeman 链码之间的转换 …… 154
5.1.3 与其他链码的比较 …… 156
5.2 组合顶点链码 …… 157

5.2.1 第一种改进的顶点链码 …… 157
5.2.2 第二种改进的顶点链码 …… 158
5.2.3 组合压缩链码 …… 159
5.3 链码的评价与比较 …… 160
5.3.1 一种定量评价链码的方法 …… 160
5.3.2 链码的比较 …… 161
第6章 六角网格及其图形算法 …… 164
6.1 六角网格及其特点 …… 164
6.2 六角网格上的直线生成算法 …… 166
6.3 六角网格上的椭圆生成算法 …… 171
6.4 六角网格上的圆弧生成算法 …… 173
6.5 六角网格上的裁剪算法 …… 175
6.6 六角网格上的图像处理 …… 178
6.6.1 六角网格上的数字化 …… 178
6.6.2 几何失真校正算法 …… 179
6.6.3 轮廓跟踪算法 …… 182
参考文献 …… 184

第 1 章　直线与曲线的生成

直线与曲线是组成图形的基本元素。其生成(或称绘制)算法的优劣对整个图形系统的效率是至关重要的,因为在产生一幅图形的过程中要反复多次调用这些算法。曲线的生成算法可分为线生成和像素级(或称点)生成两类。所谓线生成,就是以小的直线段来逼近和代替实际曲线而显示之;而像素级生成则是逐个像素地选择距离实际曲线最近的点而显示之。前者是随着随机扫描显示器的产生而出现的;后者则是为目前普遍使用的光栅扫描显示器而设计的,具有精度高、计算量小等特点。

1.1　圆及椭圆的多边形逼近及线式生成[1,2]

当圆的内接多边形边数足够多时,该多边形可以和圆接近到任意程度。因此在允许的误差范围内(可用圆周与多边形之间的最大距离来度量),可以用显示多边形代替显示圆。

设要绘制的圆的圆心为 $P_c(x_c,y_c)$,半径为 R。又设内接正多边形的一个顶点为 $P_i(x_i,y_i)$,内接正多边形两个顶点至圆心连线的夹角为 α(如图 1-1 所示),则

$$x_i = x_c + R\cos i\alpha$$

$$y_i = y_c + R\sin i\alpha$$

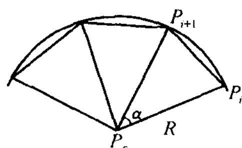

图 1-1　用内接正多边形逼近圆弧

可用递推关系得出下一个顶点 P_{i+1} 的坐标为

$$x_{i+1} = x_c + R\cos(i\alpha+\alpha) = x_c + (x_i - x_c)\cos\alpha - (y_i - y_c)\sin\alpha$$

$$y_{i+1} = y_c + R\sin(i\alpha+\alpha) = y_c + (x_i - x_c)\sin\alpha + (y_i - y_c)\cos\alpha$$

可用矩阵形式表示为

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} x_i - x_c \\ y_i - y_c \end{bmatrix} + \begin{bmatrix} x_c \\ y_c \end{bmatrix}$$

上式又可进一步简化成

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = 2\cos\alpha \begin{bmatrix} x_i \\ y_i \end{bmatrix} - \begin{bmatrix} x_{i-1} \\ y_{i-1} \end{bmatrix}$$

该式就是计算圆的内接多边形的各顶点的递推公式。因为 α 是常数,$\cos\alpha$ 只需在开始时算一次。这样,算一个顶点只需做两次乘法。

以上是众所周知的结果。这里要指出的是,如果用圆的相交正多边形代替内

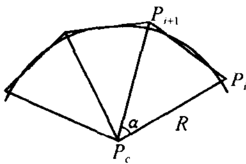

图 1-2　用与圆相交正多边形逼近圆弧

接正多边形逼近圆(如图 1-2 所示),会产生更好的效果。

比较图 1-1 和图 1-2 可以看出:在多边形的顶点数相同的前提下,用相交多边形逼近圆比用内接多边形逼近圆的误差更小,或者说逼近程度更高。而在算法具体实现时,只需将圆的半径 R 增加一个小的增量 x,其他没有任何变化。这相当于对半径为 $R+x$ 的圆执行内接多边形算法。下面我们将看到,恰当地选择这个增量可使多边形最佳逼近圆弧。

设图 1-2 的圆相交多边形中各直线段端点(即多边形的顶点)距圆弧的法向距离为 x。下面计算 x 为多大时才能使多边形最大限度地逼近圆弧,即最佳逼近。

一般来说,最佳逼近有两种衡量方法。一种是使多边形与圆弧所夹的面积最小;另一种是使多边形与圆弧之间的法向最远点的距离最近。下面先计算在面积逼近意义下最佳的 x 值。

由于一个 α 角扇区是对称的,所以只需计算 $\alpha/2$ 角扇区部分的多边形与圆弧所围成的面积。为了能得出 x 的直观的、简洁的表达式,将这一段圆弧用直线代替(如图 1-3 所示)。图中还为各线段设了标记,以便计算。

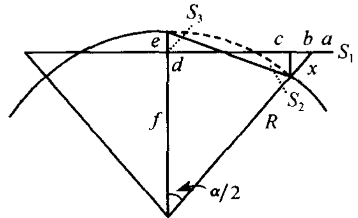

图 1-3　最佳 x 值的计算

从图 1-3 可知

$$f=(R+x)\cos\frac{\alpha}{2}$$

$$e=R-f=R-(R+x)\cos\frac{\alpha}{2}$$

$$d=e\cdot\tan\frac{\pi-\frac{\alpha}{2}}{2}=\left[R-(R+x)\cos\frac{\alpha}{2}\right]\cot\frac{\alpha}{4}$$

$$a=x\cdot\sin\frac{\alpha}{2}$$

$$b=x\cdot\cos\frac{\alpha}{2}$$

$$\begin{aligned}c&=(R+x)\sin\frac{\alpha}{2}-a-d\\&=R\sin\frac{\alpha}{2}-\left[R-(x+R)\cos\frac{\alpha}{2}\right]\cot\frac{\alpha}{4}\end{aligned}$$

多边形与圆弧所夹面积为

$$S=S_1+S_2+S_3=\frac{1}{2}ab+\frac{1}{2}bc+\frac{1}{2}de$$
$$=\frac{x^2}{4}\sin\alpha+x^2\cos^2\frac{\alpha}{2}\cot\frac{\alpha}{4}+\frac{R}{4}x\sin\alpha-\frac{3R}{2}x\cos\frac{\alpha}{2}\cot\frac{\alpha}{4}$$
$$+\frac{3R}{2}x\cos^2\frac{\alpha}{2}\cot\frac{\alpha}{4}+\frac{1}{2}R^2\cot\frac{\alpha}{4}-R^2\cos\frac{\alpha}{2}$$
$$\times\cot\frac{\alpha}{4}+\frac{R^2}{2}\cos^2\frac{\alpha}{2}\cot\frac{\alpha}{4}$$

将上式对 x 求导

$$\frac{\mathrm{d}S}{\mathrm{d}x}=\frac{1}{2}x\sin\alpha+2x\cos^2\frac{\alpha}{2}\cot\frac{\alpha}{4}+\frac{R}{4}\sin\alpha$$
$$-\frac{3R}{2}\cos\frac{\alpha}{2}\cot\frac{\alpha}{4}+\frac{3R}{2}\cos^2\frac{\alpha}{2}\cot\frac{\alpha}{4}$$

令上式为零，求出 x 为

$$x=-\frac{\frac{R}{4}\sin\alpha-\frac{3R}{2}\cos\frac{\alpha}{2}\cot\frac{\alpha}{4}+\frac{3R}{2}\cos^2\frac{\alpha}{2}\cot\frac{\alpha}{4}}{\frac{1}{2}\sin\alpha+2\cos^2\frac{\alpha}{2}\cot\frac{\alpha}{4}}$$
$$=\frac{1-\cos\frac{\alpha}{2}}{1+\cos\frac{\alpha}{2}}R$$

x 的这个值使多边形与圆弧所夹面积最小。

下面计算使算法在点逼近意义下最佳的 x 值。

由于圆相交多边形与圆弧之间的法向最远点距离为 x 和 e 的最大者，而从图 1-3 可见，x 与 e 互补，即如果 x 增大，则 e 必然减小，反之亦然。这样只有当 x 等于 e 时，$\max(x,e)$ 才会最小。因此，令 $x=e=R-(R+x)\cos\alpha/2$，求出的 x 值就是使算法在点逼近意义下最佳的 x 值，即

$$x=\frac{1-\cos\frac{\alpha}{2}}{1+\cos\frac{\alpha}{2}}R$$

幸运的是，该值刚好与面积最佳逼近意义下求出的 x 值完全相同，这更证实了它是使多边形最佳逼近圆弧的唯一的取值。

综上所述，在用线生成算法产生半径为 R 的圆弧时，将 R 增大到

$$R+x=R+\frac{1-\cos\frac{\alpha}{2}}{1+\cos\frac{\alpha}{2}}R=\frac{2}{1+\cos\frac{\alpha}{2}}R$$

试读结束：需要全本请在线购买： www.ertongbook.com

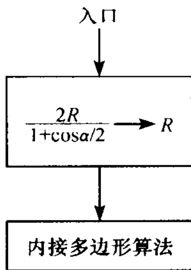

图 1-4　相交多边形逼近算法的执行过程

然后用传统的内接多边形算法生成的圆弧将比不增大 R 的内接多边形算法更逼近参照圆弧，而且是最佳逼近。由此得出本算法的执行过程如图 1-4 所示。

下面我们来分析算法的逼近误差，在内接多边形算法中，参照圆弧与生成的多边形之间的最大误差（即最近点距离）是

$$e' = \left(1 - \cos\frac{\alpha}{2}\right)R$$

而由上面的计算可知，新算法的最大误差是

$$e = \frac{1 - \cos\frac{\alpha}{2}}{1 + \cos\frac{\alpha}{2}}R$$

二者之比为

$$\frac{e}{e'} = \frac{1}{1 + \cos\frac{\alpha}{2}}$$

由于 α 值在实用中是很小的，因此，这个比值约为 1/2。这表明相交多边形逼近算法的最大误差比内接多边形算法几乎缩小了一半。

相交多边形方法同样适用于对椭圆弧的逼近。设椭圆弧的中心为 $P_c(x_c, y_c)$，长半轴为 A，短半轴为 B，长轴与 x 轴的夹角为 φ，则做变换

$$\bar{x} - x_c = (x - x_c)\cos\varphi + (y + y_c)\sin\varphi$$
$$\bar{y} - y_c = -(x - x_c)\sin\varphi + (y - y_c)\cos\varphi$$

椭圆方程便表示为

$$\frac{(\bar{x} - x_c)^2}{A^2} + \frac{(\bar{y} - y_c)^2}{B^2} = 1$$

再做变换

$$x' - x_c = \sqrt{\frac{B}{A}}(\bar{x} - x_c)$$

$$y' - y_c = \sqrt{\frac{A}{B}}(\bar{y} - y_c)$$

则椭圆弧变成圆弧

$$(x' - x_c)^2 + (y' - y_c)^2 = AB$$

由于这两个变换式的系数行列式的值均为 1，因此能保持区域的面积不变。这样在面积最佳逼近的意义下，对变成的这个圆弧的最佳逼近就是对原椭圆弧的最佳逼近。

现在，圆弧的半径是 $\sqrt{AB}$，根据前述的圆弧逼近方法，要对圆弧进行最佳逼

近，应将这个半径增大到

$$\frac{2\sqrt{AB}}{1+\cos\frac{\alpha}{2}}$$

上式可变换成

$$\sqrt{\frac{4AB}{\left(1+\cos\frac{\alpha}{2}\right)^2}}=\sqrt{\frac{2A}{1+\cos\frac{\alpha}{2}}\cdot\frac{2B}{1+\cos\frac{\alpha}{2}}}$$

可见，对椭圆弧进行最佳逼近，应先将其长、短半径 A 和 B 分别增大到 $\frac{2A}{1+\cos\frac{\alpha}{2}}$ 和 $\frac{2B}{1+\cos\frac{\alpha}{2}}$，然后用传统的内接多边形算法可生成最佳逼近的椭圆弧。此过程与圆弧逼近非常统一。

我们知道，对曲线进行逼近时，逼近误差与算法的执行速度是一对矛盾的因素。也就是说，要使逼近误差小一些，就要增加取点的频度，即多取点。这无疑会因计算较多的点而增加计算量，从而降低算法的执行速度。而上述的相交多边形逼近算法在没有增加取点频度的前提下却减少了近一半的逼近误差。那么，换言之，用该方法生成的圆和椭圆弧与用传统的内接多边形算法生成的圆弧或椭圆弧逼近误差相同时，该算法会因计算的点少而比内接多边形算法的执行速度快。

从理论上来讲，用这种与曲线相交的直线段来逼近曲线的思想同样也可应用于其他曲线的线生成及逼近。这里关键是如何确定增量 x，以使直线段与曲线距离最近或所夹面积最小。

对于一般曲线的线生成，已经有一些较有效的算法，如我国学者提出的正负法[3]和 T-N 方法[4]等。最简单的一种生成曲线的方法就是，对某一个变量（x，y 坐标或参数 t）均匀地取一些点而计算其他变量的值，以求出曲线上的点，然后将这些点连接就是曲线的线生成。一种可供选择的取点方法是：在曲线弯曲程度大的地方取点的密度也大；而在弯曲程度小的部分则可使取点间隔大一些。具体实现时是使取点的间隔与曲线的曲率成反比来计算。

1.2　直线的像素级生成算法

我们目前普遍使用的显示器是光栅扫描显示器。其显示屏是由许多被称为像素的点组成的。这些像素是以水平方向和垂直方向成直线排列的。或者说显示屏是由一些水平线和垂直线构成的网格形成的，每个网格点（垂直线与水平线的交点）就是一个像素。在显示器上所显示的曲线图形或图像就是通过这些像素的“亮”与“不亮”（或不同的颜色）的各种组合而形成的。基于光栅显示的这一特点，

我们要在其上绘制曲线(或图形)就要与像素打交道。例如我们要生成一条曲线，最好是逐点地选择那些距离曲线最近的像素并将其“点亮”，这就是像素级(或称点)的生成方法。这种方法的优点之一是生成的曲线误差小，精度高，所显示的像素与实际曲线之间的距离一般不大于二分之一个像素单位。另一个优点就是下面我们将看到的，其算法一般只使用整数运算(至少在主循环内只使用整数运算)，因而执行速度很快。

1.2.1 Bresenham 直线生成算法

说到直线的生成算法，最著名的就是 Bresenharm 算法[5]。设要生成的直线的两个端点坐标为 $P_1(x_1,y_1)$ 和 $P_2(x_2,y_2)$，并令 $\mathrm{d}x=x_2-x_1$；$\mathrm{d}y=y_2-y_1$。现以 $0\leqslant \mathrm{d}y/\mathrm{d}x\leqslant 1$ 的情况为例简要说明 Bresenham 算法的基本原理。

对于斜率在这个范围内的直线，x 坐标第一步增加 1，即向右或右上方走；而 y 坐标就应增加 $\mathrm{d}y/\mathrm{d}x$(如图 1-5 所示)。至于是向右走到像素(x_1+1,y_1)还是向右上走到像素(x_1+1,y_1+1)则取决于这两个像素中哪一个距离实际直线近。我们以这两点连线的中点$(x_1+1,y_1+1/2)$为判断标准。如果实际直线与垂直网格线 $x=x_1+1$ 的交点$(x_1+1,y_1+\mathrm{d}y/\mathrm{d}x)$在上述中点之下，即 $\mathrm{d}y/\mathrm{d}x<1/2$，则下一步应走到右邻像素上，并且下一步加 $\mathrm{d}y/\mathrm{d}x$(即 $2\mathrm{d}y/\mathrm{d}x$)并与 1/2 进行比较以决定再下一步的走向；否则，如果 $\mathrm{d}y/\mathrm{d}x\geqslant 1/2$，就走到右上邻像素上，而下一步就应该用 $2\mathrm{d}y/\mathrm{d}x$ 与 $1+1/2$ 进行比较以决定再下一步的方向。这相当于用 $2\mathrm{d}y/\mathrm{d}x-1$ 与 1/2 进行比较。

现在我们可以总结出直线生成过程的规律为:用 e 对 $\mathrm{d}y/\mathrm{d}x$ 进行累加，其初值为 $\mathrm{d}y/\mathrm{d}x$。直线如图 1-5 每向右走一步 e 就增加一个 $\mathrm{d}y/\mathrm{d}x$ 值，写成递推关系是 $e_{i+1}=e_i+\mathrm{d}y/\mathrm{d}x$。另外，如果是走向右上邻像素；则 e 还要减 1，即 $e_{i+1}=e_1+\mathrm{d}y/\mathrm{d}x-1$。之后，用 e 与 1/2 进行比较。如果 $e\geqslant 1/2$，则下一步走到右上邻像素；否则下一步走到右邻像素。如此反复进行，这就是 Bresenham 算法的基本思想。

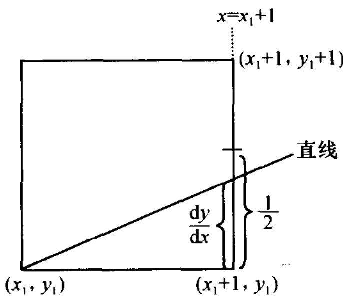

图 1-5 Bresenham 算法

在算法的具体实现时，我们将 e 与零进行比较。这样只需将 e 的初值减 1/2 变为 $\mathrm{d}y/\mathrm{d}x-1/2$。另外，一般来说 $\mathrm{d}y/\mathrm{d}x$ 是一个实数值。为了只使用整数运算，可将 e 放大 $2\mathrm{d}x$ 倍(由于只利用 e 值的正负性，所以放大若干倍并不影响其符号)。这样，e 的初值就应该是 $2\mathrm{d}y-2\mathrm{d}x$。当 $e<0$ 时，下一步应向右邻像素移动，然后修改 e 值，$e=e+2\mathrm{d}y$；否则，下一步向右上邻像素移动，更新 e 值，$e=e+2(\mathrm{d}y-\mathrm{d}x)$。下面是完整的 Bresenham 直线生成算法的形

式描述。

```
dx = abs(x2 - x1);
dy = abs(y2 - y1);
if x2>x1 then sx = 1 else sx = -1;
if y2>y1 then sy = 1 else sy = -1;
x = x1;
y = y1;
if dx>dy then      {t1 = 2 * dy;
                    t2 = 2 * (dy - dx);
                    e = 2 * dy - dx;
                    for i = 1 to dx{
                               Plot(x,y);/ * 画点 * /
                               x = x + sx;
                               if e> = 0 then {y = y + sy;
                                               e = e + t2
                                              }
                               else e = e + t1
                               }
                    }
else {
     t1 = 2 * dx;
     t2 = 2 * (dx - dy);
     e = 2 * dx - dy;
     for i = 1 to dy{
                   y = y + sy;
                   if e> = 0 then {
                                  x = x + sx;
                                  e = e + t2
                                  }
                   else e = e + t1
                   }
     }
```

Bresemham 算法可以说是生成直线的最有效的方法。它只使用整数运算，并且从上面的形式描述可以看到：算法每生成一点，只需要改变 e 值的一次加法运算（而画点及改变 x 和 y 坐标值的语句在任何算法中都是一样的）。

尽管如此，仍可通过对 Bresennham 算法的灵活运用而加快直线生成的速度。因为直线是最基本的图形元素，所以在图形软件中直线生成的速度将极大地影响着整个软件的效率。从这一点来看，直线生成的任何一个微小的进步都是很有意义的。

另外，Bresenham 直线算法在生成直线时，从一端开始生成的直线与从另一端开始生成的直线有可能在个别点上不同（实际直线通过光栅网格线的中点时，即 $e=1/2$ 时，会出现这种情况）。这样在一些应用场合就要出现问题，例如在图形软件包中擦掉一个图形往往是从反方向进行的，这样就可能剩下一些点擦不掉。Thong 对 Bresenham 算法进行了改进，可以解决上述问题[6]。

1.2.2 单点直线生成算法已无优化的余地

每次循环生成直线上一点的算法我们称为单点直线生成算法。本节我们要说明，作为单点直线生成算法，Bresenham 算法已使用了最小的计算量。或者说，Bresenham 直线生成算法的计算量已无再减少的余地。

直线生成算法的计算量主要体现在主循环中。由下面 Bresenham 算法的主循环语句可见，每生成一点所需的计算量是：

（1）循环语句所需的一次结束判断。对于一个循环语句来说，一次结束判断是必不可少的，否则它将无法结束。而一个通用的直线生成算法不可能没有循环语句。

（2）对 x 坐标的一次加法和对 y 坐标的 $\mathrm{d}y/\mathrm{d}x$ 次加法（整条直线共需对 y 坐标进行 $\mathrm{d}y$ 次加 1 操作，由于共有 $\mathrm{d}x$ 个点，因此每点平均是 $\mathrm{d}y/\mathrm{d}x$ 次加法）。改变坐标的这 $1+\mathrm{d}y/\mathrm{d}x$ 次加法对任何直线算法都是一样的，因为它就是跟踪直线上点的轨迹的操作。每一次加法都是与直线上的点是一一对应的，少任何一次加法都会少画一点。因此我们在比较算法时，不考虑这 $1+\mathrm{d}y/\mathrm{d}x$ 次计算。

（3）对于判断变量 e 的一次加法和一次比较操作。这一次加法和一次减法操作都是必不可少的。如果没有 $e>0$ 的比较操作（条件语句），则对坐标值的增加只能有一种模式，因此这样的算法只能生成一条与这种模式对应的固定直线（如水平直线或 45°角直线等）而不能作为一般的直线算法产生任意的直线。而如果没有对 e 的加法操作来不断更新 e 的值，那么 e 就是一个常值，因此它对 0 的比较也是常量，这同样是上面的结果。而对于一个值的更新而言，除了赋值操作之外，一次加法操作是最小的计算量。而要对 e 进行赋值更新，赋值号右边只有 x 或 y 坐标变量可供选择（否则又需要额外的计算）。如果这样，x（或 y）坐标变量与 e 的值则是相同的。由于 x（或 y）变量是单调变化的，所以 e 与一个常量的比较结果只能有一次变化。对于一个直线生成算法来说，e 值比较结果的一次变化是不可能产生任意直线的。另外这些运算都是整数运算，且其他运算（如乘、除法）的计算量都不小于加法和比较操作（减法与加法和比较的计算量是同一数量级的）。因此对于单

点直线生成算法来说，Bresenham 算法已经使用了最少的计算量，即 $2+dy/dx$ 次加法和 2 次比较操作。

1.2.3　一个双点 Bresenham 直线生成算法

前面介绍了 Bresenham 直线生成算法，它使用一个判断参量 e 来选择下一个像素的方向。在主循环中的每一步，都要计算更新 e 的值，并判断 e 值的正负，以便确定离直线最近的下一个像素。如此反复，便生成一条最接近于实际直线（理想直线）的直线。可见，Bresenham 算法每生成一点的工作量是判断 e 的正负和计算更新 e 的值。下面要用直线链码理论，对 Bresenham 算法进行改进。直线链码理论与直线绘制相结合，可使我们得到如下有用的提示：用 Bresenham 算法绘制直线时，并不必每生成一点，都要进行上述的工作。进一步来说，在生成直线上的某些点之后，我们便可知道其下一点的位置。这时，既不用计算更新 e 的值，也不用判断 e 的符号，就可生成下一点。因此，在算法主循环中的某些步中，可以连续地生成两点。

Freeman[7] 出于模式识别的目的，提出了用图 1-6 所示的不同斜率的 8 个小直线段作为基元，来描述线画图形，即 Freeman 链码，图中圆点可认为是光栅点或像素中心，并指出一条直线的链码应满足 3 个条件：①最多只有两个相邻的基元（码）出现；②上述两个码的其中之一只能单个出现（不会连续出现）；③这个单独出现的码一定是均匀分布在整个直线链码中。

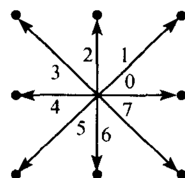

图 1-6　Freeman 链码

实际上，在光栅显示器上绘制直线的过程，也可以理解为对直线编码的过程。从直线上的一个点移到下一个点的过程，一定对应着 Freeman 链码的 8 个基元之一，这是因为每个光栅点（像素）的 8 个相邻光栅点的方向及距离，恰恰与 Freeman 链码的 8 个基元的方向及长度相一致。因此，我们可以利用上述的直线链码应满足的条件，来研究直线的绘制。由条件①和②可知：在绘制一条直线时，也只需进行两个方向的移动，其中一个方向一定是单独出现的。或者说，这个方向出现时，紧随其后的一定是另一个方向。这样当直线绘制算法通过单独出现的方向来到一个像素点时，则下一个像素点的方向就是已知的了（即另一个方向）。例如有一直线，其端点为(1,1)和(10,4)。它在屏幕上的显示如图 1-7 所示。如果用 Freeman 链码对其进行编码应为 010010010，可见只有 0 和 1 两个基元出现，而基元 1 是单独出现的。对于该直线的绘制，算法的每一步要判断下一个点是向方向 0 移动（即 x 坐标加 1）还是向方向 1 移动（x 和 y 坐标均加 1）。但是当算法判断出是向方向 1（单独出现的方向）移动时，则再下一个移动方向就

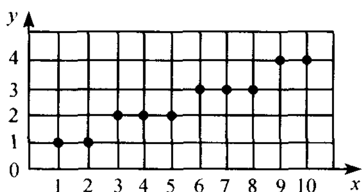

图 1-7　一条直线的例子

试读结束：需要全本请在线购买：　www.ertongbook.com