IBM PC 系列机
图形程序设计方法

龚治源　　译
苏鸿根　　审校

北京科海培训中心

图形程序设计方法

(美) L.Ammeral 著

龚治源　译

苏鸿根　审校

前　言

如果你是一位C语言程序员，或者刚学完C语言教程后打算开发图形应用，我们推荐你阅读一下本书，你将能从中得到极大的收获和提高。

本书作者在使用C语言编制图形程序方面具有丰富的经验。通过大量的实例分析，作者由浅入深地讲述了各种图形程序设计方法，特别强调怎样解决编程中的难点或内部要害问题，例如图形程序与图形适配器，矩阵打印机和鼠标器之间的衔接，弥补C语言本身的不足之处，提高绘图速度的技巧等。文笔流畅，内容精练，是本书的又一优点。

本书末尾附有一个完整的交互式绘图程序包，列出了详细的源程序及其注释，这在以前有关CAD软件资料中是十分罕见的。而且你只要有一台配有图形适配器的IBMPC／XT，AT或其兼容机，就能很容易地实现，绘制出令人满意的图形。

1990年11月

目　录

第一章 绪 言

§1—1 本书的内容和目的

本书译自美国L·Ammeral 著《Graphics Programming for IBM-PC／XT Programmer》一书。在写这本书之前，作者一直在小型电子计算机上工作，1986年曾出版了两本书，一本是《Programming Principles in Computer Graphics》（计算机图形程序设计原理）；一本是《C for Programmers》（C语言程序员手册）。作者用C语言在小型电子计算机上编制图形程序很有经验，在每本书中，作者都列出了很多源程序，便于读者练习和运用。

在小型电子计算机上编制图形程序时，作者经常使用下面四条基本子程序：

initgr ( ) 初始化图形输出；

move (x, y) 将笔（真实或虚拟的）移到点 (x, y)；

draw (x、y) 从笔当前所在位置到点 (x, y) 画一条线段；

endgr ( ) 执行最后的动作。

因为这四个基本子程序不依赖硬件，在小型电子计算机上建立DIPlot绘图库很容易，用Fortran语言也可以实现。由于不依赖硬件，用这4个基本子程序便可以直观地建立两个软件层，高层是子程序块，低层则当工具使用。只要事先编好了这些子程序，许多实际工作可以交给子程序去做，而程序员不必了解这些子程序内部如何工作。

作者写本书的目的，是专门为使用IBM—PC／XT 微型电子计算机的程序员，使他们很容易地在微机上建立一套绘图子程序库。为了使读者易于领会和掌握，尽量将源程序列出。鉴于以前用C语言编写的图形程序都不大好用，作者根据自己的经验，用这四个基本子程序重新建立了绘图子程序库，这对于程序员来说，是建立低层图形程序的很好的练习。为了避免混乱，本书只讲低层程序设计，不讲高层程序块。

第一、二章的内容是用四个基本子程序建立一个微型软件包，从基本操作、基本定义到画点、画直线逐渐形成。

第三、四章讲解“光栅图形”，在萤光屏上设计一枝软件“笔”，画出的线段可被删除掉。输出用矩阵打印机。如果你现在使用的就是矩阵打印机，则不用“笔”也可以打印出绘图结果的硬拷贝。

第五章讲述将若干绘图结果拼合起来的程序编写方法，并且告诉你如何将特殊字符加入的方法，例如整数符号等。

第六章将前面几章的结果发展成一个不算复杂的绘图程序系统，并将全部源程序列在后面。由于这个程序系统对硬件没有特殊要求，在IBM—PC／XT 机上实现不会遇到困难，读者不妨试试。

如果你打算从本书中学到高深的图形研究结果，那是会失望的。羡慕别人的成果是一件快事，但总不如自己的成果给你带来的欢乐多。自己动手吧，你会从中得到乐趣的。

§1—2 C语言程序员应注意的几个问题。

本书假定读者已经熟悉C语言程序设计，工作环境是IBM—PC或其兼容机，本书中出现的C语言程序是在lattice C 3.0版的编译器支持下工作的。如果使用别的编译器，则必须使你的编译器与源程序相适配。本书下面几节将提供有关资料。

§1—2—1 无符号字符类型

在图形程序设计中经常用到

unsigned Char（无符号字符）。

在lattice C语言中，Char（字符）之前的关键字unsigned是字符的最左边一位(bit)。即是说，在编译时，保留字符最左边一位为符号位，在一个字符转换成int（整型）时，符号位放在最左边。例如：在

```
Unsigned char  K = 0xC0;
        /* In binary = 1100  0000 */
```

之后，右边操作的结果

$$K \gg 4$$

将是0x 0C(=0000 1100，2进制），在大多数应用中，这正是我们想要的结果。但是如果K是Char（字符）型，而不是unsigned Char（无符号字符）型，则“符号位”1就用来将1100 0000扩展到

1111 1111 1100，0000

这就是说，现在右边8位bit的值是0xFC。如果这不是我们所要的结果，则关键字 unsigned就是一个很好的补救措施，因为它使1100 0000 扩展为0000 0000 1100 0000，在右移后则产生正确值0000 0000 0000 1100。

§1—2—2 从键盘直接输入

如果要想从键盘输入，正常情况下是用文件指针stdin作“标准输入”。例如：

SCanf(“%d” &n)等价于fscanf(stdin，“%d”，&n)，并且

getchar( )等价于getc(stdin)。

这些函数要用缓存，即是说，我们打入的字符只有打了回车键后才起作用，这样，我们就可以用回车键去修改错误。有时我们想让打进去的字符只用一会儿，能否不打回车键？在lattice C语言中有两个函数可达此目的：

getch ( ) 从键盘上打入一个字符，不作响应；

getche ( ) 从键盘上打入一个字符，要作响应。

这里说的“响应”（echo）的意思是指将打入的字符在萤光屏上显示出来。用getche和getchar得到的结果是一样的。有时会看到机器处于图形工作方式（graphic mode），此时输入的字符就不是你所希望的，在这种情况下用getch较合适。

还有一个很有用的非标准函数：

kbhit ( ) 检验键盘是否击入，

与正规的输入函数不一样，kbhit不等打入其它键就将值转为0；如果打入了其它键，转为1。

当一个字符已经输入，Kbhit则跳过刚输入的字符，换句话说，字符仍然可以用正规方法去读。在打入一个键以后，函数kbhit可以用来终止一个循环。例如在下列程序中：

```
main ( )
{int n=0; double x=1.0;
  while (1)
    {X*=1.000000001; n++;
      if(Kbhit( ))break;
    }
    printf( "n=%d x=%" ,n, X) ;
}
```

还有一个函数ungetch，类似于标准输入输出函数ungetc，当调用getch或getche将一个字符压入堆栈以后要用到，下次再调用getch或getche时还要用这个函数。堆栈的深度只有一层，因此不能压第二个字符进去。在下面这个例子中：

```
Ch1=getch( ); ungetch(ch1);
Ch2=getch( );
```

从键盘上读只有一个字符有效，且这个字符分别赋予Ch1和Ch2。在第§1—2—4中要用到ungetch。

§1—2—3 存储模式及其存取

8088/8086采用段地址技术，每个地址由两个16位bit的二进制数组成，一个是段，一个是偏移量。段向左移4位，右边就空出4个0，然后就把偏移量加上去，用这种方法可以获得20位bit的地址，从而满足了一兆字节地址存储空间的需要。下面是四个“段寄存器”的名称及其作用：

CS 代码段
DS 数据段
SS 堆栈段
ES 附加段

只要程序不超过64k字节，就可能在代码段CS中保持段常数，变动的仅仅是偏移量。与此类似，只要数据区没有超过64k字节，数据寄存器DS才能保持常数。用这种方法，只需16位地址就可以为指令和数据存取服务。这就形成了比20位地址更有效的代码，在大量的应用中，有64k的程序区和64k的数据区也就够用了。我们把这种方法叫做S模式（S即Small，小的意思）。除S模式外，如果程序存储量超过64k字节可以用P模式，如果数据超过64k字节可以用D模式，如果程序和数据都超过64k字节可用L模式（L即large，大的意思）。存储模式确定后，就向编译说明，每个存储模式有自己的库结构，在编译和链接时，必须与有关的存贮模式保持一致。S是缺省模式，即在没有说明时就按S模式进行编译。

上面讨论的是面向机器的，如果不是处理图形问题可以不予考虑。在下一章要讨论直接存取图形适配器，就象正规程序和数据存入内存一样，起始地址是0xB80000。注意，这是个20位的地址，用16进制表示。由于图形适配器要求存储容量大，要用到“屏幕存储区(Screen memory)”这个术语。如果使用C或D模式，这个术语是很好的。幸好lattice C语

言中有两个函数可以在存储器中移动数据，在这样的存储区要用到下面两个相关的函数：

```
Poke (Segment, offset, source, nbytes)
Peek (Segment, offset, destination, nbytes)
```

头一个变量用的是16位数0xB800，就象前面说过的那样，可以在右边扩展出4个0。前两个变量是整型（无符号）。第二个变量是偏移量，用来扩展第一个变量。第三个变量是正式指向字符的指针，可以用一个字符数组来命名。第四个变量也是整型，用来说明拷贝多少字节。

§1—2—4 控制台中断

在程序运行中，有时要用Ctrl—Break去中断程序的运行。操作方法是首先按下Ctrl键不放，再按下Break键。如果机器上没有Break键，用Ctrl C也可以使程序运行中断。我们把这种中断叫做控制台中断。这里有两个问题，尤其在绘图程序中要很好解决。

第一个问题是，当按下Break键后，机器可能不中断，而操作系统只有在某种条件下才去检查控制台中断。即是说，只有事先设置了中断标识符，操作系统才能按照“服务请求”或“控制台中断请求”去检查。如果仅仅是执行计算任务，用下面这个循环语句就够了：

```
for (i=0; i<30000; i++)   S+=1+2* (i/2);
```

但因实际上没有这种请求，程序将拒绝执行控制台中断。作者遇到过比计算更有趣的问题，然后着手去寻找一种无害的服务请求（一种更可取的控制台服务请求），我们可将它插入循环体中，让操作系统去执行我们希望的检查。首先试验了一次简单调用kbhit（参看§1—2—2），结果证明在大多数情况下是令人满意的。当操作时多打入了一些键，在发生控制台中断前是不起作用的。有时要输入某些字符，不仅打入字母就够了，还要打回车键才起作用。作者因此用下列函数来扩展了这个功能：

```
Checkbreak ( )
{ Char ch;
  if (kbhit ( ) ) { Ch=getch ( ); kbhit ( ); ungetch (ch); }
}
```

如果Checkbreak是在循环体内部调用，程序就会执行控制台中断命令。

第二个问题，当机器执行我们的控制台中断命令时，在执行过程中和以后会发生什么事情？如果对自己的动作不加任何说明，则控制台中断即将激活缺省中断处理程序，即简单地停止程序的执行。在大多数程序中，也正是操作员希望的，但图形程序还需要作其它动作，在§2—9节中要讲这个问题。一般情况下我们可以“生成”一个中断陷阱，即写一个特殊函数，说明在发生控制台中断时应该做什么。这种函数的地址，简单地写成它的名字，作为函数onbreak的一个变量，即lattice C中的一个变量。如果这个函数的值是0，则在中断点恢复执行程序。否则程序立即失败。函数onbreak也可以给一个空指针作变量；也可以写成0。在这种情况下，当控制台中断发生时，缺省中断处理器会起作用。如果onbreak函数的变量不是0而是一个函数，则在使用之前必须说明，否则编译器会误认为是一个简单的整变量。这里有一个例证列在§1—2—2中，这个程序就是基于这种情况的。我们现在是用一个正规的控制台中断，而不是打任意键。

```
/*BREAKDEMO.C: Console-break demonstration*/
#include "dos.h"
int n=0, my_function ( ) ;
double  x=1.0;
main ( )
{onbreak (my_function) ; /* Replace default interrupt handler */
   while (n<30000)   /*  with my_function.                          */
   {X *=1.000000001; n++;
    checkberak ( ) ;
   }
    onbreak (0) ;
    /* Any program text inserted here, when interrupted, would */
    invoke the default interrupt handler.
    printf ( "Normal program end, n=30000  x=%f" ,  x) ;
}
int my  function ( )
{printf ( "n=%d      x=%f" ,n,x) ;
 exit (0) ;
}
 checkbreak ( )
{char  ch;
 if (kbhit ( ) )   {ch=getch ( ) ; kbhit ( ) ; ungetch (ch) ; }
}
```

注意，这里的my－function调用标准函数exit，调用后不返回主程序，因此这里不需写一个返回语句说明是否要返回并恢复执行主程序，如前面提到过的那样。

最后，还要注意到IBM—PC机上的Ctrl Breack和Ctrl C有点微妙的区别。如果在运行BREACKDEMO·C程序（中断检查）时多打了一个键，Ctrl Break照样起作用，而Ctrl·C无效。如果在控制台中断前多打了任意键，或在打入中断命令时没有多打任意键则Ctrl Break和Ctrl C都起作用。

§1—2—5　8088 I／O端口寻址

在最低级输入输出能用程序来实现，这是基于机器有初级机器指令IN（输入）和OUT（输出）。这些是为汇编语言用的（甚至在高层的I／O设备也有这两条指令，下一节要谈）。指令IN从输入接口读数据，指令OUT将数据写到输出接口。这些I／O接口都是特有的硬件线路，计算机依靠这些线路与终端通信。所有为了输入输出的高级（high—level）子程序最终都是执行IN和OUT指令，这些指令在lattice C中也有，用在下面两条中：

```
V=inp (P) ;
outp (P, V) ;
```

这里的P和V是整型（无符号），P是端口地址，V是端口值。当使用函数inp和outp时，必须将下列一行

```
#include "dos.h"
```

用在程序中。

若没有正确使用I／O端口直接寻址（或出了错误），可能会引起各种系统问题。因此最

好是使用高级I／O函数。如果在机器上有彩色图形适配器，本书中的软件就不用直接I／O端口寻址。若机器带有大力神卡（Hercules Card）（或兼容的单色图形适配器），就可以用out函数作开关，将函数从文本方式（text mode）转换成图形方式（graphics mode），反之亦然。

§1—2—6 寄存器和软中断

现在来看看计算机不借助于I／O端口直接寻址，如何在高层（high level）与外界通信。有组子程序，调用BASIC输入输出系统，或迳直地调用BIOS。转向高级请求，程序员不须要有很多有关计算机结构方面的知识，在某种程度上说这是事实。但如果要用BIOS程序，就要具备8088（或8086）微处理器通用寄存器的一般知识。这一点很容易明白，因为这些程序是为汇编语言程序调用而设计的，而不是为高级语言（如C语言）程序的调用服务的。不须讨论8088的全部寄存器，这里只限于讨论四种寄存器，如下图：

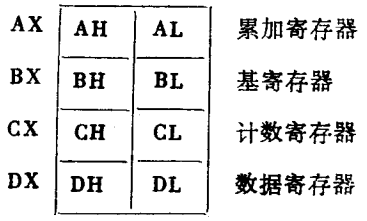

图1.1 8088的数据寄存器

每个寄存器有16位bit，分为高8位字节和低8位字节，每个都可作为8位bit的寄存器来使用。例如FA3B（16进制数）装入DX寄存器后，DH的内容将是FA，DL存放的是3B。

另一个技术问题是调用BIOS。BIOS是不能作为子程序调用的，但可以作为“软件中断”调用，这个术语是根据机器从外界得到一个信号从而使机器中断而引伸出来的。正在运行的程序被中断以后，正规情况下，还可恢复运行。当这种中断发生时，所有寄存器中的内容都被推入堆栈，并从表中找到新的地址，跳到新地址去执行程序，这叫做中断向量，新的程序片称为中断程序。中断程序执行完后，在结束处有“返回中断点”的指令，此时原先压入堆栈的寄存器的内容就被弹出，程序就从中断点处恢复并继续运行。用这种方法可以在程序中设置内部中断，即在程序中写一条特别的中断指令，这种方法叫做软中断。由于这种软中断是由程序员决定在什么地方发生中断，在概念上类似于子程序调用，但其方法类似内部中断处理。在软中断时，子程序中的地址不用写真实地址，只要一个很小的数去作中断标记就行了。将所有软中断与屏幕显示联系起来的数是16，通常写成16进制数的10。在汇编程序中用指令

```
INT  10H
```

作初始化视频显示器软中断。因为中断标识符10H可用于很多目的，还必须设置一些传送参数（Parameter—passing）。这就是为什么在本节中需要图1—1的数据寄存器的原因。用lattice C语言，可以写成“int86(0x10, &regsin, &regout);”，头一个变量是中断号10H，第二、三个变量是对应于寄存器的数据结构的地址，这个地址可用于相同情况下的汇编程序。任何通过程序的信息都必须提供给regsin，任何由程序返回的信息都必须通过regsout。为此必须写

```
#include "dos.h"
```

和说明

```
union REGS regsin, regsout;
```

在文件开头的dos.h中，union REGS的类型由下述三条线的第三条来定义：

```
struct XREG{ short ax, bx, cx, dx, si, di; };
struct HREG{ byte al, ah, bl, bh, cl, ch, dl, dh; };
union REGS{ struct XREG X; struct HRECT h; };
```

在union REGS中，结构x和h占相同的内存量，这正是我们所希望的。例如

```
regsin, x, ax
regsin, h, ah   } 共享内存，
regsin, h, al
```

且以同样方式将两个字节的AX寄存器分为两个一字节的寄存器AH和AL，等。（请不要急于了解第一行中的si和di和第二行中的al和ah，我们在这里不打算详细地讨论8088的细节）。

在中断（10H）以前，必须在寄存器AH中放一个代码，用C语言最好写成：

```
regsin, h, ah = code;
int86 (Ox10, &regsin, &regsout) ;
```

这个代码告诉中断程序做哪几个中断动作，为达此处目的，不需要regsin和regsout两个变量，用一个结构regs就够了，所以写个简单说明如下：

```
union REGS regs;
```

并且用regs代替regsin和regsout。除了AH外，对其他寄存器也必须说明我们的用途，在§2—3中将讨论这个问题。

§1—2—7 堆栈的最大容量

函数的返回地址和自动变量是放在堆栈中的，一个堆栈是一片连续的存贮单元，堆栈的大小有限，可能不够用，尤其在使用递归函数时。lattice C语言提供了一种扩展堆栈的功能，程序员要写一段程序，例如

```
unsigned int_STAK = 15000;
main ( )
{ ...
  ...
}
```

假定堆栈原来限定2048个字节，而我们想扩充到15000个字节，则用上述语句可以实现。当执行程序时，就按扩充后的大小存放。例如，我们的程序调用MYPROG.EXE，就可以将堆栈限制在20000个字节，只要输入命令

```
MYPROG = 20000
```

即可。

如果我们不愿详细计算堆栈的大小，就先规定一个最大的堆栈，如果20000个字节就够了，可以打入一条简单命令

```
MYPROG
```

§1—3 图形适配器

在萤光屏上有很多点，这些点都有两种状态，可以是亮的，也可以是不亮的（或者叫做黑的和白的）。现在有彩色显示器，这里只讲单色显示器。屏幕上显示的黑（或白）的点的区域叫做象素（pixels，有时也写作pels）。点和象素的区别是：象素在屏幕上是一个小方形区域，点在象素中。点越多，图象越清晰。

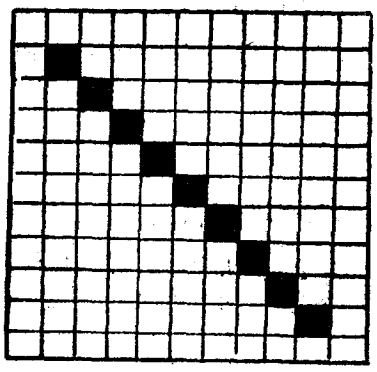

图1—2 象素和点的关系

图形适配器是计算机硬件的一块插件，也叫做图形卡。现在流行三种图形适配器，它们的特性是：

1.单色显示适配器，只用于文本，能显示25行，每行80个字符。

2.单色图形适配器（例如大力神卡），可用于文本，也可以用于图形。绘图时有720×348象素（348行、每行720个点）。

3.彩色图形适配器，既可以用于文本，也用于图形。当用于文本时，显示25×80个字符；绘图时显示640×200个象素（如果除了黑白还有别的颜色，还要少些）。这些适配器中包含的存贮内容叫做地址，用通常的方法存放，只要内存中有相应的数据，既可以显示文本也可以显示图形，但文本的编码与图形编码是完全不一样的。

对文本来说，一个字符的ASCII码用一个字节来存储，后面跟着一个属性字节，其中含有如何显示这个字符的信息（例如字符的下划线）。把这个字符转换成象素型是靠特殊的硬件来执行的，这种硬件叫做字符发生器，这是显示字符很有效的手段。在显示屏上有25×80＝2000个字符位置，每个字符两个字节，共需4000个字节。我们知道每个字符显示到屏幕上是9×14的象素区域（box 9共9个象素14条横线），这个数是很低的，如果每个象素一位bit，共是2000×14×9／8＝31500个字节。4000个字节的地址是B0000,…,B0F9F。使用单显时要记住，我们不能删除字符发生器的动作，这就是为什么单色显示适配器不能用来绘图的原因，单显这个名字就不含绘图的概念。

单色图形适配器是最不单纯的设备（为了不跟上面提到的单色显示适配器相混，常用“大力神卡”这个名字，但只有在适配器用大力神卡的计算机技术时才正确）。首先是单色图形适配器完全使用与单色显示器相同的方法产生字符，可以用单色图形适配器删除字符发生器的动作，并由文本工作方式（text mode）转为图形工作方式(graphics mode)，即所谓位映射方式（bit—mapped mode）。在图形方式时，每个象素对应于存贮器中的一位bit，如果这位bit是1，对应的象素就亮，如果是0，对应的象素就暗。每条横线有720个象素，对应720×8＝90个字节，总共348行，需要348×90＝31320个字节的存储空间，刚好凑

够32k = 32768个字节，用16进制来表示这些地址为B0000，…，B7FFF。此外单色图形适配器还有32k页，地址为B8000，…，BFFF。这两种页都可以用，但当前页只显示一个，如果只用一个则在二者中选一就行了，二种页的起始地址分别为B0000和B8000，页式的值为0和1。用页式1有两个理由，首先是起始地址B8000与图形适配器一致，可用相同现成地址；其次是页式0复盖显示程序的存贮区，而页式1则无此缺陷。若某变量在页式1上，当转到文本工作方式时会失效。在第四章要用这个结果去产生“算后检查”（Post—morten），即图形的屏幕转储检查。

如果要绘制各种颜色的图，只好买彩色图形适配器了。与单色图形适配器一样，彩色绘图适配器可以在文本方式下工作，也可在“位映射”（bit mapped mode）方式下工作。有三个问题值得考虑：

1.彩色图形适配器产生一个8×8的字符块（box），而不象其它两种适配器那样产生9×14的字符块，可读性较好。

2.在文本工作方式下，当屏幕向上滚动时闪烁，令人不舒服。

3.在图形工作方式时，清晰度较低，最多640×200。实际上这种适配器有三种“位映射”方式：

- ●640×200　单色（叫做“高清晰度”）
- ●320×200　4色
- ●160×100　16色

在640×200清晰度时，需要640×200／8 = 16000个字节，大约16k，地址是B8000，…，BBFFF。

如果你想买一个图形适配器或一台完整的PC机，必须在单色还是彩色图形适配器之间选择（或者干脆两个都买），如果主要是绘彩色图，则必须买彩色图形适配器。有一点要注意，不管你买哪种适配器，都要结合你将要使用的软件来考虑。因为我们这本书以IBM—PC为背景，在PC DOS中的BIOS视频显示程序是为彩色图形适配器而不是为单色图形适配器服务的。这就是说，当你在为单色图形适配器编程序时，必须使其在比彩色图形适配器低一级工作。

在单色图形适配器环境下工作，我们必须对“屏幕存储区”（screen memory）和I／O端口进行直接存取操作，而不是借助BIOS例程。这似乎是单色图形适配器的一个缺陷。在彩色图形适配器上，两种方式都可以用，在第二章要谈到，但在这里还是用直接写进屏幕存储区的方法为好。在有些应用中，速度在堆栈，用汇编语言编一些子程序，当屏幕修改时绕过BIOS程序是可能的，这种应用相当慢。因此使用C语言程序同时用汇编语言写的BIOS例程同时工作，在这种情况下直接存取方法不一定比BIOS快。但理应快一些，用C语言的程序员能更清楚地看到下面该做什么，而汇编语言则不行。在下面几章中，还要用汇编语言去重写一些最常用的C函数（叫做dot）以加快存取速度，到那时C语言源程序已经使用直接存取方法，所以编译得很漂亮。

注意，这些低级的和依赖硬设备的问题只涉及很少一部份模块，即包含我们的四条基本图形函数initgr，move，draw和endgr的模块，因此不必操心I／O端口和BIOS例程。

最初作者开发了这种模块的两个版本，用户必须从其中选一个，实际上用机器去选择只有一个能工作，用这种方法写了一个简明的软件，使其可以在IBM—PC及其兼容机上运行，而不管PC机上是彩色还是单色图形适配器。在第二章我们要发展这种一般化的模块。

第二章　直线绘制

§2—1　屏幕和象素坐标

在很多应用中，如果将坐标在纵横方向上都用同一长度单位是很方便的。我们现在要建立一个坐标系，在纵横坐标轴上都以英寸作为单位长度。象普通的数学一样，坐标轴指向右方和上方，坐标原点则放在屏幕左下角，这样做的好处是纵横坐标的最大值立刻就知道了，每个人使用的监视器的屏幕各有自己的准确尺寸，但我们为了方便都用10和7作为最大值。这些数据容易记住，并且与大多数屏幕在两个方向上的尺寸近似。用英寸作为单位长度，如果你的监视器屏幕正好是10×7英寸则与本书程序中用到的数值相吻合。在本书的源程序里，定义了x－max＝10.0，y－max＝7.0，不论哪里出现x－max，y－max，都要符合下列约定：

$0 \leqslant x \leqslant x-max$　　$(x-max=10.0)$

$0 \leqslant y \leqslant y-max$　　$(y-max=7.0)$

这两个坐标是实数，因此写成10.0和7.0，而不写成10和7。

此外还要用到象素坐标，都是整数，用大写字母X和Y表示，其最大值用X--max和Y--max表示。注意这里的X和Y的下划线是两条短线--，以免和前面定义的x－max和y－max相混。同样，在纵横两个方向上，屏幕上的象素坐标取值范围都是：

$0 \leqslant X \leqslant X--max$　(X--max等于719或639)

$0 \leqslant Y \leqslant Y--max$　(Y--max等于347或199)

Y轴的方向指向下，象素坐标的原点在屏幕的左上角，因此将X、Y转换成x、y要用减法，当然不能直接将x写成X或将y写成Y。现在我们用C语言的两个转换函数来进行这种转换，即

```
int IX (x) float x; {return(int) (x * horfact + 0.5); }
int IY (y) float y; {return Y__max - (int) (y * vertfact + 0.5); }
```

上面语句中的horfacf和verfact是两个因子，在initgr函数中由下面两个式子来计算：

```
horfact = X__max/x - max;
verfact = Y__max/y - max;
```

记住，由关键字int组成的造型操作命令（Cast-operator）必须将取整时截取的操作数用括号括起来。截取的值最接近整数值，然后将0.5加到这些非负的整数上去。对于这对庞大的x，y值，要用下列程序来进行检验：

```
IX(0.0) = (int) (0.0 * X__max/x - max + 0.5) = 0
IX(x - max) = (int) (x - max * X__max/x - max + 0.5) = X__max
IY(0.0) = Y__max - (int) (0.0 * Y__max/y - max + 0.5) = Y__max
IY(y - max) = Y__max - (int) (y - max * Y__max/y - max + 0.5) = 0
```

函数IX和IY是非常有用的，因它们将给定的屏幕坐标x和y由高层的用户级转换成低层的X和Y（低级），这正是我们绘图适配器处理所需要的。

函数move和draw如同在《计算机图形程序设计原理》一书中使用那样，假定存在一个“当前笔的位置”，然后将全程变量X1和Y1指向这个位置，X1和Y1也就是象素当前的坐标，当然是整型的，尤其在微型计算机上，一般没有进行浮点运算的硬件，尽可能用整型数是件很重要的事。为了这些变量以及另外一些变量，我们还要使用关键字static。这就是说只有在文件中定义了的变量才能使用。一套函数及其变量定义好了以后就属于一个文件（或一个模块，在§2—10中会看到）。这些函数有对（外部）象X1，Y1这些状态变量的存取，但用户程序没有这种存取，因此它们的值不能丢失或破坏。函数move主要由修改当前笔的位置的语句组成，或者象人们有时说的“当前点”：

```
static int X1, Y1;
move (x, y) double x, y;
{ X1=IX (x) ; Y1=IY (y) ; Check (X1, Y1) ;
}
```

函数Check的定义将在后面给出。读者可能已经猜到Check函数是用来检验由变量X1，Y1给出的点是否在屏幕的边界以内。对x和y我们这里使用的是双精度数而不是浮点数。这里不仅表明此处要用高精度数，而且在C语言中浮点变量总是转换成双精度数，即使我们写成floatx，y（浮点数x，y），参数x，y将还是会转成双精度型。

现在让我们回头来看看我们最感兴趣的四个基本函数之一的draw。记得当初给它的任务是从笔当前的位置到新的点（x，y）画出一条线段。这条线段的起始点由全程变量X1和Y1给出，用浮点参数x，y给出其终点，用整型变量来处理这两个点的象素坐标和两个点的函数参数无疑是很好的。draw这个重要函数是§2—2的主题，但我们这里已准备使用它，现在将draw写在下面：

```
draw (x, y) double x, y;
{ int X1, Y1;
    X2=IX (x) ; Y2=IY (y) ; check (x2, y2) ;
    draw-line (X1, Y1, X2, Y2) ;
    X1=x1; Y2=y2;
}
```

§2—2 用整型数算法画线

函数draw还有待于补充说明。因为在堆栈环境工作，速度是个问题，发展到现在的版本使draw这个函数已经相当复杂了，不加说明是很难读懂的。我们将讨论几个版本，每个版本有其独特的问题。Beasenham 1965年出版的书在算法上尽管形式稍有不同，但其成绩是值得重视的。

在点（X1，Y1）和（X2，Y2）之间，提供给draw—line（画线）的变量，通常总有大量的计算。现在假定每个中介点（X，Y）都可以写一个函数dot（点），并且在屏幕上调用

```
dot (x, y) ;
```

这个低级函数的中介点（intermediatepoint）是基于我们使用的图形适配器的，在§2—4和§2—5要讨论这个题目。现在的任务是找出在屏幕上要用哪个网格上的点（X，Y）。由于有大量的点要计算，最好只用整数算法，这种算法比浮点算法要快些，由于第一个文本是简

单算法就忽略不讲。在计算点P1 (X_1, Y_1) 和P2(X_2, Y_2)的相对位置时，这种算法得出了很好的结果。如图2—1

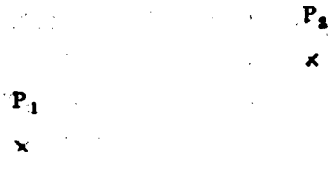

图2—1 点P_1和P_2之间的相对位置

要将点P_1移到点P_2，要向右移，还可能向上移，但P_1和P_2之间的纵坐标最大的可能值是等于横坐标，换句话说，即可能有如下关系：

$$X_1 < X_2$$

$$Y_1 \leqslant Y_2$$

$$Y_2 - Y_1 \leqslant X_2 - X_1$$

按照普通数学的约定，Y轴是指向上方的，在这里临时忽略这个事实，令Y轴指向下方，这不影响我们现在讨论的问题。对draw—line的第一个说明现在可以写成：

```
draw linel (X1, Y1, X2, Y2) int X1, Y1, X2, Y2;
{ float t; int X, Y;
  t = (float) (Y2 - Y1) / (float) (X2 - X1) ;
  for (X = X1; X <= X2; X + +)
  { Y = Y1 + (int) (t * (X - X1) + 0.5) ;
    dot (X, Y) ;
  }
}
```

一般情况下，上面的程序没有提到限制条件，我们必须搞清楚在X，Y中哪一个是由循环控制的独立变量。当然首先想到的是找出两个绝对值之差$|X_2 - X_1|$和$|Y_2 - Y_1|$中哪个较大。控制变量应该是减少而不是增加，如果t* (X—X_1) 是负数，变量就应该在取整前减少0.5（而不是增加）。我们现在不打算去追究这些细节，因为不可能推导出一个实用函数，使其能代替这种具有整数算法的浮点数。如果把最后结果马上写出恐怕很难懂，因此最好是一步步逐渐增加其复杂内容。现在假定P1和P2的位置象图2—1所示，在那种情况下仍然可以用浮点变量来发展一个代数算法，但很快就会看到这种方法很容易被淘汰。现在来考虑图2—2这个例子，在图中给定了点P1(1,1)P2 (12,5)（即当$X_1 = 1$时$Y_1 = 1$，$X_2 = 12$时，$Y_2 = 5$）

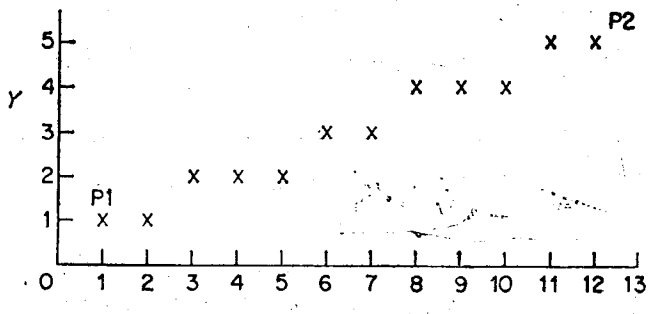

图2—2 P1，P2及其中间的点