第3卷 第11期
1998年11月

中国图象图形学报
Journal of Image and Graphics

Vol. 3, No. 11
Nov. 1998

实用技术

图形图象文件格式解码实用程序*(续9)

张明敏

9 PC Paint/Pictor PIC文件

下面介绍PC Paint和Pictor软件所使用的图形文件PIC格式。首先描述PIC文件的结构,接着介绍解码PIC文件的方法,包括相应的例子程序。

PIC文件格式产生较早,当时可供使用的机器为低档PC机,图形卡主要是CGA。随着计算机硬件设备的发展,出现了多种高级图形卡(如EGA、VGA以及Trident公司的TGUI等)。最初的PIC文件格式已不适应发展的需要,因而对PIC文件格式进行了扩充。值得注意的是虽然有很多应用系统都使用扩充PIC文件,但是不同的应用系统所用的含义也各不相同。例如PIC文件可用于保存电子表格软件Lotus 1-2-3所产生的图表,Lotus 1-2-3所用的图表是一种向量图画,而不是一般含义下的位图图象。在Lotus 1-2-3中使用线段和数字来组成表格,从而为用户提供"可视"显示。

在这里讨论的PIC文件是PC Paint/Pictor PIC,它是一种位图格式。它最初由PC Paint 2.0软件发明并使用,注意PC Paint软件与Z-Soft公司的PC Paint Paintbrush不一样,虽然PC Paint软件用户没有PC Paintbrush软件用户多,但是PIC文件仍然是一种有用的图形格式。一些动画软件经常使用这种文件格式来保存图形画面,例如Grasp图形动画系统就使用PIC文件。

PIC文件格式能支持有1位彩色、4位彩色或8位彩色的图象,图象的尺寸不受限制。它不把图象数据压缩为原有的线段形式,而是压缩成大块,这样使得解码PIC文件相对复杂。当然只要仔细设计算法,仍可以快速对PIC文件进行解码。在有些情况下,对特定种类的图形PIC可获得较好的压缩效果。

PIC格式使用行程编码(RLE)方法的一种变种对图象进行编码。

PIC格式的最显著特征是它把16种彩色的图象当作离散平面处理(这里的平面意指位平面),而不是作为交错行(interleavedline)处理。这种特征的一个缺点就是解码较为困难。在这种方式下,当对16色PIC文件进行解码时,必须取出第一个平面的所有信息数据后,再取出第二个平面的所有数据,…等等。

如果希望快速显示16色PIC图象,那么这种存储结构将比较有效。我们知道,修改处于16色模式下的显示卡通常有两种方式:第一种(也是正常情况下使用的一种是每次修改一行,这里包括为每个平面线都要切换4个显示页面,或者对每个完整的图象进行行4次切换(swap)。如果我们书写修改分辨率为640×480像素的整幅图象的代码,那么必须切换显示位平面近2 000次。虽然每次切换仅包括很少的指令,但是把图象数据写到屏幕的速度将相对减慢。

另一种方法是每次写一完整的位平面数据,即先写第一位平面上的所有数据,切换到下一个页面(即第二个位平面),写所有的数据,再切换到下一个页面,…,看到所有位平面数据都写完为止。这样做显然速度比较快。但是由于颜色不停地在改变,所以在修改时屏幕上显示的图象可能也会变化,有抖动或闪烁的感觉。

平面图象(按交错行方式保存的)很适合用前一

* 需源程序者请与浙江大学CAD&CG国家重点实验室张明敏联系,邮编: 310027,电话:(0571) 7951045,7990451

种方法显示,16 色 PIC 文件中的 4 个不同位平面就是离散平面,易于处理。当然上述两种结构可以互相转换。

在对用 4 个离散位平面保存的图象和用交错行方式保存的图象进行互相转换时,需要开设两个大的图象缓冲区——一个用于保存源图象,一个用于保存目标图象。这种方法有时并不实用,特别是当图象很大时。Graphics Workshop 软件在处理这个问题时,是通过把 PIC 数据写到一个临时磁盘文件中(压缩或对 16 色图象进行解码时)。这种方法并不是一种好方法,但它的优点是对内存空间限制较少,从而使得该软件可运行于多种平台之上。

最后要指出的是,对于分辨率为 640×480 的以交错行方式存储的 16 色位图图象而言,将需要大小为 150K 的缓冲区。我们知道,单个存储对像如果所需的空间大于 64K,那么在 PC 面上处理就需要一定的技巧。当然用户可以使用 farPtr 函数来避开这一点,但是这样做将使得速度大为减慢。

PIC 文件的 4 个位平面中的每一个平面可以放在 38K 大小的缓冲区中。虽然需要 4 个这种大小的缓冲区且整个缓冲区大小之和与保存相同大小的用交错行方式保存的图象所需的单个缓冲区大小一样,但是由于缓冲区是分开的,所以每个缓冲区所需的大小都不会大于一个存储段,从而使得编程相对容易。这样的话,每个位平面行可以使用 C 语言的指针进行存储。

当然,也可以把以交错行方式保存的平面图象用多个缓冲区保存,但这需要一定的编程控制,而 16 色 PIC 文件的结构则使得这样做比较容易。

9.1　PIC 文件结构

生成 PIC 文件的 PC Paint 应用程序在 80 年代初期就出现了,当时 CGA 卡属于高技术,而软盘只有一种规格。受当时条件影响,在 PIC 文件头标结构中也反映了这一点。PIC 格式不是简单地定义图象的尺寸和彩色级数,而是允许用户在 PIC 文件中指定源显示器(即生成相应 PIC 文件的显示器)的类型以及最适合处理该 PIC 文件的目标显示器类型。

随着后来显示卡的不断出现以及越来越多的图形显示模式变得可用,再试图把卡的显示模式与图象文件进行匹配就变得有点不切实际,因为用户不太可能使用显示模式信息来选择用来观察 PIC 文件所定义的图象的显示器。

PIC 文件以下面的头标开始,如果读者读了本书的前几章内容,那么对这种结构一定不陌生,即使不阅读下面的解释内容,也能大致知道这种格式的含义。尽管每种图形文件有各不相同的特征,但是只要它们是位图图象文件,那么它们实质上都是处理同样的信息类型。

```
typedef struct {
        unsigned int mark;
        unsigned int xsize;
        unsigned int ysize;
        unsigned int xoff;
        unsigned int yoff;
        char bitsinf;
        char emark;
        char evideo;
        unsigned int edesc;
        unsigned int esize;
        } PICHEAD;
```

上面的 mark 元素总是 1234H,该关键字(也可称为标识符)告知 PIC 阅读程序它是真正的 PIC 文件,而不是保存 Lotus 图表的文件。xsize 和 ysize 元素是图象尺寸(以像素为单位);而 xoff 和 yoff 值是显示图象的左上角坐标(相对于屏幕的左上角位置)。在绝大多数情形下,这些值都为 0。在后面讨论的 PIC 阅读程序忽略这 2 个值。

bitsinf 元素保存图象中彩色的位数,供解码用。不过这个值以一种非常奇怪且复杂的方式保存:低 4 位保存每个图象平面的每像素位数;而高 4 位则保存图象平面数。如果要根据比值推导出图象的彩色深度值,那么要用下面的公式:

```
bits = (((pic.bitsinf &0xf0) >>4)+1)
        * (pic.bitsinf &0xf0);
```

这里假定 pic 是一类型为 PICHEAD 的结构,它的值用 PIC 文件的开始部分加载。

emark 元素总是 0,evideo 元素是一字母,它表明最适合显示 PIC 文件的显示器类型,它可能包含下述字母中的一个:

A——CGA,4 种彩色
B——PCjr/Tandy 1000
C——CGA,2 种彩色
H——Hercules,单色
D——EGA,低分辨率
E——EGA,2 种彩色
F——EGA,4 种彩色
G——EGA,16 种彩色

H——Hercules,单色
I——Plantronics
J——EGA,低分辨率
K——AT&T 或 Toshiba 3100
L——VGA,256 种彩色
M——VGA,16 种彩色
N——Hercues InColor
O——VGA,单色

值得注意的是,虽然软件系统在需要时可使用域 evideo 中的信息,但是这不是必须的。用户可忽略它,即可以在其它相应显示器上显示图形,这是因为当解码后位图图象在本质上是与设备无关的,在后面介绍的 PIC 文件阅读程序将使用 VGA 模式 13H 显示上述所有类型的 PIC 文件,按照上面的说法(即根据 PIC 类型表),应是模式 L。

edesc 元素定义了解释跟在 PICHEAD 头标后的调色板信息的方法,有以下选择方案:

0——无调色板
1——CGA 的一字节彩色
2——PC_{jr}调色板
3——EGA 调色板
4——VGA 调色板

在这里中给出的程序代码并不处理 CGA 和 PCjr 文件。

最后,esize 值指定跟在头标后面的调色板信息的字节数:对单色文件而言该值为 0;对 16 色文件而言该值对 16 或 48;对于 256 色文件而言该值为 768。

我们知道,16 色显示器的彩色调色板可用两种方法表示,最简易的方法是 16 个 3 字节 RGB 彩色,VGA 卡允许它的 16 色调色板用这种方法设置,这样设置结果是:它的 16 种彩色可从 256 色显示模式下可用的 262 144 种彩色中的彩色集中选择。

旧的 EGA 卡支持的调色板设置方法比较粗糙,每种彩色用一单个字节定义,每个基本色(R、G、B)都用 2 位表示。通过使它的 2 位都置为 0 可把相应的彩色去掉,也可通过使它的 2 位值都为 1 从而使相应的彩色能显示出来,也可以通过把 2 位中的 1 位置为 1 从而设置中间彩色。

PIC 格式允许用户定义有任一种调色板的 16 色图象,PIC 文件写程序将用彩色号来指定调色板从而写 16 色 PIC 文件,读者可按自己的意愿改写这个程序从而使用更精确的调色板。

正确读出 PIC 文件中的调色板信息非常重要,原因是只有这样才能正确地反映出图象的彩色,至少应该读对由 esize 元素指定的调色板字节数。在调色板后的 2 个字节是定义图象中图象块数的整数值,该值一般为 12 左右,具体值依赖于图象的复杂性。

PIC 文件的块结构也比较复杂,一般情况下块忽略了图象行的边界,块最多可以包含 8192 个非压缩图象数据,因此必须由软件来负责对 PIC 文件进行解码,从而把它正确地分成一个或多个图象行。

另外,PIC 文件中的图象数据从画面的最后一行开始保存,从而使问题变得更复杂了。

每个图象块以下面的头标开始:

```
typedef struct {
        int pbsize;
        int bsize;
        char mbyte;
        } PICBLOCK;
```

其中,Pbsize 元素定义块中装配数置据(packed data)的大小,bsize 元素定义块将被拆卸后所占的空间总量,这个值不能超过 8 192 字节。典型情况下,除了最后一个块以外的所有块的 bsize 值都近似于 8 192 字节这个值。

mbyte 值是一唯一标记,从技术上讲,它应该是被压缩的块中最不经常出现的一个字节。事实上,在创建 PIC 文件时确定这个值比较麻烦。这里介绍的 PIC 文件写程序按一定的策略确定了这个值,相信读者看了程序后即可知道该值究竟为多少。

PIC 块中的图象数据是按行程编码方法压缩的数据,在压缩时使用了一种非常巧妙的算法。在 PICBLOCK 后的第一个字节是第一个压缩域的第一个字节,如果该字节与该块的 PICBLOCK 头标的 mbyte 值相同,那么域就为一组字节。下一个字节就可能是长度。如果下面一个字节为 0,那么随后的 2 个字节即为表示长度的整数值。这就允许行程长度在需要时超过 255 字节,在长度后的字节就是要重复的字节。

如果域中的第一个字节不是 mbyte,那么域即为 1 字节(1-byte,意为长度为 1 个字节)串,它被直接写到要被拆卸的块中。

如果要读的图象是一单色图象或者是一 256 色图象,那么一完全未压缩的块将被分成它的图象行并写到目标块中。如果它是一 16 色图象,解码后的块中的每个平面行都必须被增加到完整的图象行上,这里假定用户还要对交错行进行排序处理。在这

里讨论的 PIC 阅读程序中,所有行都被转换为彩色字节,当分别读取它的 4 个平面时,每个字节将被调整 4 次。

注意 16 色 PIC 文件的平面总是用离散块保存,因而,平面的结尾总是与块的结尾重合,16 色图象通常有 4 个短结束块(end block),而不是只有一个。

9.2 PIC 文件的解码

READPIC 程序代码对前面讨论的 PIC 文件进行解码。该程序很容易读懂,这是因为 PIC 文件的结构已在上面详细介绍过了。和前面一样,该程序将使用 VGA 卡的模式 13H 显示 PIC 文件。

```
/* * * * * * * * * * * * * * * * * * * * * * * * *
        PIC 解码程序
* * * * * * * * * * * * * * * * * * * * * * * * * *
*/
main(argc,argv)
    int argc;
    char *argv[];
{FILE *fp;
    static char results[4][16] = {  "Ok",
"Bad file","Bad read","Memory error"};
    char path[80];
    int r;
    if(argc > 1) {
        strmfe(path,argv[1],"PIC");
        strupr(path);
        if((fp = fopen(path,"rb")) != NULL) {
            fi.setup=dosetup;
            fi.closedown=doclosedown;
            r=unpackpic(fp,&fi);
            printf("\%s",results[r]);
            fclose(fp);
        }else printf("Error opening %s",path);
    }else puts("Argument:  path to a PIC file");
}/* unpack a PIC file */
unpackpic(fp,fi)
          FILE *fp;
          FILEINFO *fi;
{    PICHEAD pic;
    PICBLOCK pbk;
    char b[768], *block, *p;
    unsigned long l;
    int bcount,c,i,j,ldex=0,len;
    int line=0,pass,pos,r=GOOD_READ;
    if((block=malloc(8192)) == NULL)
    return(MEMORY_ERROR);
if(fread((char *)&pic,1,sizeof(PICHEAD),
fp) == sizeof(PICHEAD)) {
    if(pic.mark == 0x1234) {
      fi->width = pic.xsize;
      fi->depth = pic.ysize;
      fi->bits=(((pic.bitsinf & 0xf0)>>4)+
      1) * (pic.bitsinf & 0x0f);
      if(pic.esize > 768) return(BAD_FILE);
      if(fi->bits==8) fi->bytes=fi->width;
      else fi->bytes=pixels2bytes(fi->width);
      fread(b,1,pic.esize,fp);
      if(fi->bits==1)
      memcpy(fi->palette,"\000\000\000\377\
      377\377",6);
    else if(fi->bits==4 && pic.evideo != 'M')
      ega2vgapalette(fi->palette,b,16);
    else {memcpy(fi->palette,b,768);
      for(i=0;i<pic.esize;++i)
        fi->palette[i]=fi->palette[i] << 2;
    }
    if((fi->setup)(fi) != GOOD_READ) {
      free(block);
      return(MEMORY_ERROR);
    }
    if((p=malloc(fi->width)) != NULL) {
      memset(p,0,fi->width);
      for(i=0;i<fi->width;++i) putline(p,i);
      free(p);
    }
    else {free(block);
        return(MEMORY_ERROR);
    }
    bcount=fgetword(fp);
    if(bcount) {
      for(i=0;i<bcount;++i) {
        l=ftell(fp);
        pos=0;
          if(fread((char *)&pbk,1,
          sizeof(PICBLOCK),fp) !=
          sizeof(PICBLOCK)) {
          freebuffer();
          free(block);
          return(BAD_READ);
        }
  do { c=fgetc(fp);
        len=1;
        if(c==pbk.mbyte) {
          len=fgetc(fp);
```

```
                if(len==0) len=fgetword(fp);
                c=fgetc(fp);
                memset(block+ldex,c,len);
                ldex+=len;
            } else block[ldex++]=c;
            pos+=len;
            while(ldex >= fi->bytes) {
                ldex-=fi->bytes;
                if(fi->bits==8) putline(block,
                fi->depth-line-1);
                else if(fi->bits==1) {
                    if((p=mono2vga(block,
                    fi->width)) != NULL) {
                    putline(p,fi->depth-line-1);
                    free(p);
                     }
                    else {
                    freebuffer();
                    free(block);
                    return(MEMORY_ERROR);
                    }
                }
                else {
                  if((p=getline(fi->depth-
                  (line%fi->depth)-1)) !=
                   NULL) {
                    pass=line/fi->depth;
                    for(j=0;j<fi->width;++j) {
                    if(block[j>>3]
                    & masktable[j & 0x0007])
                    p[j] |= bittable[pass];
                    else p[j] &= ~bittable[pass];
                    }
                    putline(p,fi->depth-
                    (line%fi->depth)-1);
                  }
                }
        ++line;
        movmem(block+fi->bytes,block,ldex);
  }
} while(pos < pbk.bsize && c != EOF);
fseek(fp,1+(long)pbk.pbsize,SEEK_SET);
  }
(fi->closedown)(fi);
} else r=BAD_FILE;
        } else r=BAD_FILE;
    } else r=BAD_READ;
    free(block);
        return(r);
}
/* convert a monochrome line into an eight bit line */
char *mono2vga(p,width)
    char *p;
    int width;
{char *pr;
    int i;
    if((pr=malloc(width)) != NULL) {
      for(i=0;i<width;++i) {
       if(p[i >> 3] &
       masktable[i & 0x0007])
       pr[i]=1;
       else pr[i]=0;
       }
       return(pr);
       } else return(NULL);
    }
    fgetword(fp)
    FILE *fp;
    {return((fgetc(fp) & 0xff) +
    ((fgetc(fp) & 0xff) << 8));
    }
    /* set the VGA palette and background */
    setvgapalette(p,n,b)
        char *p;
        int n,b;
        {
          union REGS r;
          int i;
          outp(0x3c6,0xff);
          for(i=0;i<n;++i) {
             outp(0x3c8,i);
             outp(0x3c9,(*p++) >> 2);
             outp(0x3c9,(*p++) >> 2);
             outp(0x3c9,(*p++) >> 2);
          }
          r.x.ax=0x1001;
          r.h.bh=b;
          int86(0x10,&r,&r);
        }
    ega2vgapalette(dest,source,n)
        char *dest,*source;
        int n;
        {
          int i,r,g,b;
          for(i=0;i<n;++i) {
            r=g=b=0;
```

```
            if(*source & 0x01) b+=0x80;
            if(*source & 0x08) b+=0x40;
            if(*source & 0x02) g+=0x80;
            if(*source & 0x10) g+=0x40;
            if(*source & 0x04) r+=0x80;
            if(*source & 0x20) r+=0x40;
            ++source;
            *dest++=r;
            *dest++=g;
            *dest++=b;
        }
    }
/* make file name with specific extension */
strmfe(new,old,ext)
    char *new,*old,*ext;
    {
      while(*old != 0 && *
      old != '.') *new++=*old++;
       *new++='.';
      while(*ext) *new++=*ext++;
       *new=0;
    }
```

程序 READPIC 中的 unpackpic 函数完成主要的解码工作。它先读取 PIC 文件的头标,并对头标的参数值进行解释,即使文件非常短也必须这样做,由于 VGA 卡需要一个调色板,所以必须为 1 位彩色的 PIC 文件设置缺省的单色调色板。如果在通用的单色屏幕模式下显示单色 PIC 文件,那么将不需要这种设置操作。

READPIC 代码中有一定难度的是随后的块读取代码部分(直到 unpackpic 的结束处为止)。除了对块信息进行解压缩外,它还必须输出图象行数据。事实上,真正的工作是对域进行译码,直到在 block 缓冲区中有至少一行的图象数据,然后把图象行写到主图象缓冲区中并向左移动,直到到达 block 缓冲区的开始位置。随后域中的未压缩数据即被添加进来。

注意 16 色图象行必须从主图象缓冲区中读取,经修改再回写缓冲区四次。

从程序中可看到 READPIC 比前面介绍的读图形文件程序需要更多的存储空间。除了分配图象缓冲区外,它还必须开设一 8K 内存区作为块缓冲区。在绝大多数具有 640K 可用内存的基础上这应不成问题,但是如果在其它环境下(非 DOS 系统下,如 Turbo C 中),处理大的图象文件时即会出现问题。因为 READPIC 把所有的图象都作为 VGA 行保存,所以即使当前处理的是单色文件,也需要较大的内存空间。例如,对一个分辨率为 576×200 的单色图象而言,将需要 50K 存储区来保存 1 位平面图象,而 READPIC 程序将需要 405K 的内存工作区。

如果在 Turbo C 集成编程环境中工作,那么内存容量就成问题,很可能出现"out of memory"错。当用户在测试自己的程序代码(指读 PIC 文件的程序时),应使用较小的例子文件。