实用技术

# 图形图象文件格式解码实用程序*(续三)

张明敏

## 3 WordPerfect 的图形文件(WPG)

WordPerfect 是一种常用的字处理软件,它使用 WPG 格式来保存它的图形文件. WPG 格式把位图图象看作离散的图形对象,这些对象是 WPG 文件可以包含的内容之一。所以说它不是一种真正的图象文件格式,它是元文件。一个 WPG 文件可以包含多种图形信息。这种格式用于存储向量图形,向量图形是一组控制绘图机在何处绘制圆、线、弧和其它图元的指令。如果用户见过 AutoCAD 或 CorelDraw 如何重新生成它们的屏幕对象,那么就会熟悉这种向量图形的概念。

与另外几种基于向量的元文件一样,WPG 文件可以包含的一种对象类型是位图块。事实上,WPG 文件可以包含一个单一的位图块而不包含其它块。因此,完全有理由认为 WPG 文件能支持位图图象。WPG 格式的真正的目的在于把图表、图形等放入文字报告中。在 WordPerfect 生成的文档中经常会用到扫描的照片,读者也乐于接受这样的文档。

下面提供的程序代码允许用户读位映像的 WordPerfect 文件,它不能读向量的 WPG 文件。WordPerfect 软件中包含的 WPG 例子文件包含向量图形,并且许多公共域 WordPerfect 图形文件也包含向量描述。该程序代码可处理具有 1、4 和 8 个彩色位的 WordPerfect 图形。

### 3.1 WPG 文件的结构

WPG 文件由一个头标和一个"记录"表组成。WPG 文件中的记录个数任意,在一个向量图形中,每个记录是一个图形基元。例如,用一个存放向量图形的 WPG 文件绘制一个五角星,那么这个 WPG 文件至少需要 5 个记录。事实上,WordPerfect 实际需要的记录比这要多一些,因为还需要几个记录用于内部处理。

由于多数 WPG 记录类型包含向量图形元素,而这里不准备讨论向量图形,所以可以忽略 WPG 文件说明的大部分内容。与该程序代码相关的记录类型如下:

类型 15——标识开始的记录。

类型 14——彩色映像记录。

类型 11——位图记录。

类型 16——标识结束的记录。

一个只包含单个位图图象的 WordPerfect 文件,就有以上所列的每一种记录。

为了把一个 WPG 文件进行解码,用户首先要阅读 WPG 文件的头标,头标如下所示:

```
typedef struct{
        char id[4];
        long start;
        char product;
        char filetype;
        char majorversion;
        char minorversion;
        unsigned int encrypt;
        unsigned int reserved;
        } WPGHEAD;
```

**id 域**: WPGHEAD 结构体变量的 id 域总是存放字符串"1377WPC"。注意这是一个 4 个字节的字符串——字符串中的第一个元素是一个八进制常量。通过检查 id 域的内容,WPG 文件的解码程序就可以判定所读文件是一个真的还是假的 WPG 文件。

**start 域**: start 域说明了从文件开始到第一个记录的偏移量,这个值通常为 16L。

**product 域**: product 域应该为 1。

**filetype 域**: filetype 域应该存放 16H。

**majorversion 域**: 目前,majorversion 域的值为 1.

**minorversion 域**: minorversion 域的值为 0。

**encrypt 域**: encrypt 域存放 0,表示文件不加密。

* 注: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

**reserved 域：** reserved 域设置为 0。

WordPerfect 的图形记录中的第一个字节表示记录的类型，接下来的几个字节定义记录的大小。如果图形记录中的第二个字节保存的值在 0 到 254 之间，那么它表示记录的长度。如果它保存的值是 255，那么随后的 2 个字节表示一个整数，它定义记录的长度。如果 WordPerfect 的图形记录相当大，那么它用长整数来定义记录的长度。在读一个用整数保存长度的记录时，如果整数的最高位为 1，那么该整数和 7FFFH 相与的结果构成一个长整数的前 2 个字节，随后在 WPG 文件中读取的 2 个字节为一个长整数的后 2 个字节。

决定了一个记录的长度和类型之后，WPG 文件的解码程序可以开始对相关类型的内容进行解码或跳过该记录。

(1)标识开始的记录

一个"start WPG Date"记录的类型值为 15，它总是 6 个字节长，它的数据结构如下所示：

```
typedef struct {
        char version;
        char flags;
        int screenwidth;
        int screendepth;
        } STARTRECORD;
```

**version 域：** version 域的值应该为 1。

**flags 域：** 如果 flags 域与 01H 相与的结果为真，那么在被阅读的 WPG 文件中有 postscript 代码。

**screenwidth 域和 screendepth 域：** 如果在文件中存在向量图形，那么 screenwidth 和 screendepth 域给出了当前文件的向量图形的虚拟空间大小。

(2)彩色映像(Color Map)记录

"Color Map"记录的类型值为 14。记录中的数据是如下所示的结构：

```
typedef struct {
        int startindex;
        int palettesize;
        } COLORMAP;
```

这个数据结构之后是许多 3 字节的调色板入口。

**startindex 域：** startindex 域定义了调色板的入口数，其中装入由这个彩色映像所定义的第一种颜色，这个值一般为 0。

**palettesize 域：** palettesize 域定义了在这个彩色映像中的颜色数目。假设 startindex 的值为 0，那么对 4 位的文件 palettesize 的值为 16，对 8 位的文件该值为 256。

(3)位图记录

一个"Bitmap"记录的类型值为 11。它包含如下的数据结构：

```
typedef struct {
        int width;
        int depth;
        int bits;
        int xresolution;
        int yresolution;
        } BITMAP;
```

**width 域和 depth 域：** width 和 depth 域定义了位图图象的大小。

**bits 域：** bits 域存放彩色位的数目，它可以是 1，2，4 或 8。

**xresolution 域和 yresolution 域：** xresolution 和 yresolution 准备打印该图形时每英寸的点数。

该数据结构之后是实际的编码图形数据。

位图记录需要一个长整数来给出其长度。该记录通常包含了位图 WPG 文件中的大多数数据。

### 3.2 WPG 文件的解码

WPG 文件中的图象数据使用了一个专用的行程编码过程来存储。与其它的行程编码行一样，一个 WPG 行由一系列的域组成，每个域为四个字节。每个域由一个关键字节(用于说明它的类型)和 3 个数据字节组成。

(1)如果关键字节的最高位和其它位都被设置，那么该域是一个字节行程。在这种情况下，长度为关键字节与 7FH 相与所得的结果值。文件中的随后的字节将重复若干次(即上面计算得到的结果值)。

(2)如果关键字节的最高位被设置而其它位没有被设置，那么这个域是一个实心行程(solid run)。文件中接下来的字节为行程的长度，字节 FFH 将重复由长度指定的次数。

(3)如果关键字节的最高位没被设置而其它位被设置，那么这个域是一个字符串，关键字节的值为串的长度。

(4)如果关键字节为 0，那么文件中的下一个字节为重复的次数，这表示以前的译码行应该重复下一个字节所指定的次数。

单色的 WPG 文件把它的行存放为单个单色的平面，8 位的 WPG 文件以字节阵列存放它的行。4

位的 WPG 文件以一个专用的方法来保存它的数据。一般说来,4 位的图象文件在处理位图时,要特别小心,原因是保存位图有几种方法,并且对每一种方法有相应的参数进行说明。

这里讨论的 WPG 文件解码程序使用模式 13 来显示它所读到的图象。在这种情况下,一个 4 位的 WPG 行以通常的方式翻译——即每个字节由 readline 函数(参见后面的程序)翻译成两个不同的字节,而没有位平移操作。在大多数情况下,用户不想用这种效率低的方法来保存图象数据。

### 3.3 WPG 文件的解码程序

下面给出了一个简单的阅读位图 WPG 文件的程序,其中一个主要的子程序是 unpackwpg。

```
    /* * * * * * * * * * * * * * * * * * * * * * *
                WPG 解码程序
     * * * * * * * * * * * * * * * * * * * * * * */
main(argc,argv)
    int argc;
    char *argv[];
{FILE *fp;
    static char results[8][16] = {  "Ok","Bad file",
        "Bad read","Memory error"};
    char path[81];
    int r;
    if(argc > 1) {
        strmfe(path,argv[1],"WPG");
        strupr(path);
        if((fp = fopen(path,"rb")) != NULL) {
            fi.setup=dosetup;
            fi.closedown=doclosedown;
            r=unpackwpg(fp,&fi);
            printf("\%s",results[r]);
            fclose(fp);
        } else printf("Error opening %s",path);
    } else puts("Argument:     path to a WPG file");
}
/* unpack a WPG file */
unpackwpg(fp,fi)
    FILE *fp;
    FILEINFO *fi;
{WPGHEAD wpg;
    char *p, *pr;
    unsigned long offset=0L;
    int i;
    fi->width=fi->depth=fi->bytes=
    fi->bits=0;
    memcpy(fi->palette,"\000\000\000\377\377\377",
    6);
    if(fread((char *)&wpg,1,sizeof(WPGHEAD),fp)=
    =sizeof(WPGHEAD)) {
      if(! memcmp(wpg.id,"\377WPC",4)) {
          fseek(fp,wpg.start,SEEK_SET);
          do {i=readrecord(fi,fp,&offset);
          } while(i != 16 && i != EOF);
          if(i==16 &&fi->width != 0 &&fi->
            depth != 0 &&offset != 0L) {
          if((p=malloc(fi->bytes)) != NULL) {
          if((fi->setup)(fi) != GOOD_READ) {
            free(p);
            return(MEMORY_ERROR);
          }
          /* find the start of the image data */
          fseek(fp,offset,SEEK_SET);
          /* read all the lines */
          for(i=0;i<fi->depth;++i) {
            if(readwpgline(p,fp,fi->bytes)
              != fi->bytes) {
                freebuffer();
                free(p);
                return(BAD_READ);
            }
            /* translate the line types into VGA */
            switch(fi->bits) {
              case 1:
                pr=mono2vga(p,fi->width);
                if(pr != NULL) {
                    putline(pr,i);
                    free(pr);
                }
                else {
                    freebuffer();
                    free(p);
                    return(MEMORY_ERROR);
                }
                break;
                case 4:
                pr=ega2vga(p,fi->width);
                if(pr != NULL) {
                    putline(pr,i);
                    free(pr);
                }
                else {
                    freebuffer();
                    free(p);
```

```
                    return(MEMORY_ERROR);
                }
                break;
            case 8:
                putline(p,i);
                break;
            }
        }
        (fi->closedown)(fi);
        free(p);
        return(GOOD_READ);
        } else return(MEMORY_ERROR);
    } else return(BAD_FILE);
    } else return(BAD_FILE);
  } else return(BAD_READ);
}
/* 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);
}
/* convert a four bit line into an eight bit line */
char *ega2vga(p,width)
    char *p;
    int width;
{
    char *pr;
    int i,j=0;
    if((pr=malloc(width)) != NULL) {
        for(i=0;i<width;) {
            pr[i++]=(p[j] >> 4) & 0x0f;
            pr[i++]=p[j] & 0x0f;
            ++j;
        }
        return(pr);
    } else return(NULL);
}
/* read one record of a wpg file */
readrecord(fi,fp,offset)
    FILEINFO *fi;
    FILE *fp;
    unsigned long *offset;
{
    unsigned long l,t;
    unsigned int i,j,type,fc;
    type=fgetc(fp);
    t=ftell(fp);
    i=fgetc(fp) & 0x00ff;
    if(i == 0xff) {
        i=fgetword(fp);
        if(i & 0x8000) {
            l = (unsigned long)(i & 0x7fff) << 16;
            i = fgetword(fp);
            l += (((unsigned long)i)+4L);
        } else l=(((unsigned int)i)+2L);
    } else l=(unsigned long)i;
    switch(type) {
        case 11:            /* bitmap */
            fi->width=fgetword(fp);
            fi->depth=fgetword(fp);
            fi->bits=fgetword(fp);
            fgetword(fp);
            fgetword(fp);
            *offset=ftell(fp);
            if(fi->bits == 8) fi->bytes=fi->
            width;
            else fi->bytes=pixels2bytes(fi->width)
            *fi->bits;
            break;
        case 14:            /* palette */
            fc=fgetword(fp);
            j=fgetword(fp);
            for(i=0;i<j;++i) {
              if(((fc+i)*RGB_SIZE) >= 768) break;
              fi->palette[((fc+i)*RGB_SIZE)+
              RGB_RED]=fgetc(fp);
              fi->palette[((fc+i)*RGB_SIZE)+
              RGB_GREEN]=fgetc(fp);
              fi->palette[((fc+i)*RGB_SIZE)+
              RGB_BLUE]=fgetc(fp);
            }
            break;
    }
    fseek(fp,t+l+1L,SEEK_SET);
```

```
    return(type);
}
/* uncompress one line of a wpg image */
readwpgline(p,fp,bytes)
    char *p;
    FILE *fp;
    int bytes;
{
    static int repeat;
    int c,d,i,n=0;
    if(repeat) {
        --repeat;
        n=bytes;
    }
    else {
        do {
            c=fgetc(fp);
            if((c & 0x0080) && (c & 0x007f)) {
              d=fgetc(fp) & 0xff;
              for(i=0;i<(c & 0x7f);++i) p[n++]=
              d;
            }
            else if((c & 0x0080) && ! (c & 0x007f)) {
              d=fgetc(fp) & 0xff;
              for(i=0;i<d;++i) p[n++]=0xff;
            }
            else if(! (c & 0x0080) && (c & 0x007f)) {
            for(i=0;i<(c & 0x7f);++i) p[n++]=
            fgetc(fp);
            }
            else {
              repeat=fgetc(fp);
              n=bytes;
            }
        } while(n < bytes);
    }
    return(n);
}
fgetword(fp)
    FILE *fp;
{
    return((fgetc(fp) & 0xff) + ((fgetc(fp) & 0xff) <<
        8));
}
/* this function is called before an image is unpacked */
dosetup(fi)
    FILEINFO *fi;
{
    union REGS r;
    if(! getbuffer((long)fi->width * (long)fi->depth,
    fi->width,fi->depth))
    return(MEMORY_ERROR);
    r.x.ax=0x0013;
    int86(0x10,&r,&r);
    setvgapalette(fi->palette,1<<fi->bits,fi->
    background);
    return(GOOD_READ);
}
/* This function a called after an image has been un-
packed. It must display the image and deallocate memory.
*/
doclosedown(fi)
    FILEINFO *fi;
{
    union REGS r;
    int c,i,n,x=0,y=0;
    if(fi->width > SCREENWIDE) n=SCREEN
    WIDE;
    else n=fi->width;
    do {
        for(i=0;i<SCREENDEEP;++i) {
            c=y+i;
            if(c>=fi->depth) break;
            memcpy(MK_FP(0xa000,SCREENWIDE *
            i),getline(c)+x,n);
        }
        c=GetKey();
        switch(c) {
        case CURSOR_LEFT:
            if((x-STEP) > 0) x-=STEP;
            else x=0;
            break;
        case CURSOR_RIGHT:
            if((x+STEP+SCREENWIDE) < fi->
            width) x+=STEP;
            else if(fi->width > SCREENWIDE)
              x=fi->width-SCREENWIDE;
            else x=0;
            break;
        case CURSOR_UP:
            if((y-STEP) > 0) y-=STEP;
            else y=0;
            break;
        case CURSOR_DOWN:
            if((y+STEP+SCREENDEEP) < fi->
            depth) y+=STEP;
```

```
            else if(fi->depth > SCREENDEEP)
              y=fi->depth-SCREENDEEP;
            else y=0;
            break;
        case HOME:
            x=y=0;
            break;
        case END:
            if(fi->width > SCREENWIDE)
              x=fi->width-SCREENWIDE;
            else x=0;
            if(fi->depth > SCREENDEEP)
              y=fi->depth-SCREENDEEP;
            else y=0;
            break;
        }
    } while(c != 27);
    freebuffer();
    r.x.ax=0x0003;
    int86(0x10,&r,&r);
    return(GOOD_READ);
}
/* get one extended key code */
GetKey()
{
    int c;
    c = getch();
    if(! (c & 0x00ff)) c = getch() << 8;
    return(c);
}
/* 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);
}
/* 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;
}
```

在编制 WPG 文件解码程序时，尽管对一些不符合要求的输入，程序已做了警告处理，但程序使用者还肯定会给解码程序提供某些向量 WPG 文件。如果 WPG 文件的头标包括一个标志来表示在每个文件中是否存在一个位图记录，那么编制 WPG 解码程序就会很方便。但情况并非如此。因而，既然需要一个位图记录来设置位图图象的大小，那么程序可以通过阅读一个 WPG 文件的所有记录和检查图像的大小来决定是否已发现了一个位图。

unpackwpg 函数一开始设置图象的大小和彩色位的值为 0，这是一个非法值。如所有记录被解码后，这两个值仍为 0，那么正在被读的文件没有位图记录，并可假定该文件只包含向量图形元素。

readrecord 函数处理文件中的每一个记录。事实上，如果用户仔细地分析该函数时，就会注意到该函数只处理两种类型的记录——类型 11 的位图记录和类型 14 的彩色映像记录，该函数会跳过其它类型的记录。当读到一个位图记录时，readrecord 函数阅读位图数据结构的固定部分而不将位图数据译码，接着把 offset 的值设置为指向位图数据的地址。当所有的记录写完以后，unpackwpg 函数返回到 offset 指向的地方并去处理图象数据。

对文件中的每一行需调用 readwpgline 函数一次。由于这个文件解码程序要求它的行数据按每个像素一个字节的方式存放在存储器中，所以单色的行必须传送给 mono2vga 函数进行处理，4 位的行由 ega2vga 函数来处理。因为 ega2vga 函数给出了如何还原 WPG 文件行的每字节 2 个像素的格式，因此读者有必要分析一下该函数。

如果用户未仔细地阅读 ega2vga 函数，那么使用该函数可能会导致用户破坏 WPG 文件的图象数据。当然，这里给出的 ega2vga 函数是正确的。如果用户自己编制相应的函数，那么要注意在每个被译码的字节中以正确的顺序抽取两个像素。