数字图象处理编程入门

——做一个自己的 Photoshop

吕凤军　编著

清 华 大 学 出 版 社

(京)新登字 158 号

内 容 简 介

数字图象处理是一门非常有趣而且实用的技术,一些常用的软件如 Photoshop, CorelPhotoPaint 等的核心就是图象处理技术;然而市面上介绍图象处理的书大多理论性太强,涉及到大量的数学公式,读过很多章后也没有一个感性认识,使得许多对图象处理感兴趣的读者望而生畏。本书正是为这样的读者编写的。

本书首先介绍了 Windows 位图结构和调色板的概念,为后面的学习作准备。随后,由浅入深、结合实例、生动形象地介绍了图象处理中最常用的算法,如几何变换、平滑、锐化、半影调、抖动、直方图修正、彩色变换、腐蚀和膨胀、细化、骨架的提取、边沿检测与抽取、hough 变换、轮廓跟踪、图象检测、模板匹配、图象压缩编码、JPEG 压缩编码标准等。最后,介绍了非常实用的图象处理编程控件 Lead、视频游戏编写常用工具 DirectDraw 以及简单的多媒体编程技术。

本书语言简洁流畅、风趣幽默,复杂的算法、深奥的公式均生动形象地呈现在读者面前,使读者在轻松愉快的学习气氛中逐步领略数字图象处理技术的魅力。附盘中给出了书中实例的 C 语言源程序及图象文件,可在 Windows 环境下直接运行。

本书适用于数字图象处理技术的初学者及 C 语言编程爱好者。

书　名: 数字图象处理编程入门——做一个自己的 Photoshop
作　者: 吕凤军　编著
出版者: 清华大学出版社(北京清华大学校内,邮编 100084)
http: // www. tup. tsinghua. edu. cn
印刷者: 昌平环球印刷厂
发行者: 新华书店总店北京发行所
开　本: 787× 1092 1/ 16　印张: 11. 75　字数: 276 千字
版　次: 1999 年 9 月第 1 版　1999 年 9 月第 1 次印刷
书　号: ISBN 7-302-03523-7/ TP・1928
印　数: 0001～3000
定　价: 24.00 元(附光盘)

前　言

当你看到那些用 Photoshop 或 CorelPhotoPaint 等软件制作出的精美图片，并被它们表现出的神奇效果所折服时，是否曾想到做一个自己的图象处理软件？就像 Photoshop 那样。“怎么可能，这是吹牛吧！”你一定会这么说。呵呵，别着急，待我慢慢讲来。

我是一名清华大学计算机系的研究生，一直对图象处理、多媒体的东西非常着迷，选修过不少这方面的课程，做过许多有意思的实验。我常有这样一种冲动：把我做过的这些东西拿出来与大家一起分享，把有关的原理、算法、程序介绍给大家。

有这种想法的另一个原因是：数字图象处理（digital image processing）技术是一门非常实用的技术。Photoshop 的核心就是图象处理技术。而目前有关这方面的资料太少了，已有的书不是内容太陈旧（还停留在 DOS 下的 CGA，EGA 甚至汇编编程阶段），就是理论性太强，不容易懂，没有例子，学过以后也不知道该如何编程。我想，如果能够通过实际的例子来介绍这些图象处理算法，大家就很容易理解了。

这本书的侧重点是介绍数字图象处理编程，因此在内容的选择上也有所考虑，介绍的都是数字图象处理中的基本算法。这些算法大多可以通过程序实现。而那些理论性很强、不易编程的算法，在本书里是找不到的。书中的每一章可以看作是一个专题，后面都附有实际运行通过的源程序例子，全部程序都采用 Windows 编程（不过，我并不打算介绍 Windows 的编程，这已经超出了本书的范围）。随附的软盘中提供了全部源程序、可执行文件和例图，详细说明在文件 readme.txt 中。

下面是本书的主要内容：

(1) Windows 位图 bitmap（即 bmp 文件）的结构和调色板的概念；

(2) 图象的平移、旋转、镜象变换、转置变换、缩放；

(3) 图象的平滑（去噪声）、锐化；

(4) 图象的半影调、抖动技术；

(5) 图象的直方图修正、彩色变换；

(6) 图象的腐蚀和膨胀效果、细化算法、骨架的提取；

(7) 图象的边沿检测与抽取、hough 变换、轮廓跟踪；

(8) 图象的检测、模板匹配；

(9) 图象的压缩编码、JPEG 压缩编码标准；

(10) 图象处理编程工具 lead.ocx，DirectDraw 及简单的多媒体编程技术。

这里面有一些非常有趣的应用。例如在第 4 章介绍了一个将一幅图象转换成 ASCII 码的算法。对于喜欢上 BBS 的读者，这个算法是非常有用的。在第 10 章，介绍了一款非常实用的图象处理编程工具 lead.ocx，利用它可以很快地开发出非常棒的图象处理软件。

学习了这些内容，你就掌握了数字图象处理中的一些最常用的算法。当然，这些内容还不足以编出像 Photoshop 那样“牛”的软件（人家毕竟是 Adobe 公司的看家宝贝嘛！）。

但是要知道:万丈高楼平地起,很多非常复杂的功能可能是一些简单方法的叠加。相信你自己吧,你一定能够成为图象处理大师的。

是不是还有些信心不足?好,让我变一个戏法给你瞧瞧。

图 1　普通文本

图 2　抽取骨架后的文本

上面有两幅图,图 1 是未经处理的普通文字,经过骨架抽取,变成了图 2 的样子。这可不是用 Photoshop 做的,而是用我自己编的程序处理的。怎么样?还不错吧。

有人会问:“在编这样的程序之前有什么要求吗?”回答是:“有,只有两条:(1) 对 C 语言比较熟悉;(2) 曾经编过 Windows 程序。”

有三点要说明:

(1) 文中出现的所有例子都在我自己的机器上编译运行通过,我使用的编程语言为 Visual C++ 4.1,运行环境为中文 Windows 95 或 Windows 98。程序采用的是 Windows API 接口,全部采用 C 语言编写,并未用到 C++ 的东西,所以也可以在其他的 C 编译器,如 Borland C,Watcom C 下编译通过(可能有些函数的名称有些差别,所以建议使用 Visual C++ ,4.0、4.1、4.2、5.0 版本都可以)。尽管在 Windows 3.x 平台上也能编译运行这些程序,但强烈建议使用 Windows 95 或 Windows 98,32 位的虚拟内存环境用起来爽极了。

(2) 既然是编图象处理的程序,当然要把机器的分辨率和颜色数调大一点了,这样显示出来的图象才显得漂亮(我用的是 800× 600,16 位即 64K 色)。另外,装备一些好的图象软件是绝对必要的。我经常使用以下几种软件:

- Sea,在 DOS 下的看图工具,而且可以很方便的转换图象格式;
- AcdSee,一个小巧玲珑的看图软件;
- Ulead IphotoPlus,最大的优点是可以进行调色板的处理;
- Windows PaintBrush,不要以为画笔的功能很弱,其实很多情况下还是很有用的;
- Photoshop,就不用我多说了。

(3) 图象处理的算法中不可避免地要遇到一些数学公式,霍金说过:“每多一个公式就要吓跑一半的读者”,我将尽可能用通俗的语言将这些原理、公式讲解出来,力求做到公

式尽可能的少，但遇到只有用公式才能讲明白的时候，我也绝不回避，希望大家能耐着性子看下去。

本书主要参考了我上数字图象处理课时的教材，该教材的作者是朱志刚老师，在此表示感谢。还要感谢我的好朋友诸晓文和袁昱，没有他们的帮助，这本书的出版是不可能的。

好了，不多说了，现在就让我们进入五彩缤纷的图象世界吧！

作　者

1999 年 6 月于清华大学

目　　录

第 1 章　Windows 位图和调色板 …… 1
1.1　位图和调色板的概念 …… 1
1.2　bmp 文件格式 …… 3
1.3　显示一个 bmp 文件的 C 程序 …… 6
第 2 章　图象的几何变换 …… 20
2.1　平移 …… 20
2.2　旋转 …… 28
2.3　镜象 …… 33
2.4　转置 …… 36
2.5　缩放 …… 39
第 3 章　图象的平滑(去噪声)、锐化 …… 43
3.1　平滑 …… 43
3.2　中值滤波 …… 46
3.3　锐化 …… 49
第 4 章　图象的半影调和抖动技术 …… 54
4.1　图案法 …… 54
4.2　抖动法 …… 59
4.3　将 bmp 文件转换为 txt 文件 …… 63
第 5 章　直方图修正和彩色变换 …… 67
5.1　反色 …… 67
5.2　彩色图转灰度图 …… 71
5.3　真彩图转 256 色图 …… 75
5.4　对比度扩展 …… 80
5.5　削波 …… 85
5.6　阈值化 …… 86
5.7　灰度窗口变换 …… 87
5.8　灰度直方图统计 …… 89
5.9　灰度直方图均衡化 …… 92
第 6 章　腐蚀、膨胀、细化算法 …… 99
6.1　腐蚀 …… 101
6.2　膨胀 …… 105
6.3　开 …… 109
6.4　闭 …… 111

6.5 细化 …… 112
第7章 边沿检测与轮廓、提取跟踪 …… 118
7.1 边沿检测 …… 118
7.2 Hough 变换 …… 121
7.3 轮廓提取 …… 125
7.4 种子填充 …… 127
7.5 轮廓跟踪 …… 132
第8章 图象的检测及模板匹配 …… 136
8.1 投影法 …… 136
8.2 差影法 …… 140
8.3 模板匹配 …… 144
第9章 图象压缩编码及 JPEG 压缩编码标准 …… 145
9.1 哈夫曼编码 …… 146
9.2 行程编码 …… 147
9.3 LZW 算法的大体思想 …… 153
9.4 JPEG 压缩编码标准 …… 153
第10章 图象处理编程工具及简单的多媒体编程 …… 163
10.1 LeadTools …… 163
10.2 DirectDraw …… 173
10.3 简单的多媒体编程 …… 174
参考文献 …… 177
后记 …… 178

第 1 章　Windows 位图和调色板

1.1　位图和调色板的概念

如今 Windows(3.x 以及 95, 98, NT) 系列已经成为绝大多数用户使用的操作系统, 它比 DOS 成功的一个重要因素是可视化的漂亮界面。那么 Windows 是如何显示图象的呢? 这就要谈到位图(bitmap)。

我们知道, 普通的显示器屏幕是由许许多多的点构成的, 我们称之为象素。显示时采用扫描的方法: 电子枪每次从左到右扫描一行, 为每个象素着色, 然后从上到下这样扫描若干行, 就扫过了一屏。为了防止闪烁, 每秒要重复上述过程几十次。例如我们常说的屏幕分辨率为 640× 480, 刷新频率为 70Hz, 意思是说每行要扫描 640 个象素, 一共有 480 行, 每秒重复扫描屏幕 70 次。

我们称这种显示器为位映象设备。所谓位映象, 就是指一个二维的象素矩阵, 而位图就是采用位映象方法显示和存储的图象的。举个例子, 图 1.1 是一幅普通的黑白位图, 图 1.2 是被放大后的图, 图中每个方格代表了一个象素。我们可以看到: 整个骷髅就是由这样一些黑点和白点组成的。

图 1.1　骷髅

图 1.2　放大后的骷髅位图

那么, 彩色图是怎么回事呢? 我们先讲解三元色 RGB 的概念。

我们知道, 自然界中的所有颜色都可以由红、绿、蓝(R, G, B) 组合而成。有的颜色含有红色成分多一些, 如深红; 有的含有红色成分少一些, 如淡红。针对含有红色成分的多少, 可以分成 0 到 255 共 256 个等级; 0 级表示不含红色成分, 255 级表示含有 100% 的红

色成分。同样,绿色和蓝色也被分成 256 级。这种分级的概念被称作量化。

这样,根据红、绿、蓝各种不同的组合我们就能表示出 256×256×256,约 1 600 万种颜色。这么多颜色对于我们人眼来说已经足够了。

表 1.1 是常见的一些颜色的 RGB 组合值。

表 1.1 常见颜色的 RGB 组合

颜色	R	G	B
红	255	0	0
蓝	0	255	0
绿	0	0	255
黄	255	255	0
紫	255	0	255
青	0	255	255
白	255	255	255
黑	0	0	0
灰	128	128	128

你大概已经明白了,当一幅图中每个象素赋予不同的 RGB 值时,就能呈现出五彩缤纷的颜色了,这样就形成了彩色图。对,是这样的,但实际上的做法还有些差别。

让我们来看看下面的例子。

有一个长宽各为 200 个象素,颜色数为 16 的彩色图,每一个象素都用 R,G,B 三个分量表示。因为每个分量有 256 个级别,要用 8 位(bit),即一个字节(byte)来表示,所以每个象素需要用 3 个字节。整个图象要用 200×200×3,约 120K 字节,可不是一个小数目呀!如果我们用下面的方法,就能节省很多。

因为是一个 16 色图,也就是说这幅图中最多只有 16 种颜色,因此可以用一个表,表中的每一行记录一种颜色的 R,G,B 值。这样当我们表示一个象素的颜色时,只需要指出该颜色是在第几行,即该颜色在表中的索引值。举个例子,如果表的第 0 行为 255,0,0(红色),那么当某个象素为红色时,只需要标明 0 即可。

让我们再来计算一下:16 种状态可以用 4 位表示,所以一个象素要用半个字节。整个图象要用 200×200×0.5,约 20K 字节,再加上表占用的字节为 3×16=48 字节,整个占用的字节数约为前面的 1/6,省很多吧?

这张 R,G,B 的表,就是我们常说的调色板,另一种叫法是颜色查找表 LUT(look up table),似乎更确切一些。Windows 位图中便用到了调色板技术。其实不光是 Windows 位图,许多图象文件格式如 pcx,tif,gif 等都用到了调色板。所以很好地掌握调色板的概念是十分必要的。

有一种图,它的颜色数高达 256×256×256,也就是说包含我们上述提到的 R,G,B 颜色表示方法中所有的颜色,这种图叫作真彩色图(true color)。真彩色图并不是说一幅

图包含了所有的颜色,而是说它具有显示所有颜色的能力,即最多可以包含所有颜色。表示真彩色图时,每个象素直接用 R,G,B 三个分量字节表示,而不采用调色板技术。原因很明显:如果用调色板,表示一个象素也要用 24 位。这是因为每种颜色的索引要用 24 位(因为总共有 2^{24} 种颜色,即调色板有 2^{24} 行),和直接用 R,G,B 三个分量表示的字节数一样,不但没有任何便宜,还要加上一个 256×256×256×3 个字节的大调色板。所以真彩色图直接用 R,G,B 三个分量表示,它又叫作 24 位色图。

1.2 bmp 文件格式

介绍完位图和调色板的概念,下面就让我们来看一看 Windows 位图文件(bmp 文件)的格式是什么样子的。

bmp 文件大体上分成四个部分,如图 1.3 所示。

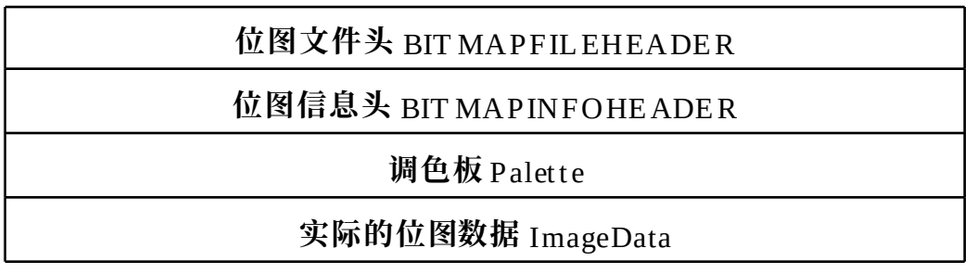

图 1.3 Windows 位图文件结构示意图

第一部分为位图文件头 BITMAPFILEHEADER,是一个结构,其定义如下:

```
typedef struct tagBITMAPFILEHEADER {
    WORD    bfType;
    DWORD   bfSize;
    WORD    bfReserved1;
    WORD    bfReserved2;
    DWORD   bfOffBits;
} BITMAPFILEHEADER;
```

这个结构的长度是固定的,为 14 个字节(WORD 为无符号 16 位整数,DWORD 为无符号 32 位整数),各个域的说明如下:

bfType

指定文件类型,必须是 0x424D,即字符串“BM”。也就是说所有 bmp 文件的头两个字节都是“BM”。

bfSize

指定文件大小,包括这 14 个字节。

bfReserved1,bfReserved2

为保留字,不用考虑。

bfOffBits

为从文件头到实际的位图数据的偏移字节数,即图 1.3 中前三个部分的长度之和。

第二部分为位图信息头 BITMAPINFOHEADER,也是一个结构,其定义如下:

```
typedef struct tagBITMAPINFOHEADER{
  DWORD    biSize;
  LONG     biWidth;
  LONG     biHeight;
  WORD     biPlanes;
  WORD     biBitCount
  DWORD    biCompression;
  DWORD    biSizeImage;
  LONG     biXPelsPerMeter;
  LONG     biYPelsPerMeter;
  DWORD    biClrUsed;
  DWORD    biClrImportant;
} BITMAPINFOHEADER;
```

这个结构的长度也是固定的,为 40 个字节(LONG 为 32 位整数),各个域的说明如下:

biSize

指定这个结构的长度,为 40。

biWidth

指定图象的宽度,单位是象素。

biHeight

指定图象的高度,单位是象素。

biPlanes

必须是 1,不用考虑。

biBitCount

指定表示颜色时要用到的位数,常用的值为 1(黑白二色图),4(16 色图),8(256 色),24(真彩色图)。(新的 bmp 格式支持 32 位色,这里就不讨论了。)

biCompression

指定位图是否压缩,有效的值为 BI_RGB,BI_RLE8,BI_RLE4,BI_BITFIELDS(都是一些 Windows 定义好的常量)。要说明的是,Windows 位图可以采用 RLE4 和 RLE8 的压缩格式,但用的不多。我们今后所讨论的只是第一种不压缩的情况,即 biCompression 为 BI_RGB 的情况。

biSizeImage

指定实际的位图数据占用的字节数,其实也可以从以下的公式中计算出来:

biSizeImage= biWidth' * biHeight

要注意的是: 上述公式中的 biWidth' 必须是 4 的整数倍(所以不是 biWidth,而是 biWidth',表示大于或等于 biWidth 的,最接近 4 的整倍数。举个例子,如果 biWidth= 240,则 biWidth'= 240;如果 biWidth= 241,则 biWidth'= 244)。

如果 biCompression 为 BI- RGB,则该项可能为零。

biXPelsPerMeter

指定目标设备的水平分辨率,单位是每米的象素个数,关于分辨率的概念,我们将在打印部分详细介绍。

biYPelsPerMeter

指定目标设备的垂直分辨率,单位同上。

biClrUsed

指定本图象实际用到的颜色数。如果该值为零,则用到的颜色数为 2 的 biBitCount 次方。

biClrImportant

指定本图象中重要的颜色数。如果该值为零,则认为所有的颜色都是重要的。

第三部分为调色板 Palette,当然,这里是对那些需要调色板的位图文件而言的。有些位图,如真彩色图,前面已经讲过,是不需要调色板的,BITMAPINFOHEADER 后直接是位图数据。

调色板实际上是一个数组,共有 biClrUsed 个元素(如果该值为零,则有 2 的 biBitCount 次方个元素)。数组中每个元素的类型是一个 RGBQUAD 结构,占 4 个字节,其定义如下:

```
typedef struct tagRGBQUAD {
    BYTE    rgbBlue; //该颜色的蓝色分量
    BYTE    rgbGreen; //该颜色的绿色分量
    BYTE    rgbRed; //该颜色的红色分量
    BYTE    rgbReserved; //保留值
    } RGBQUAD;
```

第四部分就是实际的图象数据了。对于用到调色板的位图,图象数据就是该象素在调色板中的索引值。对于真彩色图,图象数据就是实际的 R, G, B 值。下面对 2 色、16 色、256 色位图和真彩色位图分别介绍。

对于 2 色位图,用 1 位就可以表示该象素的颜色(一般 0 表示黑,1 表示白),所以 1 个字节可以表示 8 个象素。

对于 16 色位图,用 4 位可以表示 1 个象素的颜色,所以 1 个字节可以表示 2 个象素。

对于256色位图,1个字节刚好可以表示1个象素。

对于真彩色图,3个字节才能表示1个象素,哇噻,好费空间呀! 没办法,谁叫你想让图的颜色显得更亮丽呢,有得必有失嘛。

要注意两点:

(1) 每一行的字节数必须是4的整数倍,如果不是,则需要补齐。这在前面介绍biSizeImage时已经提到了。

(2) 一般来说,bmp文件的数据是从下到上、从左到右的。也就是说,从文件中最先读到的是图象最下面一行的左边第一个象素,然后是左边第二个象素……接下来是倒数第二行左边第一个象素,左边第二个象素……依次类推,最后得到的是最上面一行的最右一个象素。

好了,终于介绍完bmp文件结构了,是不是觉得头有些大? 别着急,对照着下面的程序,你就会很清楚了(我可最爱看源程序了,呵呵)。

1.3 显示一个bmp文件的C程序

下面的函数LoadBmpFile,其功能是从一个bmp文件中读取数据(包括BITMAPINFOHEADER、调色板和实际图象数据),将其存储在一个全局内存句柄hImgData中,这个hImgData将在以后的图象处理程序中用到。同时填写一个类型为HBITMAP的全局变量hBitmap和一个类型为HPALETTE的全局变量hPalette。这两个变量将在处理WM_PAINT消息时用到,用来显示出位图。该函数的两个参数分别是用来显示位图的窗口句柄和bmp文件名(全路径)。当函数成功时,返回TRUE,否则返回FALSE。

```
BITMAPFILEHEADER   bf;
BITMAPINFOHEADER   bi;
BOOL LoadBmpFile (HWND hWnd, char * BmpFileName)
{
    HFILE                   hf; //文件句柄
    LPBITMAPINFOHEADER lpImgData; //指向BITMAPINFOHEADER结构的指针
    LOGPALETTE              * pPal; //指向逻辑调色板结构的指针
    LPRGBQUAD               lpRGB; //指向RGBQUAD结构的指针
    HPALETTE                hPrevPalette; //用来保存设备中原来的调色板
    HDC                     hDc; //设备句柄
    HLOCAL                  hPal; //存储调色板的局部内存句柄
    DWORD                   LineBytes; //每一行的字节数
    DWORD                   ImgSize; //实际图象数据占用的字节数
    DWORD                   NumColors; //实际用到的颜色数,即调色板数组中的颜色个数
    DWORD                   i;

    if((hf=_lopen(BmpFileName, OF_READ))==HFILE_ERROR){
```

```
    MessageBox(hWnd, "File c:\\test.bmp not found!", "Error Message",
                MB_OK|MB_ICONEXCLAMATION);
    return FALSE; //打开文件错误,返回
}
//将 BITMAPFILEHEADER 结构从文件中读出,填写到 bf 中
_lread(hf,(LPSTR)&bf,sizeof(BITMAPFILEHEADER));
//将 BITMAPINFOHEADER 结构从文件中读出,填写到 bi 中
_lread(hf,(LPSTR)&bi,sizeof(BITMAPINFOHEADER));

//我们定义了一个宏 #define WIDTHBYTES(i)    ((i+31)/32*4)
//上面曾经提到过,每一行的字节数必须是 4 的整数倍,
//只要调用 WIDTHBYTES(bi.biWidth*bi.biBitCount)就能完成这一换算。
//举一个例子,对于 2 色图,如果图象宽是 31,则每一行需要 31 位存储,合 3 个字节加
//7 位,因为字节数必须是 4 的整倍数,所以应该是 4,而此时的
//biWidth=31,biBitCount=1,WIDTHBYTES(31*1)=4,和我们设想的一样。
//再举一个 256 色的例子,如果图象宽是 31,则每一行需要 31 个字节存储,因为字节数
//必须是 4 的整数倍,所以应该是 32,而此时的
//biWidth=31,biBitCount=8,WIDTHBYTES(31*8)=32,和我们设想的一样。你可以多举
//几个例子来验证一下。
//LineBytes 为每一行的字节数
LineBytes=(DWORD)WIDTHBYTES(bi.biWidth*bi.biBitCount);
//ImgSize 为实际的图象数据占用的字节数
ImgSize=(DWORD)LineBytes*bi.biHeight;
//NumColors 为实际用到的颜色数,即调色板数组中的颜色个数
if(bi.biClrUsed!=0)
    NumColors=(DWORD)bi.biClrUsed; //如果 bi.biClrUsed 不为零,就是本图象实际
//用到的颜色数;
else //否则,用到的颜色数为 2 的 biBitCount 次方。
    switch(bi.biBitCount){
    case 1:
        NumColors=2;
        break;
    case 4:
        NumColors=16;
        break;
    case 8:
        NumColors=256;
        break;
    case 24:
        NumColors=0; //对于真彩色图,没用到调色板
        break;
      default:
```

```
        //不处理其它的颜色数,认为出错
        MessageBox (hWnd, "Invalid color numbers! ", "Error
                Message",MB_OK|MB_ICONEXCLAMATION);
                _lclose(hf);
        return FALSE; //关闭文件,返回FALSE
    }
if (bf.bfOffBits! = (DWORD)(NumColors * sizeof(RGBQUAD) + sizeof
                (BITMAPFILEHEADER)+ sizeof(BITMAPINFOHEADER)))
{
    //计算出的偏移量与实际偏移量不符,一定是颜色数出错
    MessageBox(hWnd, "Invalid color numbers! ", "Error Message",MB_OK|
            MB_ICONEXCLAMATION);
    _lclose(hf);
    return FALSE; //关闭文件,返回FALSE
}
bf.bfSize = sizeof(BITMAPFILEHEADER)+ sizeof(BITMAPINFOHEADER)+ NumColors
        * sizeof(RGBQUAD)+ ImgSize;
//分配内存,大小为BITMAPINFOHEADER结构长度加调色板+ 实际位图数据
if((hImgData= GlobalAlloc(GHND,(DWORD)(sizeof(BITMAPINFOHEADER)+
    NumColors* sizeof(RGBQUAD)+ ImgSize)))= NULL)
{
    //分配内存错误
    MessageBox(hWnd, "Error alloc memory! ", "ErrorMesaage",MB_OK|
            MB_ICONEXCLAMATION);
    _lclose(hf);
    return FALSE; //关闭文件,返回FALSE
}
//指针lpImgData指向该内存区
lpImgData= (LPBITMAPINFOHEADER)GlobalLock(hImgData);

//文件指针重新定位到BITMAPINFOHEADER开始处
_llseek(hf,sizeof(BITMAPFILEHEADER),SEEK_SET);

//将文件内容读入lpImgData
_hread(hf,(char *)lpImgData,(long)sizeof(BITMAPINFOHEADER)
        + (long)NumColors* sizeof(RGBQUAD)+ ImgSize);
_lclose(hf); //关闭文件

if(NumColors! = 0)  //NumColors不为零,说明用到了调色板
{
    //为逻辑调色板分配局部内存,大小为逻辑调色板结构长度加NumColors个
    //PALETTENTRY大小
```

```
        hPal = LocalAlloc (LHND, sizeof (LOGPALETTE) + NumColors * sizeof
                           (PALETTEENTRY));

    //指针 pPal 指向该内存区
    pPal = (LOGPALETTE *)LocalLock(hPal);

    //填写逻辑调色板结构的头
    pPal-> palNumEntries = NumColors;
    pPal-> palVersion    = 0x300;

    //lpRGB 指向的是调色板开始的位置
        lpRGB = (LPRGBQUAD) ((LPSTR) lpImgData + (DWORD) sizeof
                                 (BITMAPINFOHEADER));

    //填写每一项
    for (i = 0; i < NumColors; i++)
    {
        pPal-> palPalEntry[i].peRed= lpRGB-> rgbRed;
        pPal-> palPalEntry[i].peGreen= lpRGB-> rgbGreen;
        pPal-> palPalEntry[i].peBlue= lpRGB-> rgbBlue;
        pPal-> palPalEntry[i].peFlags= (BYTE)0;
        lpRGB++; //指针移到下一项
    }

    //产生逻辑调色板, hPalette 是一个全局变量
    hPalette= CreatePalette(pPal);

    //释放局部内存
    LocalUnlock(hPal);
    LocalFree(hPal);
}

//获得设备上下文句柄
hDc= GetDC(hWnd);

if(hPalette) //如果刚才产生了逻辑调色板, 初始化为 NULL 的 hPalette 将不再为 NULL
{
    //将新的逻辑调色板选入 DC, 将旧的逻辑调色板句柄保存在 hPrevPalette
    hPrevPalette= SelectPalette(hDc, hPalette, FALSE);
    RealizePalette(hDc);
}
```

```
    //产生位图句柄
    hBitmap= CreateDIBitmap (hDc, (LPBITMAPINFOHEADER)lpImgData, (LONG) CBM-
                             INIT, (LPSTR)lpImgData+ sizeof(BITMAPINFOHEADER) +
                             NumColors* sizeof(RGBQUAD), (LPBITMAPINFO)lpImgData,
                             DIB- RGB- COLORS);

    //将原来的调色板(如果有的话)选入设备上下文句柄
    if(hPalette && hPrevPalette)
    {
        SelectPalette(hDc, hPrevPalette, FALSE);
        RealizePalette(hDc);
    }

    ReleaseDC(hWnd, hDc); //释放设备上下文
    GlobalUnlock(hImgData); //解锁内存区
    return TRUE; //成功返回
}
```

对上面的程序,要说明两点:

(1) 对于需要调色板的图,要想正确地显示,必须根据 bmp 文件,产生逻辑调色板。产生的方法是: ① 为逻辑调色板指针分配内存,大小为逻辑调色板结构(LOGPALETTE)长度加 NumColors 个 PALETTENTRY 大小(调色板的每一项都是一个 PALETTEENTRY 结构); ② 填写逻辑调色板结构的头 pPal- > palNumEntries= NumColors; pPal- > palVersion= 0x300; ③ 从文件中读取调色板的 RGB 值,填写到每一项中; ④ 产生逻辑调色板: hPalette= CreatePalette(pPal)。

(2) 产生位图(BITMAP)句柄,该项工作由函数 CreateDIBitmap 来完成。

```
hBitmap= CreateDIBitmap(hDc, (LPBITMAPINFOHEADER)lpImgData, (LONG)CBM- INIT,
    (LPSTR)lpImgData+ sizeof(BITMAPINFOHEADER) + NumColors* sizeof(RGBQUAD),
    (LPBITMAPINFO)lpImgData, DIB- RGB- COLORS);
```

CreateDIBitmap 的作用是产生一个和 Windows 设备无关的位图。该函数的第一项参数为设备上下文句柄,如果位图用到了调色板,要在调用 CreateDIBitmap 之前将逻辑调色板选入该设备上下文中,产生 hBitmap 后,再把原调色板选入该设备上下文中,并释放该上下文; 第二项为指向 BITMAPINFOHEADER 的指针; 第三项就用常量 CBM-INI, 不用考虑; 第四项为指向调色板的指针; 第五项为指向 BITMAPINFO(包括 BITMAPINFOHEADER、调色板及实际的图象数据)的指针; 第六项就用常量 DIB-RGB- COLORS, 不用考虑。

上面提到了设备上下文,相信编过 Windows 程序的读者对它并不陌生,这里再简单介绍一下。Windows 操作系统统一管理着诸如显示、打印等操作,将它们看作是一个个的设备,每一个设备都有一个复杂的数据结构来维护。所谓设备上下文就是指这个数据结构。然而,我们不能直接和这些设备上下文打交道,只能通过引用标识它的句柄(实际上是