程序功能说明书

程序功能说明：

1.BMP图像的获取 与 生成

2.图像的 水平镜像 和 垂直镜像变化

3.图像彩色转黑白

4.图像的平滑和锐化处理

5. 24位真彩色图 和 非真彩色图的自动判定

程序框架流程图

马赛克处理

转化后BMP文件的生成

锐化处理

平滑处理

转换黑白

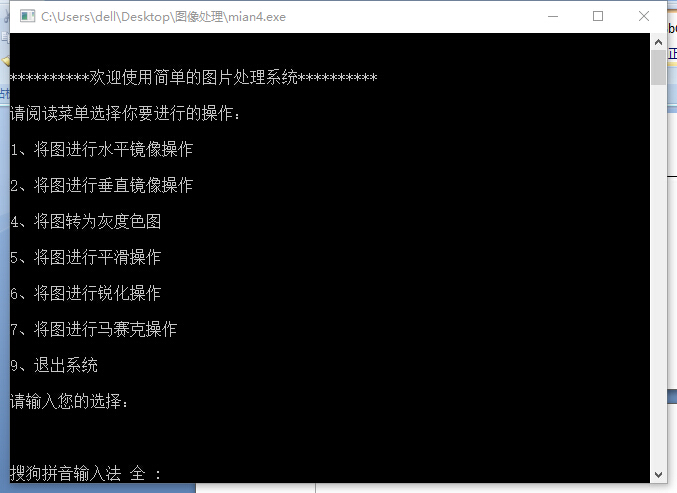
垂直镜像

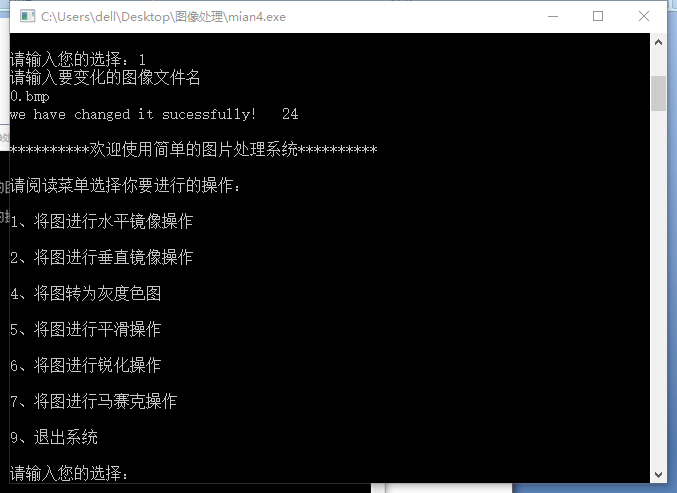
水平镜像

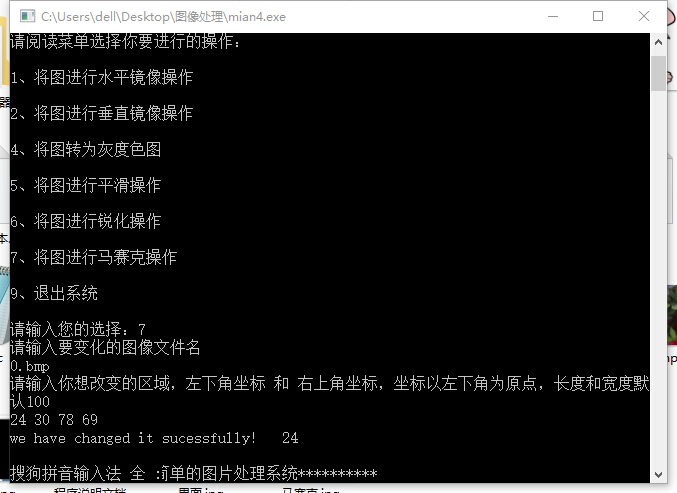
功能的选择

指定图片文件的读取

**功能截图**







**处理结果样例**



24色真彩图 0.bmp

效果图



水平镜像 垂直镜像



转灰度图 平滑

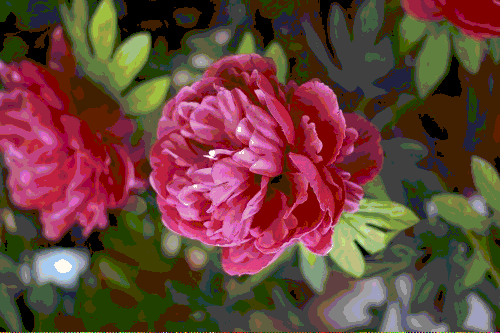
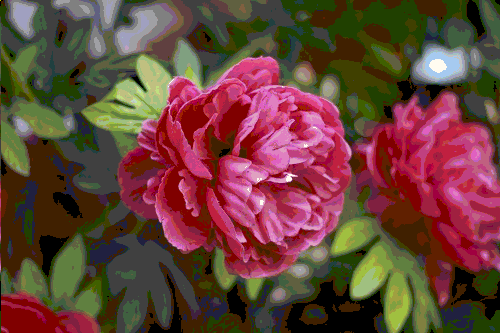


处理样例二

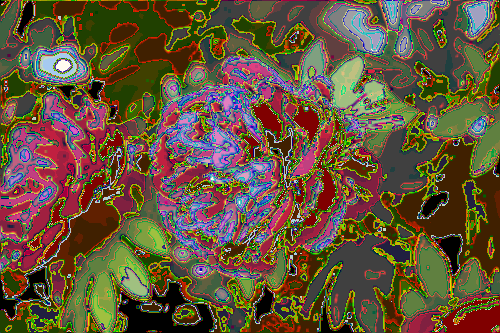


256色图

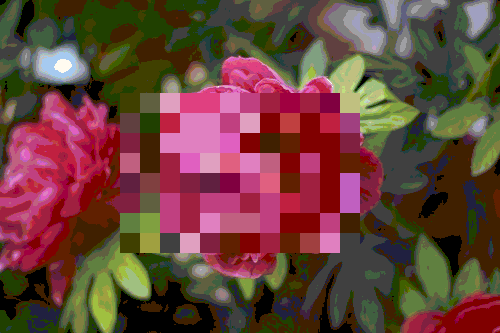
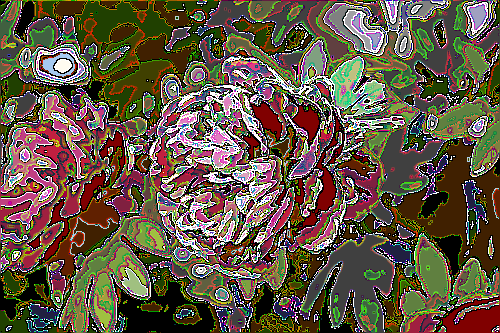
处理效果



水平镜像 垂直镜像



转化黑白 平滑



**三、本程序主要使用的的方法汇总**

void LevelOrvertical1();

void LevelOrvertical2();

void ColorToGray();

void ColorControl(int flag);

void Mosaic();

void Mosaicchange(int row,int col,int width)

void Mosaicchange2(int row,int col,int width);

void pInput();

void pOutput(char OutName[20],int length);

方法应用详细说明

1.LevelOrvertical()

水平镜像变化

将Imagedate位图数据（存储在PictureIn中）左右交交换

2. void LevelOrvertical2()

垂直镜像变化即上下像素点变化位置

3. void ColorToGray()灰度图变化

对真彩色：算每个像素点的灰度值求出，然后使RGB三色全部赋值为此灰度

对非真彩色图：直接修改对调色板RGB三色的数值为三者和的平均数

四、本程序使用的变量，数据结构

数据结构

由于头文件windows.h中已经有相关BMP文件所需的数据结构，所以无需定义

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)，各个域的说明略（见《数字图像处理编程入门》1章）

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;

这个结构的长度是固定的，为40个字节(LONG为32位整数)

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2biBitCount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颜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。下面针对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)，所以一个字节可以表示8个象素。

对于16色位图，用4位可以表示一个象素的颜色，所以一个字节可以表示2个象素。

对于256色位图，一个字节刚好可以表示1个象素。

对于真彩色图，三个字节才能表示1个象素

typedef struct tagPICSIZE {

int width;

int height;

int length;

int offset;

}PICSIZE;

Width：图像宽带字节数

Height：图像高度字节数

Length：图像大小字节数

Offset：前三部分字节数

全局变量的使用

BYTE PictureIn[M\*N];//读BMP文件第四部分ImageDate的存储数组

BYTE PictureOut[M\*N];//变化后BMP文件第四部分的ImageDate存储数组

FILE \*fInput; //输入文件

FILE \*fOutput; //输出文件

BITMAPFILEHEADER FileHeader; //图像第一部分存储变量

BITMAPINFOHEADER InfoHeader; //图像第二部分存储变量

RGBQUAD Quad[256]; //调色板存储的数组

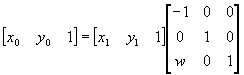
PICSIZE PiSize; //存储图像大小信息的结构体变量

**五、程序使用的核心算法和实现方法**

**镜像变化**

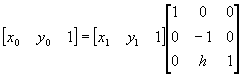
镜象的变换矩阵很简单。设原图宽为w，高为h，变换后，图的宽和高不变。

水平镜象的变化矩阵为：



(2.10)

垂直镜象的变化矩阵为：



程序代码实现方式：将左右像素点交换位置

if(InfoHeader.biBitCount==24) //如果是真彩色图（因为一个像素点三个字节）

for(j=0;j<PiSize.height;j++)

{

for(i=0;i<PiSize.width/3;i++) //将左右像素点镜像 交换

{ PictureOut[j\*PiSize.width+i\*3]=PictureIn[j\*PiSize.width+(PiSize.width/3-i)\*3]; PictureOut[j\*PiSize.width+i\*3+1]=PictureIn[j\*PiSize.width+(PiSize.width/3-i)\*3+1]; PictureOut[j\*PiSize.width+i\*3+2]=PictureIn[j\*PiSize.width+(PiSize.width/3-i)\*3+2];

}

}

else //如果不是真彩色图 左右像素点交换

{

for(j=0;j<PiSize.height;j++)

{

for(i=0;i<PiSize.width;i++)

{

PictureOut[j\*PiSize.width+i]=PictureIn[j\*PiSize.width+PiSize.width-i];

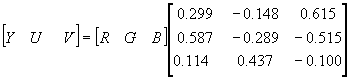
}

}

}

**灰度图变化**

YUV的颜色表示方法，在这种表示方法中，Y分量的物理含义就是亮度，它含了灰度图(grayscale)的所有信息，只用Y分量就完全能够表示出一幅灰度图来。YUV和RGB之间有着如下的对应关系：



我们利用上式，根据R、G、B的值求出Y值后，将R、G、B值都赋值成Y，就能表示出灰度图来，这就是彩色图转灰度图的原理。

先看看真彩图。我们知道真彩图不带调色板，每个象素用3个字节，表示R、G、B三个分量。所以处理很简单，根据R、G、B的值求出Y值后，将R、G、B值都赋值成Y，写入新图即可。

再来看看带调色板的彩色图，我们知道位图中的数据只是对应调色板中的一个索引值，我们只需要将调色板中的彩色变成灰度，形成新调色板，而位图数据不用动，就可以了。

程序实现方式：

if(InfoHeader.biBitCount==24)

for(j=0;j<PiSize.height;j++)

{

for(i=0;i<PiSize.width/3;i++)

{

int sum;//求每个像素点的灰度值

sum=PictureIn[j\*PiSize.width+i\*3]\*0.299+PictureIn[j\*PiSize.width+i\*3+1]\*0.578+PictureIn[j\*PiSize.width+i\*3+2]\*0.114;

PictureOut[j\*PiSize.width+i\*3]=sum;//将RGB三通道 全部赋值为 同一灰度值 实现照片变黑白

PictureOut[j\*PiSize.width+i\*3+1]=sum;

PictureOut[j\*PiSize.width+i\*3+2]=sum;

}

}

else

{

for(i=0;i<PiSize.length;i++)PictureOut[i]=PictureIn[i];

BYTE sum;

for(i=0;i<pow(2,InfoHeader.biBitCount);i++)//将调色板不同颜色转化为灰度

{

sum=(Quad[i].rgbBlue+Quad[i].rgbGreen+Quad[i].rgbRed)/3;

Quad[i].rgbBlue=sum;

Quad[i].rgbGreen=sum;

Quad[i].rgbRed=sum;

}

}

**平滑与锐化**

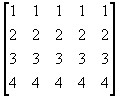
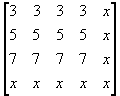
实现的是一种简单的低通滤波器(low pass filter)。在灰度连续变化的图象中，如果出现了与相邻象素的灰度相差很大的点，比如说一片暗区中突然出现了一个亮点，人眼能很容易觉察到。就象看老电影时，由于胶片太旧，屏幕上经常会出现一些亮斑。这种情况被认为是一种噪声。灰度突变在频域中代表了一种高频分量，低通滤波器的作用就是滤掉高频分量，从而达到减少图象噪声的目的。

为了方便地叙述上面所说的“将原图中的每一点的灰度和它周围八个点的灰度相加，然后除以9，作为新图中对应点的灰度”这一操作，我们采用如下的表示方法：

****

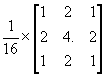
(3.1)

这种表示方法有点象矩阵，我们称其为模板(template)。中间的黑点表示中心元素，即，用哪个元素做为处理后的元素。例如[2. 1]表示将自身的2倍加上右边的元素作为新值，而[2 1.]表示将自身加上左边元素的2倍作为新值。

通常，模板不允许移出边界，所以结果图象会比原图小，例如模板是 image007，原图是 ，经过模板操作后的图象为 ；其中数字代表灰度，x表示边界上无法进行模板操作的点，通常的做法是复制原图的灰度，不进行任何处理。

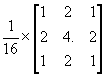
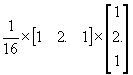
模板操作实现了一种邻域运算(Neighborhood Operation)，即某个象素点的结果灰度不仅和该象素灰度有关，而且和其邻域点的值有关。在以后介绍的细化算法中，我们还将接触到邻域运算。模板运算的数学涵义是一种卷积(或互相关)运算，你不需要知道卷积的确切含义，只要有这么一个概念就可以了。

模板运算在图象处理中经常要用到，可以看出，它是一项非常耗时的运算。以



(3.2)

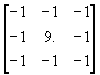
为例，每个象素完成一次模板操作要用9个乘法、8个加法、1个除法。对于一幅n×n(宽度×高度)的图象，就是9n2个乘法，8n2个加法和n2个除法，算法复杂度为O(n2)，这对于大图象来说，是非常可怕的。所以，一般常用的模板并不大，如3×3，4×4。有很多专用的图象处理系统，用硬件来完成模板运算，大大提高了速度。另外，可以设法将二维模板运算转换成一维模板运算，对速度的提高也是非常可观的。例如，(3.2)式可以分解成一个水平模板和一个垂直模板，即，

= image016× image018= 

锐化

锐化(sharpening)和平滑恰恰相反，它是通过增强高频分量来减少图象中的模糊，因此又称为高通滤波(high pass filter)。锐化处理在增强图象边缘的同时增加了图象的噪声。

常用的锐化模板是拉普拉斯(Laplacian)模板(见(3.4)式)，又是个数学家的名字，可见学好数学，走遍天下都不怕。



(3.4)

容易看出拉普拉斯模板的作法：先将自身与周围的8个象素相减，表示自身与周围象素的差别；再将这个差别加上自身作为新象素的灰度。可见，如果一片暗区出现了一个亮点，那么锐化处理的结果是这个亮点变得更亮，增加了图象的噪声。

程序实现方法

if(flag==1)//flag=1进行平滑处理 //24色真彩图处理代码

{

for(j=1;j<PiSize.height;j++)

{

for(i=3;i<(PiSize.width-3);i++)

{

int y=0;

y+=PictureIn[(j-1)\*PiSize.width+i-3]; //套用模板

y+=PictureIn[(j-1)\*PiSize.width+i];

y+=PictureIn[(j-1)\*PiSize.width+i+3];

y+=PictureIn[j\*PiSize.width+i-3];

y+=PictureIn[j\*PiSize.width+i];

y+=PictureIn[j\*PiSize.width+i+3];

y+=PictureIn[(j+1)\*PiSize.width+i-3];

y+=PictureIn[(j+1)\*PiSize.width+i];

y+=PictureIn[(j+1)\*PiSize.width+i+3];

PictureOut[j\*PiSize.width+i]=y/9;

}

}

pOutput("smooth.bmp",PiSize.length);

}

else//flag=0 进行锐化

{

for(j=1;j<PiSize.height-1;j++)

{

for(i=3;i<(PiSize.width-3);i++) //套用锐化功能模板，使用拉普拉斯模板

{

int sum;

sum=9\*PictureIn[j\*PiSize.width+i];

sum-=PictureIn[(j-1)\*PiSize.width+(i-3)];

sum-=PictureIn[(j-1)\*PiSize.width+i];

sum-=PictureIn[(j-1)\*PiSize.width+i+3];

sum-=PictureIn[j\*PiSize.width+i-3];

sum-=PictureIn[j\*PiSize.width+i+3];

sum-=PictureIn[(j+1)\*PiSize.width+(i-3)];

sum-=PictureIn[(j+1)\*PiSize.width+i];

sum-=PictureIn[(j+1)\*PiSize.width+i+3];

if(sum>255)

PictureOut[j\*PiSize.width+i]=255;

else if(sum<0)

PictureOut[j\*PiSize.width+i]=0;

else

PictureOut[j\*PiSize.width+i]=sum;

}

}

pOutput("sharpening.bmp",PiSize.length);

}

}

马赛克

设定马赛克方块大小为20\*20像素的方格，将图像划分为20\*20方格，将需要打上马赛克的地方的方格颜色变为这个方块的平均色调 或者中间色调。

程序实现方法：

for(j=PiSize.height\*ly/2000;j<PiSize.height\*ry/2000;j++)

{

for(i=PiSize.width\*lx/6000;i<PiSize.width\*rx/6000;i++)

{

Mosaicchange(j,i,PiSize.width); //计算每一个方块颜色的平均值，并将方块中像素点色彩全部赋值为 平均值

}

}

24位真彩图 方块平均色彩颜色计算获取

void Mosaicchange(int row,int col,int width)

{

int i,j;

int averRed=0;

int averBlue=0;

int averGreen=0;

int sum=0;

for(i=row\*20;i<row\*20+20;i++)

{

for(j=col\*20;j<col\*20+20;j++)

{

averRed+=PictureIn[i\*width+j\*3];

averGreen+=PictureIn[i\*width+j\*3+1];

averBlue+=PictureIn[i\*width+j\*3+2];

++sum;

}

}

averRed/=sum;

averGreen/=sum;

averBlue/=sum;

for(i=row\*20;i<row\*20+20;i++)

{

for(j=col\*20;j<col\*20+20;j++)

{

PictureOut[i\*width+j\*3]=averRed;

PictureOut[i\*width+j\*3+1]=averGreen;

PictureOut[i\*width+j\*3+2]=averBlue;

}

}

}