

# 浙江大学实验报告

专业：计算机科学与技术  
姓名：龙永奇  
学号：3220105907  
日期：2023/11/22

课程名称：图像信息处理 指导老师：宋明黎 成绩：

实验名称：图像的几何变换

## 一、实验目的和要求

1. 图像的平移 (Translation)
2. 图像的旋转 (Rotation)
3. 图像的缩放 (Scale)
4. 图像的剪切 (Shear)
5. 图像的镜像 (Mirror)

## 二、实验内容和原理

### ➤ 图像几何变换基本概念与原理

图像的几何变换，即图像的空间变换，本质就是将原图像的像素坐标通过一定变换算法转换到新的坐标上。图像的几何变换原理就是建立原图像与目标图像的映射关系，目标就是确定变换关系以及变换参数，常见的变换形式有平移、旋转、缩放、剪切以及镜像等，也就是本次实验的目标。

由于原图像的坐标为整数坐标，但是经过不同形式的坐标映射后，可能会将某些像素映射到非整数坐标，因此需要在变换的过程中使用插值算法。

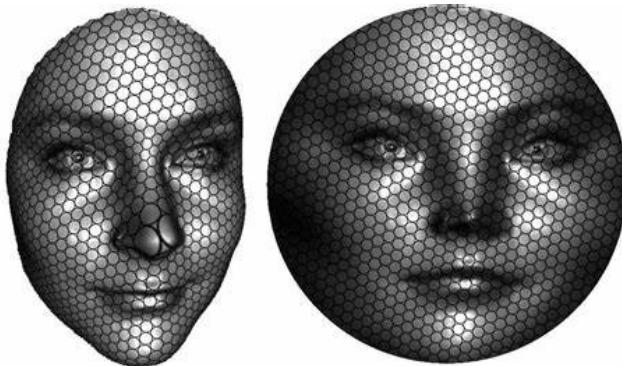


图 2.1 图像的映射

## ➤ 插值算法

插值算法即在已知的离散数据点基础上，在一定范围内求出新数据点的方法，是一种几何变换工具，主要的插值算法有最邻近插值 (Nearest Neighbor)、线性插值 (Linear interpolation) 以及 RBF 插值 (RBF interpolation)。

### 1. 最邻近插值 (Nearest Neighbor)

最邻近插值，即输出像素的灰度值等于距离其所映射到位置最近的输入像素的灰度值，因此需要先对图像进行几何变换，计算出  $P'$  对应的原位置  $P$ 。由于  $P$  的坐标通常不是整数，因此需要寻找  $P$  点最接近的像素  $Q$ ，将  $Q$  像素作为转换后的图像的  $P'$  点的像素，计算过程如下：

$$(x', y') \xrightarrow{\text{inverse transformation}} (x, y) \xrightarrow{\text{rounding operation}} (x_{int}, y_{int}) \\ \xrightarrow{\text{assign value}} I_{new}(x', y') = I_{old}(x_{int}, y_{int})$$

图 2.2 计算过程

效果如下：



图 2.3 效果图

### 2. 线性插值 (Linear interpolation)

- 1 维线性插值

$$g_3 = \frac{g_2 - g_1}{x_2 - x_1} (x_3 - x_1) + g_1$$

其中  $g_3$  为  $x_3$  的灰度值， $g_1$  和  $g_2$  为  $x_1$  和  $x_2$  的灰度值

- 2 维线性插值

定义双线性方程  $g(x, y) = ax + by + cxy + d$

分别将 A、B、C、D 四点的位置和灰度代入方程，得到方程组

解方程组，得到  $a$ 、 $b$ 、 $c$ 、 $d$  四个系数

将  $P$  点位置带入方程即可得到灰度

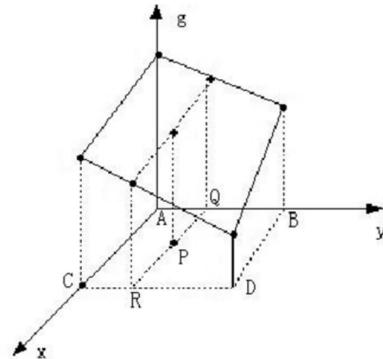


图 2.4 2 维线性插值

2 维线性插值的本质和 1 维情况相同，在代码实现的过程中可以分别计算 A、B 和 C、D 的线性插值，然后再进行一次线性插值即可得到 P 点灰度值，因此只需要先计算每行的插值，再计算每列的插值，即可对图像进行插值运算。

➤ 以下是图像的变换算法

## 1. 图像平移 (Translation)

图像平移变换的原理非常简单，即根据图像当前的位置  $(x_0, y_0)$ ，以及平移量  $(\Delta x, \Delta y)$ ，得到图像新的位置  $(x_0 + \Delta x, y_0 + \Delta y)$ ，其数学表达式如下：

$$\begin{cases} x' = x + x_0 \\ y' = y + y_0 \end{cases} \quad \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_0 \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

图 2.5 图像平移数学表达

由于图像被平移到新的位置，原来的位置空出，此时可以选择删除空余位置，也可直接增大图像大小，例如下图：

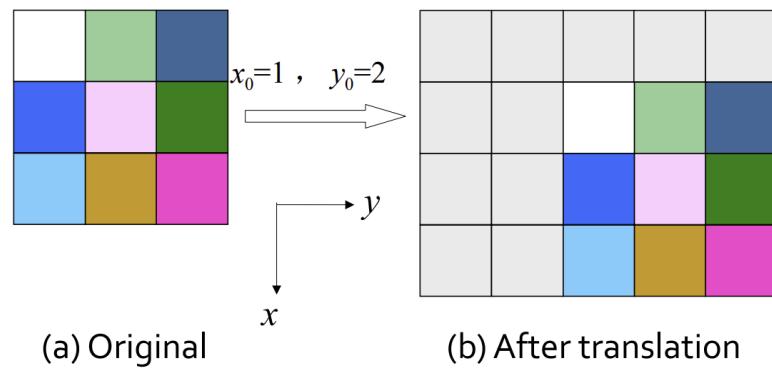


图 2.6 图像的平移

本次实验使用第二种方法。

## 2. 图像的旋转 (Rotation)

图像的旋转原理即以图像某一点为旋转轴，将图像整体旋转固定角度。由于旋转过后图像的参考坐标系发生了变化，其垂直、水平对称轴以及宽度、高度和具体的旋转角度有关，因此需要使用三角函数非线性运算，才能避免图像变形，其数学表达式为：

$$\begin{cases} x' = x \cos\theta - y \sin\theta \\ y' = x \sin\theta + y \cos\theta \end{cases} \quad \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

图 2.7 图像的旋转数学表达

旋转后的结果如下：

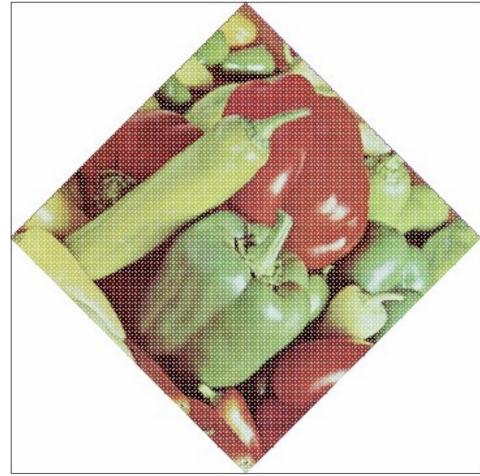


图 2.8 图像的旋转 1

可以发现一个很明显的问题，在旋转变换后，图像中会出现很多空洞，这是因为在变换过程中会产生浮点数坐标（非整数），但是图像的储存数据是离散型整数，因此需要使用插值法对图像进行填充，如下图所示：



图 2.9 图像的旋转 2

此外由于在旋转的过程中是以某一旋转中心为坐标原点，因此需要新的变换对其进行平移。

### 3. 图像的缩放 (Scale)

图像的缩放就是将原图像的尺寸进行缩小或增大，本质上就是改变原图像的像素个数，通过增减某些像素点实现，数学表达式如下：

$$\begin{cases} x' = cx \\ y' = dy \end{cases} \quad \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} c & 0 & 0 \\ 0 & d & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

图 2.10 图像的缩放数学表达

其中  $c$  为水平缩放系数， $d$  为垂直缩放系数，其具体含义为：

- 将图像沿  $x$  方向缩放  $c$  倍， $y$  方向缩放  $d$  倍
- 当  $c = d$  时，图像为等比缩放，否则为非等比缩放，会导致图像变形，例如：



图 2.11 图像的缩放

其中左图为原图像，中图为等比缩放，右图为非等比缩放

- 缩小：按一定间隔选取行和列的像素构成缩小后的图像
- 放大：使用插值的方法填补新图像出现的空行、列，但是会有马赛克(mosaics)现象

### 4. 图像的剪切 (Shear)

图像的剪切变换是场景在平面上的非垂直投影效果，可以将原图像的任意一边拉长，形成各种平行四边形，可分别对  $x$ 、 $y$  坐标进行剪切，也可以同时剪切，其数学表达如下：

Shear on  $x$  axis

$$\begin{cases} a(x, y) = x + d_x y \\ b(x, y) = y \end{cases}$$

Shear on  $y$  axis

$$\begin{cases} a(x, y) = x \\ b(x, y) = y + d_y x \end{cases}$$

图 2.12 图像的剪切

其中  $d_x$  与  $d_y$  为剪切变换参数，具体效果为：



图 2.13 图像的剪切变换

## 5. 图像的镜像 (Mirror)

图像的镜像变换就是以图像的垂直中线或水平中线为轴，将两部分的像素进行调换，可通过将图像绕 x 轴或 y 轴翻转实现，镜像变换的公式如下：

$$\text{Flip around } x \text{ axis:} \quad \begin{cases} x' = x \\ y' = -y \end{cases}$$

$$\text{Flip around } y \text{ axis:} \quad \begin{cases} x' = -x \\ y' = y \end{cases} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

图 2.14 图像的镜像数学表达

当  $s_x = 1, s_y = -1$  时，绕 x 轴旋转；当  $s_x = -1, s_y = 1$  时，绕 x 轴旋转，具体效果为：

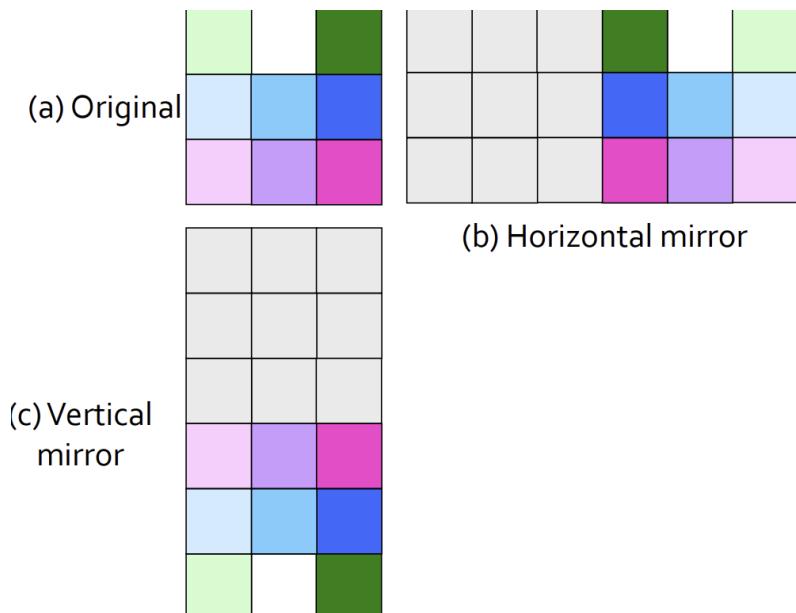


图 2.15 图像的镜像变换

### 三、实验步骤与分析

#### ➤ 图像的读取

本次实验使用变量 Org 储存原图像，Ans 储存变换后的图像，分别将两图像传入变换函数中，在函数内进行图像的信息复制以及输出，读取函数如下：

```
FILE *fp;
FILESTRUCT Org, Ans; // 原图像 Org, 变换后的图像 Ans
int BMPSize, BMPheight, BMPwidth;
fp = fopen("zjz.bmp", "rb"); // 读取本目录下的 bmp 文件
if (!fp) printf("憨憨丢掉了:(\n");
else printf("憨憨在这里:)\n";
fread(&(Org.FH), sizeof(FILE_HEAD), 1, fp);
fread(&(Org.IH), sizeof(INF_HEAD), 1, fp);
if (!Org.IH.biSizeImage) Org.IH.biSizeImage = Org.FH.bfSize - Org.FH.bfOffBits;
BMPheight = Org.IH.biHeight;
BMPwidth = Org.IH.biWidth;
BMPSize = Org.IH.biSizeImage;
// 给八位灰度图加上调色盘，本次实验分别对 24 位和 8 位进行图像几何变换
if(Org.IH.biBitCount == 8){
    for(int i = 0; i < 256; i++){
        Org.Color[i].rgbBlue = Org.Color[i].rgbGreen = Org.Color[i].rgbRed = i;
        Ans.Color[i].rgbBlue = Ans.Color[i].rgbGreen = Ans.Color[i].rgbRed = i;
    }
}
fseek(fp, Org.FH.bfOffBits, SEEK_SET);
Org.pix_val = (unsigned char *)malloc(sizeof(unsigned char) * BMPSize);
fread(Org.pix_val, BMPSize * sizeof(unsigned char), 1, fp);
fclose(fp);
// 平移
TrAnslate(&Org, &Ans, BMPwidth/2, BMPheight/2);
free(Ans.pix_val);
// 旋转
Rotate(&Org, &Ans, 45);
free(Ans.pix_val);
// 缩放
Scale(&Org, &Ans, 1.7, 1.7);
free(Ans.pix_val);
// 剪切
Shear(&Org, &Ans, 0, 0.6);
free(Ans.pix_val);
// 镜像
Mirror(&Org, &Ans, 'y');
free(Ans.pix_val);
```

## 1. 图像的平移

需要注意的一点是，由于平移后会有空余的部分，因此函数内先对整个 Ans 进行初始化，将每个像素点设置为白色作为背景色，具体实现如下：

```
void TrAnslate(FILESTRUCT *Org, FILESTRUCT *Ans, int del_x, int del_y){
    memcpy(&(Ans->FH), &(Org->FH), sizeof(FILE_HEAD));
    memcpy(&(Ans->IH), &(Org->IH), sizeof(INF_HEAD));
    // 计算新图像大小
    int new_BMPwidth = Org->IH.biWidth + del_x;
    int new_BMPheight = Org->IH.biHeight + del_y;
    // 每行字节
    int Org_row_byte = (Org->IH.biBitCount / 8 * Org->IH.biWidth + 3) / 4 * 4;
    int Ans_row_byte = (Ans->IH.biBitCount / 8 * new_BMPwidth + 3) / 4 * 4;
    Ans->IH.biWidth = new_BMPwidth;
    Ans->IH.biHeight = new_BMPheight;
    Ans->IH.biSizeImage = new_BMPheight * Ans_row_byte;
    Ans->FH.bfSize = Ans->IH.biSizeImage + Ans->FH.bfOffBits;
    Ans->pix_val = (unsigned char *)malloc(Ans->IH.biSizeImage *
    sizeof(unsigned char));
    // 设置新生成的背景颜色为白色（平移后空出来的）
    for(int i = 0; i < Ans->IH.biSizeImage; i++) Ans->pix_val[i] = 255;
    if(Ans->IH.biBitCount == 8){
        for(int i = 0; i < Org->IH.biHeight; i++){
            for(int j = 0; j < Org->IH.biWidth; j++){
                Ans->pix_val[(i + del_y) * Ans_row_byte + (j + del_x)] =
                Org->pix_val[i * Org_row_byte + j];
            }
        }
        Output(Ans, "TrAnslate_Gray.bmp");
    }else{ // 24 位彩色图
        for(int i = 0; i < Org->IH.biHeight; i++){
            for(int j = 0; j < Org->IH.biWidth; j++){
                Ans->pix_val[(i + del_y) * Ans_row_byte + (j + del_x) * 3] =
                Org->pix_val[i * Org_row_byte + j * 3];
                Ans->pix_val[(i + del_y) * Ans_row_byte + (j + del_x) * 3 +
                1] = Org->pix_val[i * Org_row_byte + j * 3 + 1];
                Ans->pix_val[(i + del_y) * Ans_row_byte + (j + del_x) * 3 +
                2] = Org->pix_val[i * Org_row_byte + j * 3 + 2];
            }
        }
        Output(Ans, "TrAnslate_Color.bmp");
    }
}
```

## 2. 图像的旋转

由于图像的旋转涉及到坐标系的变换，因此首先需要计算出新的图像大小以及新坐标系下各像素点对应的坐标。

点 P 以坐标原点 O 为旋转中心，逆时针旋转角度  $\beta$  后得到点 Q：

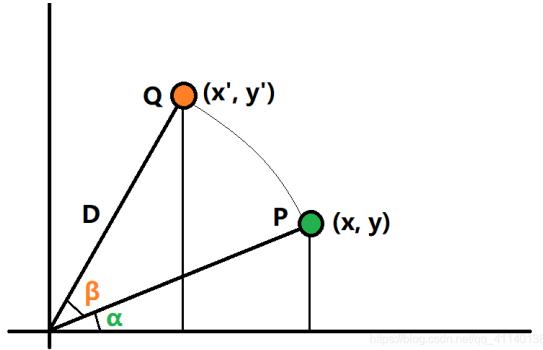


图 3.1 坐标系旋转

对应的坐标如下：

$$y' = D \sin(\alpha + \beta) = D(\cos\alpha \sin\beta + \sin\alpha \cos\beta) = x \sin\beta + y \cos\beta$$

$$x' = D \cos(\alpha + \beta) = D(\cos\alpha \cos\beta - \sin\alpha \sin\beta) = x \cos\beta - y \sin\beta$$

接下来我们计算新图像的大小：

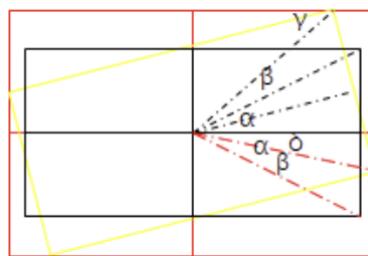


图 3.2 图像的旋转

在图 3.2 中，原始图为黑色矩形，假设旋转  $\beta$  之后得到黄色的矩形，若要完全显示这个图像就需要红色矩形的空间

$$\sin\alpha = \frac{height}{2R} \quad \cos\alpha = \frac{width}{2R}$$

$$\sin(\alpha + \beta) = \frac{newHeight}{2R}$$

$$newHeight = height \times \cos\beta + width \times \sin\beta$$

$$newWidth = height \times \sin\beta + width \times \cos\beta$$

在插值方法的选取上使用了最邻近插值，通过逆变换寻找原图像中的点，若没有对应的点则使用四舍五入找到临近像素值，赋值给 Ans，如果回溯的点不在原图范围内，则设置为白色，具体实

现如下：

```
void Rotate(FILESTRUCT *Org, FILESTRUCT *Ans, double angle){
    memcpy(&(Ans->FH), &(Org->FH), sizeof(FILE_HEAD));
    memcpy(&(Ans->IH), &(Org->IH), sizeof(INF_HEAD));
    angle *= pi / 180.0;
    int new_BMPwidth = Org->IH.biWidth * fabs(cos(angle)) + Org->IH.biHeight
* fabs(sin(angle));
    int new_BMPheight = Org->IH.biWidth * fabs(sin(angle)) +
Org->IH.biHeight * fabs(cos(angle));
    // 坐标系变换
    int center_x = Org->IH.biWidth / 2;
    int center_y = Org->IH.biHeight / 2;
    int new_center_x = center_x * cos(angle) - center_y * sin(angle);
    int new_center_y = center_x * sin(angle) + center_y * cos(angle);
    // 计算变换产生的偏移量
    int del_x = new_BMPwidth / 2 - new_center_x;
    int del_y = new_BMPheight / 2 - new_center_y;
    int Org_row_byte = (Org->IH.biBitCount / 8 * Org->IH.biWidth + 3) / 4 *
4;
    int Ans_row_byte = (Ans->IH.biBitCount / 8 * new_BMPwidth + 3) / 4 * 4;
    Ans->IH.biWidth = new_BMPwidth;
    Ans->IH.biHeight = new_BMPheight;
    Ans->IH.biSizeImage = new_BMPheight * Ans_row_byte;
    Ans->FH.bfSize = Ans->IH.biSizeImage + Ans->FH.bfOffBits;
    Ans->pix_val = (unsigned char *)malloc(Ans->IH.biSizeImage *
sizeof(unsigned char));
    if(Ans->IH.biBitCount == 8){
        for(int i = 0; i < new_BMPheight; i++){
            for(int j = 0; j < new_BMPwidth; j++){
                // 计算当前像素点在原图像中的位置
                int Org_x = round((j - del_x) * cos(angle) + (i - del_y) *
sin(angle));
                int Org_y = round((del_x - j) * sin(angle) + (i - del_y) *
cos(angle));
                // 如果在原图像中这个点出界，则设为白色
                if(Org_x < 0 || Org_x >= Org->IH.biWidth || Org_y < 0 ||
Org_y >= Org->IH.biHeight){
                    Ans->pix_val[i * Ans_row_byte + j] = 255;
                }else{
                    Ans->pix_val[i * Ans_row_byte + j] = Org->pix_val[Org_y
* Org_row_byte + Org_x];
                }
            }
        }
    }
}
```

```

        Output(Ans, "Rotate_Gray.bmp");
    }else{
        for(int i = 0; i < new_BMPheight; i++){
            for(int j = 0; j < new_BMPwidth; j++){
                int Org_x = round((j - del_x) * cos(angle) + (i - del_y) * sin(angle));
                int Org_y = round((del_x - j) * sin(angle) + (i - del_y) * cos(angle));
                if(Org_x < 0 || Org_x >= Org->IH.biWidth || Org_y < 0 || Org_y >= Org->IH.biHeight){
                    Ans->pix_val[i * Ans_row_byte + j * 3] = 255;
                    Ans->pix_val[i * Ans_row_byte + j * 3 + 1] = 255;
                    Ans->pix_val[i * Ans_row_byte + j * 3 + 2] = 255;
                }else{
                    Ans->pix_val[i * Ans_row_byte + j * 3] =
Org->pix_val[Org_y * Org_row_byte + Org_x * 3];
                    Ans->pix_val[i * Ans_row_byte + j * 3 + 1] =
Org->pix_val[Org_y * Org_row_byte + Org_x * 3 + 1];
                    Ans->pix_val[i * Ans_row_byte + j * 3 + 2] =
Org->pix_val[Org_y * Org_row_byte + Org_x * 3 + 2];
                }
            }
        }
        Output(Ans, "Rotate_Color.bmp");
    }
}

```

### 3. 图像的缩放

仍然使用最邻近插值法，对 Ans 中的像素点进行溯源，其中 dx、dy 为 x、y 方向上的缩放系数，实现如下：

```

void Scale(FILESTRUCT *Org, FILESTRUCT *Ans, double dx, double dy){
    memcpy(&(Ans->FH), &(Org->FH), sizeof(FILE_HEAD));
    memcpy(&(Ans->IH), &(Org->IH), sizeof(INF_HEAD));
    // 缩放后图像大小，四舍五入
    int new_BMPwidth = round(Org->IH.biWidth * dx);
    int new_BMPheight = round(Org->IH.biHeight * dy);
    int Org_row_byte = (Org->IH.biBitCount / 8 * Org->IH.biWidth + 3) / 4 *
4;
    int Ans_row_byte = (Ans->IH.biBitCount / 8 * new_BMPwidth + 3) / 4 * 4;
    Ans->IH.biWidth = new_BMPwidth;
    Ans->IH.biHeight = new_BMPheight;
    Ans->IH.biSizeImage = new_BMPheight * Ans_row_byte;
    Ans->FH.bfSize = Ans->IH.biSizeImage + Ans->FH.bfOffBits;
}

```

```

Ans->pix_val = (unsigned char *)malloc(Ans->IH.biSizeImage *
sizeof(unsigned char));
if(Ans->IH.biBitCount == 8){
    for(int i = 0; i < new_BMPheight; i++){
        for(int j = 0; j < new_BMPwidth; j++){
            int Org_x = round(j / dx);
            int Org_y = round(i / dy);
            Ans->pix_val[i * Ans_row_byte + j] = Org->pix_val[Org_y *
Org_row_byte + Org_x];
        }
    }
    Output(Ans, "Scale_Gray.bmp");
}else{
    for(int i = 0; i < new_BMPheight; i++){
        for(int j = 0; j < new_BMPwidth; j++){
            int Org_x = round(j / dx);
            int Org_y = round(i / dy);
            Ans->pix_val[i * Ans_row_byte + j * 3] = Org->pix_val[Org_y *
* Org_row_byte + Org_x * 3];
            Ans->pix_val[i * Ans_row_byte + j * 3 + 1] =
Org->pix_val[Org_y * Org_row_byte + Org_x * 3 + 1];
            Ans->pix_val[i * Ans_row_byte + j * 3 + 2] =
Org->pix_val[Org_y * Org_row_byte + Org_x * 3 + 2];
        }
    }
    Output(Ans, "Scale_Color.bmp");
}
}

```

#### 4. 图像的剪切

其中 dx、dy 为 x、y 方向上的剪切系数，实现如下：

```

void Shear(FILESTRUCT *Org, FILESTRUCT *Ans, double dx, double dy){
    memcpy(&(Ans->FH), &(Org->FH), sizeof(FILE_HEAD));
    memcpy(&(Ans->IH), &(Org->IH), sizeof(INF_HEAD));
    // 剪切后图像大小
    int new_BMPwidth = Org->IH.biWidth + round(Org->IH.biHeight * dx);
    int new_BMPheight = Org->IH.biHeight + round(Org->IH.biWidth * dy);
    int Org_row_byte = (Org->IH.biBitCount / 8 * Org->IH.biWidth + 3) / 4 *
4;
    int Ans_row_byte = (Ans->IH.biBitCount / 8 * new_BMPwidth + 3) / 4 * 4;
    Ans->IH.biWidth = new_BMPwidth;
    Ans->IH.biHeight = new_BMPheight;
    Ans->IH.biSizeImage = new_BMPheight * Ans_row_byte;
    Ans->FH.bfSize = Ans->IH.biSizeImage + Ans->FH.bfOffBits;
}

```

```

Ans->pix_val = (unsigned char *)malloc(Ans->IH.biSizeImage *
sizeof(unsigned char));
if(Ans->IH.biBitCount == 8){
    for(int i = 0; i < new_BMPheight; i++){
        for(int j = 0; j < new_BMPwidth; j++){
            int Org_x = j - round(i * dx);
            int Org_y = i - round(j * dy);
            if(Org_x < 0 || Org_x >= Org->IH.biWidth || Org_y < 0 ||
Org_y >= Org->IH.biHeight)
                Ans->pix_val[i * Ans_row_byte + j] = 255;
            else
                Ans->pix_val[i * Ans_row_byte + j] = Org->pix_val[Org_y
* Org_row_byte + Org_x];
        }
    }
    Output(Ans, "Shear_Gray.bmp");
}else{
    for(int i = 0; i < new_BMPheight; i++){
        for(int j = 0; j < new_BMPwidth; j++){
            int Org_x = j - round(i * dx);
            int Org_y = i - round(j * dy);
            if(Org_x < 0 || Org_x >= Org->IH.biWidth || Org_y < 0 ||
Org_y >= Org->IH.biHeight){
                Ans->pix_val[i * Ans_row_byte + j * 3] = 255;
                Ans->pix_val[i * Ans_row_byte + j * 3 + 1] = 255;
                Ans->pix_val[i * Ans_row_byte + j * 3 + 2] = 255;
            }else{
                Ans->pix_val[i * Ans_row_byte + j * 3] =
Org->pix_val[Org_y * Org_row_byte + Org_x * 3];
                Ans->pix_val[i * Ans_row_byte + j * 3 + 1] =
Org->pix_val[Org_y * Org_row_byte + Org_x * 3 + 1];
                Ans->pix_val[i * Ans_row_byte + j * 3 + 2] =
Org->pix_val[Org_y * Org_row_byte + Org_x * 3 + 2];
            }
        }
    }
    Output(Ans, "Shear_Color.bmp");
}

```

## 5. 图像的镜像

Flag 处输入 x 则为 x 轴镜像，输入 y 则为 y 轴镜像，实现如下：

```

void Mirror(FILESTRUCT *Org, FILESTRUCT *Ans, char flag){
    memcpy(&(Ans->FH), &(Org->FH), sizeof(FILE_HEAD));
    memcpy(&(Ans->IH), &(Org->IH), sizeof(INF_HEAD));

```

```

// 镜像后图像大小不变
int new_BMPwidth = Org->IH.biWidth;
int new_BMPheight = Org->IH.biHeight;
int Org_row_byte = (Org->IH.biBitCount / 8 * Org->IH.biWidth + 3) / 4 *
4;
int Ans_row_byte = Org_row_byte;
Ans->pix_val = (unsigned char *)malloc(Ans->IH.biSizeImage *
sizeof(unsigned char));
if(Ans->IH.biBitCount == 8){
    for(int i = 0; i < new_BMPheight; i++){
        for(int j = 0; j < new_BMPwidth; j++){
            int Org_y, Org_x;
            if(flag == 'x'){
                Org_x = j;
                Org_y = Org->IH.biHeight - i;
            }else{
                Org_x = Org->IH.biWidth - j;
                Org_y = i;
            }
            if(Org_x < 0 || Org_x >= Org->IH.biWidth || Org_y < 0 ||
Org_y >= Org->IH.biHeight)
                Ans->pix_val[i * Ans_row_byte + j] = 255;
            else
                Ans->pix_val[i * Ans_row_byte + j] = Org->pix_val[Org_y
* Org_row_byte + Org_x];
        }
    }
    Output(Ans, "Mirror_Gray.bmp");
}else{
    for(int i = 0; i < new_BMPheight; i++){
        for(int j = 0; j < new_BMPwidth; j++){
            int Org_y, Org_x;
            if(flag == 'x'){
                Org_x = j;
                Org_y = Org->IH.biHeight - i;
            }else{
                Org_x = Org->IH.biWidth - j;
                Org_y = i;
            }
            if(Org_x < 0 || Org_x >= Org->IH.biWidth || Org_y < 0 ||
Org_y >= Org->IH.biHeight){
                Ans->pix_val[i * Ans_row_byte + j * 3] = 255;
                Ans->pix_val[i * Ans_row_byte + j * 3 + 1] = 255;
                Ans->pix_val[i * Ans_row_byte + j * 3 + 2] = 255;
            }
        }
    }
}

```

```
        }else{
            Ans->pix_val[i * Ans_row_byte + j * 3] =
Org->pix_val[Org_y * Org_row_byte + Org_x * 3];
            Ans->pix_val[i * Ans_row_byte + j * 3 + 1] =
Org->pix_val[Org_y * Org_row_byte + Org_x * 3 + 1];
            Ans->pix_val[i * Ans_row_byte + j * 3 + 2] =
Org->pix_val[Org_y * Org_row_byte + Org_x * 3 + 2];
        }
    }
}
Output(Ans, "Mirror_Color.bmp");
}
```

## 四、实验环境及运行方法

### 1. 实验环境

系统 Windows11 编译器 gcc 10.3.0x86\_mingw32

### 2. 运行方法

在文件夹中，Lab4.c 为源文件，运行代码，如果出现“憨憨在这里: )”说明读取图片文件成功，否则会出现“憨憨丢掉了:(”。

随后根据读取的图象是彩色或者灰度程序会分别输出：

- 平移原图像一半大小的灰度、彩色图像：Translate\_Gray.bmp, Translate\_Color.bmp
- 旋转 45° 灰度、彩色图像：Rotate\_Gray.bmp, Rotate\_Color.bmp
- 缩放 1.7 倍灰度、彩色图像：Scale\_Gray.bmp, Scale\_Color.bmp
- 剪切 y 方向 0.6 倍灰度、彩色图像：Shear\_Gray.bmp, Shear\_Color.bmp
- Y 方向镜像灰度、彩色图像：Mirror\_Gray.bmp, Mirror\_Color.bmp,

## 五、实验结果展示

### 1. 原图像



2. 平移原图像一半



3. 旋转 45°



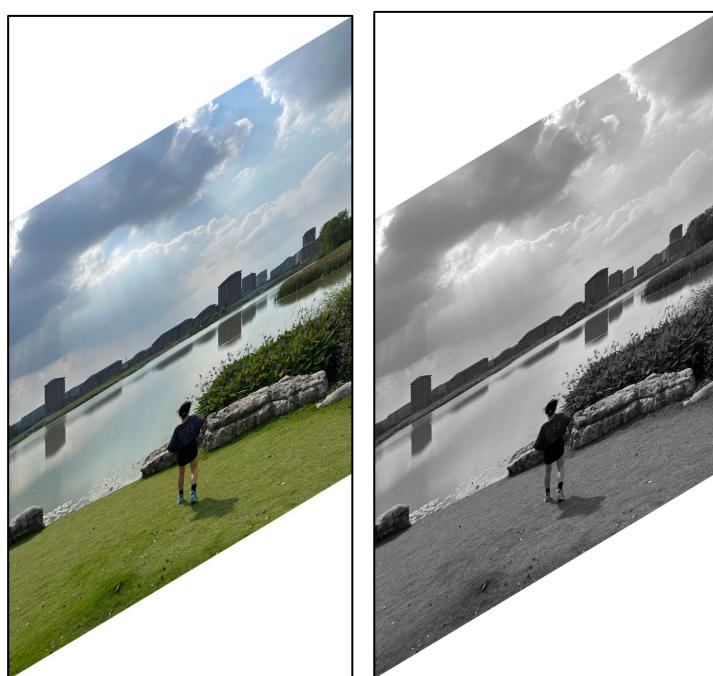
4. 旋转 90°



5. 缩放 1.7 倍



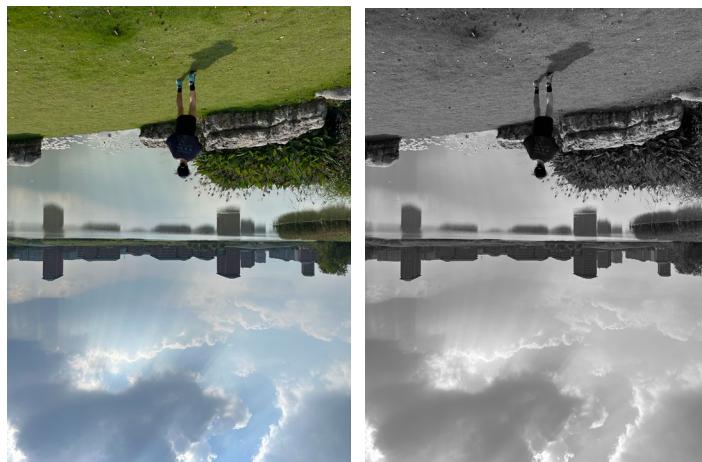
6. Y 方向剪切 0.6 倍



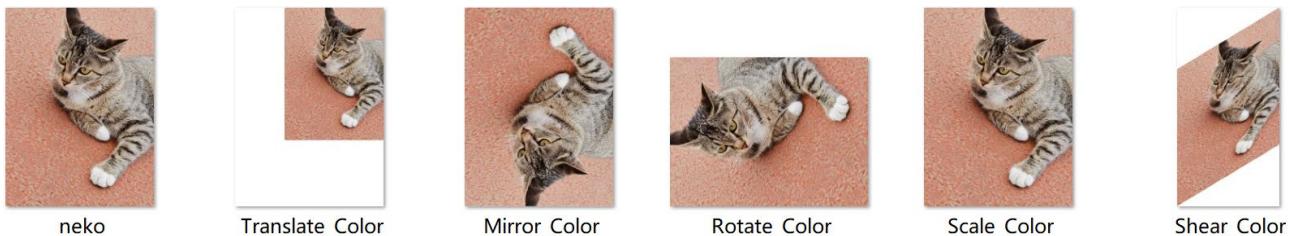
## 7. Y 轴镜像



## 8. X 方向镜像



### ➤ 测试样例一：动物



### ➤ 测试样例二：人像



## 六、心得体会

本次实验涉及到图像的几何变换，由于是对图像本身的几何空间性质进行改变，因此和之前改变图像的像素参数实验不同：

- 需要根据具体的变换形式增减图像大小，否则可能会出现图像的缺失
- 需要建立起坐标之间的映射
- 在变换过后可能由于图像大小的改变出现空洞，需要使用不同的插值方法进行填补

此外，在课程中学习了解到，图像的几何变换在消除几何失真方面有重要的应用，在两副相同的图像可能会由于透视、角度等原因造成图像失真，为图像识别带来干扰，因此可以通过图像的几何变换消除这类几何失真。与此同时，几何变换在预处理、归一化也起到重要作用。