

浙江大学实验报告

专业：计算机科学与技术
姓名：张祎迪
学号：3220102157
日期：2023/11/15

课程名称：图像信息处理 指导老师：宋明黎 成绩：
实验名称：图像几何变换

一、实验目的和要求

- 通过编写简单的几何变换程序，加深对几何变换原理的理解，掌握基本的图形学编程技能，以及通过实际操作掌握几何变换的应用。
- 实现平移、旋转、缩放、剪切、镜像等基本几何变换的图形学算法。

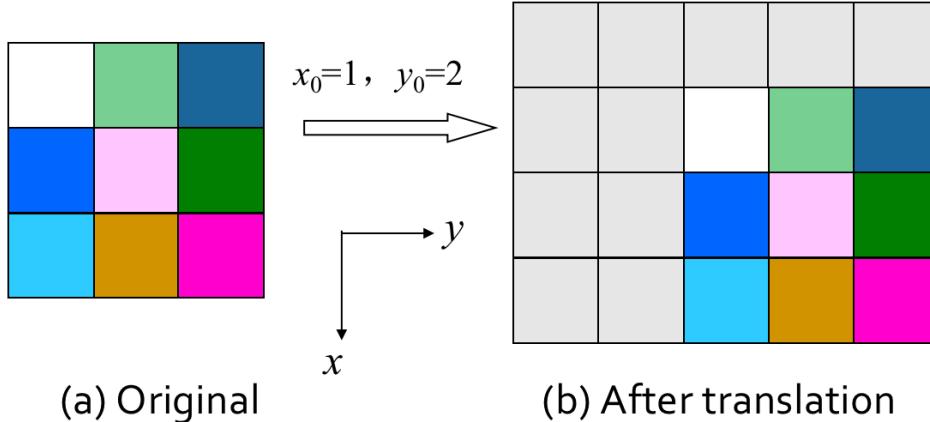
二、实验内容和原理

1. 基本几何变化

(1) 平移 (Translation)

将图像沿水平和竖直方向移动，从而产生新图像的过程。

平移后的景物与原图像相同，但“画布”一定是扩大了，否则就会丢失信息。

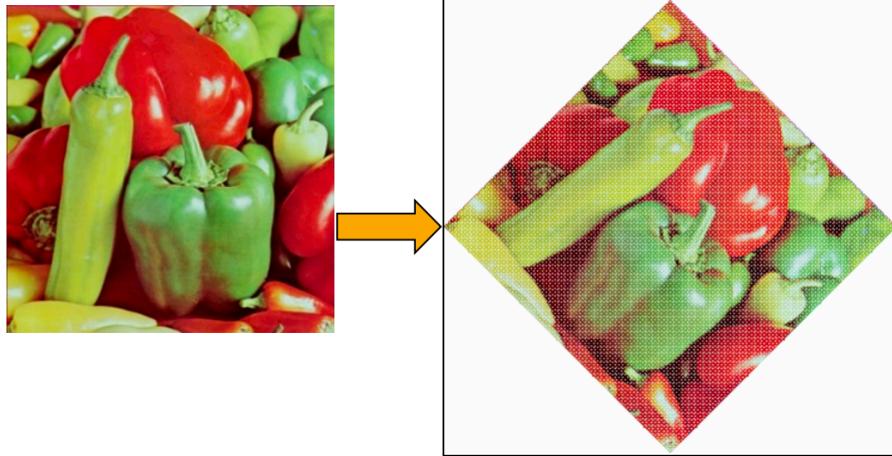


- 矩阵表示

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

(2) 旋转 (Rotation)

- 绕原点旋转 θ 角，得到新图像的过程



- 注意：旋转变换的过程中，图像中会产生空洞，用插值法补全

- 矩阵表示

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (2)$$

(3) 缩放 (Scale)

将图像乘以一定系数，从而产生新图像的过程



(a) Original



(b) Scale with the same ratio



(b) Scale with different ratios

- 沿x轴方向缩放c倍 ($c>1$ 时为放大, $0<c<1$ 时为缩小)

沿y轴方向缩放d倍 ($d>1$ 时为放大, $0<d<1$ 时为缩小)

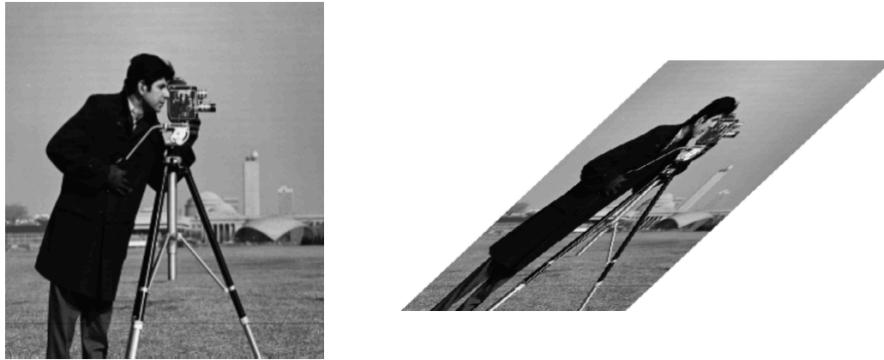
- 当 $c=d$ 时，图像等比缩放；否则为非等比缩放，导致图像变形。
- 缩小：按一定间隔选取某些行和列的像素构成缩小后的新图像；
- 放大：新图像出现空行和空列，可采用插值的方法加以填补，但存在“马赛克”现象。

- 矩阵表示

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} c & 0 & 0 \\ 0 & d & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (3)$$

(4) 图像剪切 (Shear)

图像的错切变换实际上是景物在平面上的非垂直投影效果



$$\text{Shear on } x\text{-axis} : \begin{cases} a(x, y) = x + d_x y \\ b(x, y) = y \end{cases} \quad (4)$$

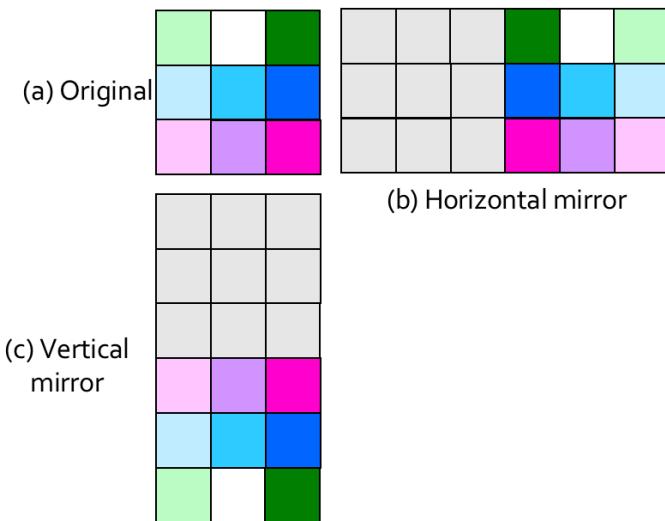
$$\text{Shear on } y\text{-axis} : \begin{cases} a(x, y) = x \\ b(x, y) = y + d_y x \end{cases} \quad (5)$$

- 矩阵表示

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & d_x & 0 \\ d_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (6)$$

(5) 镜像变换 (Mirror)

绕x轴或y轴翻转，从而产生与原图像对称的新图像的过程



- 矩阵表示

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (7)$$

- 当 $s_x = 1$, 且 $s_y = -1$ 时实现绕x轴的镜像变换
- 当 $s_x = -1$, 且 $s_y = 1$ 时实现绕y轴的镜像变换

(6) 复合几何变换

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (8)$$

2. 插值

(1) 最邻近插值 (Nearest neighbor)

- 最邻近插值，即输出像素的灰度值等于离它所映射到的位置最近的输入像素的灰度值。

为了计算几何变换后新图像中某一点 P' 处的像素值，可以首先计算该几何变换的逆变换，计算出 P' 所对应的原图像中的位置 P

通常情况下， P 的位置不可能正好处在原图像的某一个像素位置上(即 P 点的坐标通常都不会正好是整数)

寻找与 P 点最接近的像素 Q ，把 Q 点的像素值作为新图像中 P' 点的像素值。

- Steps

$$\begin{aligned} (x', y') &\xrightarrow{\text{Inverse Transtion}} (x, y) \\ &\xrightarrow{\text{rounding operation}} (x_{int}, y_{int}) \\ &\xrightarrow{\text{assign value}} I_{new}(x', y') = I_{old}(x_{int}, y_{int}) \end{aligned} \quad (9)$$

- 当图像中包含明显的几何结构时，结果将不太光滑连续，从而在图像中产生人为的痕迹

(2) 线性插值 (Linear interpolation)

- 在一维情况下，已知 x_1 和 x_2 处的灰度值分别为 g_1 和 g_2 ，则 x_3 处的灰度值 g_3 为：

$$g_3 = \frac{g_2 - g_1}{x_2 - x_1} (x_3 - x_1) + g_1 \quad (10)$$

- 在二维情况下，称为双线性插值

- 注：可由一维的情况推导而来，得到双线性方程 $g(x,y)$ 为

$$g(x, y) = \frac{(y_1 - y)(x_1 - x)}{(y_1 - y_0)(x_1 - x_0)} g(0, 0) + \frac{(y_1 - y)(x - x_0)}{(y_1 - y_0)(x_1 - x_0)} g(1, 0) + \frac{(y - y_0)(x_1 - x)}{(y_1 - y_0)(x_1 - x_0)} g(0, 1) + \frac{(y - y_0)(x - x_0)}{(y_1 - y_0)(x_1 - x_0)} g(1, 1) \quad (11)$$

即满足 $g(x, y) = ax + by + cxy + d$ 的形式

- 已知图像的正方形网格上四个点A、B、C、D的灰度，求P点的灰度：

- 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，解出 $a b c d$ 四个系数
- 将P点的位置代入方程，得到P点的灰度

(3) 径向基函数插值 [Radial Basis Function (RBF) based interpolation]

- 径向基函数 $\phi(x)$

- 最常见为高斯函数 $\varphi(r) = e^{\frac{-r^2}{2\sigma^2}}$

- Multiquadratics: $\phi(r) = \sqrt{1 + \frac{r^2}{\sigma^2}}$

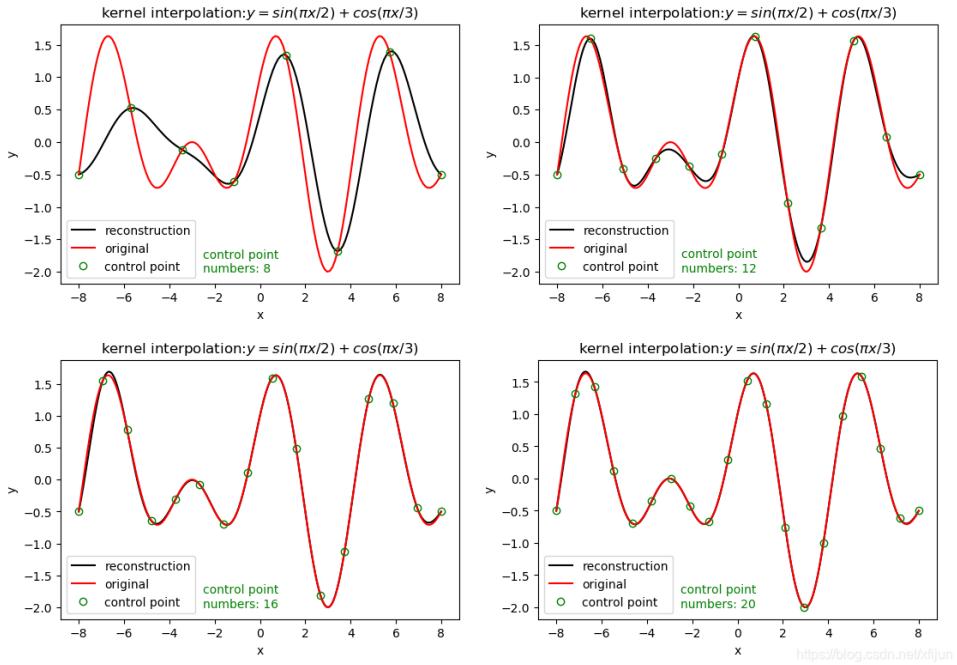
- Linear: $\phi(r) = r$

- Cubic: $\phi(r) = r^3$

.....

- 插值函数

$$\hat{f}(x) = \sum_{i=1}^N w_i \phi(||x - x_i||)$$



<https://blog.csdn.net/xijun>

三、实验步骤与分析

- 注明：本次使用了实验一的函数，在提交的DIP.h中可见。

1.Translate

- 函数对输入的"bmp"进行平移：
 - 沿x轴向右（ $x < 0$ 则向左）平移x个像素点距离
 - 沿y轴向上（ $y < 0$ 则向下）平移y个像素点距离
- 几点注意：
 - 由于平移操作，需要对图片的尺寸进行处理：
 - 如果 $x > 0 y > 0$ 则新生成的图片左下角位置应为原图片平移前的位置.
 - 如果 $x > 0 y < 0$ 则新生成的图片左上角位置应为原图片平移前的位置.
 - 如果 $x < 0 y > 0$ 则新生成的图片右下角位置应为原图片平移前的位置.
 - 如果 $x < 0 y < 0$ 则新生成的图片右上角位置应为原图片平移前的位置.
 - 一定要注意，在对图像进行逐个像素遍历的时候， i 对应行数，即对应 y, j 对应 x .
 - 需要注意，在y坐标上平移的时候，如果 $y > 0$ 即原图像向上平移，应做 $i + y$ 操作，因为实际上bmp图像的存储是倒序的，即为第一个像素点其实是右下角的像素.

```

1 void translate(BMP bmp, int x, int y) {
2     // 创建一个新的BMP结构体用于存储平移后的图像
3     BMP translate;
4     // 将传入的bmp图像的信息拷贝到新的结构体中
5     memcpy(&translate, &bmp, sizeof(translate));
6     // 获取原始图像的宽度和高度
7     int width = translate.bmpih.width;
8     int height = translate.bmpih.height;
9     // 计算平移后的图像宽度和高度
10    int width1 = width + abs(x);
11    int height1 = height + abs(y);
12    // 计算每行的字节数
13    int row_byte1 = (translate.bmpih.bitCount / 8 * width1 + 3) / 4 * 4;
14    int row_byte = (translate.bmpih.bitCount / 8 * width + 3) / 4 * 4;
15    // 更新平移后图像的宽度、高度、图像大小等信息
16    translate.bmpih.height = height1;

```

```

17 translate.bmpih.width = width1;
18 translate.bmpih.imageSize = height1 * row_byte1;
19 translate.bmpih.bfSize = translate.bmpih.bfOffset + translate.bmpih.imageSize;
20 // 分配内存用于存储平移后的图像数据，并将像素值初始化为255（白色）
21 translate.bitmap = (byte *)calloc(translate.bmpih.imageSize, sizeof(byte));
22 for (int i = 0; i < translate.bmpih.imageSize; i++) {
23     translate.bitmap[i] = 255;
24 }
25 // 根据图像的位深度分情况处理像素值
26 if (translate.bmpih.bitCount == 8) {
27     for (int i = 0; i < height; i++) {
28         for (int j = 0; j < width; j++) {
29             int posi = i * row_byte + j;
30             int i1, j1;
31             if (y <= 0) {
32                 i1 = i;
33             } else {
34                 i1 = i + y;
35             }
36             if (x >= 0) {
37                 j1 = j + x;
38             } else {
39                 j1 = j;
40             }
41             int posi1 = i1 * row_byte1 + j1;
42             translate.bitmap[posi1] = bmp.bitmap[posi];
43         }
44     }
45 } else {
46     for (int i = 0; i < height; i++) {
47         for (int j = 0; j < width; j++) {
48             int posi = i * row_byte + j * 3;
49             int i1, j1;
50             if (y <= 0) {
51                 i1 = i;
52             } else {
53                 i1 = i + y;
54             }
55             if (x >= 0) {
56                 j1 = j + x;
57             } else {
58                 j1 = j;
59             }
60             int posi1 = i1 * row_byte1 + j1 * 3;
61             translate.bitmap[posi1] = bmp.bitmap[posi];
62             translate.bitmap[posi1 + 1] = bmp.bitmap[posi + 1];
63             translate.bitmap[posi1 + 2] = bmp.bitmap[posi + 2];
64         }
65     }
66 }
67 // 将平移后的图像保存为"translate.bmp"
68 FILE *fp = fopen("translate.bmp", "wb");
69 OUTPUT(&translate, fp);
70 // 释放动态分配的内存
71 free(translate.bitmap);
72 }

```

2.Rotation

- 注意

```
1 // 在逆推原坐标点的过程中，需要考虑平移问题，因为变换了参考系
2 double x = (double)(width1 / 2) - cx;
3 double y = (double)(height1 / 2) - cy;
```

这一步是非常必要的，一定要注意在旋转过程中也改变了图像的尺寸，也就是说做了坐标系的变换，在原图像中心点旋转后在原坐标系下位于 (c_x, c_y) 而在新图像位于新图像的正中央，即为 $(width_1/2, height_1/2)$ 处

```
1 void rotate(int *x, int *y, double cost, double sint) {
2     // 保存原始坐标
3     double tmp1 = *x;
4     double tmp2 = *y;
5
6     // 计算旋转后的新坐标
7     *x = (tmp1 * cost) - (tmp2 * sint);
8     *y = (tmp1 * sint) + (tmp2 * cost);
9 }
10 void rot(BMP bmp, double angle) {
11     // 创建一个新的BMP结构体用于存储旋转后的图像
12     BMP rot;
13     // 将传入的bmp图像的信息拷贝到新的结构体中
14     memcpy(&rot, &bmp, sizeof(rot));
15     // 获取原始图像的宽度和高度
16     int width = rot.bmpih.width;
17     int height = rot.bmpih.height;
18     // 将角度转换为弧度
19     angle = angle * M_PI / 180.0;
20     // 计算cos和sin值
21     double cos_a = cos(angle);
22     double sin_a = sin(angle);
23     // 计算旋转后的图像宽度和高度
24     int width1 = width * fabs(cos_a) + height * fabs(sin_a);
25     int height1 = width * fabs(sin_a) + width * fabs(cos_a);
26     // 计算旋转中心点的坐标
27     int cx = width / 2;
28     int cy = height / 2;
29     // 对旋转中心点进行坐标变换
30     rotate(&cx, &cy, cos_a, sin_a);
31     // 计算每行的字节数
32     int row_byte1 = (rot.bmpih.bitCount / 8 * width1 + 3) / 4 * 4;
33     int row_byte = (rot.bmpih.bitCount / 8 * width + 3) / 4 * 4;
34     // 更新旋转后图像的宽度、高度、图像大小等信息
35     rot.bmpih.height = height1;
36     rot.bmpih.width = width1;
37     rot.bmpih.imageSize = height1 * row_byte1;
38     rot.bmph.bfSize = rot.bmph.bfOffset + rot.bmpih.imageSize;
39     // 分配内存用于存储旋转后的图像数据
40     rot.bitmap = (byte *)calloc(rot.bmpih.imageSize, sizeof(byte));
41     // 在逆推原坐标点的过程中，需要考虑平移问题，因为变换了参考系
42     double x = (double)(width1 / 2) - cx;
43     double y = (double)(height1 / 2) - cy;
44     // 根据图像的位深度分情况处理像素值
45     if (rot.bmpih.bitCount == 8) {
46         for (int i = 0; i < height1; i++) {
47             for (int j = 0; j < width1; j++) {
48                 int pos1 = i * row_byte1 + j;
49                 double x1 = j - x;
```

```

50     double y1 = i - y;
51     int x2 = round((x1 * cos_a) + (y1 * sin_a));
52     int y2 = round(-(x1 * sin_a) + (y1 * cos_a));
53     int posi = y2 * row_byte + x2;
54     if (x2 < 0 || y2 < 0 || x2 >= width || y2 >= height) {
55         rot.bitmap[posi1] = 255;
56         continue;
57     }
58     rot.bitmap[posi1] = bmp.bitmap[posi];
59 }
60 }
61 } else {
62     for (int i = 0; i < height1; i++) {
63         for (int j = 0; j < width1; j++) {
64             int posi1 = i * row_byte1 + j * 3;
65             double x1 = j - x;
66             double y1 = i - y;
67             int x2 = round((x1 * cos_a) + (y1 * sin_a));
68             int y2 = round(-(x1 * sin_a) + (y1 * cos_a));
69             int posi = y2 * row_byte1 + x2 * 3;
70             if (x2 < 0 || y2 < 0 || x2 >= width || y2 >= height) {
71                 rot.bitmap[posi1] = 255;
72                 rot.bitmap[posi1 + 1] = 255;
73                 rot.bitmap[posi1 + 2] = 255;
74                 continue;
75             }
76             rot.bitmap[posi1] = bmp.bitmap[posi];
77             rot.bitmap[posi1 + 1] = bmp.bitmap[posi + 1];
78             rot.bitmap[posi1 + 2] = bmp.bitmap[posi + 2];
79         }
80     }
81 }
82 // 将旋转后的图像保存为"rotate.bmp"
83 FILE *fp = fopen("rotate.bmp", "wb");
84 OUTPUT(&rot, fp);
85 // 释放动态分配的内存
86 free(rot.bitmap);
87 }

```

3.Scale

- 一定要注意，在对图像进行逐个像素遍历的时候， i 对应行数，即对应 y , j 对应 x .

```

1 void scale(double x, double y, BMP bmp) {
2     // 创建一个新的BMP结构体用于存储缩放后的图像
3     BMP scale;
4     // 复制原始图像信息到新的结构体
5     memcpy(&scale, &bmp, sizeof(scale));
6     // 获取原始图像的宽度和高度
7     int width = scale.bmpih.width;
8     int height = scale.bmpih.height;
9     // 计算缩放后的宽度和高度
10    int width1 = round(width * x);
11    int height1 = round(height * y);
12    // 计算每行的字节数（包含补齐的字节）
13    int row_byte = (bmp.bmpih.bitCount / 8 * width + 3) / 4 * 4;
14    int row_bytel = (bmp.bmpih.bitCount / 8 * width1 + 3) / 4 * 4;
15    // 更新新BMP结构体的宽度和高度信息
16    scale.bmpih.width = width1;
17    scale.bmpih.height = height1;
18    // 分配内存以存储缩放后的图像数据

```

```

19 scale.bitmap = (byte*)malloc(row_byte1 * height1 * sizeof(byte));
20 // 更新新BMP结构体的图像大小信息
21 scale.bmpih.imageSize = height1 * row_byte1;
22 scale.bmpih.bfSize = scale.bmpih.bfOffset + scale.bmpih.imageSize;
23 // 根据图像的位深度进行缩放操作
24 if (scale.bmpih.bitCount == 24) {
25     // 24位深度的图像
26     for (int i = 0; i < height1; i++) {
27         for (int j = 0; j < width1; j++) {
28             // 计算缩放后的位置
29             int posi = i * row_byte1 + j * 3;
30             int posi1 = round(i / y) * row_byte + round(j / x) * 3;
31             // 复制像素值到新的位置
32             scale.bitmap[posi] = bmp.bitmap[posi1];
33             scale.bitmap[posi + 1] = bmp.bitmap[posi1 + 1];
34             scale.bitmap[posi + 2] = bmp.bitmap[posi1 + 2];
35         }
36     }
37 } else {
38     // 其他位深度的图像
39     for (int i = 0; i < height1; i++) {
40         for (int j = 0; j < width1; j++) {
41             // 计算缩放后的位置
42             int posi = i * row_byte1 + j;
43             int posi1 = round(i / y) * row_byte + round(j / x);
44             // 复制像素值到新的位置
45             scale.bitmap[posi] = bmp.bitmap[posi1];
46         }
47     }
48 }
49 // 打开一个文件用于写入缩放后的图像数据
50 FILE* fp = fopen("scale.bmp", "wb");
51 // 输出新的BMP结构体到文件
52 OUTPUT(&scale, fp);
53 // 释放分配的内存
54 free(scale.bitmap);
55 }

```

4.Rear

- 不涉及 $height1 - i$, 由于原图像新图像素均为倒序储存「即为第一个点在图像左下角」
- 根据公式计算得到逆过程

$$\begin{cases} x = \frac{x' - d_x * y'}{1 - d_x * d_y} \\ y = \frac{y' - d_y * x'}{1 - d_x * d_y} \end{cases} \quad (12)$$

```

1 void affine(double dx, double dy, BMP bmp) {
2     // 创建一个新的BMP结构体用于存储仿射变换后的图像
3     BMP affine;
4     // 复制原始图像信息到新的结构体
5     memcpy(&affine, &bmp, sizeof(affine));
6     // 获取原始图像的宽度和高度
7     int width = affine.bmpih.width;
8     int height = affine.bmpih.height;
9     // 计算仿射变换后的宽度和高度
10    int width1 = round(width + dx * height);
11    int height1 = round(height + dy * width);
12    // 计算每行的字节数 (包含补齐的字节)
13    int row_byte = (bmp.bmpih.bitCount / 8 * width + 3) / 4 * 4;

```

```

14 int row_byte1 = (bmp.bmpih.bitCount / 8 * width1 + 3) / 4 * 4;
15 // 更新新BMP结构体的宽度和高度信息
16 affine.bmpih.width = width1;
17 affine.bmpih.height = height1;
18 // 分配内存以存储仿射变换后的图像数据
19 affine.bitmap = (byte*)malloc(row_byte1 * height1 * sizeof(byte));
20 // 更新新BMP结构体的图像大小信息
21 affine.bmpih.imageSize = height1 * row_byte1;
22 affine.bmph.bfSize = affine.bmph.bfOffset + affine.bmpih.imageSize;
23 // 根据图像的位深度进行仿射变换操作
24 if (affine.bmpih.bitCount == 24) {
25     // 24位深度的图像
26     for (int i = 0; i < height1; i++) {
27         for (int j = 0; j < width1; j++) {
28             // 计算仿射变换后的位置
29             int posi = i * row_byte1 + j * 3;
30             int x1 = round((j - dx * (i)) / (1 - dx * dy));
31             int y1 = round((i - dy * j) / (1 - dx * dy));
32             // 检查新位置是否在原始图像范围内，如果不在则填充白色
33             if (x1 < 0 || y1 < 0 || x1 >= width || y1 >= height){
34                 affine.bitmap[posi] = 255;
35                 affine.bitmap[posi + 1] = 255;
36                 affine.bitmap[posi + 2] = 255;
37                 continue;
38             }
39             // 复制像素值到新的位置
40             int posi1 = y1 * row_byte + x1 * 3;
41             affine.bitmap[posi] = bmp.bitmap[posi1];
42             affine.bitmap[posi + 1] = bmp.bitmap[posi1 + 1];
43             affine.bitmap[posi + 2] = bmp.bitmap[posi1 + 2];
44         }
45     }
46 } else {
47     // 其他位深度的图像
48     for (int i = 0; i < height1; i++) {
49         for (int j = 0; j < width1; j++) {
50             // 计算仿射变换后的位置
51             int posi = i * row_byte1 + j;
52             int x1 = round((j - dx * (i)) / (1 - dx * dy));
53             int y1 = round((i - dy * j) / (1 - dx * dy));
54             // 检查新位置是否在原始图像范围内，如果不在则填充白色
55             if (x1 < 0 || y1 < 0 || x1 >= width || y1 >= height){
56                 affine.bitmap[posi] = 255;
57                 continue;
58             }
59             // 复制像素值到新的位置
60             int posi1 = y1 * row_byte + x1;
61             affine.bitmap[posi] = bmp.bitmap[posi1];
62         }
63     }
64 }
65 // 打开一个文件用于写入仿射变换后的图像数据
66 FILE* fp = fopen("affine.bmp", "wb");
67 // 输出新的BMP结构体到文件
68 OUTPUT(&affine, fp);
69 // 释放分配的内存
70 free(affine.bitmap);
71 }

```

5.Mirror

```
1 void mirror(int axis, BMP bmp) {
2     // 创建一个新的BMP结构体用于存储镜像操作后的图像
3     BMP mirror;
4     // 复制原始图像信息到新的结构体
5     memcpy(&mirror, &bmp, sizeof(mirror));
6     // 获取原始图像的宽度和高度
7     int height = bmp.bmpih.height;
8     int width = bmp.bmpih.width;
9     // 计算每行的字节数（包含补齐的字节）
10    int row_byte = (bmp.bmpih.bitCount / 8 * width + 3) / 4 * 4;
11    // 分配内存以存储镜像操作后的图像数据
12    mirror.bitmap = (byte*)calloc(mirror.bmpih.imageSize, sizeof(byte));
13    // 根据图像的位深度进行镜像操作
14    if (mirror.bmpih.bitCount == 24) {
15        // 24位深度的图像
16        for (int i = 0; i < height; i++) {
17            for (int j = 0; j < width; j++) {
18                // 计算当前位置的索引
19                int posi = i * row_byte + j * 3;
20                int posil;
21                // 根据镜像轴选择镜像位置
22                if (axis) {
23                    posil = i * row_byte + (width - j) * 3;
24                } else {
25                    posil = (height - i) * row_byte + j * 3;
26                }
27                // 复制像素值到新的位置
28                mirror.bitmap[posi] = bmp.bitmap[posil];
29                mirror.bitmap[posi + 1] = bmp.bitmap[posil + 1];
30                mirror.bitmap[posi + 2] = bmp.bitmap[posil + 2];
31            }
32        }
33    } else {
34        // 其他位深度的图像
35        for (int i = 0; i < height; i++) {
36            for (int j = 0; j < width; j++) {
37                // 计算当前位置的索引
38                int posi = i * row_byte + j;
39                int posil;
40                // 根据镜像轴选择镜像位置
41                if (axis) {
42                    posil = i * row_byte + (width - j);
43                } else {
44                    posil = (height - i) * row_byte + j;
45                }
46                // 复制像素值到新的位置
47                mirror.bitmap[posi] = bmp.bitmap[posil];
48            }
49        }
50    }
51    // 打开一个文件用于写入镜像操作后的图像数据
52    FILE* fp = fopen("mirror.bmp", "wb");
53    // 输出新的BMP结构体到文件
54    OUTPUT(&mirror, fp);
55    // 释放分配的内存
56    free(mirror.bitmap);
57 }
```

四、实验环境及运行方法

实验环境：

MacBook Air M2 Sonoma 14.0

Apple clang version 15.0.0(arm64-apple-darwin23.0.0)

运行方法：

打开lab04文件夹，用vscode打开其中的code文件夹，其中包含源文件 `lab4.c`，头文件 `DIP.h`，可执行文件 `lab4mac`, `lab4.exe` 和24位BMP图像 `Lena.bmp`。

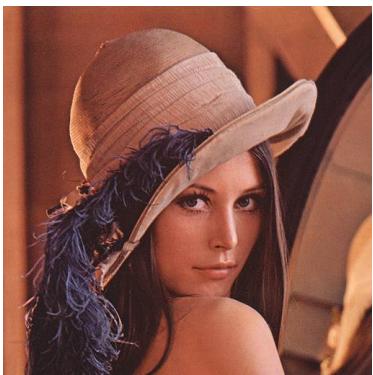
- (1) 打开 `lab3.c`, `DIP.h`, 进入 `lab3.c`, 修改希望处理的图像名为 `input.bmp` (如把 `Lena.bmp` 修改为 `input.bmp`) , 点击 Run Code 可开始运行。输出 "successfully loaded!" 表示文件正常读入，「由于在 `DIP.h` 中改变了 `#pragma pack alignment value`」, 会产生 warning, 但不影响程序运行」
- (2) 程序会输出五张处理后的图片，分别为：

五、实验结果展示

- Translate

		
Input	$dx=100 \ dy=200$	$dx=-100 \ dy=-200$

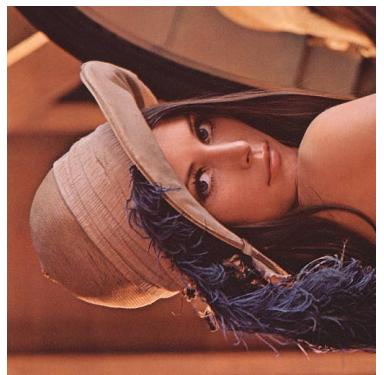
- Rotate



Input

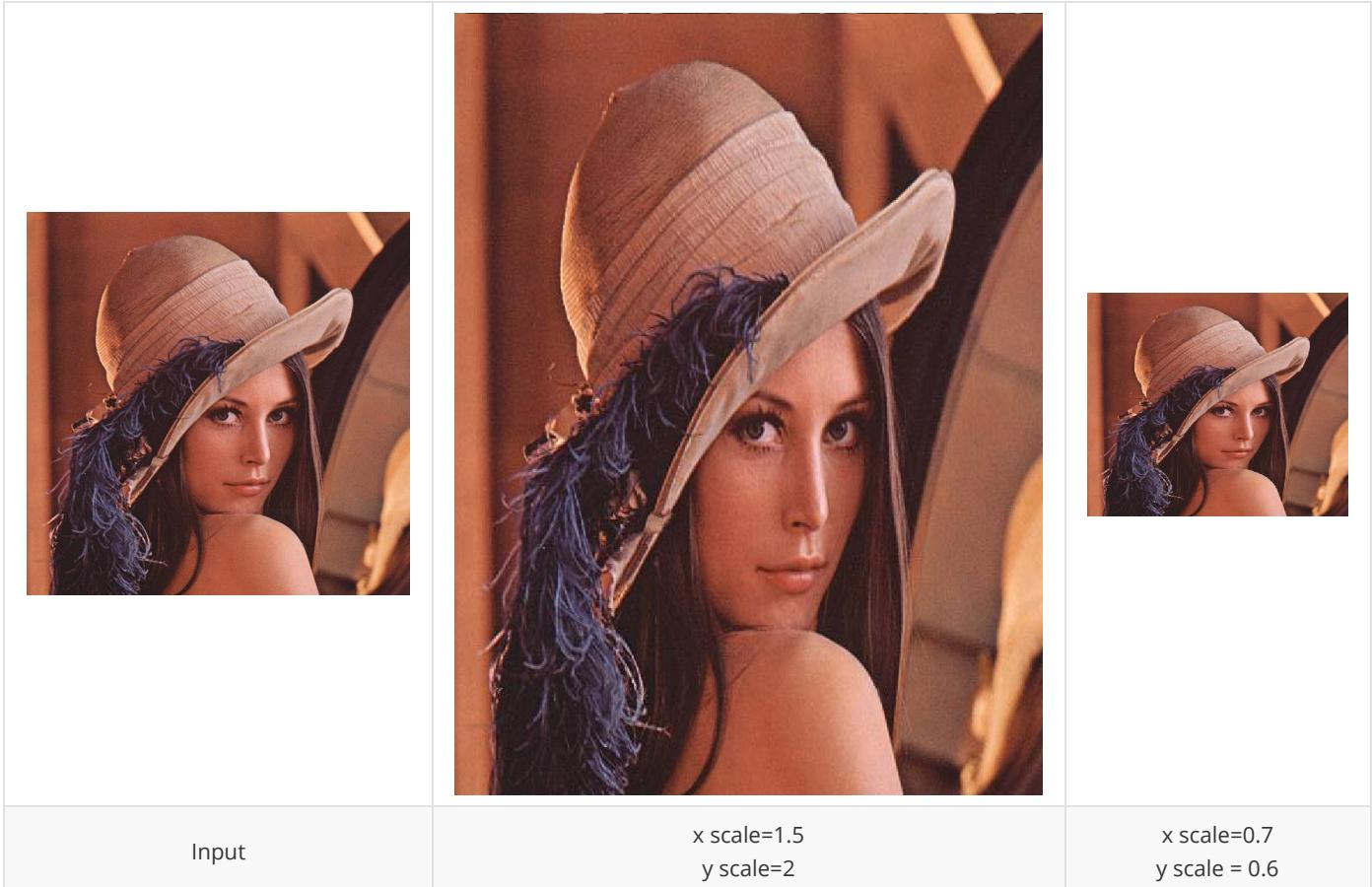


逆时针60度



逆时针90度

- Scale



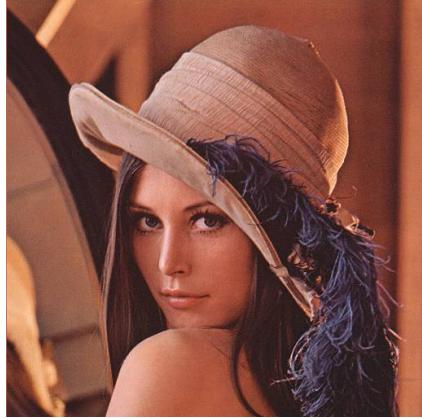
- Rear



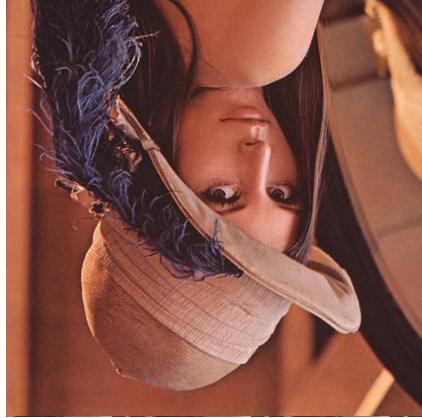
- Mirrow



Input



沿x轴



沿y轴

六、心得体会