

浙江大学实验报告

专业：计算机科学与技术

姓名：

学号：

日期：2021/10/29

课程名称：____ 图像信息处理 ____ 指导老师：____ 宋明黎 ____ 成绩：____

实验名称：____ 简单的图像几何变换 ____

一、实验目的和要求

1. 平移
2. 旋转
3. 缩放
4. 错切
5. 镜像

二、实验内容和原理

1. 平移

在 X 轴和 Y 轴上平移图像。使用的矩阵如下：

Translation—Equation

$$\begin{cases} x' = x + x_0 \\ y' = y + y_0 \end{cases}$$

OR

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

Each pixel in the original image is translated x_0 and y_0 respectively.

2. 旋转

将图片旋转，使用如下矩阵：

Rotation——Equation

$$\begin{cases} x' = x \cos \theta - y \sin \theta \\ y' = x \sin \theta + y \cos \theta \end{cases}$$

OR

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

但是简单地使用矩阵变换导致变换后的点不一定是整点，会导致整张图片出现很多漏洞，对于这种情况笔者使用行插值法，即将漏洞处像素值等于前一个像素的像素值，这样可以大致还原图片，虽然会导致图片边上出现一些锯齿。

3. 缩放

将图片旋转，使用如下矩阵：

Scale——Equation

$$\begin{cases} x' = cx \\ y' = dy \end{cases}$$

OR

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

但是在放大时候也会出现漏洞，而缩小的时候可能会导致像素值的缺失。面对这种情况要使用双线性插值。主要就是用某一像素点周围的四个像素值来建立线性方程：

$$g(x, y) = ax + by + cxy + d$$

然后代入四个点得到方程，最后代入变换后点的坐标，接触这个点的像素值。

经过计算，得到某一点相邻四个点（这四个点的横纵坐标差为 1 或 0）的双线性插值方

程如下：

$$f(x, y) = f(Q_{11})(x_2 - x)(y_2 - y) + f(Q_{21})(x - x_1)(y_2 - y) \\ + f(Q_{12})(x_2 - x)(y - y_1) + f(Q_{22})(x - x_1)(y - y_1)$$

为确保源图像和目标图像几何中心对齐，再结合上面的矩阵，使用下面的公式找到目标图像 (x', y') 对应源图像 (x, y) 的坐标关系：

$$x = (x' + 0.5) \div c + 0.5$$

$$y = (y' + 0.5) \div d + 0.5$$

再使用 $(\text{floor}(x), \text{floor}(y))$ 及周围三个点放入双线性插值公式即可。

4. 错切

将图片错切，产生立体感。使用如下公式：

Shear—Equation

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

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

5. 镜像

将图片关于 X 轴或 Y 轴镜像变化，使用如下矩阵：

Mirror—Equation

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

When $S_x = 1$, and $S_y = -1$, flip around the x axis

When $S_x = -1$, and $S_y = 1$, flip around the y axis

三、实验步骤与分析

1. 矩阵操作

注意到这几个图像几何操作都使用了矩阵，因此，笔者使用 `GeometricTransform` 函数来统一做矩阵操作，再对旋转和缩放另做处理。笔者使用的矩阵是上面原理部分变换矩阵的前两行，即一个 2×3 的矩阵，因为最后一行对 x' 和 y' 没有影响。

注意到缩放和其他的操作有一定的不同，那就是针对背景。其他的操作为了表现出几何变化的效果，需要在图片以外描绘出背景，而缩放操作不需要背景，单纯对图片做一个缩放。所以笔者的这个函数不仅仅需要原始图片和变换矩阵，还需要一个 `canvas` 作为 `flag`，当需要背景时，传入 1 即可。

该函数中部分和之前的 `Copy` 函数接近，主要是重新创建一个 `Image` 对象，对其赋值的部分就略过。介绍一下主要的算法实现部分。

首先是确定新图像的长宽。对于不需要背景的图片，就直接对 4 个顶点施加矩阵，得到的横纵坐标的最大值即为最后图片的长宽；对于需要背景的图片，在以上基础上，需要并上原来的矩形范围。最后变换矩阵坐标的范围是 $[minX, maxX] \times [minY, maxY]$ ，但是 $minX$ 和 $minY$ 可能是负值。

```
// apply the matrix to the 4 corner vertexs to determine the size of the target image
for (i = 0; i < 4; i++) {
    x = vertexs[i][0] * Matrix[0][0] + vertexs[i][1] * Matrix[0][1] + Matrix[0][2];
    y = vertexs[i][0] * Matrix[1][0] + vertexs[i][1] * Matrix[1][1] + Matrix[1][2];
    if (x < minX)
        minX = x;
    else if (x > maxX)
        maxX = x;
    if (y < minY)
        minY = y;
    else if (y > maxY)
        maxY = y;
}
// get the width and height
bmpImageNew->bmpInfo->biWidth = maxX - minX + 1;
bmpImageNew->bmpInfo->biHeight = maxY - minY + 1;
```

得到长宽之后就对图像中所有像素运用矩阵。注意这里了 $x - minX$ 和 $y - minY$ 来作为变换后矩阵的坐标。

```
// apply the matrix to all the pixels
for (i = 0; i < bmpImage->bmpInfo->biHeight; i++) {
    for (j = 0; j < bmpImage->bmpInfo->biWidth; j++) {
        x = j * Matrix[0][0] + i * Matrix[0][1] + Matrix[0][2];
        y = j * Matrix[1][0] + i * Matrix[1][1] + Matrix[1][2];
        for (k = 0; k < bmpImage->bmpInfo->biBitCount / 8; k++)
            bmpImageNew->bmpData[(y - minY) * dataPerLineNew + (x - minX) * bmpImageNew->bmpInfo->biBitCount / 8 + k] =
                bmpImage->bmpData[i * dataPerLine + j * bmpImage->bmpInfo->biBitCount / 8 + k];
    }
}

return bmpImageNew;
```

2. 平移、镜像、错切

这几个操作在实现了上述矩阵操作之后是简单的，只要将原理部分的矩阵代入即可。

平移传入的参数就是在 X 轴和 Y 轴上移动的距离。

```
Image *Translation(Image *bmpImage, double x, double y)
{
    double translationMatrix[2][3] = {{1, 0, x}, {0, 1, y}};
    Image *bmpImageTranslation = GeometricTransform(bmpImage, translationMatrix, 1);
    return bmpImageTranslation;
}
```

镜像传入的参数：sX 若为-1，则关于 X 轴对称；若 sY 为-1，则关于 Y 轴对称，当其中之一为 1 的时候就不发生镜像对称。

```
Image *Mirror(Image *bmpImage, int sX, int sY)
{
    if ((abs(sX) != 1) || (abs(sY) != 1))
        return NULL;
    double mirrorMatrix[2][3] = {{sX, 0, 0}, {0, sY, 0}}; // mirror matrix
    Image *bmpImageMirror = GeometricTransform(bmpImage, mirrorMatrix, 1);
    return bmpImageMirror;
}
```

错切传入的参数：dx 和 dy 即为上述原理部分提到的 dx 和 dy 参数。

```
Image *Shear(Image *bmpImage, double dx, double dy)
{
    double shearMatrix[2][3] = {{1, dx, 0}, {dy, 1, 0}}; // shear matrix
    Image *bmpImageShear = GeometricTransform(bmpImage, shearMatrix, 1);
    return bmpImageShear;
}
```

3. 旋转

首先像上述几种操作也对图像运用矩阵操作函数。传入的参数是弧度制。

```
Image *Rotation(Image *bmpImage, double theta)
{
    int i, j, k;
    double rotMatrix[2][3] = {{cos(theta), -sin(theta), 0}, {sin(theta), cos(theta), 0}}; // rot matrix
    Image *bmpImageRot = GeometricTransform(bmpImage, rotMatrix, 1);
    int dataPerLineRot = (bmpImageRot->bmpInfo->biWidth * (bmpImageRot->bmpInfo->biBitCount / 8) + 3) / 4 * 4;
```

接下来进行邻近插值，笔者判断某一个点像素值为 0 且同一行后一个不为 0，就用前一个像素值来替代。

```
// interpolation with the nearest pixels of the same row
for (i = 1; i < bmpImageRot->bmpInfo->biHeight; i++) {
    for (j = 1; j < bmpImageRot->bmpInfo->biWidth; j++) {
        for (k = 0; k < bmpImageRot->bmpInfo->biBitCount / 8; k++)
            if (bmpImageRot->bmpData[i * dataPerLineRot + j * bmpImageRot->bmpInfo->biBitCount / 8 + k] == 0 &&
                bmpImageRot->bmpData[i * dataPerLineRot + (j + 1) * bmpImageRot->bmpInfo->biBitCount / 8 + k] != 0)
                bmpImageRot->bmpData[i * dataPerLineRot + j * bmpImageRot->bmpInfo->biBitCount / 8 + k] =
                    bmpImageRot->bmpData[i * dataPerLineRot + (j + 1) * bmpImageRot->bmpInfo->biBitCount / 8 + k];
    }
}
return bmpImageRot;
```

4. 缩放

首先像上述几种操作也对图像运用矩阵操作函数。参数中 `cx` 和 `cy` 分别是 X 和 Y 的缩放比例。

```
Image *Scale(Image *bmpImage, double cx, double cy)
{
    int i, j, k;
    double srcX, srcY;
    int x1, y1, x2, y2;
    double scaleMatrix[2][3] = {{cx, 0, 0}, {0, cy, 0}}; // scale matrix
    Image *bmpImageScale = GeometricTransform(bmpImage, scaleMatrix, 0);
}
```

然后接下来利用原理部分的双线性插值来进一步优化图像。

第一步是得到当前目标坐标(i, j)在源图像的对应坐标($srcX, srcY$)，主要是使用原理部分提到的公式。

```
// interpolation
int dataPerLineScale = (bmpImageScale->bmpInfo->biWidth * (bmpImageScale->bmpInfo->biBitCount / 8) + 3) / 4 * 4;
int dataPerLine = (bmpImage->bmpInfo->biWidth * (bmpImage->bmpInfo->biBitCount / 8) + 3) / 4 * 4;
for (i = 0; i < bmpImage->bmpInfo->biHeight * cy; i++) {
    if (i == bmpImageScale->bmpInfo->biHeight - 1)
        break;
    for (j = 0; j < bmpImage->bmpInfo->biWidth * cx; j++) {
        if (j == bmpImageScale->bmpInfo->biWidth - 1)
            break;
        // srcX and srcY is the relevant pixels in the original image
        srcX = (j + 0.5) / cx - 0.5;
        srcY = (i + 0.5) / cy - 0.5;
    }
}
```

第二步是得到($srcX, srcY$)周围的四个坐标，主要使用 `floor` 函数，也就是以下四个坐标 ($\text{floor}(srcX), \text{floor}(srcY)$), ($\text{floor}(srcX) + 1, \text{floor}(srcY)$), ($\text{floor}(srcX), \text{floor}(srcY) + 1$), ($\text{floor}(srcX) + 1, \text{floor}(srcY) + 1$)。但要确保这四个坐标都不能越过边界。

```
// figure out the 4 pixels around the (srcX, srcY)
x1 = min(max(floor(srcX), 0), bmpImage->bmpInfo->biWidth - 2);
y1 = min(max(floor(srcY), 0), bmpImage->bmpInfo->biHeight - 2);
x2 = x1 + 1;
y2 = y1 + 1;
```

最后使用上述提到的双线性插值公式即可。

```
// bilinear interpolation
for (k = 0; k < bmpImageScale->bmpInfo->biBitCount / 8; k++) {
    bmpImageScale->bmpData[i * dataPerLineScale + j * bmpImageScale->bmpInfo->biBitCount / 8 + k] =
        bmpImage->bmpData[y1 * dataPerLine + x1 * bmpImage->bmpInfo->biBitCount / 8 + k] * (x2 - srcX) * (y2 - srcY) +
        bmpImage->bmpData[y1 * dataPerLine + x2 * bmpImage->bmpInfo->biBitCount / 8 + k] * (srcX - x1) * (y2 - srcY) +
        bmpImage->bmpData[y2 * dataPerLine + x1 * bmpImage->bmpInfo->biBitCount / 8 + k] * (x2 - srcX) * (srcY - y1) +
        bmpImage->bmpData[y2 * dataPerLine + x2 * bmpImage->bmpInfo->biBitCount / 8 + k] * (srcX - x1) * (srcY - y1);
}
```

四、实验环境及运行方法


编译环境：

gcc 6.3.0、Windows 11 Insider Preview 22483.1011

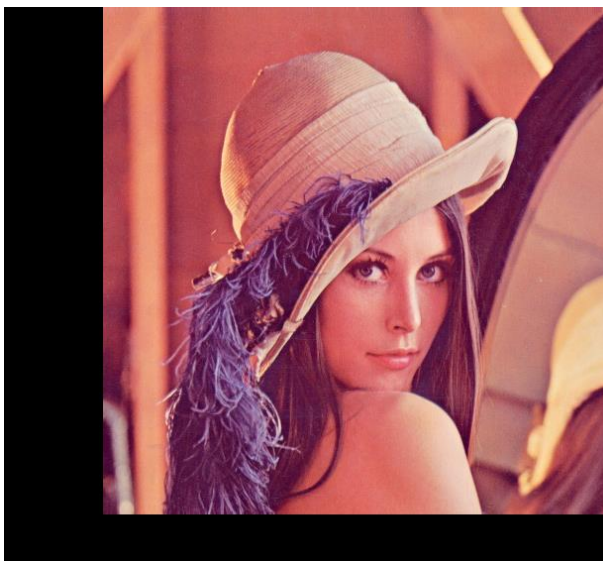
测试方法：

在命令行中输入 gcc image.c main.c，然后运行.\a.exe。输入和输出图像都放在了 output 文件夹，输入为 24 位 BMP 文件，文件名为 input.bmp，输出的图片有平移图片 outputTrans.bmp、旋转图片 outputRot.bmp、镜像图片 outputMirror.bmp、错切图片 outputShear.bmp、缩小图片 outputS1.bmp 和放大图片 outputS2.bmp。

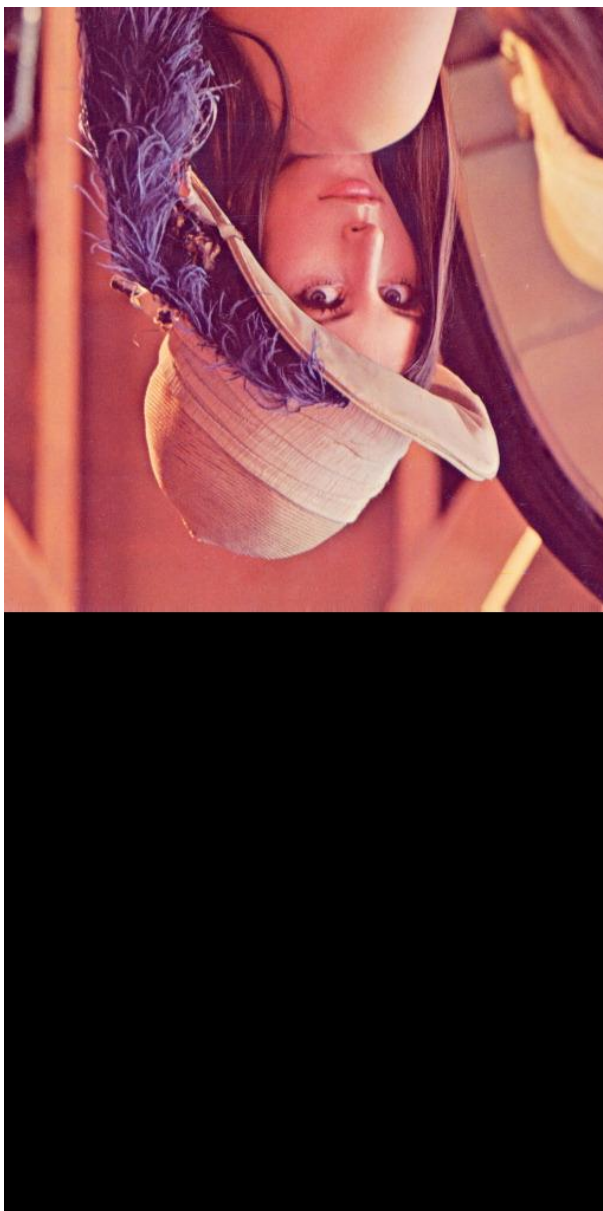
五、实验结果展示

24 位 BM P 图	
----------------------	---

平移



镜像



旋转



错切



缩小



放大



六、心得体会

在本次实验中我对图片的几何操作有了较深的理解，特别是之前课上学过双线性插值，在这里实验中对双线性插值的推导有了数学上的认识，更对插值有了更全面一些的认识。