成绩	

广东工业大学

实验报告

课程名称 现代图像处理技术

题目名称 图像旋转原理及 C++实现

学院 计算机学院

专业班级 电子信息专硕 1 班

学生姓名 梁增国

学号 2112005119

指导老师 战荫伟

日期 2020 年 10 月 7 日

目 录

1. 摘要

图像旋转是图像处理中最常见的操作之一, 图像旋转是指图像以某一点为中心旋转一定的角度, 形成一幅新的图像的过程。本文主要介绍图像旋转的原理, 包括前向映射算法、常见的两种后向映射算法, 以及用 C++编程实现图像旋转。

关键词：图像旋转算法、前向映射算法、后向映射算法、最邻近法、双线性插值法

2. 实验内容与相关平台

2.1 实验内容

- 推导图像旋转某个角度后, 新图像的宽与高的计算公式
- 推导图像像素点的旋转变换公式, 包括前向映射公式、后向映射公式
- 编写程序, 使用前向映射算法、最邻近算法、双线性插值算法实现图像旋转
- 对比使用前向映射算法、最邻近算法生成旋转的图像及效果
- 对比使用最邻近算法、双线性插值算法生成的旋转图像及效果

2.2 实验的相关平台与工具

C++、VS Code、OpenCV（仅用于读写图像）

3. 计算旋转后图像的宽与高

3.1 公式推导

计算旋转后图像的宽与高, 通过画草图的方式来模拟旋转的过程, 可很容易算出旋转后的图像的宽与高。旋转过程模拟见图 3-1, 公式推导过程如下文所示。

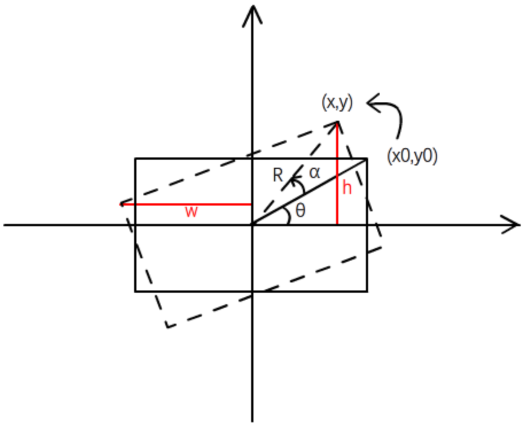

图 3-1

假设原图的宽为 W，高为 H，旋转后图像的宽为 W'，高为 H'，并根据上图可知：

$$
\begin{aligned}
H' &= 2h \\
&= 2R \cdot \sin(\alpha + \theta) \\
&= 2R \cdot (sin\alpha \cdot cos\theta + cos\alpha \cdot sin\theta) \\
&= W \cdot sin\alpha + H \cdot cos\alpha
\end{aligned}
$$

同理可得：

$$\mathrm{W}' = 2w = W \cdot cos\alpha + H \cdot sin\alpha$$

3.2 关键代码

ImgRotate.cpp > main(void)

```cpp
int main(void)
{
    Mat imgMat = imread("C:/Users/ZXX-PC/Desktop/cat.png");

    float rotateAngle = 33 * 3.14159 / 180.0;
    int imgW = imgMat.cols;
    int imgH = imgMat.rows;

    //计算旋转后的图像的宽与高
    int newImgW = int(imgW * abs(cos(rotateAngle)) + imgH * abs(sin(rotateAngle))) + 1;
    int newImgH = int(imgW * abs(sin(rotateAngle)) + imgH * abs(cos(rotateAngle))) + 1;

    Mat fmapImgMat(newImgH, newImgW, CV_8UC3, Scalar::all(0));
    Mat bmapImgMat1(newImgH, newImgW, CV_8UC3, Scalar::all(0));
    Mat bmapImgMat2(newImgH, newImgW, CV_8UC3, Scalar::all(0));

```

图 3-2

如图 3-2 红框所示，计算出旋转后图像的宽与高。需要注意的是，用户可以输入任意的旋转角度，因此要保证三角函数的计算结果是正值。

4. 像素点的旋转与坐标系变换

4.1 像素点的旋转公式推导

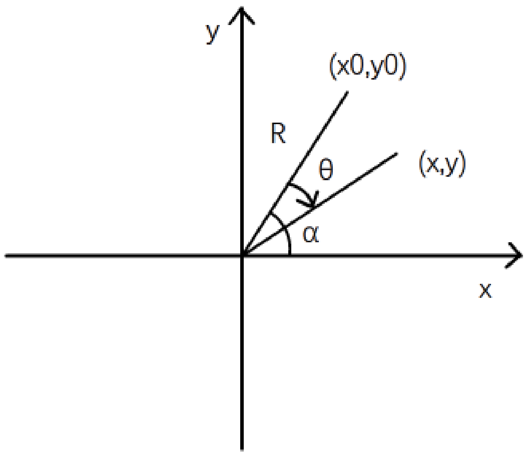

图 4-1

与计算图像旋转后的宽与高类似，计算坐标点的旋转也可以通过画草图模拟，公式推导过程如下。

假设点(x_0, y_0)旋转角度θ后到达点(x, y)处，并假设该点到原点的距离为 R，易得有以下公式：

$$x = R \cdot cos(\alpha - \theta)$$
$$y = R \cdot sin(\alpha - \theta)$$

化简，可得：

$$\begin{aligned} x &= R \cdot cos(\alpha - \theta) \\ &= R \cdot cos\alpha \cdot cos\theta + R \cdot sin\alpha \cdot sin\theta \\ &= x_0 cos\theta + y_0 sin\theta \end{aligned}$$

$$\begin{aligned} y &= R \cdot sin(\alpha - \theta) \\ &= R \cdot sin\alpha \cdot cos\theta - R \cdot cos\alpha \cdot sin\theta \\ &= y_0 cos\theta - x_0 sin\theta \end{aligned}$$

将变换公式与逆变换公式用矩阵表示，如下所示：

$$[x \quad y \quad 1] = [x_0 \quad y_0 \quad 1] \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$[x_0 \quad y_0 \quad 1] = [x \quad y \quad 1] \begin{bmatrix} cos\theta & sin\theta & 0 \\ -sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

4.2 数学坐标系与图像坐标系的转换

我们可知，图像的旋转一般是以图像的中心点为旋转点，对应着数学坐标系的原点，而图像坐标系是以图像左上角为原点，因此我们需要在图像旋转时实现数学坐标与图像坐标的相互转换。在旋转图像时，先把坐标系从图像坐标系转换到数学坐标系，旋转后再转换回图像坐标系。

假设原图像的宽高为 W，H，旋转后的图像的外接矩形宽高大小为 W'，H'，则图像坐标系的点(x_0, y_0)与数学坐标系的点(x, y)的相互转换，有以下变换公式与逆变换公式：

$$[x \quad y \quad 1] = [x_0 \quad y_0 \quad 1]\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ -0.5W & 0.5H & 1 \end{bmatrix}$$

$$[x_0 \quad y_0 \quad 1] = [x \quad y \quad 1]\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0.5W' & 0.5H' & 1 \end{bmatrix}$$

结合像素点的旋转矩阵，我们得到最终的像素点旋转变换公式与逆变换公式：

$$[x \quad y \quad 1] = [x_0 \quad y_0 \quad 1]\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ -0.5W & 0.5H & 1 \end{bmatrix}\begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0.5W' & 0.5H' & 1 \end{bmatrix}, \quad ①$$

$$[x_0 \quad y_0 \quad 1] = [x \quad y \quad 1]\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ -0.5W' & 0.5H' & 1 \end{bmatrix}\begin{bmatrix} cos\theta & sin\theta & 0 \\ -sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0.5W & 0.5H & 1 \end{bmatrix}, \quad ②$$

实际上，式①就是图像旋转算法中的前向映射变换，式②为图像旋转算法中的后向映射变换。在编程时，我们可以利用矩阵乘法函数把多个矩阵相乘，得到最终变换后的坐标，也可以对上述公式进一步化简，以此来计算结果。

化简后，可得变换公式式③与逆变换公式式④：

$$[x \quad y \quad 1]^T = \begin{bmatrix} x_0 \cdot cos\theta - y \cdot sin\theta - 0.5 \cdot W \cdot cos\theta + 0.5 \cdot H \cdot sin\theta + 0.5 \cdot W' \\ x_0 \cdot sin\theta + y \cdot cos\theta - 0.5 \cdot W \cdot sin\theta - 0.5 \cdot H \cdot cos\theta + 0.5 \cdot H' \\ 1 \end{bmatrix}, \quad ③$$

$$[x_0 \quad y_0 \quad 1]^T = \begin{bmatrix} x \cdot cos\theta + y \cdot sin\theta - 0.5 \cdot W' \cdot cos\theta - 0.5 \cdot H' \cdot sin\theta + 0.5 \cdot W \\ -x \cdot sin\theta + y \cdot cos\theta + 0.5 \cdot W' \cdot sin\theta - 0.5 \cdot H' \cdot cos\theta + 0.5 \cdot H \\ 1 \end{bmatrix}, \quad ④$$

4.3 关键代码

```
//前向映射
for (int x0 = 0; x0 < imgW; x0++) {
    for (int y = 0; y < imgH; y++) {
        int x1 = int(x0 * cos(rotateAngle) - y * sin(rotateAngle) +
                (-0.5 * imgW * cos(rotateAngle) + 0.5 * imgH * sin(rotateAngle) + 0.5 * newImgW));
        int y1 = int(x0 * sin(rotateAngle) + y * cos(rotateAngle) +
                (-0.5 * imgW * sin(rotateAngle) - 0.5 * imgH * cos(rotateAngle) + 0.5 * newImgH));

        fmapImgMat.at<Vec3b>(y1, x1)[0] = imgMat.at<Vec3b>(y, x0)[0];
        fmapImgMat.at<Vec3b>(y1, x1)[1] = imgMat.at<Vec3b>(y, x0)[1];
        fmapImgMat.at<Vec3b>(y1, x1)[2] = imgMat.at<Vec3b>(y, x0)[2];
    }
}
```

图 4-2

实现代码如上图所示，在本实验中，没有采用矩阵的直接运算，而是通过将变换矩阵化简，以此来计算得到坐标变换后的值。

5. 前向映射法与后向映射法

通俗的讲，前向映射法就是遍历原图的点，将其映射到旋转后的图像上。后向映射法，就是遍历旋转后的图像上的点，然后在原图上找到所需要的像素点，再将其填充到旋转后的图像中。

前向映射法一般不会被采用，因为经过旋转变换计算后得到的坐标值的类型是浮点型，而现实中图像的坐标值类型为整型，因此会导致旋转后的图像上的某些像素点是缺失的。

后向映射法则无上述问题，因为后向映射的原理是在原图上找对应的像素点，可以保证旋转后的图像每个像素点都可以从原图中得到或者生成。**后向映射的面临主要问题，是用什么算法在原图中寻找对应的像素点。**

在实践中，后向映射算法一般有最近邻法(Nearest Interpolation)、双线性插值(Bilinear Interpolation)、双三次插值(Bicubic interpolation)等算法，本实验会主要实现最邻近算法以及双线性插值法。

5.1 最邻近法及关键代码

最邻近算法的原理是通过旋转算法，直接找到某个离计算得到的坐标值最近的像素点，将其像素值赋给新图像对应的位置的像素点中。由于最邻近法直接寻找原图的像素点，不涉及其他额外的计算，因此计算效率最高，但由于这种“暴力”的处理，往往会破坏图像中原有的渐变关系，在图像的某些细节可能显示效果比较差。

图 5-1 为最邻近法的关键代码，注意红框代码中末端的”+0.5”再整体取整，以便取到离当前的坐标点最近的像素点。

```
//后向映射-最邻近算法
for (int x = 0; x < newImgW; x++) {
    for (int y = 0; y < newImgH; y++) {
        int x0 = int(x * cos(rotateAngle) + y * sin(rotateAngle) +
                (-0.5 * newImgW * cos(rotateAngle) - 0.5 * newImgH * sin(rotateAngle) + 0.5 * imgW) + 0.5);
        int y0 = int(-x * sin(rotateAngle) + y * cos(rotateAngle) +
                (0.5 * newImgW * sin(rotateAngle) - 0.5 * newImgH * cos(rotateAngle) + 0.5 * imgH) + 0.5);

        if (!(x0 > 0 && y0 > 0 && x0 < imgW && y0 < imgH)) continue;
        bmapImgMat.at<Vec3b>(y, x)[0] = imgMat.at<Vec3b>(y0, x0)[0];
        bmapImgMat.at<Vec3b>(y, x)[1] = imgMat.at<Vec3b>(y0, x0)[1];
        bmapImgMat.at<Vec3b>(y, x)[2] = imgMat.at<Vec3b>(y0, x0)[2];
    }
}
```

图 5-1

5.2 前向映射法与最邻近法的实验效果对比

图 5-2

图 5-3

采用前向映射算法得到的图像如图 5-2 所示，明显可见图像中存在着有规律的像素缺失。图 5-3 则采用后向映射算法中的最邻近算法，则无像素缺失的情况出现。

5.3 双线性插值法及关键代码

在上一节中，我们采用最邻近算法实现了无像素点缺失的图像旋转处理，但由于最邻近法往往会破坏图像原有的渐变效果，在某些细节上显得过渡特别生硬。因此，我们可以采用另一种算法，即**双线性插值法(Bilinear Interpolation)**。

双线性插值的坐标变换公式和最近邻法一样，**其区别在于后者是直接采用离计算后的坐标最近的 1 个像素点，而前者是通过找到离计算后的坐标值最近的 4 个像素点，以此做线性插值最终生成 1 个像素点。**因此，在图像的细节显示中，双线性插值法往往有着比较出色的效果。

同样地，我们可以通过画草图的方式来理解这个算法的原理。

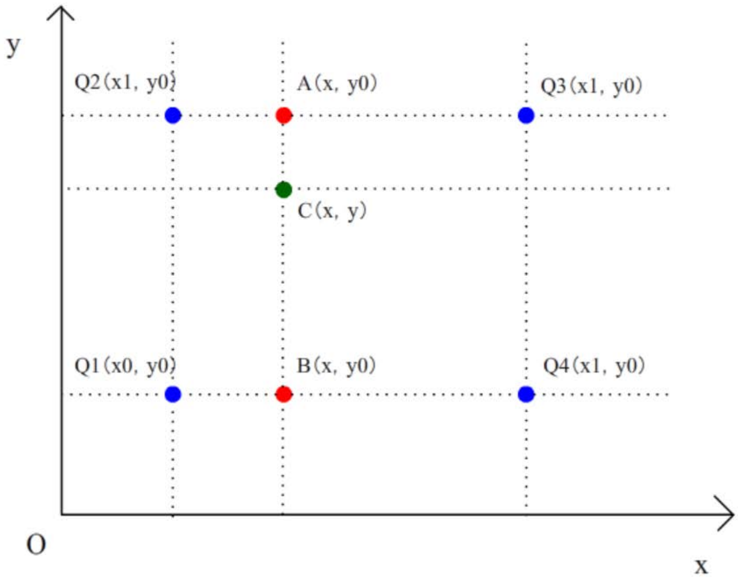

图 5-4

如上图所示，假设点 C(x,y)是通过坐标变换后得到的计算后的像素点坐标，那它的像素值会根据与之相邻的四个像素点计算得出。具体的计算步骤是先分别沿 X 轴方向，根据像素点 Q1 与点 Q4、点 Q2 与点 Q3 的像素值计算了两次单线性插值得到点 A 与点 B 的像素值，再根据点 A 与点 B 的像素值与坐标，再沿 Y 轴方向上计算一次单线性插值，进而得到像素点 C 的像素值。

若使用函数 f(X)表示该点 X 的像素值，则有以下公式：

$$f(B)=\frac{x_1-x}{x_1-x_1}\cdot f(Q1)+\frac{x-x_0}{x_1-x_0}\cdot f(Q4), \quad ①$$

$$f(A)=\frac{x_1-x}{x_1-x_1}\cdot f(Q2)+\frac{x-x_0}{x_2-x_0}\cdot f(Q3), \quad ②$$

$$f(C)=\frac{y_1-y}{y_1-y_0}\cdot f(B)+\frac{y-y_0}{y_1-y_0}\cdot f(A), \quad ③$$

因为 4 个像素点是相邻的，所以有关系式：$y_1-y_0=1, x_1-x_0=1$，并将式①，式②代入式③中并化简，可得像素点 C 的像素值计算如下：

$$\begin{aligned}f(C)&=\frac{y_1-y}{y_1-y_0}\cdot f(B)+\frac{y-y_0}{y_1-y_0}\cdot f(A)\\&=f(Q1)\cdot(x_1-x)\cdot(y_1-y)+f(Q4)\cdot(x-x_0)\cdot(y_1-y)+\\&\quad f(Q2)\cdot(x_1-x)\cdot(y-y_0)+f(Q3)\cdot(x-x_0)\cdot(y-y_0)\end{aligned}$$

关键代码如下图 4-5 所示：

ImgRotate.cpp

ImgRotate.cpp > main(void)

```cpp
//后向映射-双线性插值法
for (int x = 0; x < newImgW; x++) {
    for (int y = 0; y < newImgH; y++) {
        double x0_d = x * cos(rotateAngle) + y * sin(rotateAngle) + (-0.5 * newImgW * cos(rotateAngle) - 0.5 * newImgH * sin(rotateAngl
        double y0_d = -x * sin(rotateAngle) + y * cos(rotateAngle) + (0.5 * newImgW * sin(rotateAngle) - 0.5 * newImgH * cos(rotateAngl

        int x0 = int(x0_d);
        int y0 = int(y0_d);

        if (!(x0 > 0 && y0 > 0 && x0 < imgW - 1 && y0 < imgH - 1))
            continue;
        int x1 = x0 + 1;
        int y1 = y0 + 1;

        bmapImgMat2.at<Vec3b>(y, x)[0] = imgMat.at<Vec3b>(y0, x0)[0] * (x1 - x0_d) * (y1 - y0_d) +          //Q1
                                         imgMat.at<Vec3b>(y0, x0 + 1)[0] * (x0_d - x0) * (y1 - y0_d) +      //Q4
                                         imgMat.at<Vec3b>(y0 + 1, x0)[0] * (x1 - x0_d) * (y0_d - y0) +      //Q2
                                         imgMat.at<Vec3b>(y0 + 1, x0 + 1)[0] * (x0_d - x0) * (y0_d - y0);   //Q3

        bmapImgMat2.at<Vec3b>(y, x)[1] = imgMat.at<Vec3b>(y0, x0)[1] * (x1 - x0_d) * (y1 - y0_d) +
                                         imgMat.at<Vec3b>(y0, x0 + 1)[1] * (x0_d - x0) * (y1 - y0_d) +
                                         imgMat.at<Vec3b>(y0 + 1, x0)[1] * (x1 - x0_d) * (y0_d - y0) +
                                         imgMat.at<Vec3b>(y0 + 1, x0 + 1)[1] * (x0_d - x0) * (y0_d - y0);

        bmapImgMat2.at<Vec3b>(y, x)[2] = imgMat.at<Vec3b>(y0, x0)[2] * (x1 - x0_d) * (y1 - y0_d) +
                                         imgMat.at<Vec3b>(y0, x0 + 1)[2] * (x0_d - x0) * (y1 - y0_d) +
                                         imgMat.at<Vec3b>(y0 + 1, x0)[2] * (x1 - x0_d) * (y0_d - y0) +
                                         imgMat.at<Vec3b>(y0 + 1, x0 + 1)[2] * (x0_d - x0) * (y0_d - y0);
    }
```

图 5-5

5.4 最邻近法与双线性插值法的实验效果对比

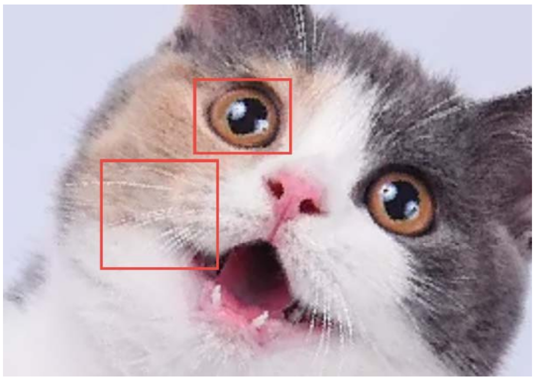

图 5-6 最邻近法

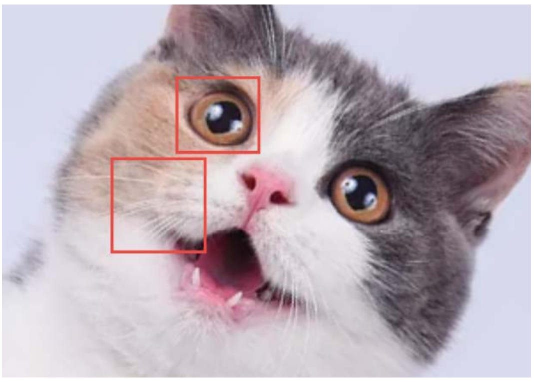

图 5-7 双线性插值法

图 5-6 展示的是采用最邻近法生成的图像放大后的局部细节，可见红框处渐变效果较差，过渡生硬。反之，采用双线性插值生成的图像 5-7 在对应的位置则有比较好的过渡与渐变效果。

6. 总结

- 前向映射法会导致旋转后的图像出现某部分像素的缺失。
- 后向映射法中常见的算法包括最邻近法、双线性插值法、双三次插值插值等算法。
- 最近邻法(Nearest Interpolation)的评价：计算速度最快，但在图像细节中显示的效果比较差。
- 双线性插值(Bilinear Interpolation)评价：双线性插值算法在性能与效果之间取得比较好的平衡，也是很多图像处理框架中属于默认算法。

7. 参考文章

[1] 双线性插值法，最邻近法 处理图像的旋转，放大 – (PIL+numpy 处理）. https://blog.csdn.net/HLW0522/article/details/81541583

[2] 一篇文章为你讲透双线性插值.https://zhuanlan.zhihu.com/p/110754637