Ex3：收集收据，并完成 Hough 变换

学号：17343124
姓名：伍斌
完成时间：2020.10.13

项目要求：

任务 1 - 收集名片收据:
1. 收集 20 张正常名片，用手机拍照保存，要求照片清晰，背景干净整洁.
2. 每张照片的文件名为：名片的电话号码+姓名.(参考本作业的例子数据)
（见文件夹中的 test 例）

任务 2 - 完成名片图像的边缘和角点检测:
根据上面的 20 张名片图，拍照时可能角度不正，需要对名片图像进行矫正，完成如下功能并输出结果（用 C++完成，把图像转为 BMP 格式，建议使用 CIMG 图像库完成图像读写和相关操作)。
1. 图像的边缘.
2. 提取并计算名片的各直线方程.
3. 提取名片图像的四个角点.

实现过程：

首先对图片进行处理，使用上次的 canny 算法，编写封装 Hough.h 和 Hough.cpp ，其中包括边缘直线的检测和交点的计算，并在图中画出直线和交点，最后导出图片。

Hough 变换（直线检测）算法原理：

在原始图像坐标系下的一个点对应了参数坐标系中的一条直线，同样参数坐标系的一条直线对应了原始坐标系下的一个点，然后，原始坐标系下呈现直线的所有点，它们的斜率和截距是相同的，所以它们在参数坐标系下对应于同一个点。这样在将原始坐标系下的各个点投影到参数坐标系下之后，看参数坐标系下有没有聚集点，这样的聚集点就对应了原始坐标系下的直线。

在实际应用中，y=k*x+b 形式的直线方程没有办法表示 x=c 形式的直线(这时候，直线的斜率为无穷大)。所以实际应用中，是采用参数方程 p=x*cos(theta)+y*sin(theta)。这样，图像平面上的一个点就对应到参数 p---theta 平面上的一条曲线上，其它的还是一样。

Hough 变换（直线检测）算法具体实现过程：

1. 将图片转化为灰度图；

```
CImg<float> Hough::RGBtoGray(const CImg<float>& srcImg) {
    CImg<float> grayImage = CImg<float>(srcImg._width, srcImg._height, 1, 1, 0);
    cimg_forXY(grayImage, x, y) {
        grayImage(x, y, 0) = (int)round((double)srcImg(x, y, 0, 0) * 0.299 +
```

```
                                    (double)srcImg(x, y, 0, 1) * 0.587 +
                                    (double)srcImg(x, y, 0, 2) * 0.114);
    }
    return grayImage;
}
```

2. 初始化霍夫空间；（这里用到了 EX2 Code1 中的 sobel 算子，查阅往年师兄的资料发现其实 Code0 的效果更好，但是由于 Code0 接口不够清晰所以还是用了 sobel 算子，编写完成之后的调参过程中发现效果确实不好，经常出现边缘识别错误。）

```
CImg<float> Hough::initHoughSpace() {
    // sobel 算子
    CImg<float> sobelx(3, 3, 1, 1, 0);
    CImg<float> sobely(3, 3, 1, 1, 0);
    sobelx(0, 0) = -1, sobely(0, 0) = 1;
    sobelx(0, 1) = 0, sobely(0, 1) = 2;
    sobelx(0, 2) = 1, sobely(0, 2) = 1;
    sobelx(1, 0) = -2, sobely(1, 0) = 0;
    sobelx(1, 1) = 0, sobely(1, 1) = 0;
    sobelx(1, 2) = 2, sobely(1, 2) = 0;
    sobelx(2, 0) = -1, sobely(2, 0) = -1;
    sobelx(2, 1) = 0, sobely(2, 1) = -2;
    sobelx(2, 2) = 1, sobely(2, 2) = -1;

    CImg<float> gradient_x = blurred_img;
    gradient_x = gradient_x.get_convolve(sobelx); // 计算x方向上的梯度
    CImg<float> gradient_y = blurred_img;
    gradient_y = gradient_y.get_convolve(sobely); // 计算y方向上的梯度

    int maxp = (int)sqrt(src._width*src._width + src._height*src._height);
    CImg<float> hough_space(360, maxp, 1, 1, 0); // 初始化hough space

    cimg_forXY(src, i, j) {
        double grad = sqrt(gradient_x(i, j)*gradient_x(i, j) + gradient_y(i, j)*gra
dient_y(i, j));
        if (grad > gradient_threshold) {
            src(i, j) = grad;
            cimg_forX(hough_space, alpha) {
                double theta = ((double)alpha*cimg::PI) / 180;
                int p = (int)(i*cos(theta) + j*sin(theta));
                if (p >= 0 && p < maxp) {
                    hough_space(alpha, p)++; // 累加矩阵
                }
            }
        }
```

```
    }
    return hough_space;
}
```

3. 投票算法找出霍夫空间中直线经过最多的点；

```
void Hough::findPeaks() {
    peaks.clear();
    cimg_forXY(houghspace, theta, p) {
        if (houghspace(theta, p) > vote_threshold) {
            bool flag = true;
            double alpha = (double)theta*cimg::PI / 180;
            // y 的范围
            const int y0 = ((double)p / (sin(alpha))) - double(x_min)*(1 / tan(alph
a));
            const int y1 = ((double)p / (sin(alpha))) - double(x_max)*(1 / tan(alph
a));
            // x 的范围
            const int x0 = ((double)p / (cos(alpha))) - double(y_min)*(tan(alpha));
            const int x1 = ((double)p / (cos(alpha))) - double(y_max)*(tan(alpha));

            if (x0 >= x_min && x0 <= x_max || x1 >= x_min && x1 <= x_max ||
                y0 >= y_min && y0 <= y_max || y1 >= y_min && y1 <= y_max) {
                for (int i = 0; i < peaks.size(); i++) {
                    if (sqrt((peaks[i].x - theta)*(peaks[i].x - theta)
                        + (peaks[i].y - p)*(peaks[i].y - p)) < peak_dis) {
                        flag = false;
                        if (peaks[i].cnt < houghspace(theta, p)) {
                            Point temp(theta, p, houghspace(theta, p));
                            peaks[i] = temp;
                        }
                    }
                }
                if (flag) {
                    Point temp(theta, p, houghspace(theta, p));
                    peaks.push_back(temp);
                }
            }
        }
    }
}
```

4. 寻找并画出直线；

```
void Hough::drawLines() {
    lines.clear();
```

```
    for (int i = 0; i < peaks.size(); i++) {
        double theta = double(peaks[i].x)*cimg::PI / 180;
        double k = -cos(theta) / sin(theta); // 直线斜率
        double b = double(peaks[i].y) / sin(theta);
        Line templine(k, b);
        lines.push_back(templine);
        cout << "Line " << i << ": y = " << k << "x + " << b << endl;
    }

    const double lines_color[] = { 255, 0, 0 };
    for (int i = 0; i < lines.size(); i++) {
        const int x0 = (double)(y_min - lines[i].b) / lines[i].k;
        const int x1 = (double)(y_max - lines[i].b) / lines[i].k;
        const int y0 = x_min*lines[i].k + lines[i].b;
        const int y1 = x_max*lines[i].k + lines[i].b;

        if (abs(lines[i].k) > 1) {
            result.draw_line(x0, y_min, x1, y_max, lines_color);
        }
        else {
            result.draw_line(x_min, y0, x_max, y1, lines_color);
        }
    }

}
```

5. 寻找并画出直线交点；

```
void Hough::drawIntersections() {
    intersections.clear();
    int k = 0;
    for (int i = 0; i < lines.size(); i++) {
        for (int j = i + 1; j < lines.size(); j++) {
            double k0 = lines[i].k;
            double k1 = lines[j].k;
            double b0 = lines[i].b;
            double b1 = lines[j].b;

            double x = (b1 - b0) / (k0 - k1);
            double y = (k0*b1 - k1*b0) / (k0 - k1);

            if (x >= 0 && x < src._width && y >= 0 && y < src._height) {
                Point tempPoint(x, y, 0);
                intersections.push_back(tempPoint);
```

```
                cout << "Intersection " << k++ << ": x = " << x << ", y = " << y <<
 endl;
            }
        }
    }

    const double intersections_color[] = { 255, 0, 0 };
    for (int i = 0; i < intersections.size(); i++) {
        result.draw_circle(intersections[i].x, intersections[i].y, 25, intersection
s_color);
    }
}
```

6. 最后把这几个函数封装为一个 Hough 变换接口。

```
CImg<float> Hough::houghProcess(CImg<float> srcImg) {
    this->src = RGBtoGray(srcImg); // 转灰度图
    this->blurred_img = src.get_blur(sigma); // 高斯滤波平滑
    this->houghspace = initHoughSpace(); // 初始化霍夫空间
    findPeaks(); // 找出霍夫空间中直线经过最多的点
    drawLines(); // 寻找并画出直线
    drawIntersections(); // 寻找并画出直线交点
    return result;
}
```

测试效果：

根据 Hough 类的构造函数：

```
Hough(CImg<float> srcImg, double sigma, double gradient_threshold, double vote_thre
shold, double peak_dis);
```

根据 20 张名片的测试，主要影响其实验效果的是参数 sigma 和 vote_threshold 。

拟合效果较好的数据，以收集到的名片（12345678910 十八纸）为例。输入图片：

1. 当输入的 sigma=10.5f，vote_threshold=1000 时：

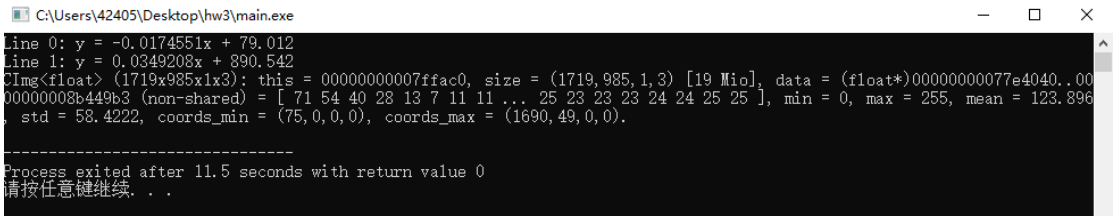
```
C:\Users\42405\Desktop\hw3\main.exe
CImg<float> (1719x985x1x3): this = 00000000007ffac0, size = (1719,985,1,3) [19 Mio], data = (float*)0000000007794040..00
00000008af49b3 (non-shared) = [ 71 54 40 28 13 7 11 11 ... 25 23 23 23 24 24 25 25 ], min = 0, max = 255, mean = 123.84,
 std = 58.2416, coords_min = (75,0,0,0), coords_max = (606,697,0,0).

--------------------------------
Process exited after 3.89 seconds with return value 0
请按任意键继续. . .
```

输出图像:

分析:

未能检测到边缘直线。应该是 sigma 参数不匹配导致 sobel 算子的边缘检测失败。

2. 当输入的 sigma=6.5f，vote_threshold=1000 时：

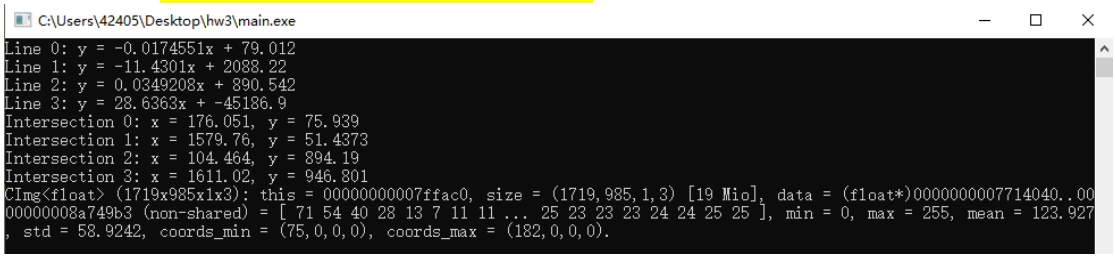


输出图像:

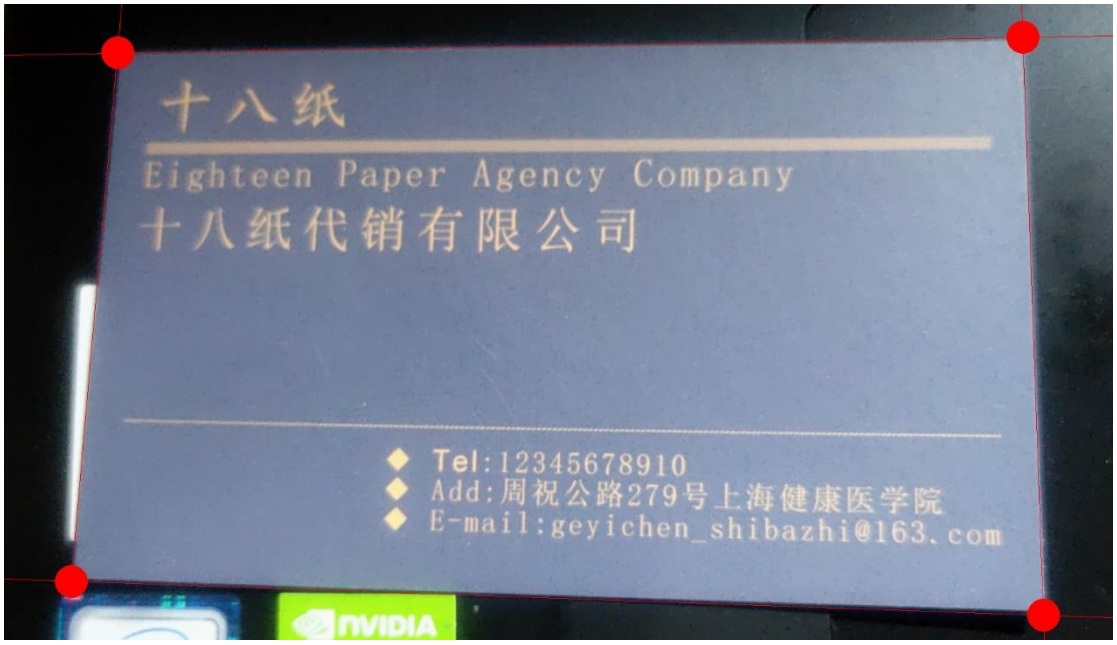
分析:

只检测画出了两条直线，sobel 算子没有问题，应该是 vote_threshold 的值太大，导致投票算法太过苛刻，检测出的直线变少。

3. 当输入的 sigma=6.5f，vote_threshold=500 时：



输出图像:



分析:

成功画出四条边缘直线，并成功标注出名片四个顶点，在此参数下 Hough 算法对本图片的拟合效果很好。

拟合效果较差的数据，以收集到的名片（18629629219 杨彩霞）为例。输入图片：

1. 当输入的 sigma=6.5f，vote_threshold=500 时：

```
C:\Users\42405\Desktop\hw3\main.exe
CImg<float> (1080x1440x1x3): this = 00000000007ffac0, size = (1080,1440,1,3) [17 Mio], data = (float*)000000000700b040..
00000000081d743f (non-shared) = [ 249 249 247 247 247 247 249 249 ... 148 149 150 151 151 150 149 148 ], min = 9, max =
251, mean = 175.909, std = 36.4735, coords_min = (671,541,0,2), coords_max = (35,8,0,0).

--------------------------------
Process exited after 43.65 seconds with return value 0
请按任意键继续. . .
```

输出图像:

分析:

未能检测到边缘直线。应该是 sigma 参数不匹配导致 sobel 算子的边缘检测失败。

2. 当输入的 sigma=4.5f，vote_threshold=500 时：

```
选择C:\Users\42405\Desktop\hw3\main.exe
Line 0: y = 0.0349208x + 178.108
Line 1: y = 11.4301x + -13561.9
Intersection 0: x = 1205.78, y = 220.215
CImg<float> (1440x1080x1x3): this = 00000000007ffac0, size = (1440,1080,1,3) [17 Mio], data = (float*)0000000007064040..
0000000023043f (non-shared) = [ 211 211 211 211 210 208 206 205 ... 188 188 187 187 187 186 186 186 ], min = 0, max =
255, mean = 175.595, std = 37.306, coords_min = (1186,0,0,1), coords_max = (1186,0,0,0).

--------------------------------
Process exited after 4.506 seconds with return value 0
请按任意键继续. . .
```

输出图像:

分析：

只检测画出了两条直线，sobel 算子没有问题，应该是 vote_threshold 的值太大，导致投票算法太过苛刻，检测出的直线变少。

3. 当输入的 sigma=4.5f，vote_threshold=300 时：

```
C:\Users\42405\Desktop\hw3\main.exe
Line 0: y = 0.781286x + 0
Line 1: y = 0.781286x + -0
Line 2: y = 0.0349208x + 178.108
Line 3: y = 0.0349208x + 530.323
Line 4: y = -0.0524078x + 592.812
Line 5: y = 0.0349208x + 713.435
Line 6: y = 11.4301x + -13561.9
Intersection 0: x = 0, y = 0
Intersection 1: x = 238.635, y = 186.442
Intersection 2: x = 710.541, y = 555.136
Intersection 3: x = 711.068, y = 555.547
Intersection 4: x = 955.879, y = 746.815
Intersection 5: x = 1273.57, y = 995.02
Intersection 6: x = 238.635, y = 186.442
Intersection 7: x = 710.541, y = 555.136
Intersection 8: x = 711.068, y = 555.547
Intersection 9: x = 955.879, y = 746.815
Intersection 10: x = 1273.57, y = 995.02
Intersection 11: x = 1205.78, y = 220.215
Intersection 12: x = 715.566, y = 555.311
Intersection 13: x = 1236.69, y = 573.509
Intersection 14: x = 1232.73, y = 528.208
Intersection 15: x = 1252.76, y = 757.182
CImg<float> (1440x1080x1x3): this = 00000000007ffac0, size = (1440,1080,1,3) [17 Mio], data = (float*)0000000007016040..
00000000081e243f (non-shared) = [ 255 255 255 255 255 255 255 255 ... 188 188 187 187 187 186 186 186 ], min = 0, max =
255, mean = 174.576, std = 40.6459, coords_min = (0,0,0,1), coords_max = (0,0,0,0).

--------------------------------
Process exited after 6.167 seconds with return value 0
请按任意键继续. . .
```

输出图像：

分析：

画出了多余的直线和点，并且左边的边缘未被检测到，应该还是 sigma 参数的值存在问题。同时 vote_threshold 的值太小，导致检测出的直线变多。

4. 当输入的 sigma=5.0f，vote_threshold=350 时：

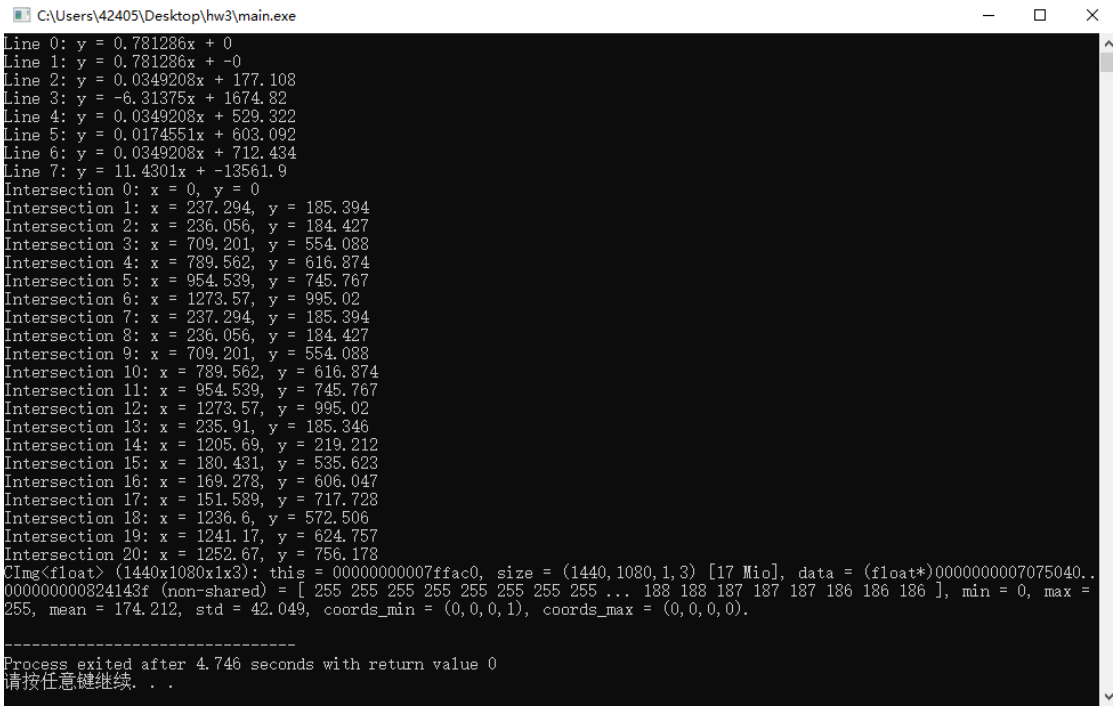
```
C:\Users\42405\Desktop\hw3\main.exe
Line 0: y = 0.781286x + 0
Line 1: y = 0.781286x + -0
Line 2: y = 0.0349208x + 178.108
Line 3: y = 0.0349208x + 530.323
Line 4: y = 11.4301x + -13550.5
Intersection 0: x = 0, y = 0
Intersection 1: x = 238.635, y = 186.442
Intersection 2: x = 710.541, y = 555.136
Intersection 3: x = 1272.49, y = 994.179
Intersection 4: x = 238.635, y = 186.442
Intersection 5: x = 710.541, y = 555.136
Intersection 6: x = 1272.49, y = 994.179
Intersection 7: x = 1204.77, y = 220.18
Intersection 8: x = 1235.68, y = 573.474
CImg<float> (1440x1080x1x3): this = 00000000007ffac0, size = (1440,1080,1,3) [17 Mio], data = (float*)000000000728b040..
000000000845743f (non-shared) = [ 255 255 255 255 255 255 255 255 ... 188 188 187 187 187 186 186 186 ], min = 0, max =
255, mean = 175.045, std = 39.1747, coords_min = (0,0,0,1), coords_max = (0,0,0,0).

--------------------------------
Process exited after 6.154 seconds with return value 0
请按任意键继续. . .
```

输出图像：

分析：

左边和下边的边缘未被检测到，并且画出了一条多余的直线，参数仍然存在问题。

4. 当输入的 sigma=3.5f，vote_threshold=375 时：



输出图像：

分析：

多画出了 3 条直线和 9 个点。但根据反复论证，这已经是本算法能输出的最佳图像。其拟合效果如此之差的主要原因：拍摄的图像背景和名片之间的色差太小，导致转化为灰度图像之后，用 sobel 算子计算边缘的难度大增。具体表现在左边那条边不易被检测到，一旦检测出这条边，多余的直线也会被投票画出来。同时由于名片字体和拍摄光线原因，“亿家尚品整体厨房”这行字无论参数如何变换都会被检测为边缘。

思考：如何在保证精度的结果情况下加快运行速度？

（这部分有参考 CSDN 博客：https://blog.csdn.net/Jeanet_1/article/details/49387967 ）

Hough 变换检测直线的原理是首先得到边缘点，对每一边缘点得到过边缘点的所有直线，直线参数为极坐标下的（row,theta），而每一个（row,theta)都对应极坐标中一个点，对每一条直线对应的点记为投票一次，最终记录投票数，投票数大于某一域值的即可看作一条直线。

由此我们可以看出，算法中计算量大，主要是因为：第一，对边缘点做一个 360 度的直线组，即经过该边缘点上的所有直线，对其对应的（row,theta)投票。第二，对每一边缘点都要重复此过程。在边缘点众多的情况下，这样的计算量是非常可怕的。

因此，对算法的改进主要针对此两个方面。

1、用两次 sobel 算子相减得到单相素宽度的边缘。sobel 算子提取边缘是不错的算法，速度相对较快。但一次 sobel 算子提取得到的边缘一般边缘点较多。用两次 sobel 算子相减可以得到单像素的边缘。这样可以大大减少边缘点的个数

2、计算每一边缘点的梯度方向，统计每个方向上的点的个数，找到边缘点个数最多的方向，做为粗略的直线方向。此处，为了容错，也可以考虑对梯度方向值进行拟合后，找几

个波峰做为备选直线方向。视具体情况。

3、得到大致方向后，将与此方向偏差较多的边缘点剔除。做完这一步后，你会发现，边缘点已经减少很多了。而你需要的信息不会丢失。

4、在直线方向的正负 n 度内（这个 n 主要看你的容错标准），做 Hough 变换。找到直线。