

HW2 Contour/Line extraction

作业要求：

对输入的一张彩色图像，检测其中的圆形与直线，并将检测结果显示在原图上。

1. 检测算法的核心功能需要自己写代码实现，**不能调用OpenCV或其他SDK里与圆形/直线检测相关的函数**；如果要用到边缘检测，可以调用OpenCV函数
2. 在原图上显示最终的检测结果
3. 单独显示一些关键的中间结果
4. 必须对指定的三张测试图像(coin, seal, highway)调试结果。此外，自己还可以自愿加一些测试图像。

信息：

姓名：鲍奕帆

学号：3180103499

邮箱：3180103499@zju.edu.cn

完成日期：2020年12月11日晚上8点30

一、软件开发说明

- 计算机与操作系统

macOS Catalina版本10.15.5

MacBook Pro(13-inch, 2020, Four Thunderbolt 3 ports)

CPU: 2 GHz 四核 Intel Core i5

内存: 16 GB 3733 MHz LPDDR4

启动磁盘: Macintosh HD

图形卡: Intel Iris Plus Graphics 1536 MB

内核(uname -a): Darwin EvandeMacBook-Pro.local 19.5.0 Darwin Kernel Version 19.5.0: Tue May 26 20:41:44 PDT 2020; root:xnu-6153.121.2~2/RELEASE_X86_64 x86_64

- opencv 版本

opencv_4.4.0

- 编程语言

C++11

- IDE

XCode Version 12.2 (12B45b)

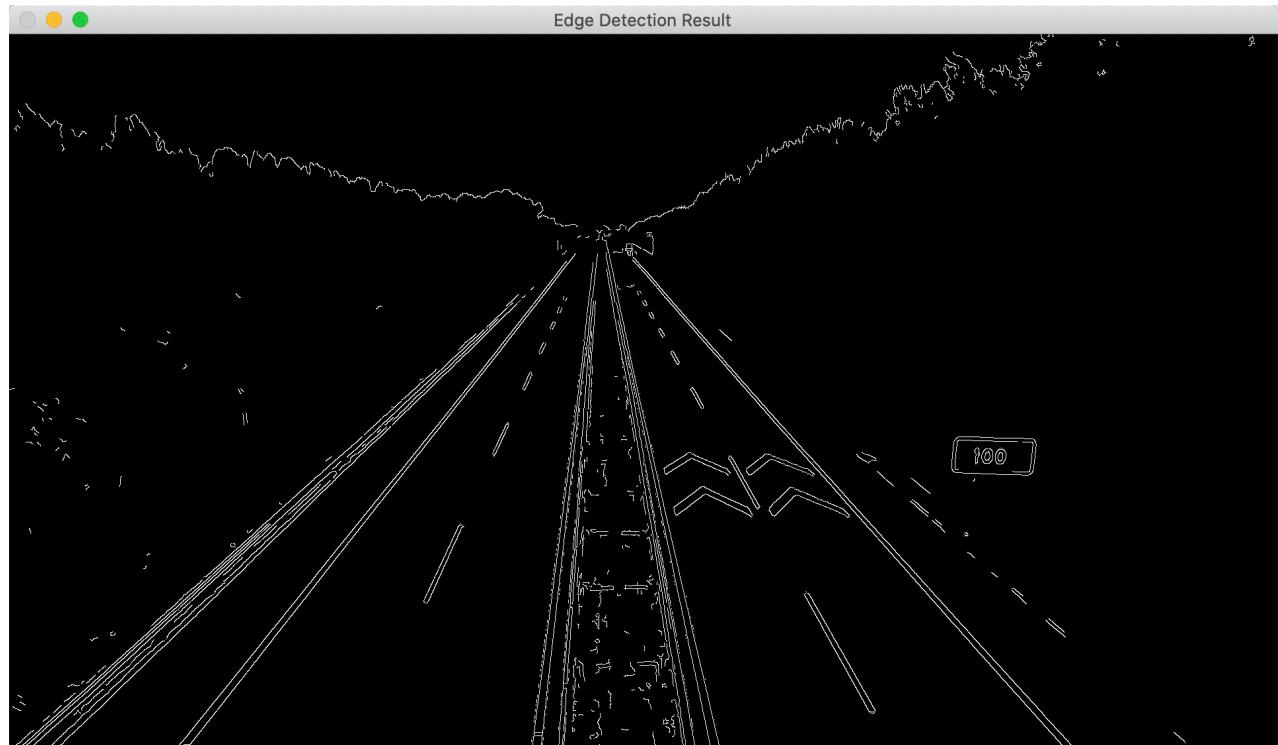
- 编译器

Apple clang version 11.0.3 (clang-1103.0.32.62)

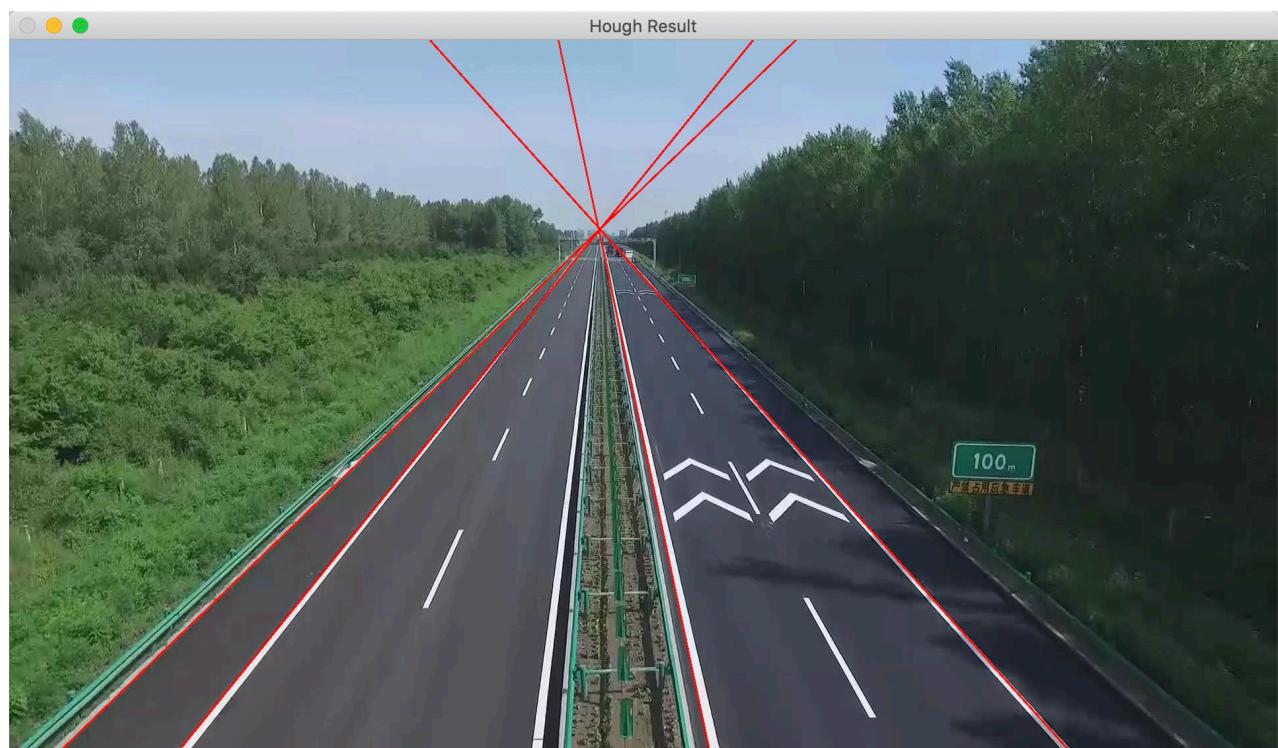
二、实验结果展示与分析

1. highway测试结果

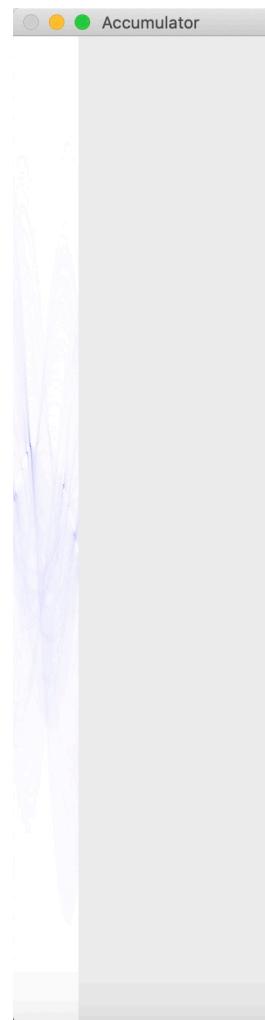
平滑后canny 检测结果



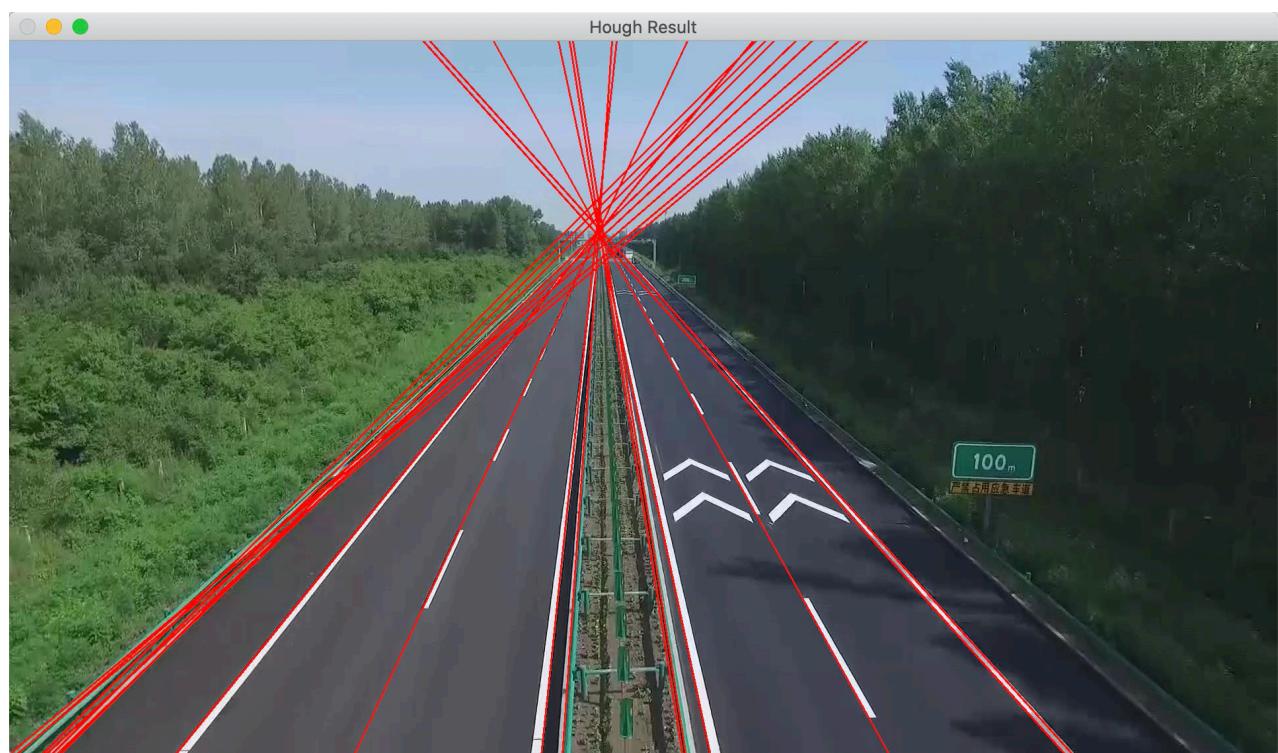
原图+threshold = width / 4;



累加器结果图。由于theta只有0到180，因此显示的比较狭窄，不够清晰。



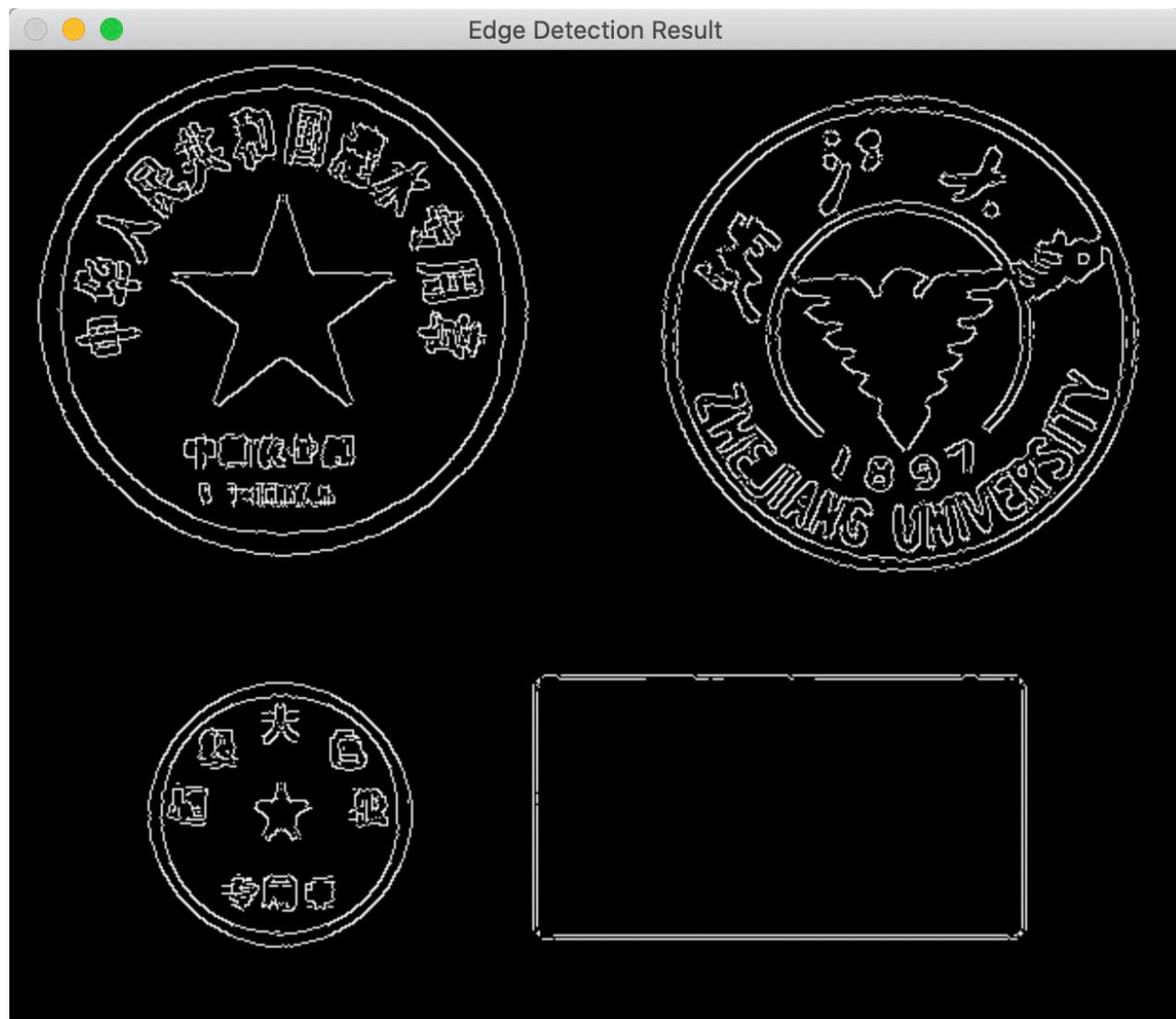
threshold = width/4 - 185



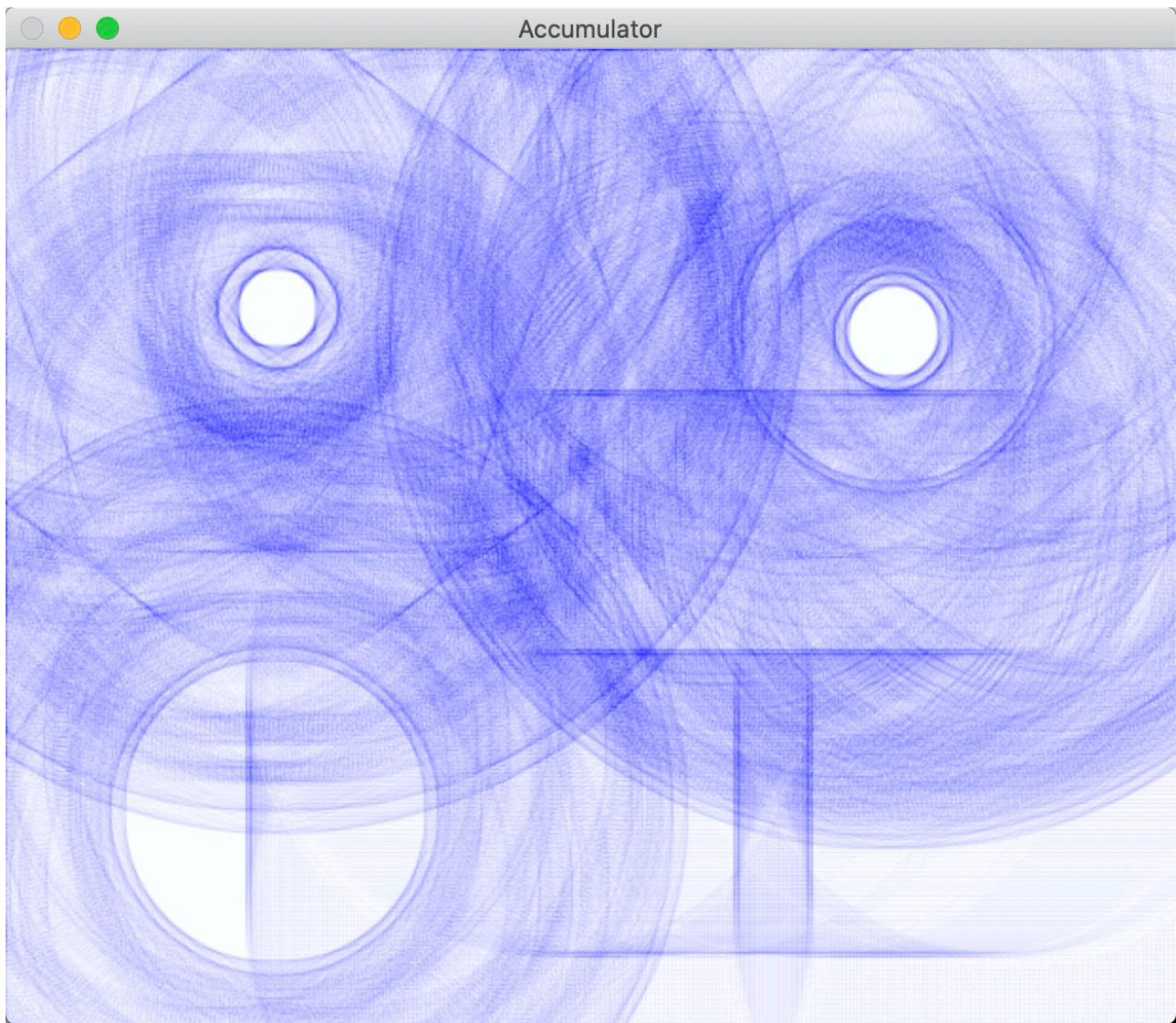
此图很好的反应了最终的结果。对比一下canny检测结果，可以看到，所有的直线都已经覆盖。结果较好。

2. seal检测结果

平滑后canny边缘检测结果图



累加器结果图。



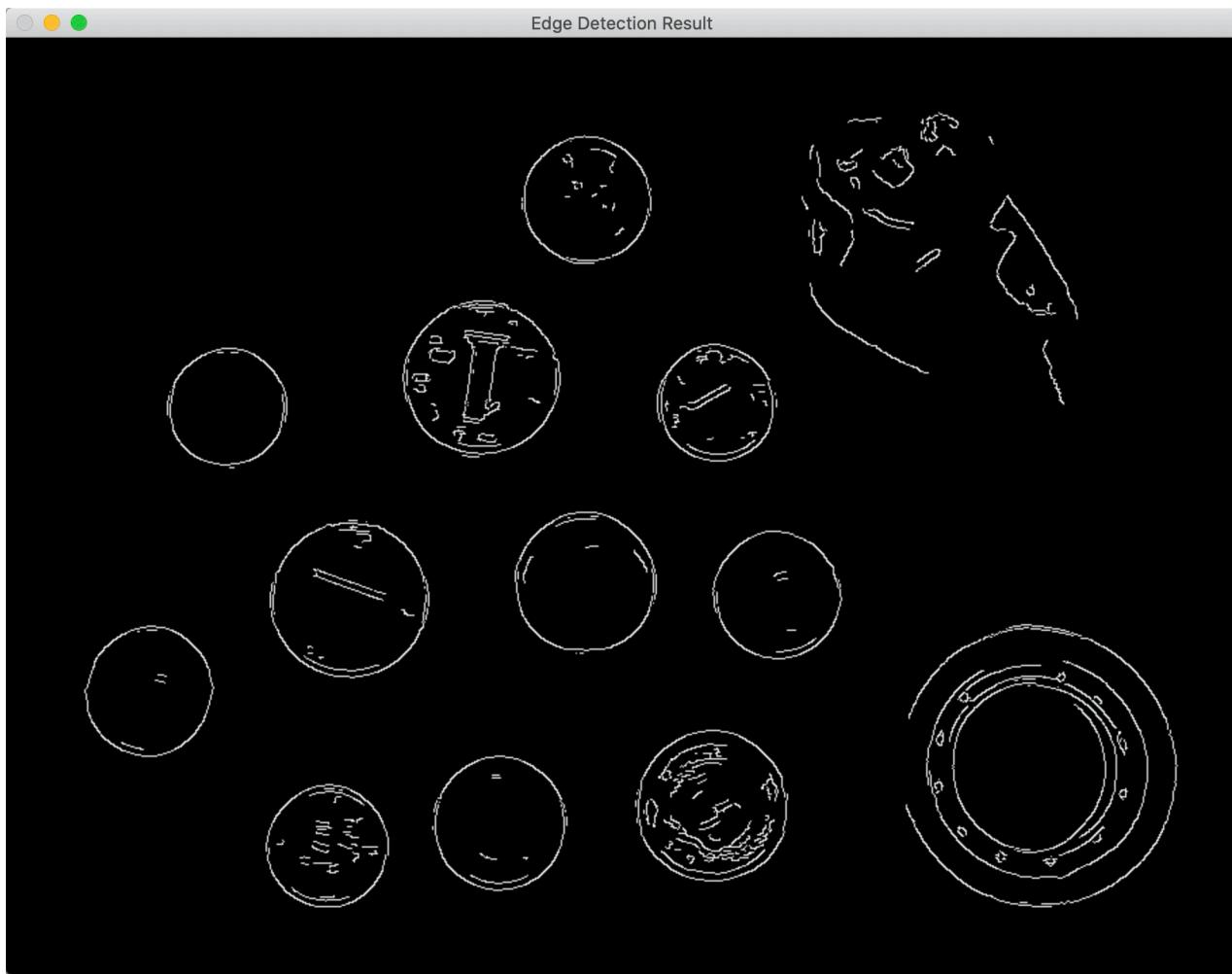
最终圆形检测图



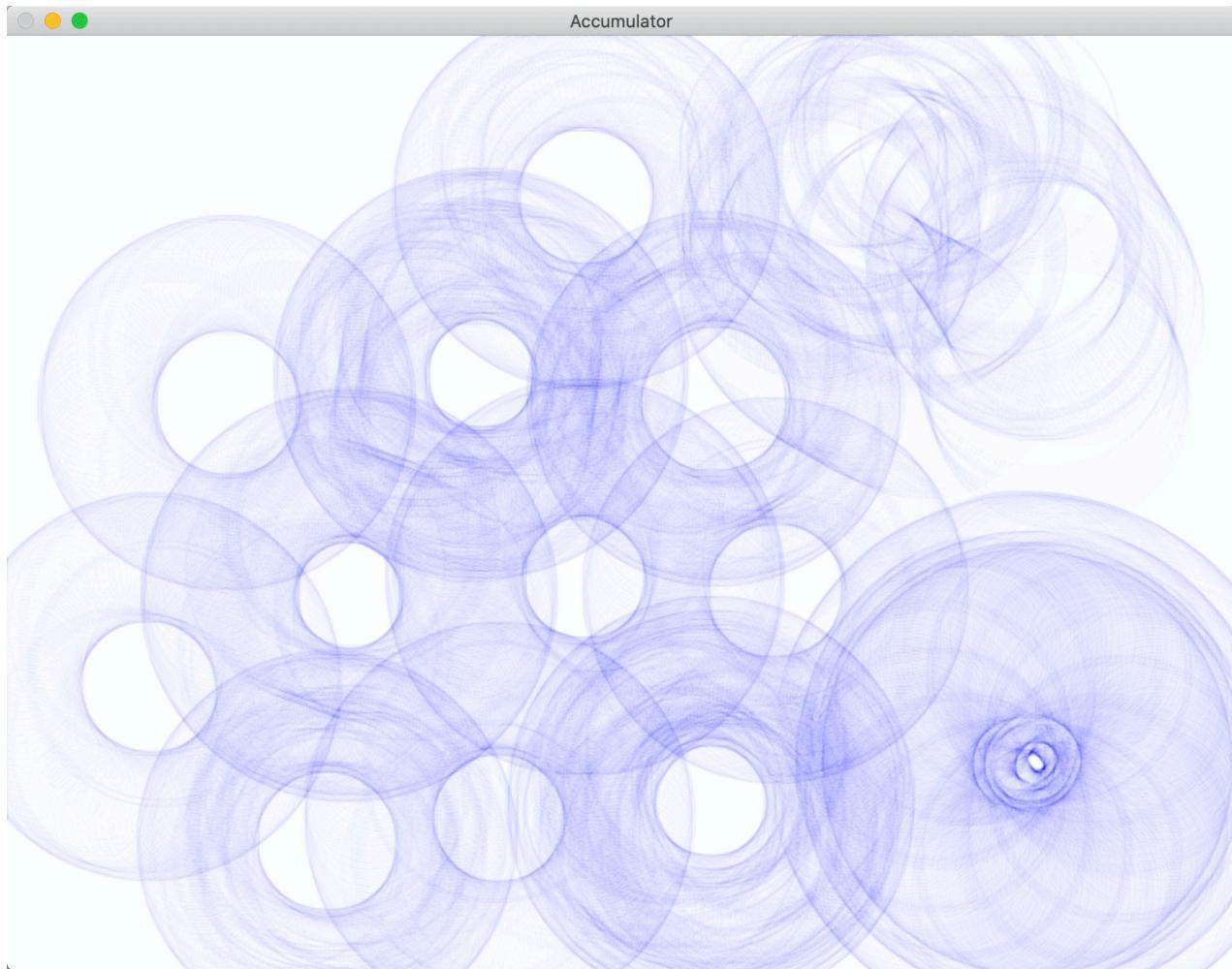
可以看到，所有的圆形都得到了检测。这里我没有特意的去舍去重复的圆形，主要是为了更好的比较效果。事实上要消除也比较简单，但作业的主要目的还是根据原理实现hough圆形检测算法。这里自己的参数设置为半径范围25到150，步进为2. threshold为周长的0.85倍。

3. coin检测结果

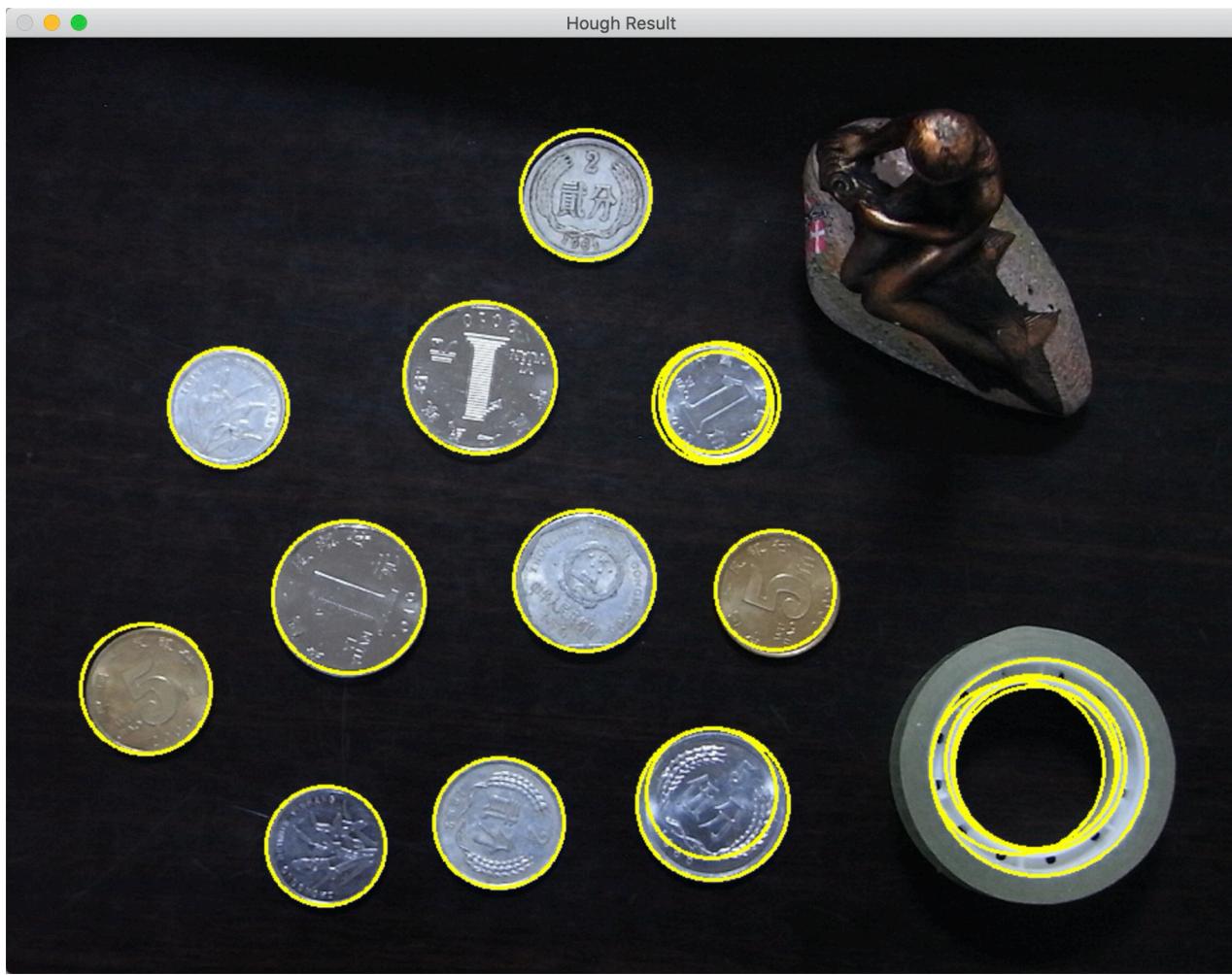
平滑后canny检测结果图



累加器结果图



最终圆形检测图

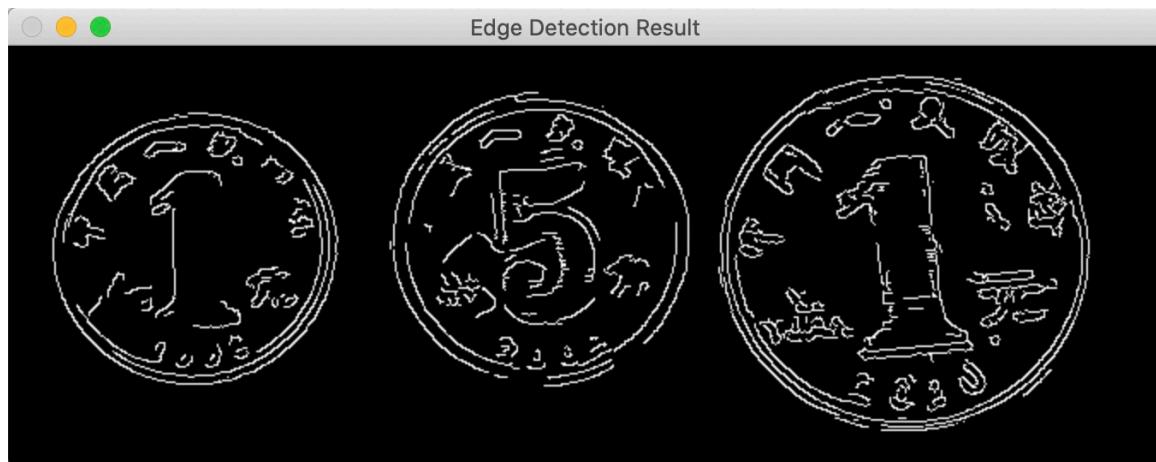


对比canny边缘检测得到的结果，可以看出，几乎所有的圆形都得到了绘制，尤其是透明胶那部分很好。当然，也有一个硬币效果不是那么好。可能还需要自己进一步调整参数。这里的参数为r的范围是20到100，步进为4.threshold为0.95的周长。

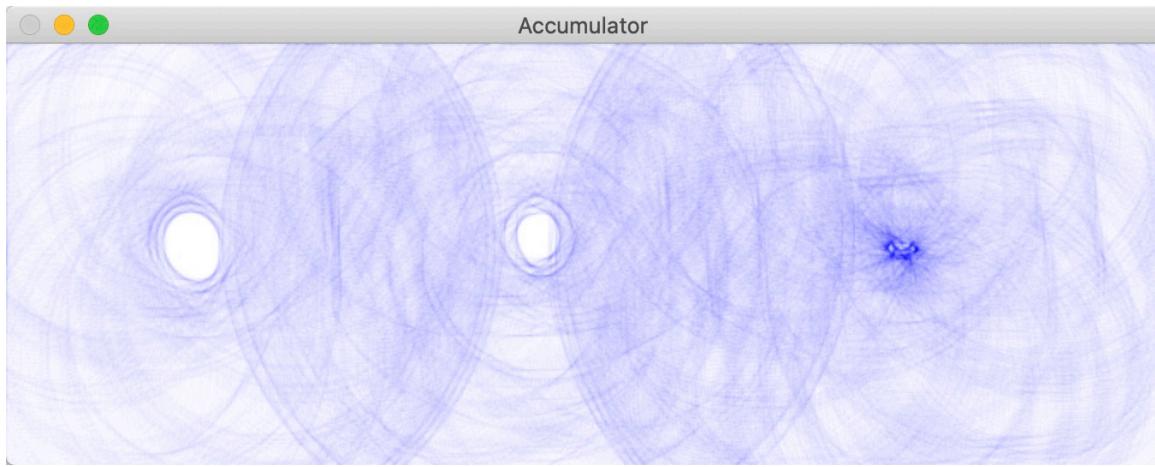
我同时将结果对比了opencv中自带的函数，结果差别也不是很大。仅仅是他的圆形绘制更加细腻和准确。总体结构类似。

4. 自愿检测--圆形--自己的硬币

canny检测结果



累加器图形化显示

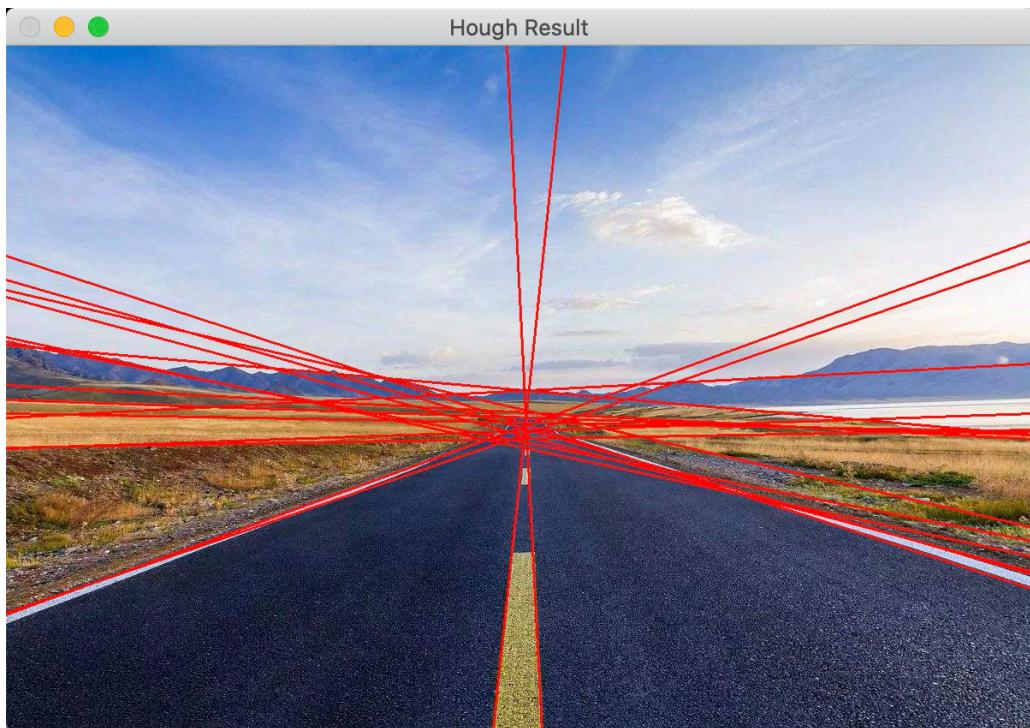


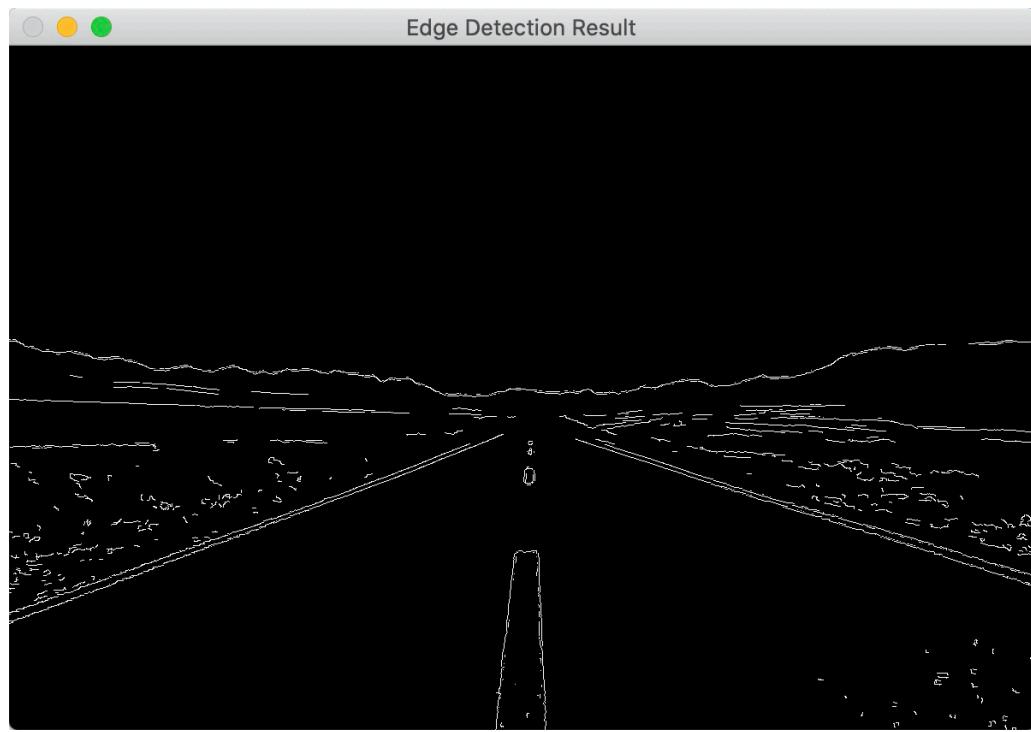
圆形检测结果

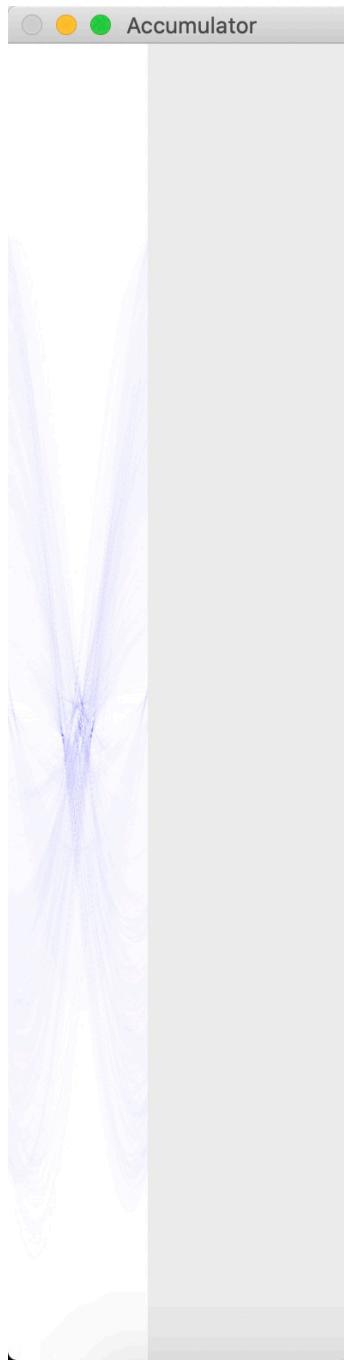


这里也已经是比较精确了。边缘中的圆都画了出来。不过对于最后的一个1元。我还是遇到了一些问题，他的检测结果一直不是很好。这里自己的参数设置为半径范围22到100, threshold为1.01倍的周长。

5. 自愿检测--直线--图像中的直线



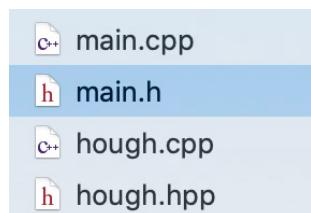




总体来看，检测结果不错。画出的直线和边缘检测中自己看到的基本一致。累加器的值，也可以相对比较明显的看出粗细，从而确定直线。

三、程序基本描述

- 源文件结构



一共有三个cpp文件，分别是main.cpp, hough.cpp, hough.hpp

下面叙述各个模块的大致功能

- main.h

```
//opencv库
#include <opencv2/highgui/highgui_c.h>
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/core/core.hpp"
//语言库
#include <stdio.h>
#include <string>
#include <map>
#include <iostream>
#include "hough.hpp"
std::string imgPath = "/Users/evan/Library/Developer/Xcode/DerivedData/cvTest-ebynplkownzjsachtzimeuxhcifw/Build/Products/Debug/myHighway.jpg";
int thisThreshold = 0;           //默认的threshold 直接在函数中调整即可
const char* CW_IMG_ORIGINAL    = "Hough Result";
const char* CW_IMG_EDGE         = "Edge Detection Result";
const char* CW_ACCUMULATOR     = "Accumulator";
void myHoughLinePer(string imgPath, int threshold);
void myHoughCirclePer(string imgPath, int threshold);
```

myHoughLinePer()函数完成了HoughLine的检测，以及各种图形的绘制。

myHoughCirclePer()函数完成了HoughCircle的检测，以及各种图形的绘制。

CW_IMG_ORIGINAL、CW_IMG_EDGE、CW_ACCUMULATOR分别为三幅图像的显示窗口名字。

thisThreshold设置为0。事实上是在函数内部修改threshold，方便调整。

imgPath为图像路径，根据测试需要修改。

- main函数

```
int main(int argc, char** argv) {

    if(imgPath.empty())
    {
        //usage(argv[0]);
        return -1;
    }
    cv::namedWindow(CW_IMG_ORIGINAL, cv::WINDOW_AUTOSIZE);
    cv::namedWindow(CW_IMG_EDGE,           cv::WINDOW_AUTOSIZE);
    cv::namedWindow(CW_ACCUMULATOR,       cv::WINDOW_AUTOSIZE);
    //myHoughLinePer(imgPath, thisThreshold);
    //myHoughCirclePer(imgPath, thisThreshold);
    myHoughLinePer(imgPath, thisThreshold);
    return 0;
}
```

main函数的基本功能就是设置直接调用对应的Hough变化函数，传入图像路径和threshold，而后根据结果进行绘制。不同的情况需要手动修改main函数代码

- myHoughLinePer()函数实现

```
void myHoughLinePer(string imgPath, int threshold)
{
    Mat gray;           //灰度图--用于边缘检测
    Mat imgCanny;       //边缘检测结果
    Mat imgBlur;        //平滑去噪音结果--用于边缘检测
    Mat imgOri = imread(imgPath, 1); //原始图片读入
    cvtColor(imgOri, gray, COLOR_BGR2GRAY);
    blur(gray, imgBlur, Size(5, 5)); //平滑
    Canny(imgBlur, imgCanny, 100, 150, 3); //Canny 边缘检测 可以进行 参数调整
    int width = imgCanny.cols;           //图像宽度
    int height = imgCanny.rows;         //图像高度
    BaoYifan::HoughLine hough;
    hough.calAccum(imgCanny.data, width, height);
    if(threshold == 0)
        threshold = width>height?width/4:height/4; //默认threshold 后面可以根据按键进行修改 从而显示更多或者更少直线
    while(1)
    {
        Mat imgRes = imgOri.clone(); //最终结果图 结合 直线
        //遍历累加器 找大于threshold的直线 返回直线的两个点对
        vector< pair< pair<int, int>, pair<int, int> >> lines =
        hough.getLines(threshold); //hough.GetLines(threshold);
        //根据点对 将直线绘制在结果图上
        vector< pair< pair<int, int>, pair<int, int> >>::iterator it;
        for(it=lines.begin();it!=lines.end();it++)
        {
            cv::line(imgRes, cv::Point(it->first.first, it->first.second),
            cv::Point(it->second.first, it->second.second), cv::Scalar( 0, 0, 255), 2, 8);
        }
        //直线绘制函数
    }
    //累加器的绘制
    int aw, ah, maxa; //累加器的宽 高
    aw = ah = maxa = 0;
    const unsigned int* accu = hough.getAccum(&aw, &ah); //返回对应累加器的数据
    //寻找最大计数值
    for(int p=0;p<(ah*aw);p++)
    {
        if((int)accu[p] > maxa)
            maxa = accu[p];
    }
    double contrast = 1.0;
    double coef = 255.0 / (double)maxa * contrast; //以最大值为系数最大值
    cv::Mat img_accu(ah, aw, CV_8UC3); //RGB图像
```

```

for(int p=0;p<(ah*aw);p++)
{
    unsigned char c = (double)accu[p] * coef < 255.0 ? (double)accu[p]
* coef : 255.0;
    img_accu.data[(p*3)+0] = 255;           //RGB三个像素点的设置
    img_accu.data[(p*3)+1] = 255-c;
    img_accu.data[(p*3)+2] = 255-c;
}
cv::imshow(CW_IMG_ORIGINAL, imgRes);
cv::imshow(CW_IMG_EDGE, imgCanny);
cv::imshow(CW_ACCUMULATOR, img_accu);
//按键响应 增大或者减小 threshold
char c = cv::waitKey(360000);
if(c == '+')
{
    threshold += 5;
    cout << "print +" << endl;
}
if(c == '-')
{
    threshold -= 5;
    cout << "print -" << endl;
}
if(c == 27)
    break;
}
}

```

其基本执行流程如下

1. 读入图片，图片预处理
2. Canny检测
3. Hough计算累加器
4. 根据累加器和threshold获得直线点对
5. 绘制检测图像、累加器图像、canny检测图像
6. 响应键盘输入

其中3和4将在后面的程序模块实现算法中具体描述，也即HoughLine核心代码。

HoughCircle与之类似，将在后面程序实现算法中描述。

四、程序模块实现算法(步骤与要点)

- Hough 直线检测算法理论

Hough变换算法：4步曲

1. 适当地**量化**参数空间（合适的精度即可）.
2. 假定参数空间的每一个单元都是一个累加器，把累加器**初始化**为零.
3. 对图像空间的每一点，在其所满足的参数方程对应的累加器上**加1**.
4. 累加器阵列的**最大值**对应模型的参数.

空间变化公式为 $r = x\cos(\theta) + y\sin(\theta)$ 。参数为r和theta

反变换计算公式如下

$$r = x.\cos(\theta) + y.\sin(\theta)$$

$$x = \frac{r - y.\sin(\theta)}{\cos(\theta)}$$

$$y = \frac{r - x.\cos(\theta)}{\sin(\theta)}$$

- Hough 直线检测算法实现

1. 累加器计算

```
int HoughLine::calAccum(unsigned char* imgData, int width, int height)
{
    imgWidth = width;           //图像宽度
    imgHeight = height;         //图像高度
    //创建计数器
    double r_max = ((sqrt(2.0) * (double)(height>width?height:width)) / 2.0);
    //对角线 最长 r的最大值
    accumHeight = r_max * 2.0; //这里 -r到r 因此总共坐标长度为2r_max--0到2r_max对应
    //到-r到r
    accumWidth = 180;           //180弧度 横坐标
```

```

    accumulator = (unsigned int*)calloc(accumWidth*accumHeight, sizeof(unsigned
int)); //二维数组
    double centerX = width / 2; //图像中心点x坐标
    double centerY = height / 2; //图像中心点y坐标

    //遍历行列像素点 转换 设置累加器
    //IMPORTATN
    for(int py = 0; py < height; py++)
    {
        for(int px = 0; px < width; px++)
        {
            if(imgData[(py*width + px)] > 250)
            {   //遍历所有的theta(0-180)计算对应的r 即表示一个像素点 贡献的他经过的所有
                直线
                    for(int theta = 0; theta < 180; theta++)
                    {
                        double r = ( ((double)px - centerX) * cos((double)theta *
DEG2RAD)) + (((double)py - centerY) * sin((double)theta * DEG2RAD)); //公式
r=xcos(theta)+ysin(theta)计算离图像中点的距离//有负有正
                        accumulator[ (int)((round(r + r_max) * 180.0)) + theta]++;
//对应的累加器计数+1
                    }
                }
            }
        return 0;
    }
}

```

主要步骤已在代码中给出注释。即遍历图像中的行和列，找到有效的像素点。对每个点，遍历所有的theta，根据前面给出公式计算r。于是就得到了经过一个像素点，所有离散化的直线对应的(r, theta)。将累加器+1.

2. 获得直线点对

```

vector< pair< pair<int, int>, pair<int, int> > > HoughLine:: getLines(int
threshold)
{
    vector< std::pair< std::pair<int, int>, std::pair<int, int> > > lines; //返
    回直线结果
    if(accumulator==NULL)
        return lines; //空
    //遍历accumulator
    for(int r = 0; r < accumHeight; r++) //行r
    {
        for(int theta = 0; theta < accumWidth; theta++) //列theta
        {
            if((int)accumulator[(r*accumWidth)+theta] >= threshold) //判断大于
threshold 则此点/直线(r theta)满足
            {

```

```

//判断是否是9x9格子中的局部最大
int max = accumulator[(r*accumWidth) + theta];
for(int ly=-4;ly<=4;ly++)
{
    for(int lx=-4;lx<=4;lx++)
    {
        if( (ly+r>=0 && ly+r< accumHeight) && (lx+theta>=0 &&
lx+theta
                                         <accumWidth) )
        {
            if( (int)accumulator[ ( (r+ly)*accumWidth) +
(theta+lx) ] > max )
            {
                max = accumulator[ ( (r+ly)*accumWidth) +
(theta+lx) ];
                ly = lx = 5;
            }
        }
    }
}
if(max > (int)accumulator[(r*accumWidth) + theta])
    continue;

//反计算 直线上两个点的值 并加入到直线中
//公式 y = (r - xcos(theta)) / sin(theta)
//公式 x = (r - ysin(theta)) / cos(theta)
int x1, y1, x2, y2;
x1 = y1 = x2 = y2 = 0;
if(theta >= 45 && theta <= 135) //这里条件判断主要是为了避开两个等于0
的情况
{
    x1 = 0; //第一个点的选取 选择y轴上点
    y1 = ((double)(r-(accumHeight/2)) - ((x1 - (imgWidth/2)) *
cos(theta * DEG2RAD))) / sin(theta * DEG2RAD) + (imgHeight / 2);
    x2 = imgWidth; //第二个点的选取 选择与图像右边缘的交点
    y2 = ((double)(r-(accumHeight/2)) - ((x2 - (imgWidth/2)) *
cos(theta * DEG2RAD))) / sin(theta * DEG2RAD) + (imgHeight / 2);
}
else
{
    y1 = 0; //第一个点的选取 x轴上的点
    x1 = ((double)(r-(accumHeight/2)) - ((y1 - (imgHeight/2)) *
sin(theta * DEG2RAD))) / cos(theta * DEG2RAD) + (imgWidth / 2);
    y2 = imgHeight; //第二个点的选取 选择与图像下边缘的交点
    x2 = ((double)(r-(accumHeight/2)) - ((y2 - (imgHeight/2)) *
sin(theta * DEG2RAD))) / cos(theta * DEG2RAD) + (imgWidth / 2);
}

```

```

        lines.push_back(pair< pair<int, int>, pair<int, int> >
(pair<int, int>(x1,y1), pair<int, int>(x2,y2)));
    }
}
return lines;
}

```

主要步骤已在代码中给出注释。即遍历累加器，判断累加器的值大于threshold，而后选取9x9方格中的local maxima。最后根据前面给出的反计算公式，计算得到有效的点对(用cv::Line绘制)。以向量形式返回。

- Hough 圆形检测算法理论

圆弧检测：6步曲

- (1) 量化关于 \mathbf{a} , \mathbf{b} 的参数空间到合适精度
- (2) 初始化所有累加器为 0
- (3) 计算图像空间中边缘点的梯度幅度 $G_{mag}(x, y)$
和角度 $\theta(x, y)$
- (4) 若边缘点参数坐标满足 $b = a \tan \theta - x \tan \theta + y$
则该参数坐标对应的累加器加 1
- (5) 拥有最大值的累加器所在的坐标即为图像空
间中的圆心之所在
- (6) 得到圆心坐标之后，我们可以很容易反求 \mathbf{r}

- 给定一个圆弧，它应该有三个参数：两个用来确定圆心，一个给出半径

$$x = a + r \cos \theta$$

$$y = b + r \sin \theta$$



令 θ 为边缘点处的梯度角

$$a = x - r \cos \theta$$

$$b = y - r \sin \theta$$



$$b = a \tan \theta - x \tan \theta + y$$



关于 a, b 的参数空间

浙江大学计算机学院

(仅供本课程内部学

习)

但这里自己实现的时候，没有采用梯度的方式。仍然是以 ab 作为参数空间，也即累加器的维度。但具体实现的时候

采用公式 $a = x - r \cos(\theta)$ 和 $b = y - r \sin(\theta)$ 。而后按步骤遍历最小半径和最大半径，遍历 θ 角度，得到在每一个特定半径下，经过一个像素点 (px, py) 的所有圆(圆心 (a, b))。这也可能导致自己实现的算法不够完善，从而最终检测结果相对来说没有直线检测那么好。

- myHoughCirclePer()函数

```
void myHoughCirclePer(string imgPath, int threshold)
{
    Mat gray; //灰度图--用于边缘检测
    Mat imgCanny; //边缘检测结果
    Mat imgBlur; //平滑去噪音结果--用于边缘检测
    Mat imgOri = imread(imgPath, 1); //原始图片读入
    cvtColor(imgOri, gray, COLOR_BGR2GRAY);
    blur(gray, imgBlur, Size(5, 5)); //平滑
    Canny(imgBlur, imgCanny, 100, 150, 3); //Canny 边缘检测 可以进行 参数调整
    int width = imgCanny.cols; //图像宽度
    int height = imgCanny.rows; //图像高度

    //

    BaoYifan::HoughCircle hough;
    vector< pair< pair<int, int>, int> > circles;
    for(int r = 22; r < 100; r+=1) //选择并遍历圆半径的大小范围
    {
        hough.calAccum(imgCanny.data, width, height, r); //计算所有的投票
        if(threshold == 0)
            threshold = 1.01 * (2.0 * (double)r * M_PI); //默认threshold
    }
}
```

```

        hough.getircles(threshold, circles);           //遍历计数器 获取圆形
的圆心

        //累加器的绘制
        int aw, ah, maxa;
        aw = ah = maxa = 0;
        const unsigned int* accu = hough.getAccum(&aw, &ah);

        for(int p=0;p<(ah*aw);p++)
        {
            if((int)accu[p] > maxa)
                maxa = accu[p];
        }
        double contrast = 1.0;
        double coef = 255.0 / (double)maxa * contrast;

        cv::Mat img_accu(ah, aw, CV_8UC3);
        for(int p=0;p<(ah*aw);p++)
        {
            unsigned char c = (double)accu[p] * coef < 255.0 ? (double)accu[p]
* coef : 255.0;
            img_accu.data[(p*3)+0] = 255;
            img_accu.data[(p*3)+1] = 255-c;
            img_accu.data[(p*3)+2] = 255-c;
        }

        cv::imshow(CW_IMG_ORIGINAL, imgOri);
        cv::imshow(CW_IMG_EDGE, imgCanny);
        cv::imshow(CW_ACCUMULATOR, img_accu);
        waitKey(1);
    }

    int a,b,r;
    a=b=r=0;
    std::vector< std::pair< std::pair<int, int>, int> > result;
    std::vector< std::pair< std::pair<int, int>, int> >::iterator it;
    //前后不相交 适当的减少多余的圆//还可以进一步改进
    for(it=circles.begin();it!=circles.end();it++)
    {
        int d = sqrt( pow(abs(it->first.first - a), 2) + pow(abs(it-
>first.second - b), 2) );
        if( d > it->second + r)
        {
            result.push_back(*it);
            //ok
            a = it->first.first;
            b = it->first.second;
            r = it->second;
        }
    }
}

```

```

    //result.push_back(*it);
}

//显示最终结果图
Mat imgRes = imgOrg.clone();
for(it=result.begin();it!=result.end();it++)
{
    circle(imgRes, cv::Point(it->first.first, it->first.second), it-
>second, cv::Scalar( 0, 255, 255), 2, 8);
}
imshow(CW_IMG_ORIGINAL, imgRes);
imshow(CW_IMG_EDGE, imgCanny);
waitKey(360000);
}

```

此函数的多数功能已经在注释中体现。与圆检测相关核心的地方在于半径的遍历。在每一个对应的半径下，都调用calAccum函数计算累加器的值。其余的为处理部分，包括圆个数的选择，累加器图的动态绘制和最终结果图的显示。

后面的累加器的值计算，以及getircles函数与直线部分思想基本类似。这里就不过多展示了。

五、编程体会

本次实验，我重新的跟着课堂上的算法要点实现了Hough变换。整个过程还是比较困难的，首先是自己上完课之后很多内容就忘记了，不能够很好的消化吸收。同时一开始我对于直线的参数表示还是有一些疑问。公式 $r = x\cos(\theta) + y\sin(\theta)$ 自己一开始也没想通是为什么，后来自己的画图做了一下也就明白了。实现的时候遇到的第二个问题是一些小的数学技巧。感觉虽然这些数学技巧看起来容易，可是自己在写代码的时候也不能立马的想出来，写起来也比较吃力。可能自己的逻辑能力还有待增强。除此之外，圆弧检测算法的内容我一开始并没有完全看懂，学习理解了网上的博客内容。

总的来说，自己还是应该努力锻炼和提高自己的编程能力。提高自己的算法实现能力。期待下一次作业的完成！

大头照



