Assignment 2

姓名：田原

学号：3180101981

Assignment 2
- 开发软件说明
- 算法具体步骤及实现要点
  - 初始化
  - 函数集成
- 结果展示及分析
- 编程体会
- 个人照片

开发软件说明

- 主要开发语言
  - C++
- 主要开发环境
  - Visual Studio 2019
  - OpenCV 4.5.0
- 编译系统环境
  - Windows 10 (64-bit)

算法具体步骤及实现要点

初始化

创建窗口；对图片进行滤波，并显示滤波结果：

```
int kvalue = 15;
int maxcolor = 255;

cvtColor(picOriginal, gray, COLOR_BGR2GRAY); // 转为灰度图
GaussianBlur(gray, gray, Size(3, 3), 3, 3); // 高斯滤波
bilateralFilter(gray, gray1, kvalue, kvalue * 2, kvalue / 2); // 双边

namedWindow("RESULT", 0); // 创建窗口
th1 = 200, th2 = 255; // canny边缘检测的参数

binary = myCanny(gray1, th1, th2); // caany边缘检测
imshow("RESULT", binary); // 显示边缘检测的图片
waitKey(0);
```

其中`myCanny`函数如下：

```
Mat myCanny(Mat picOriginal, int th1, int th2) {
    Mat picEdge;
    Canny(picOriginal, picEdge, th1, th2);
    return picEdge;
}
```

之后将边缘点记录在points中，方便后续的计算：

```
vector<Point> points; // 记录所有边缘点的坐标
for (int i = 0; i < binary.rows; i++) {
    for (int j = 0; j < binary.cols; j++) {
        if (binary.at<uchar>(i, j) > 0)
        points.push_back(Point(i, j));
    }
}
```

函数集成

为了方便后面的实现，首先实现两个形状检测函数：

- 圆形检测函数，该函数往circles中写入检测到的圆的三个参数（圆心坐标和半径）。

  ```
  vector<Vec3f> circles;
  void findCircles(
      Mat image,  // image输入
      vector<Point> points, // 边缘点坐标
      double pace, // 角度的分割
      double mindist, // 同心圆的半径差
      double thre, // 计数器的阈值
      double minradius = 0, double maxradius = 0);
  ```

  具体实现原理如下：

  首先把参数空间离散化，利用vector嵌套建立计数器：

  ```
  int sizer = (maxradius - minradius) / mindist + 1;
  int sizea = (2 * PI / pace) + 1;
  vector<vector <vector<int> > > counter(height, vector<vector<int>>(width,
  vector<int>(sizer, 0)));
  ```

  然后对于每一个边缘点，对圆心坐标和半径进行投票：

  ```
  for (int i = 0; i < points.size(); i++) {
      int x = points[i].x; // rows
      int y = points[i].y; // cols
      for (int r = 0; r < sizer; r++) {
          for (int angle = 0; angle < sizea; angle++) {
              int a = x - (maxradius - r * mindist) * sin(angle * pace);
              int b = y - (maxradius - r * mindist) * cos(angle * pace);
              if (a >= 0 && a < height && b >= 0 && b < width && counter[a][b]
  [r] >= 0) {
                  counter[a][b][r] ++; // 计数器++
                  if (counter[a][b][r] > max)
                      max = counter[a][b][r];
                  if (counter[a][b][r] >= thre) { // 如果计数超过阈值
  ```

```
                    double radius = maxradius - r * mindist;
                    circles.push_back(Vec3f(a, b, radius)); // 记录圆心坐标和
半径
                    for (int temp1 = 0; temp1 < 3; temp1++) {
                        for (int temp2 = 0; temp2 < 3; temp2++) {
                            for (int tempr = 0; tempr < 3; tempr++) {
                                counter[a+temp1][b+temp2][r-tempr] = -1; //
并把附近的一些圆形置空
                            }
                        }
                    }
                }
            }
        }
    }
}
```

这里有一个比较重要的点，在找到一个圆并且push之后，我们会把它周围的圆形对应的计数器置空。因为很多时候，画面中的一个圆形，可以检测到很多相近的圆，最后画面就会被我们检测到的这些"圆"填满，这是我们所不希望的，所以我们做了如上优化。

在检测完之后，我们需要把圆形在原图上画出来，代码如下：

```
for (int i = 0; i < circles.size(); i++) {
    Point center(circles[i].val[1], circles[i].val[0]);
    circle(picOriginal, center, circles[i].val[2], Scalar(255, 0, 0)); // 画
圆
}
```

- 第二个函数是直线检测函数：

```
void findLines(
    Mat image,
    double pace1, double pace2,double range,
    double thre);
    // pace1 距离步长
    // pace2 角度步长
    // range 容错范围
    // thre 累加器阈值
```

具体实现原理如下：

首先同样是离散化参数空间，建立对应的计数器：

```
int height = image.rows; int width = image.cols;
int maxlength = sqrt(height * height + width * width);
int countLength = (int)(maxlength / pace1 + 1);
int countTheta = (int)(2 * PI / pace2 + 1);
vector<vector <double> > counter(countLength, vector<double>(countTheta, 0)); //
建立计数器
```

然后对于每一个边缘点，对对应的 ρ 和 $theta$ 进行投票：

```
for (int i = 0; i < height; i ++) {
    for (int j = 0; j < width; j++) {
        if (image.at<uchar>(i, j) <= 0) continue;
```

```
        for (int rho = 0; rho * pace1 < maxlength; rho ++) {
            for (int theta = 0; theta * pace2 < 2 * PI; theta ++) {
                if (counter[rho][theta] == -1) continue;
                double temp = rho * pace1 - j * cos(theta * pace2) - i *
sin(theta * pace2);
                if (abs(temp) <= range) { // 差距小于容错
                    counter[rho][theta] ++; // 计数器++
                    if (counter[rho][theta] >= thre) {
                        lines.push_back(Vec2f(rho * pace1, theta * pace2));
                        counter[rho][theta] = -1;
                    }
                }
            }
        }
    }
}
```

在原图上画出直线。这里比较麻烦的点在于，直线只能通过两个点画出来，不能通过两个参数画出，所以需要计算点坐标：

```
for (size_t i = 0; i < lines.size(); i++) {
    Vec2f linex = lines[i];
    float rho = lines[i][0], theta = lines[i][1];
    Point pt1, pt2;
    double a = cos(theta), b = sin(theta);
    double x0 = a * rho, y0 = b * rho; // x = rho cos theta, y = rho sin theta
    pt1.x = cvRound(x0 + 1000 * (-b));
    pt1.y = cvRound(y0 + 1000 * a);
    pt2.x = cvRound(x0 - 1000 * (-b));
    pt2.y = cvRound(y0 - 1000 * (a));
    line(picOriginal, pt1, pt2, cv::Scalar(0, 255, 0), 1);
}
```

最后显示原图和绘图结果：

```
imshow("RESULT", picOriginal);
waitKey(0);
```

关于参数选择：

hw-coin:

```
findCircles(binary, points, 0.1, 5, 19, Min / 18, Max / 4);
```

hw-highway:

```
findLines(binary, 2, PI/32.0, 1, 100);
```

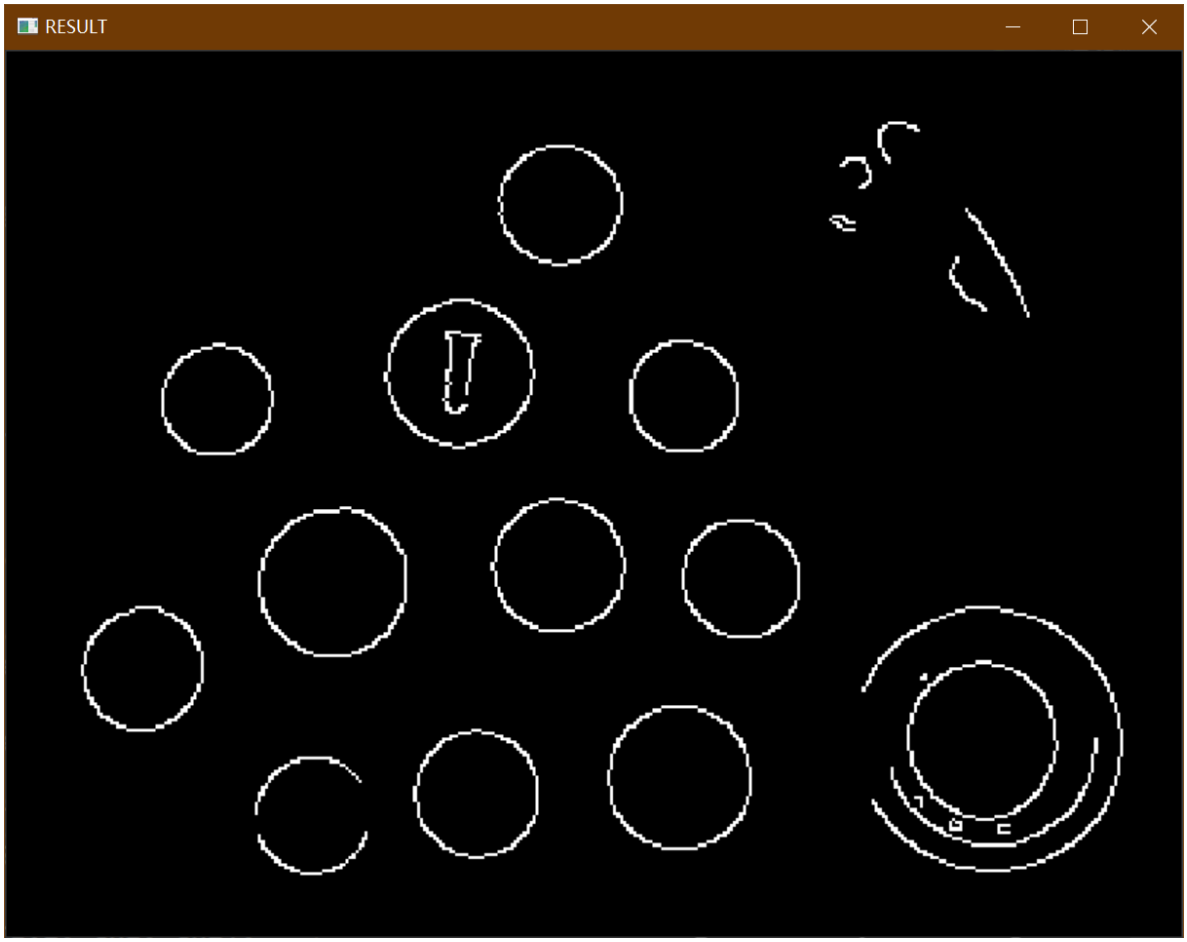
hw-seal:

```
findCircles(binary, points, 0.1, 5, 25, Min / 7, Max / 2);
findLines(binary, 2, PI/32.0, 1, 100);
```

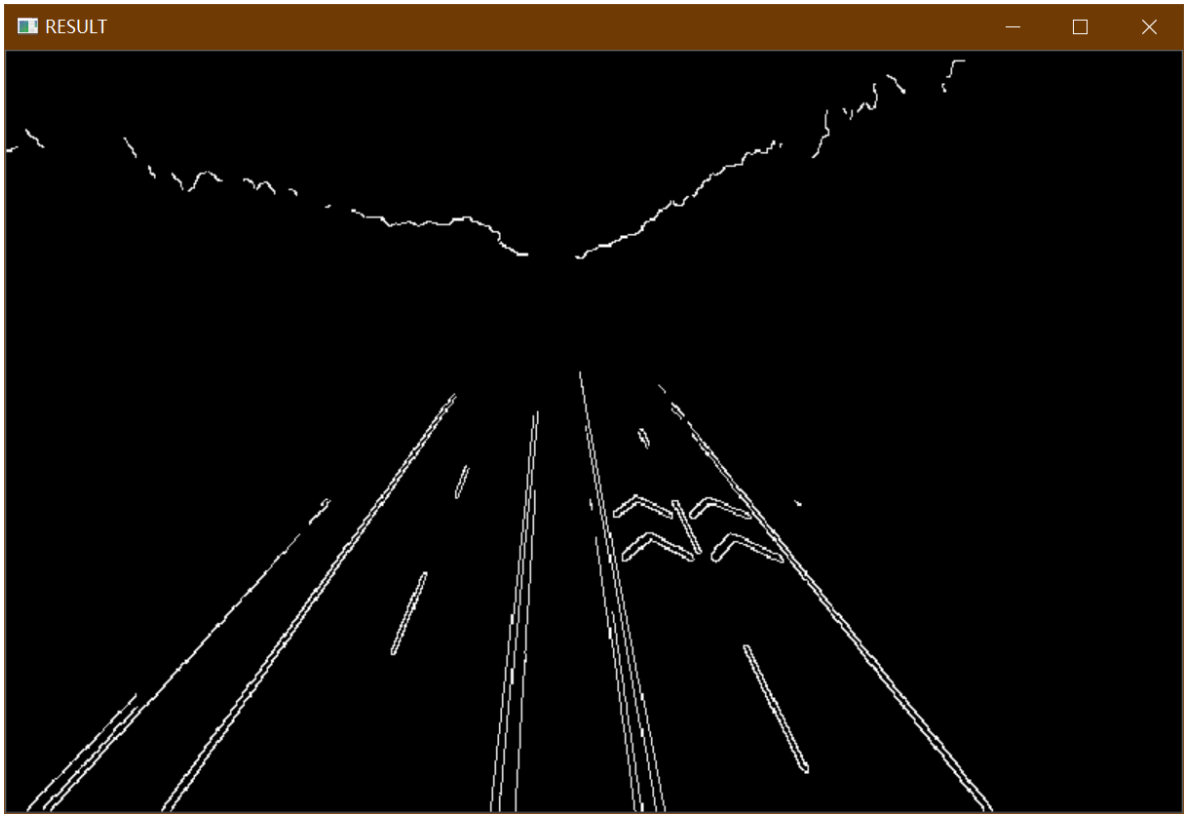
结果展示及分析

三张图片的滤波并边缘检测效果如下：

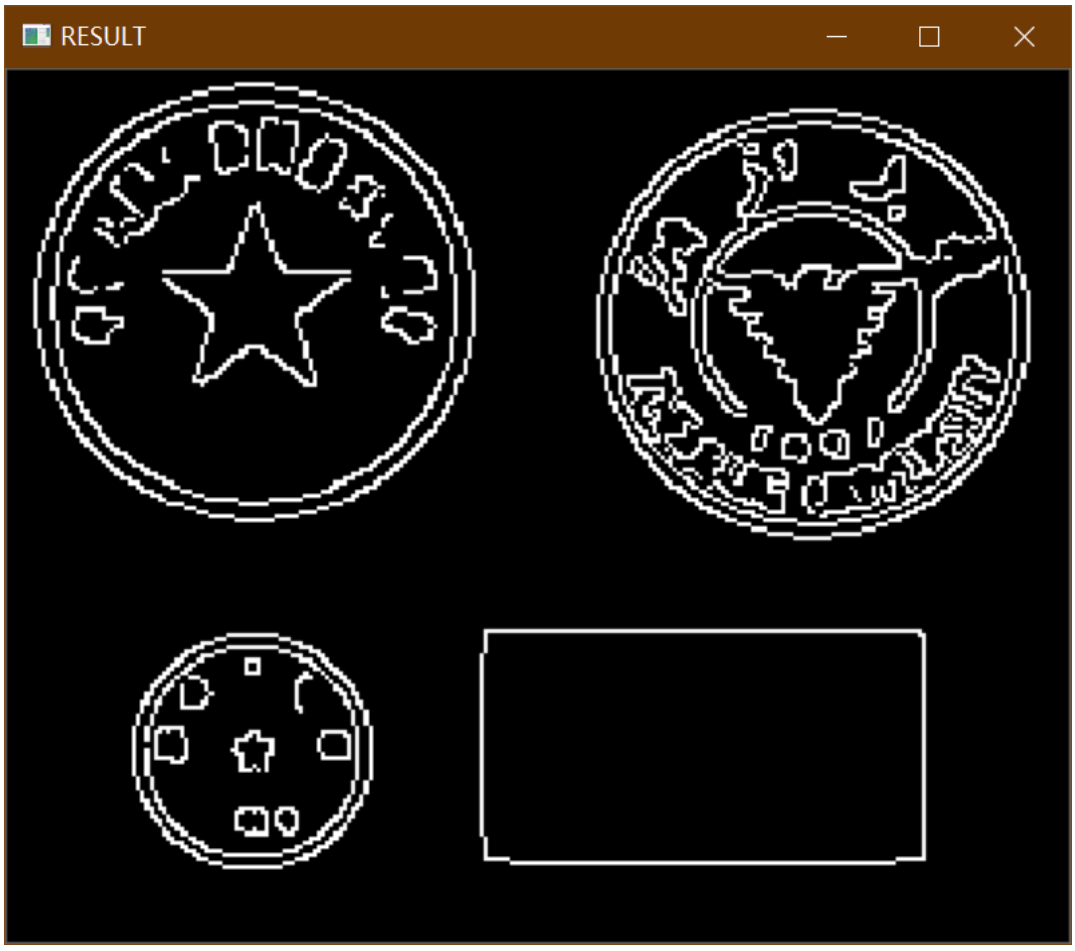
hw-coin:



hw-highway: （这里由于光线太暗，公路最右边的线其实丢失了一条）

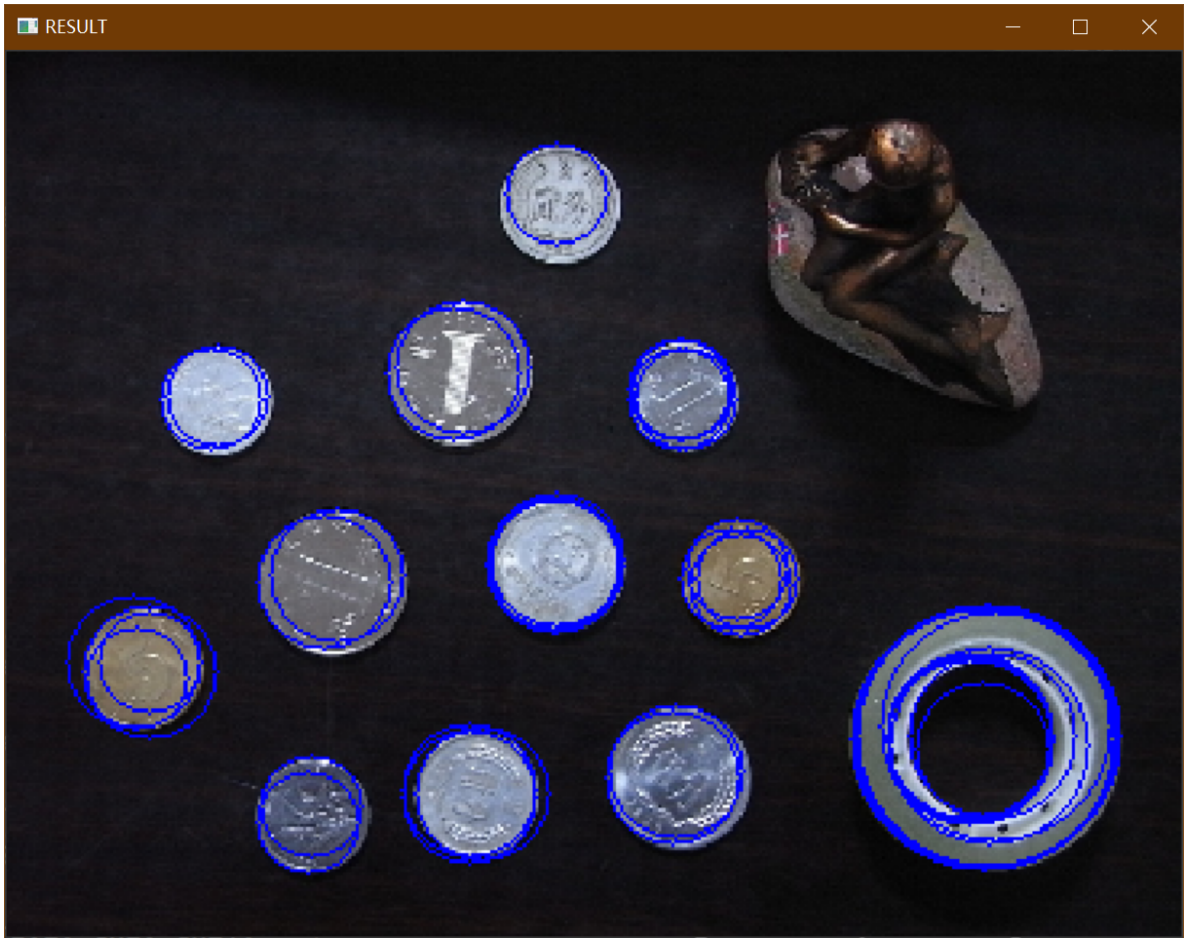


hw-seal: （这里其实还剩下比较多像素点，给后续带来挺大影响）

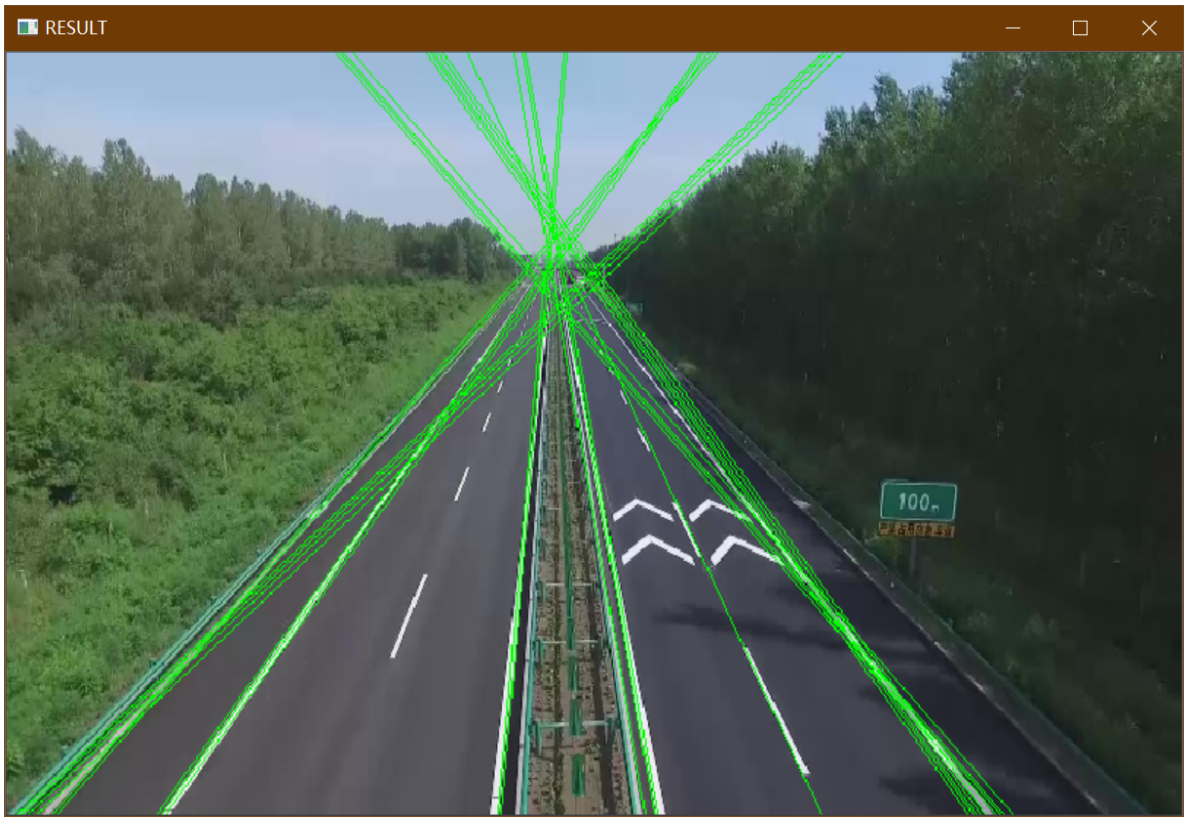


最后的检测结果图分别如下：

hw-coin：硬币基本全部检测出来，有一些圆还不够精确：



hw-highway：滤波后有剩余的线基本全部检测出来：



hw-coin：这张图是三张里面比较不完美的一张，左下的圆和长方形左边没能检测出来，原因在编程体会处分析。

总体来说我觉得效果还是比较满意的……主要是过程太痛苦了，虽然最后一张图有点不完美，但是已经满意了==。