MIN-VIS-2016

ASSIGNMENT 4 HOUGH TRANSFORM – GROUP4

Group Members:

- Sandio Fongang R
- Armaan Rustami

Hough Lines. Task 1:

a. Develop a program using OpenCV to detect lines in the image as shown in Figure 1. (image can be found on Canvas).

Consider to use filters to blur your image first. The result should look like the image as depicted in "*Figure 2. Output Image Hough Lines*". Please notice the red lines as a result of the Hough transform. See also example in OpenCV documentation on internet.

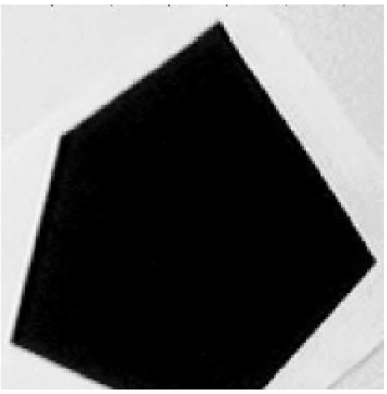


Figure 1 Image "Hough Picture".

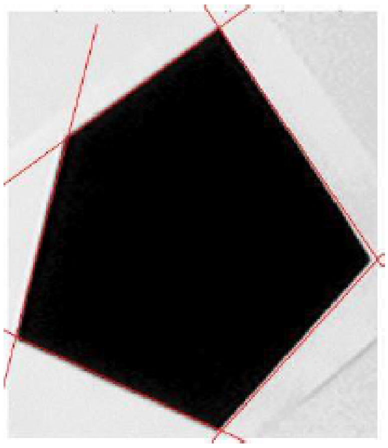


Figure 2 Output Image Hough Lines.

b. Hough transform (Hough Lines) is useful to detect parametric objects. In this task you'll be asked to calculate the intersections of the lines found in task a, also seen in *Figure 3* as an example. The pixel coordinates of A, B, C, D and E must be calculated.

Your results for the coordinates values A, B, C, D and E must be visible in the Output Image or beneath the Output Image. Please comment your results.

Task 2:

a. Please read the tutorial carefully as shown here by clicking and create a program using Hough Transform for Circles on the images "EuroCoins.jpg" and "EuroCoins2.jpg". These images can be found on Canvas.

Figure 4 EuroCoins

Figure 5 EuroCoins2

2. SOLUTION

a. Hough Line transform:

In order to detect existing lines in our picture here are the Steps we took:

- Filter the image by applying a GlaussianBlur with a kernel of 5 by 5,
- Detect the edges by using Canny() with a Threshold value of Min=100 and Max 150.
- Then we processed the image with HoughLines to detect the possible existing lines. You will notice that we had to increase the angle in order to reduce the sensibility. This is because some lines were duplicated.

```
HoughLines(img_edge, lines, 0.7, 2.61*M_PI/180, 50, 0, 0 );
```

- Afterward, we then go through each line in our accumulator and we get 2 Points from each:

```
void getFourPointsOnLines()
{
  for( size_t i = 0; i < lines.size(); 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;
   pt1.x = cvRound(x0 +1000* (-b));
   pt1.y = cvRound(y0 + 1000*(a));
   pt2.x = cvRound(x0 - 1000*(-b));
   pt2.y = cvRound(y0 - 1000*(a));
   for (int j = 0; j < lines.size(); ++j)
   {
    if (i==j){
    }
    else
    {
      float rho2 = lines[j][0], theta2 = lines[j][1];
      Point  pt3,pt4;
      double a2 = cos(theta2), b2 = sin(theta2);
      double x02 = a2*rho2, y02 = b2*rho2;
      pt3.x = cvRound(x02 +1000 *(-b2));
      pt3.y = cvRound(y02 +1000 *(a2));
      pt4.x = cvRound(x02 -1000 *(-b2));
      pt4.y = cvRound(y02 - 1000 *(a2));
      getIntersectionPoint(pt1, pt2, pt3, pt4, intPnt);
    }

  }

}
cout<<"Number of lines: "<<lines.size()<<endl;
labelling();
imshow("source", img_original);
imshow("lines with intersections", final_image);

}
```

- After getting the four Points, we calculate and check if they intersect each other.

```
bool getIntersectionPoint(Point a1, Point a2, Point b1, Point b2, Point & intPnt){
  Point p = a1;
  Point q = b1;
  Point r(a2-a1);
  Point s(b2-b1);

  if(cross(r,s) == 0) {return false;}

  double t = cross(q-p,s)/cross(r,s);

  intPnt = p + t*r;

    if (intPnt.x >0 &&intPnt.x<320&&intPnt.y>0 &&intPnt.y<400) //Storing only the points that are in our window frame
    {
      if (!checkingIfExist(intPnt))
      {
        cout<<intPnt<<endl;
        circle(final_image,intPnt, 5, Scalar(0,0,255), 2, CV_AA,0 );
        IntersectionPoints.push_back(intPnt);
      }
    }

    return true;
  }
  double cross(Point v1,Point v2){
    return v1.x*v2.y - v1.y*v2.x;
  }
  bool checkingIfExist(Point p)
  {
    for (int i = 0; i < IntersectionPoints.size(); ++i)
    {
      if (IntersectionPoints[i].x==p.x && IntersectionPoints[i].y==p.y)
      {
        return true;
      }
    }
    return false;

  }


  void labelling()
  {
```

- Finally, we process the labelling of the points by using the function Puttext:

```
        ostringstream ss;
         ss<<currentLabel;
         std::string s= ss.str();
        putText(final_image, s,ptToUse, 1, 1.2, Scalar(0,255,0), 2, 8, false );
        currentLabel++;

      }

    }
```

Final images:

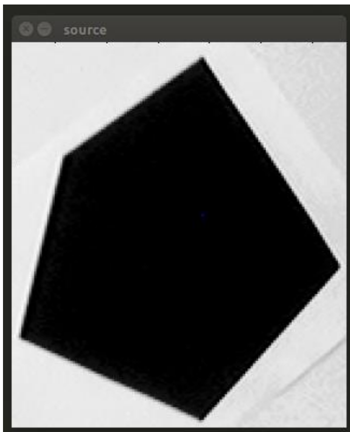


Figure 1: Original before Blurring

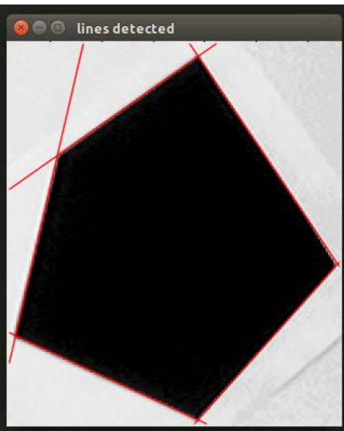


Figure 2: Image with lines

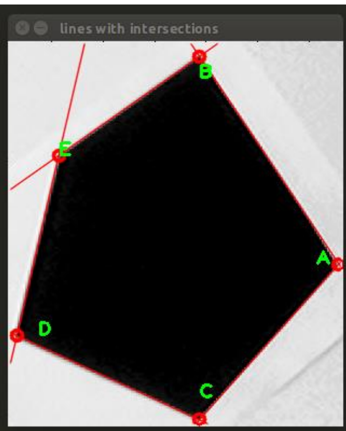


Figure 3: Lines with intersections

b. Circles detection:

The goal of this assignment is to detect circles in the image, so to achieve our goal we had to:

- Convert the picture to Grayscale,
- Apply a GlaussianBlur
- Apply HoughCircles with a Threshold of [150, 55] and store the lines in an accumulator.

```
/// Convert it to gray
cvtColor( img_original, src_gray, CV_BGR2GRAY );

/// Reduce the noise so we avoid false circle detection
GaussianBlur( src_gray, src_gray, Size(9, 9), 2, 2 );

vector<Vec3f> circles;

/// Apply the Hough Transform to find the circles
HoughCircles( src_gray, circles, CV_HOUGH_GRADIENT, 1, src_gray.rows/8, 150, 55, 0, 0);
```

- We then get the centre Point and the Radius of each circle.

```
Point center2(cvRound(circles[j][0]), cvRound(circles[j][1]));
  int radius2 = cvRound(circles[j][2]);
```

- Finally, we proceed to the drawing.

```
 // circle center
circle( img_original, center, 3, Scalar(0,255,0), -1, 8, 0 );
 // circle outline
circle( img_original, center, radius, Scalar(0,0,255), 3, 8, 0 );
```

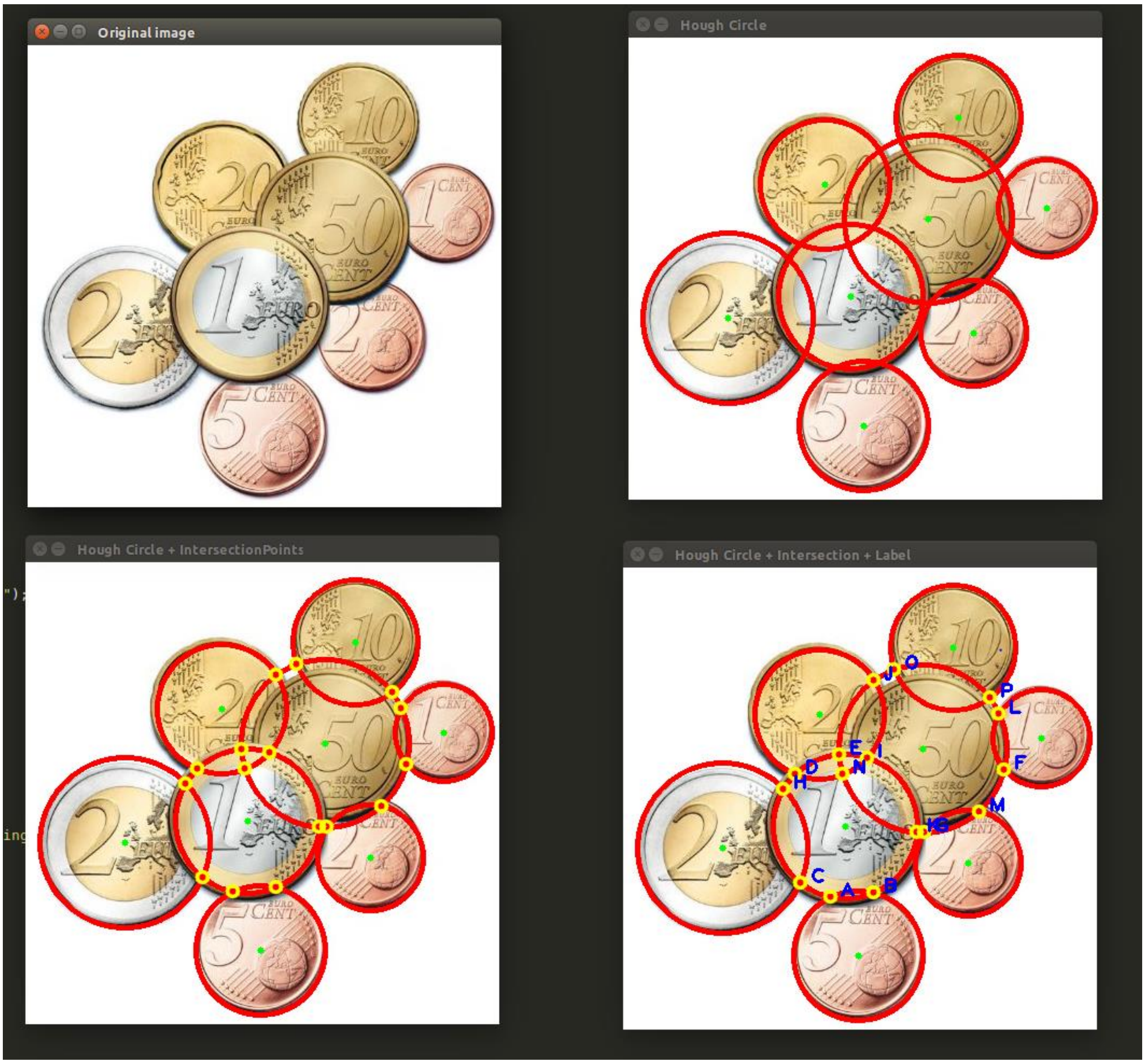
EXTRA!! Calculation and labelling of the intersection point of the circles

Although not required, we decided to calculate the intersection point of the circles. This is done in the method:

```
int circle_circle_intersection(double x0, double y0, double r0,
  double x1, double y1, double r1,
  double *xi, double *yi,
  double *xi_prime, double *yi_prime)
{
  double a, dx, dy, d, h, rx, ry;
  double x2, y2;

  /* dx and dy are the vertical and horizontal distances between
   * the circle centers.
   */
   dx = x1 - x0;
   dy = y1 - y0;
  /* Determine the straight-line distance between the centers. */
  //d = sqrt((dy*dy) + (dx*dx));
  d = hypot(dx,dy); // Suggested by Keith Briggs

  /* Check for solvability. */
  if (d > (r0 + r1))
  {
    /* no solution. circles do not intersect. */
    return 0;
  }
  if (d < fabs(r0 - r1))
  {
    /* no solution. one circle is contained in the other */
    return 0;
  }

  /* 'point 2' is the point where the line through the circle
   * intersection points crosses the line between the circle
   * centers.
   */
  /* Determine the distance from point 0 to point 2. */
   a = ((r0*r0) - (r1*r1) + (d*d)) / (2.0 * d) ;

  /* Determine the coordinates of point 2. */
   x2 = x0 + (dx * a/d);
   y2 = y0 + (dy * a/d);
  /* Determine the distance from point 2 to either of the
   * intersection points.
   */
   h = sqrt((r0*r0) - (a*a));

  /* Now determine the offsets of the intersection points from
   * point 2.
   */
   rx = -dy * (h/d);
   ry = dx * (h/d);

  /* Determine the absolute intersection points. */
   *xi = x2 + rx;
   *xi_prime = x2 - rx;
   *yi = y2 + ry;
   *yi_prime = y2 - ry;
       IntersectionPoints.push_back(Point(*xi_prime,*yi_prime));

       cout<<*xi_prime<<"  "<<*yi_prime<<endl;
   return 1;
}
```

Figure 2: Final outputs

On this picture, the Circles didn't intersect each other. That is why we have no intersection point and naturally no label.


Original image
Hough Circle
Hough Circle + IntersectionPoints
Hough Circle + Intersection + Label