Computer Vision

Assignment: 2  
Author(s): Rafał Grasman  
Date: 11-09-2018  
Subject: Canny edge, Hough lines and circles detection  
Required pre-knowledge: OpenCV, OpenCV documentation

Contents

[Task 1 3](#_Toc524473697)

[Task 2 5](#_Toc524473698)

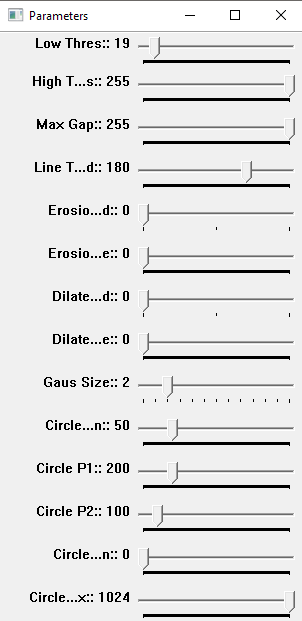
[Task 3 9](#_Toc524473699)

# Task 1

For task 1 the noise was removed using Gaussian Blur, this was sufficient for Task 2 (as can be seen in task 2). Erosion or dilation wasn’t needed. To make processing on any image faster/easier, the images are converted to grayscale before processing:

|  |
| --- |
| cvtColor(ws.src, ws.src\_gray, CV\_BGR2GRAY); |

For fine tuning the images and quickly finding working solutions a parameter control has been made:



Applying Gaussian Blur to reduce noise in images yields the following results:

|  |  |
| --- | --- |
|  |  |

Compared to the originals:

|  |  |
| --- | --- |
|  |  |

The Gaussian Blur requires an odd-numbered size, which is calculated as follow:

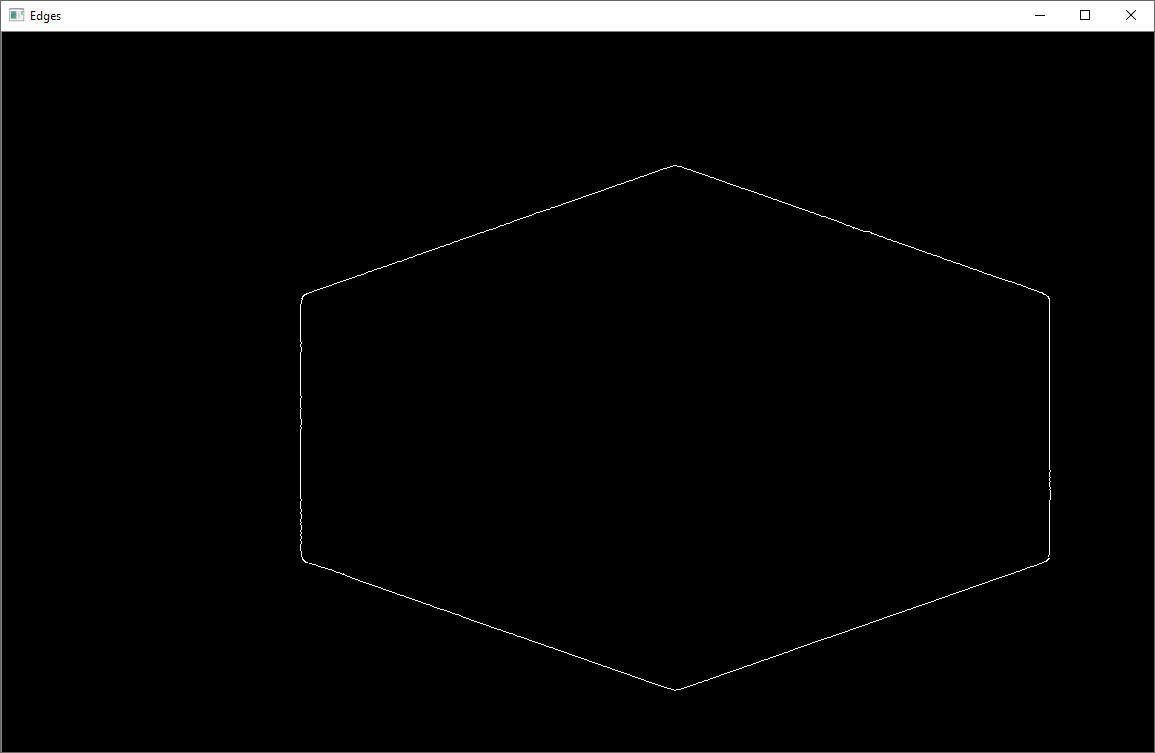
|  |
| --- |
| if (params.gaus\_size > 0)  {  GaussianBlur(ws.src\_gray, ws.dst\_clean, Size(1 + params.gaus\_size \* 2, 1 + params.gaus\_size \* 2), 0.0);  }  else  {  ws.dst\_clean = ws.src\_gray.clone();  } |

# Task 2

In task 2 lines need to be detected. After the cleaned-up image is processed, it is fed into a canny edge detector:

|  |
| --- |
| Canny(ws.dst\_clean, ws.dst\_edges, params.low\_threshold, params.high\_threshold, 3);  imshow(windows.edges, ws.dst\_edges); |

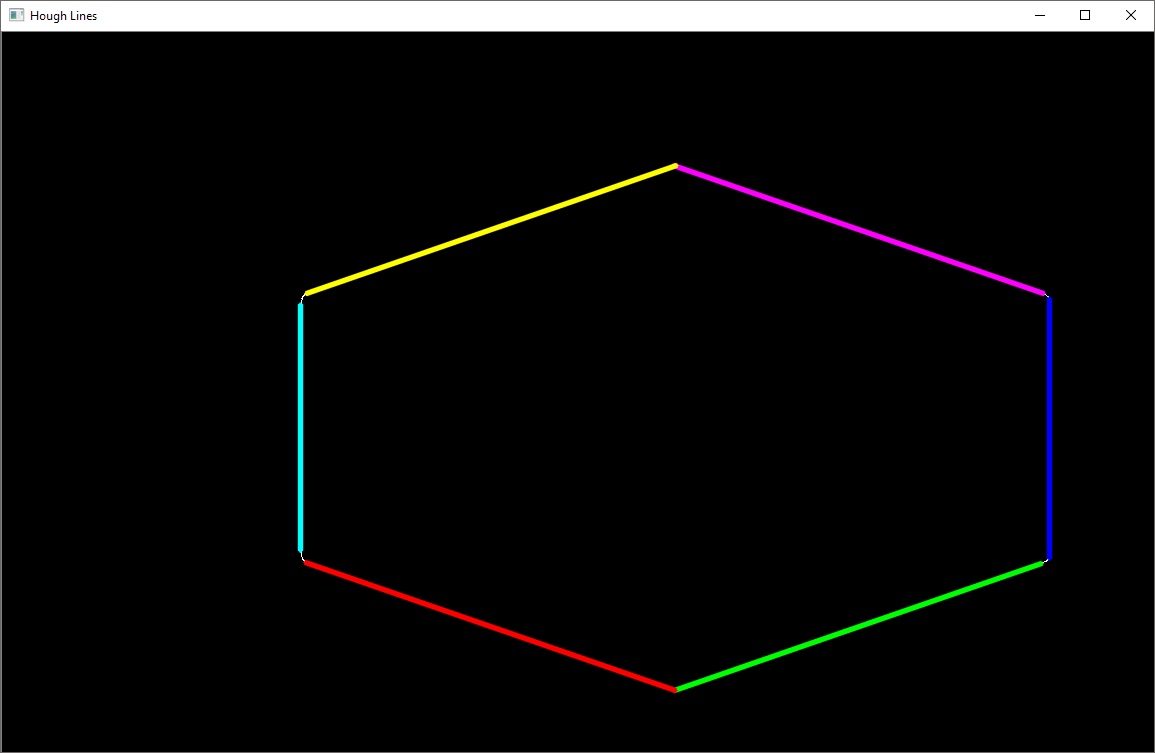
The result is:



Then the lines are detected using HoughTransformP to get the XY coordinates of line points (and the lines are overlaid onto the edges image):

|  |
| --- |
| HoughLinesP(ws.dst\_edges, lines, 1, CV\_PI / 180, params.line\_threshold, 0, params.gap);  cvtColor(ws.dst\_edges, ws.dst\_hough, CV\_GRAY2BGR);  (…)  Vec4i l = lines[i];  line(ws.dst\_hough, Point(l[0], l[1]), Point(l[2], l[3]), GetColor(i, lines.size()), 3, CV\_AA); |

And with the parameters set (as seen in the parameter window picture) the lines can be detected:



Each line is an index number in the ‘lines’ vector, by knowing the two XY points that make a line, neighbors can be found by comparing the distances (and picking the closest ones):

|  |
| --- |
| std::pair<int, int> FindNeightbour(const std::vector<Vec4i>& lines, int location, int point, double max\_distance = 25.0)  {  Point a(…)  (…) // looping through all points of all lines except point a  Point b(…);  if (cv::norm(b - a) < max\_distance)  {  return std::pair<int, int>(i, j);  }  (…) |

Then intersections are calculated for each neighbor of the lines (each line has 2 neighbors):

|  |
| --- |
| for(int j = 0; j < 2; ++j)  {  std::pair<int, int> neightbour = FindNeightbour(lines, i, j);  if (neightbour.first > i) // make sure we skip already calculated lines / intersections  {  Vec4i p = lines[neightbour.first];  Point r;  double degrees = Degrees(l, p, r);  Point mid = Mid(l, p);  (…) |

This is done by using the degrees function which immediately calculates the angle between the lines:

|  |
| --- |
| double Degrees(const Vec4i &v1, const Vec4i &v2, Point& r)  {  // get cross  Point o1(v1[0], v1[1]);  Point p1(v1[2], v1[3]);  Point o2(v2[0], v2[1]);  Point p2(v2[2], v2[3]);  Point x = o2 - o1;  Point d1 = p1 - o1;  Point d2 = p2 - o2;  double cross\_value = d1.x\*d2.y - d1.y\*d2.x;  if (abs(cross\_value) < /\*EPS\*/1e-8)  {  return 0.0;  }  double t1 = (x.x \* d2.y - x.y \* d2.x) / cross\_value;  Point b =  o1 + d1 \* t1;  Point a =  cv::norm(b - o1) > cv::norm(b - p1) ? o1 : p1;  Point c =  cv::norm(b - o2) > cv::norm(b - p2) ? o2 : p2;  // get angle  Point ab = { b.x - a.x, b.y - a.y };  Point cb = { b.x - c.x, b.y - c.y };  double dot = (ab.x \* cb.x + ab.y \* cb.y); // dot product  double cross = (ab.x \* cb.y - ab.y \* cb.x); // cross product  double alpha = atan2(cross, dot);  r = b;  return floor(alpha \* 180.0 / CV\_PI + 0.5);  } |

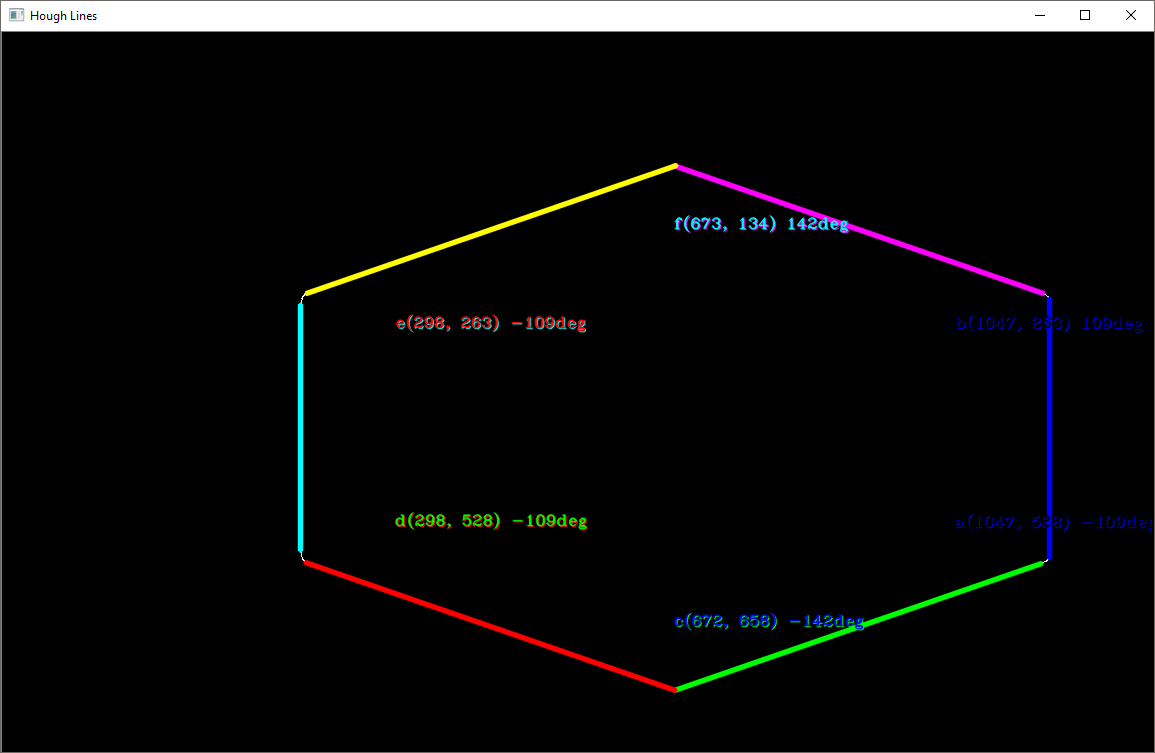
It works as follow:

1. Calculate, if any, the crossing point between the lines by calculating the cross product
2. Given the crossing point (b), for each line, assign a point (a, c) to the \*furthest\* point of the line
3. Now we can make a triangle a,b,c with b being the intersection and a and c being the outermost points, calculate the dot product and cross product, take the arctan2 of the cross and dot products and this gives the angle
4. Calculate the angle in degrees and return the intersection point so the caller can know where to draw a letter (a, b, c …) and the degree text

The mid point between two lines is calculated simply by taking the average of the line end, then of the 2 new points the average of that (so this function is called 3 times total):

|  |
| --- |
| Point Mid(const Point &v1, const Point &v2, double distance = 0.5)  {  return (v1 + v2) \* distance;  } |

This results in the following image (and correctly detected angles and cross positions):



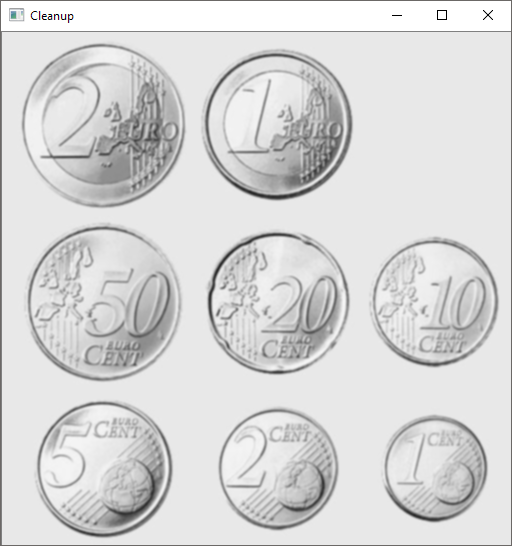
Going clockwise starting at 12:00:

F(673,134) 142 degrees  
B(1047,263) 109 degrees  
A(1047,523) -109 degrees  
C(672,658) -142 degrees  
D(298,528) -109 degrees  
E(298,263) -109 degrees

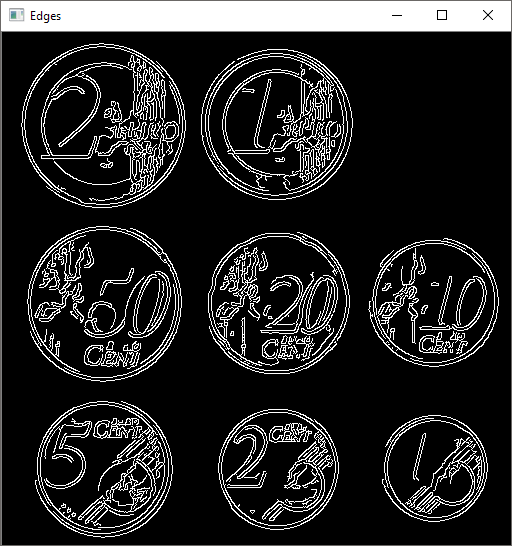
Totaling (absolute) 720 degrees which is correct for a 6-point polygon.

# Task 3

To detect circles the HoughCircles function will be used. Before this can be used a cleaned-up image is made as described in Task 1:



This image is taken into the canny edge detector to remove unwanted features and simplify the image even more for processing:



The HoughCircles algorithm can have troubles with lines that are not continuous, which requires post-processing before feeding it into the HoughCircles function. By applying some Gaussian blur, the lines are connected a bit more which increases the circle detection rate:



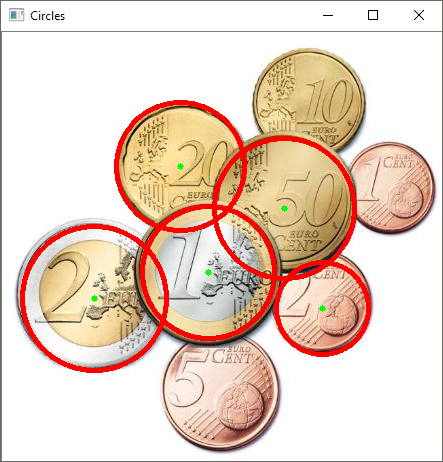
On this image the HoughCircles function is called and then the circles are drawn over the original image:

|  |
| --- |
| HoughCircles(ws.dst\_edges, circles, CV\_HOUGH\_GRADIENT, 1, params.circle\_mindist, params.circle\_param1, params.circle\_param2, params.circle\_min\_radius, params.circle\_max\_radius);    ws.dst\_circlehough = ws.src.clone();  for (size\_t i = 0; i < circles.size(); i++)  {  Point center(cvRound(circles[i][0]), cvRound(circles[i][1]));  int radius = cvRound(circles[i][2]);  circle(ws.dst\_circlehough, center, 3, Scalar(0, 255, 0), -1, 8, 0);  circle(ws.dst\_circlehough, center, radius, Scalar(0, 0, 255), 3, 8, 0);  } |

This yields the following images / results:



As for overlapping circles, not all of them can be detected unfortunately, no matter what combination of parameters (the most circles are found by reducing the initial / cleanup Gaussian blur):



Applying different values for ‘param1’ and ‘param2’ in HoughCircles function was not feasible because processing times reached 20 minutes per change on an i7-4710MQ CPU. It is suspected that lowering param2 and lowering param1 will yield more circle, but also more false positives (as can be read in the documentation for HoughCircles function).

Update 1:

After investing more time in updating the parameters according to the previously mentioned theory, all circles are detected:

|  |  |
| --- | --- |
|  |  |