Chapter 4

Feature Detection

4.1 Introduction

Segmentation is the process of separating objects or features from a digital raw image. There are different methods depending on the *feature* that need to be separated from the image [1]. Often the technique used to solve one particular set of problems is unsuitable to solve others. In this chapter we discuss the basic concepts for some specific features such as edge and lines.

The representation of the segmentation is somewhat important too. Sometimes separating entire areas that contain the silhouette of an object is enough for an application, and it can be represented as a blob of a specific colour. At other times, it is necessary to get specific data from features of the image, such as edges, lines, circles or corners.

The simplest possible segmentation method is based on thresholding. All the bright pixels in the resulting image would indicate an object. If there is more than one object an approach of separating these areas has to be used. We have already seen, for example, that the n-adjacency approach is useful in separating areas that are not contiguous.

Let us examine some specific techniques for image segmentation.

4.2 Geometric Feature Detection

Special features such as lines, corners and circles have very important applications in machine vision. Before being able to identify these features, it is necessary to find the edges in the image. We have seen already some examples of convolution operations that may separate the edges of an image. A more detailed discussion about edge detection methods follows.

4.2.1 Edge Detection

Gradient

We have seen in chapter 3 that it is possible to find edges using simple 3x3 kernels that are approximations of the gradient (first derivatives). There are different operators (kernels) that achieve that, such as Prewitt, Isotropic, Roberts and Sobel. All these operators have a common characteristic: the width of the lines representing the edges is proportional to the strength of the edges. Although still used in practise, these operators are prone to noise and are not necessarily the best option if one wants to locate the edges accurately.

[1] *Feature*: this word has two meanings in CV. One can refer to a physical property of an object (such as a line, an edge, a corner etc). But a *feature* may also refer to a number extracted from the image using a certain mathematical expression, just as one would do in machine learning. In this chapter, we use the first definition.

Laplacian of a Gaussian

A filter that combines either the first or the second derivatives and the ability to act at a certain scale would be ideal. Marr and Hildreth [1] searched for such a filter and concluded that the most useful operator is the Laplacian of a Gaussian ($\nabla^2 G$).

The Gaussian blurs the image and can make some less important edges disappear. How to build such kernels? We can start with the Gaussian equation in 2D:

$$G(x,y) = \frac{1}{\sqrt{2\pi\sigma^2}}\, e^{-\left(\frac{x^2+y^2}{2\sigma^2}\right)} \tag{4.1}$$

For a certain σ, we approximate the values and cut off to a certain kernel size. For example, with $\sigma = 0.85$, the values for $-1 < x < 1$ and $-1 < y < 1$ would be 0.118, 0.235 etc. If we divide all the values of the kernel by the smallest one and round the values to the nearest integer, the result is the well known approximation for a Gaussian. The kernel needs to be divided by 16, so the sum of the kernel elements is equal to 1. Notice that this kernel is separable.

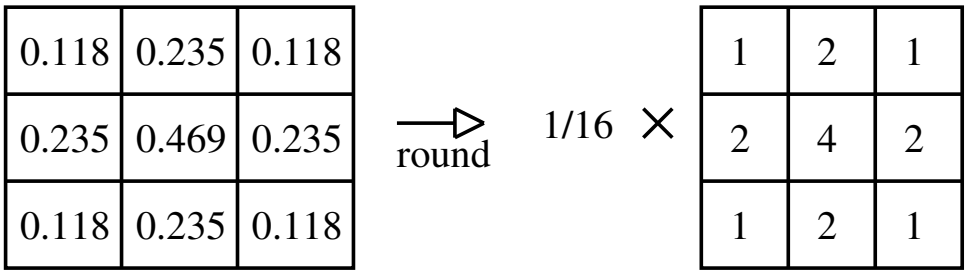

Figure 4.1: Sampling a Gaussian with a 3x3 kernel

The Laplacian of a Gaussian can be either approximated from the second derivative of equation 4.1 or by carrying out a convolution of the image separately with each kernel. The resulting kernel for a LoG using the Gaussian from figure 4.1 and figure 3.9 is in figure 4.2.

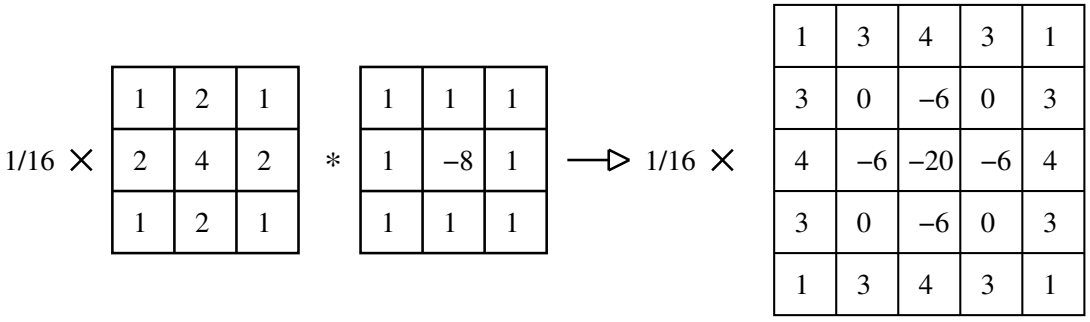

Figure 4.2: LoG 5x5 kernel

When using the 5x5 kernel the resulting image will be very close to the one produce by convolving the image separately using two 3x3 kernels. However, the results may differ due to the rounding effects. In the two kernels scenario the rounding is done before the second convolution.

The LoG can be controlled by both the kernel size and the σ of the Gaussian. The effect on the edges can be noticeable if there is a mix of thin and thick edges in the image.

Another approximation for the LoG is the so-called DoG (difference of Gaussians). In this filter two Gaussians with different σ are convolved with the image separately, and then the resulting images are subtracted from each other.

4.2.2 Edge detection functions in OpenCV

Laplacian

The OpenCV implementation for the Laplacian is called *cvLaplace*() (4.1).

Listing 4.1: Sample for cvLaplace

```
#include "highgui.h"
#include <stdio.h>
char* filename;
IplImage *image = 0, *image2 = 0, *auximage = 0;
uchar* pixel;
int main( int argc, char** argv )
{
    if (argc == 2) { filename=argv[1];}
    else exit(0);
    if( (image = cvLoadImage( filename, 1)) == 0 )
    return -1;
    cvNamedWindow("Original",1);
    cvNamedWindow("Edge Detection",2);
    image2 = cvCreateImage(cvSize(image->width, image->height),
        IPL_DEPTH_16S, 1);
    auximage = cvCreateImage(cvSize(image->width, image->height),
        IPL_DEPTH_8U, 1);
    cvCvtColor(image, auximage, CV_BGR2GRAY);
    cvLaplace(auximage,image2,3);
    cvConvertScaleAbs( image2, auximage );
    cvShowImage("Original",image);
    cvShowImage("Edge Detection",auximage);//so it shows 8UC1
        image
    cvWaitKey(0);
    cvReleaseImage(&image);
    cvReleaseImage(&image2);
    return 0;
}
```

Sobel

As seen before, Sobel's mask is capable of finding the gradients of a digital image. The edges can be found using a kernel that represents the Sobel's mask. OpenCV has an implementation edge detection called *cvSobel*(). Follows an example:

Listing 4.2: Sample for cvSobel

```
#include "cv.h"
#include "highgui.h"
#include <stdio.h>
char* filename;
IplImage *image = 0, *image2 = 0, *auximage = 0;
uchar* pixel;
int main( int argc, char** argv )
{
    if (argc == 2) { filename=argv[1];}
    else exit(0);
    if( (image = cvLoadImage( filename, 1)) == 0 )
    return -1;
    cvNamedWindow("Original",1);
    cvNamedWindow("Edge Detection",2);
    image2 = cvCreateImage(cvSize(image->width, image->height),
        IPL_DEPTH_16S, 1);
    auximage = cvCreateImage(cvSize(image->width, image->height),
        IPL_DEPTH_8U, 1);
    cvCvtColor(image, auximage, CV_BGR2GRAY);
    cvSobel(auximage,image2,1,1,5);
    cvConvertScaleAbs( image2, auximage );
    cvShowImage("Original",image);
    cvShowImage("Edge Detection",auximage);
    cvWaitKey(0);
    cvReleaseImage(&image);
    cvReleaseImage(&image2);
    return 0;
}
```

Canny

Canny detection uses a Gaussian followed by a Sobel edge detection. Then the edges are improved by finding their direction. Finally a hysteresis process is used to improve the quality of the edges. Follows an example in OpenCV (where the function is called *cvCanny*()).

Listing 4.3: Sample for cvCanny

```
#include "cv.h"
#include "highgui.h"
#include <stdio.h>
char* filename;
IplImage *image = 0, *image2 = 0, *auximage = 0;
uchar* pixel;
int main( int argc, char** argv )
{
    if (argc == 2) { filename=argv[1];}
    else exit(0);
```

```
11      if( (image = cvLoadImage( filename, 1)) == 0 )
12      return -1;
13      cvNamedWindow("Original",1);
14      cvNamedWindow("Edge Detection",2);
15      auximage = cvCreateImage(cvSize(image->width, image->height),
            IPL_DEPTH_8U, 1);
16      image2 = cvCreateImage(cvSize(image->width, image->height),
            IPL_DEPTH_8U, 1);
17      cvCvtColor(image, auximage, CV_BGR2GRAY);
18      cvCanny(auximage,image2,1,100,3);
19      cvShowImage("Original",image);
20      cvShowImage("Edge Detection",image2);
21      cvWaitKey(0);
22      cvReleaseImage(&image);
23      cvReleaseImage(&image2);
24      return 0;
25  }
```

Canny edge detection can be split into 6 steps:

- smooth the image (Gaussian)
- find raw edges (Sobel)
- find the gradient directions (except where the angle is 90)
- find the gradient where the angle is 90.
- find a thin line along the edges
- uses hysteresis to avoid breaking edges (use thresholds $T1$ and $T2$).

Examples of the three types of edge detection are illustrated in figure 4.3. In these images it is clear that the different types of edge detection are very different from each other.

Sobel and Laplacian have something in common, the edges are of different intensities. Looking carefully at the images, one can see that some edges are brighter than others. The brightness is telling how strong or weak are the slopes between the neighbouring pixels. The strength is normalised, so the edges show values from 0 to 255. It is also interesting to note that the strongest edges (the brightest) are usually separating two objects, while the weaker edges may be in the middle of the objects.

Canny shows a completely different form of edge, where the brightness of the edges are all of the same value (255 in the case of a greyscale image). The other difference is that in the Canny result the edges one pixel wide, while in Sobel and Laplacian the edges can be very thick depending on the objects. Canny is particularly useful when further processing has to be used to find geometric properties of the edges. However, with Canny we lose some information about the importance of the edges, as edges of different widths and strengths are all represented by a thin line.

4.2.3 Hough transform for straight lines

The fact that we can find edges does not necessarily imply finding straight lines. In order to confirm that a sub-set of the pixels that belong to edges form a certain geometry, further

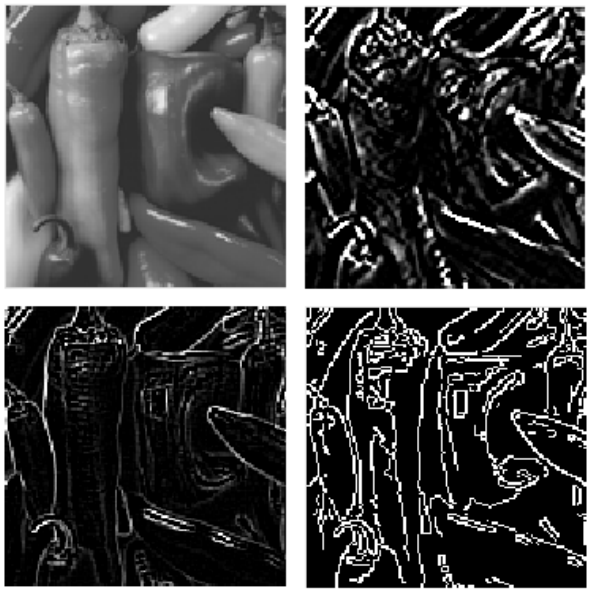

Figure 4.3: Edge detection.a) original image b) Sobel c) Laplacian d) Canny

processing is needed. Hough transforms (named after Paul Hough, who had a patent on a one of the transform earlier methods) can be used to achieve that.

The idea behind this method can be explained in four steps [2]:

- Obtain a binary image, the edges marked with 1 or 255
- Create a ρ,θ plane to represent the straight lines passing by point (x,y)
- Extract the points of the ρ,θ plane that are above a certain threshold
- Examine the continuity of the chosen lines

The first step is simple, although it may require a careful choice of parameters. Using one of the threshold methods discussed bin the previous chapter will usually work. If the binary image contains too many edges, it may be difficult to choose the correct straight lines after applying the Hough transform.

The second step relies on the fact that the binary images are usually a sparse set of points (many points of the image do not belong to any edge). For each existing point of an edge, there is an infinite number of lines that pass through it. A 2D array can be used to accumulate counters in such a way that if there is a line, several points will put a mark on a single counter.

It is convenient to use parameters different than x and y to represent the counters. A straight line passing through (0,a) and (b,0) (fig. 4.4) in Cartesian coordinates can be written as:

$$y = -\left(\frac{a}{b}\right)x + a \tag{4.2}$$

But $\rho = b\,cos(\theta)$ and $\rho = a\,sin(\theta)$, and therefore:

$$y = -\left(\frac{\rho}{sin(\theta)}\frac{cos(\theta)}{\rho}\right)x + \frac{\rho}{sin(\theta)} = -\left(\frac{cos(\theta)}{sin(\theta)}\right)x + \frac{\rho}{sin(\theta)} \tag{4.3}$$

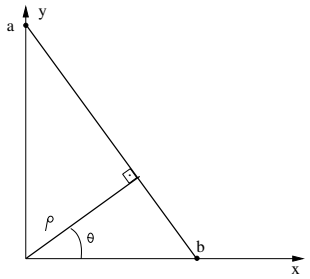

Figure 4.4: Writing a line equation with ρ and θ as parameters.

A straight line passing through point (x,y) can be represented by a parametric equation of the form (see figure 4.5):

$$x\,cos(\theta) + y\,sin(\theta) = \rho \tag{4.4}$$

In step 3, we should use accumulators to mark the parameters of lines that passes through each point. Considering a minimum threshold for these accumulators, several lines may be found.

Finally, in step 4, some of the lines found in step 3 may be interrupted. A second threshold can be defined related to a certain level of tolerance, given the length of the line (e.g., the percentage of missing points in relation to the total length of the straight line).

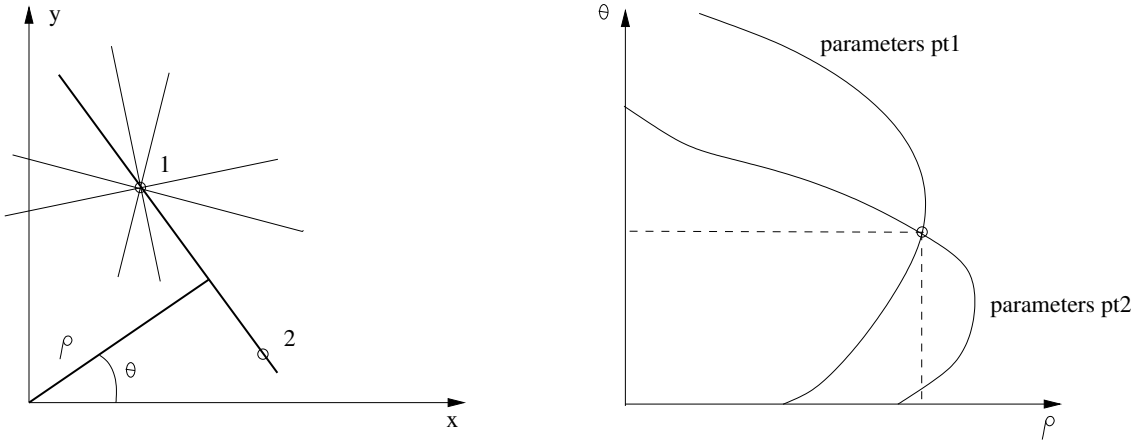

Figure 4.5: Lines for points 1 and 2 are plotted in polar coordinates. Where the curves meet there is a line to which both points belong to.

A simple implementation of the Hough transform is listed below. This sample code uses a simple parametrisation, but in order to locate the lines we need to remember that the origin in digital images is usually inverted (and so may be the angles). A binary image with two short and two long lines (figure 4.6) yields the accumulator seen in figure 4.7. In the accumulator the x-axis represents the distance from the origin (ρ) and the y-axis represents the angle θ.

Listing 4.4: A sample code for the Hough transform

```
1 void myhoughtrans(IplImage * im2)
2 {
```

```
3         int center_x, center_y, r, omega, i, j, rmax, tmax;
4         double conv;
5         double tmval=0.0;
6         conv = CV_PI/180.0;
7         center_x = im2->width/2;
8         center_y = im2->height/2;
9         rmax = (int)(sqrt((double)(im2->width*im2->width+im2->
              height*im2->height))/2.0);
10        double houghspace[180][2*rmax+1];
11        IplImage * accumulator;
12        accumulator = cvCreateImage( cvSize(180,(int)2*rmax+1), 8,
              1 );
13        for (r = 0; r < 2 * rmax+1; r++)
14           for (omega = 0; omega < 180; omega++)
15                houghspace[omega][r] = 0;
16        tmax = 0; tmval = 0;
17        for (i = 0; i < im2->height; i++) {
18          for (j = 0; j < im2->width; j++) {
19                if (pixel(im2,i,j) == 255) {
20                   for (omega = 0; omega < 180; ++omega)
21                   {
22                         r = (int)((i - center_y) * sin((double)(
                                omega*conv))
23                            + (j - center_x) * cos((double)(omega*
                                conv)));
24                         houghspace[omega][rmax+r] += 1;
25                   }
26                }
27          }
28        }
29        int imax=0,jmax=0;
30        for (i=0; i<180; i++){
31          for (j=0; j<2*rmax+1; j++){
32            if ( (houghspace[i][j]) > tmval)
33            {
34                tmval = houghspace[i][j];
35                printf("tmval %lf houghspace %d %d = %lf \n",tmval
                      ,i,j,houghspace[i][j]);
36                imax = i;
37                jmax = j;
38            }
39          }
40        }
41        printf("maxro %d largest accumulator at  %d %d =  %lf \n",
              rmax,imax,jmax,tmval);
42        for (i=0; i<180; i++){
43          for (j=0; j<2*rmax+1; j++){
44             pixel(accumulator,i,j)=cvRound(houghspace[i][j]*(255/
                   tmval));
```

```
            }
        }
}
```

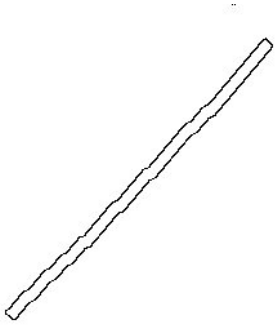

Figure 4.6: A binary image with two main straight lines.

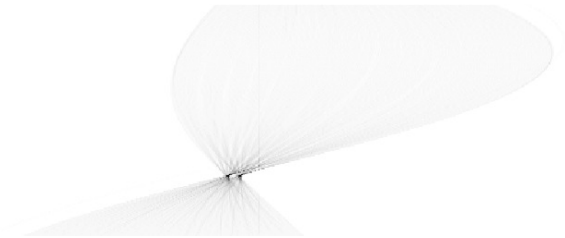

Figure 4.7: The accumulator is shown as an image (the accumulator value is normalised and rounded up). It clearly shows two darker points, which correspond to the two larger straight lines.

In OpenCV there is a function called *cvHoughLines*2() that implements Hough transforms. The function returns the parameters to a sequence, so the appropriate thresholds can be used after finding a number of lines. Follows a code example (adapted from OpenCV documentation):

Listing 4.5: Sample for cvHoughLines

```
#include <cv.h>
#include <highgui.h>
#include <math.h>
#include <stdio.h>
int main(int argc, char** argv)
{
    IplImage* src;
    if( argc == 2 && (src=cvLoadImage(argv[1], 0))!= 0)
    {
        IplImage* dst = cvCreateImage( cvGetSize(src), 8, 1 );
        IplImage* color_dst = cvCreateImage( cvGetSize(src), 8, 3
            );
```

```
12          CvMemStorage* storage = cvCreateMemStorage(0);
13          CvSeq* lines = 0;
14          int i;
15          cvCanny( src, dst, 50, 100, 3 );
16          cvCvtColor( dst, color_dst, CV_GRAY2BGR );
17          lines = cvHoughLines2( dst, storage, CV_HOUGH_STANDARD, 1,
                CV_PI/180, 180);
18          for( i = 0; i < lines->total; i++ )
19          {
20              float* line = (float*)cvGetSeqElem(lines,i);
21              float rho = line[0];
22              float theta = line[1];
23              CvPoint pt1, pt2;
24              double a = cos(theta), b = sin(theta);
25              if( fabs(a) < 0.001 )
26              {
27                  pt1.x = pt2.x = cvRound(rho);
28                  pt1.y = 0;
29                  pt2.y = color_dst->height;
30              }
31              else if( fabs(b) < 0.001 )
32             {
33                pt1.y = pt2.y = cvRound(rho);
34                pt1.x = 0;
35                  pt2.x = color_dst->width;
36              }
37              else
38              {
39                  pt1.x = 0;
40                  pt1.y = cvRound(rho/b);
41                  pt2.x = cvRound(rho/a);
42                  pt2.y = 0;
43              }
44              cvLine( color_dst, pt1, pt2, CV_RGB(255,0,0), 3, 8 );
45          }
46          printf("total number of lines %d\n",lines->total);
47          cvNamedWindow( "Source", 1 );
48          cvShowImage( "Source", src );
49          cvNamedWindow( "Hough", 1 );
50          cvShowImage( "Hough", color_dst );
51          cvWaitKey(0);
52      }
53  }
```

The sample code was used to find straight lines on an image of the office. Notice that only one line is found, although there are clearly many identifiable lines, some being better than the one that was found. However, because of the poor choices of the Canny edge detector, the edges are interrupted and therefore the Hough transform can only find one line.

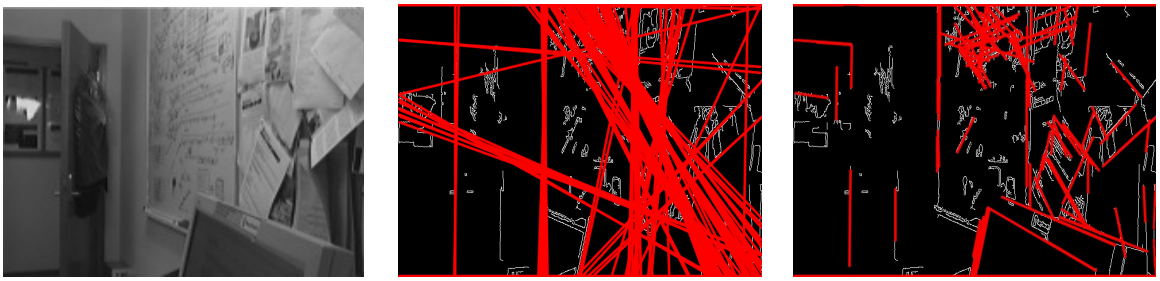

Figure 4.8: Using houghlines2 with the office image. In the first results the lines are represented as infinite lines, while the second results shows only segments of lines.

Hough transform for other curves

Hough transforms can also be used for circles, ellipses and any other shape. Generalising is conceptually easy, the difficulty is to find simple parameters that can be tested for various instances of the shape being searched via Hough transforms. The case of circle is an interesting one to study, as the parametrised equation of a circle is a simple on. One can write the circle equation as:

$$(x - x_c)^2 + (y - y_c)^2 = R^2 \tag{4.5}$$

Where x_c, y_c and R indicate the centre and the radius of the circle.

Comparing to the Hough transform of a straight line, one can see that there is an extra parameter. Therefore, the accumulator needs to be a 3D data structure rather than a 2D one. If the shape needs many parameters to be described, then the Hough transform becomes less efficient due the number of dimensions it needs to cope with. OpenCV has a version of the circle finder based on Hough transform, and a sample code follows:

Listing 4.6: Sample for finding circles

```
#include <cv.h>
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"

using namespace std;
using namespace cv;

int main(int argc, char** argv)
{
    if (argc!=2) {
      printf("needs an image\n");
      exit(0);
    }

    Mat image = imread(argv[1], 0);
    if(image.empty())
    {
        printf("Cannot open the image \n");
        exit(0);
```

```
20      }
21       Mat greyimage;
22       medianBlur(image, image, 5);//bluss the image a bit to get
             rid or noise
23       cvtColor(image, greyimage, COLOR_GRAY2BGR);
24       vector<Vec3f> circles;
25       HoughCircles(image, circles, CV_HOUGH_GRADIENT, 1, 10, 100,
             30, 1, 30);
26       //draw the circles found by the transform
27       for( size_t i = 0; i < circles.size(); i++ )
28       {
29            Vec3i c = circles[i];
30            circle( greyimage, Point(c[0], c[1]), c[2], Scalar
                 (0,0,255), 3, 1);
31       }
32       imshow("Circles detected", greyimage);
33       imshow("Original image",image);
34       waitKey(0);
35  }
```

In figure 4.9 we can see an example of applying the code above. To get the single circle in the figure, one has to change the parameters passed to the function.

Figure 4.9: An example of finding circles with HoughCircles().

One important aspect of the Hough transform is that it is completely dependent of the edges that are found by a previous processing. It is very difficult to find a perfect combination of parameters that will yield the best circles, so this method should be used with special attention to these details. Heuristics might help in some cases. For example, in figure 4.10 one of the false detections could have been found by tracing the intersection between the circles. One would then choose the circle that has the strongest result in the accumulator, meaning that there were more points in that particular contour.

4.2.4 Watershed segmentation

The methods seen so far are highly dependent on manual adjustments of parameters to perform well. Many of these methods will have more sophisticated approaches to try to automate the parameters choice. Region segmentation is an important area in computer vision because it

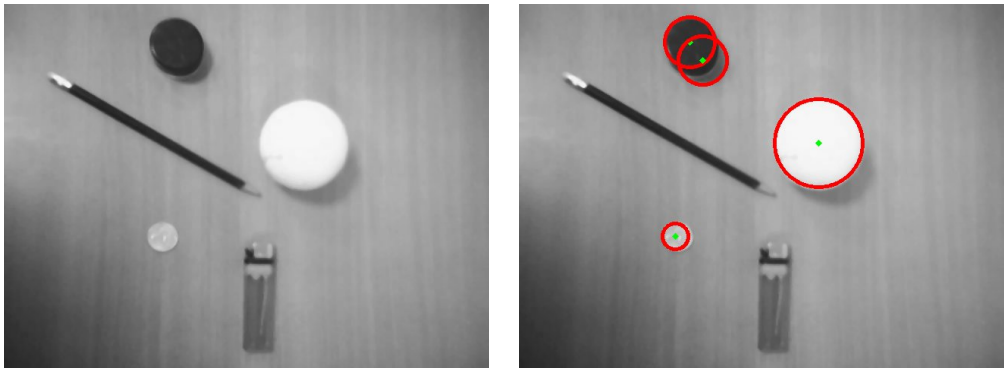

Figure 4.10: This shows an example of false circles caused by the shade of the object.

can split regions of interest and further processing can be carried out only on those regions. Watershed methods attempt to minimise the choice of parameters and it does not depend on a previous edge detection processing, rather detecting its own edges as the processing is carried out.

Watershed segmentation uses the variations in colour or greyscale levels to create a data structure that emulates a topographic representation of the image. A simulation of pouring "water" into the 3D image finds catchment basins and the watershed lines. As the "water" level increases, edges are created on the borders water-object. These edges can be marked as the process happens. Areas can then be split or joined depending on certain criteria. See an example in figure 4.11.

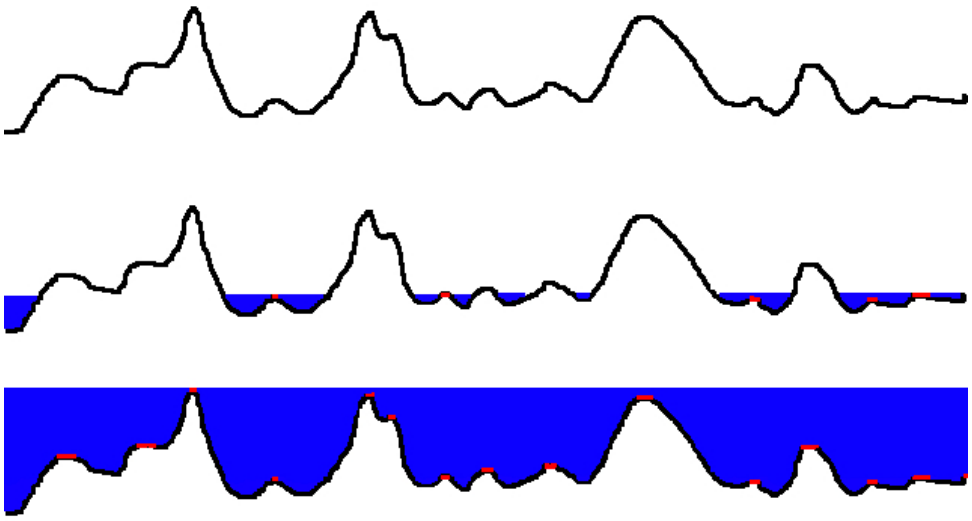

Figure 4.11: Marking the edges as the imaginary water fills up the regions.

A more meaningful example can be seen in figure 4.12. The peppers are difficult to segment due to the similarity of colours. Simple edge detection would not suffice to separate the regions.

Although there are many applications for the watershed transform, it is usually too slow to be used in real-time applications. Parallelisation can achieve good speed-ups (see for example

Moga et al.[3]).

Figure 4.12: Watershed segmentation example.

4.3 Reading

Read chapter 10 of Gonzales & Woods. Read *Johnson, M. J., 2003*, "Real Time Pipeline Profile Extraction Using Recursive Filtering and Circle Location".

4.4 Exercises

1. Write (or just modify) code to compute the LoG of a greyscale image using two options: a) a single 5x5 kernel. b) two 3x3 kernels. Compare the images. Are they identical? Compare the runtime for the two approaches using a large image (use the time utility in Linux).

2. Write (or modify) the code for edge detection. Take an argument to choose from: Sobel, Laplacian or Canny. Note that the resulting image may be of different nChannels for different OpenCV functions.

3. Test the Hough transform using cvHoughLines (or the equivalent) using different images. What happens when the thresholds are changed?

4. Write your own Hough transform code for a straight line. What considerations are you making regarding the size of the accumulators? How these decisions affect the performance of the code?

Bibliography

[1] D. Marr and E. Hildreth, "Theory of edge detection," *Proceedings of the Royal Society London B*, vol. 207, pp. 187–217, 1980.

[2] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*. Prentice Hall, 2002.

[3] A. N. Moga, B. Cramariuc, and M. Gabbouj, "Parallel watershed transformation algorithms for image segmentation.," *Parallel Computing*, vol. 24, no. 14, pp. 1981–2001, 1998.