Computer Vision

Project 1

Shrey Gupta

N16991441

Programming Language used – Java

Platform –Eclipse JEE Oxygen

Instructions to compile and run the program.

The .java files are included in the zip file. Open the files in Eclipse or any other IDE which supports Java(preferable Eclipse).

For testimage1c

1) Change the input image in line 12 of Project1.java to TestImage1c.png
2) Comment the lines 59 and 64 by inserting '//' in the beginning of the line.
3) Put in the threshold value as 68 in line 21 of Hough.java. The command will then look like
   ```
   Vector<HoughLine> lines = houghObject.getLines(68);
   ```
4) Change the output location to desktop or anywhere in Project1.java, Hough.java, nonmaxima.java
5) Run Project1.java
6) Run nonmaxima.java
7) Run Hough.java

For testimage2c

1) Change the input image in line 12 of Project1.java to TestImage2c.png
2) Put in the threshold value of 65 in line 21 of Hough.java.
3) Uncomment 59 and 64 and put the value of threshold for numPoints as 600 in line 59 of line2.java.
4) Change the output location to desktop or anywhere in Project.java, Hough.java, nonmxima.java
5) Run Project1.java
6) Run nonmaxima.java
7) Run Hough.java

For testimage3

1) Change the input image in line 12 of Project1.java to TestImage3.png
2) Put the value of threshold for numPoints as 800 in line 59 of line2.java.
3) Put in the threshold value of 84 in line 21 of Hough.java.
4) Change the output location to desktop or anywhere in Project.java, Hough.java, nonmxima.java
5) Run Project1.java
6) Run nonmaxima.java
7) Run Hough.java

What I have done-

Initially an image is taken in Project1.java. the image is accepted in png format as in RGB type. The image is converted to grayscale. The luminance is calculated according to the following formula

Luminance = *0.30R + 0.59G + 0.11B*

Every single component of the pixel i.e. red, green and blue all are set to this luminance value.

After the grayscale image is obtained, the image is smoothed with the help of mean filter. A filter of size 3X3 is used for this smoothing. Each pixel's new value is computed by convulating the value by this mask, the calculation of which is explained in the source code. The smoothed image is then used to obtain an edge image output using a sobel operator. The gradient magnitude is computed by the square root of the sum of squares of horizontal and vertical component. The calculated value is then shifted by bytes in the code so that the value remains between 0 and 255.

After obtaining an edge image with sobel, the image is converted to a binary image .Every pixel is traversed from left to right and top to bottom . The threshold value is kept at 70. If the value of pixel is greater than 70, then it is set to 255 else it is set to 0.

The binary image obtained of edges has very thick edges. As a result, a thinner image can be obtained by using non maxima suppression

The mean filtered smoothed image is taken and the gradient magnitude and angles are calculated with the help of Sobel operator. The sector value for each pixel is also calculated by analyzing the angle. The value of the pixel is determined by checking of each pixel is local maxima or not by comparing them to its neighbours along the sector value. When each pixel is computed, thresholding is done to convert the image into a binary image. The threshold is set to 30, below which the pixel is set to 255 and above which the value is set to 0.

After the thin edged non maxima image is obtained, straight lines along the edges are detected with the help of Hough Transformation

For each pixe in the image the value of p is calculated according to the formula

P = (x-xcenter)cos(theta) + (y-ycenter)sin(theta) where theta is the angle .

For each value of pixel x,y which satisfy the above equation, the value in the accumulator is incremented. To keep in check as all the points are checked the counter for number of points is increasing.

A hough array image can be also plotted to get a better idea about the accumulator(the code for which is provided.

A function of constructing lines in the image is called which returns vectors to give lines in the image. The function goes through each value of the accumulator and sees if it is a peak value in the neighbourhood of 4X4 pixels. If it is peak value, the corresponding value in peakarray is set to 1 for future operations. The theta and r value which are locally the peak value are then used as parameters to the object of HoughLine class which contains the function to construct horizontal and vertical lines. If the angle is less than PI/4 and greater than 3PI/4, then vertical lines are created else horizontal lines are created.

In such a way lines are drawn on the edges of the input image. These lines are then used to detect parellelograms in the image.

Traversal is done for each value in the peakarray which contains either value 1 or 0. If the value is 1, it indicates that the corresponding theta and r values are locally peak values and hence have a line in the image. Initially two lines are chosen with same angle. After that 2 more lines are selected having the same angle but different than the angle of first pair of lines.These 4 lines along with the image are called as parameters of object of line2 class. The line2 class contains the function to return the coordinates of the parellelograms.

In line2 class, the intersection point of all 4 lines are calculated giving us 4 points. The next thing is checked , as to how many edge points exist on these lines. If the edge point density is high enough, then the candidate parallelogram is considered to be a legit parallelogram. It will return the coordinates of the parellelograms. If the threshold value is set low , then many irrelevant parellelograms are also detected along with it because of complexity in some images.

When the code is running on TestImage1c, the output of the coordinates of the parellelograms are:

```
0 , 0 | 1 , 754 | 0 , 0 | 1006 , 754
```

When the code is running on TestImage2c, the output of the coordinates of the parellelograms are:

```
1 , 378 | 1 , 754 | 1006 , 378 | 1006 , 754
673 , 1 | 296 , 378 | 752 , 1 | 375 , 378
```

When the code is running on TestImage3, the output of the coordinates of the parellelograms are:

```
115 , 1 | 115 , 411 | 176 , 1 | 176 , 411
115 , 1 | 115 , 411 | 345 , 1 | 345 , 411
115 , 1 | 115 , 411 | 466 , 1 | 466 , 411
115 , 1 | 115 , 411 | 480 , 1 | 480 , 411
176 , 1 | 176 , 411 | 345 , 1 | 345 , 411
176 , 1 | 176 , 411 | 466 , 1 | 466 , 411
176 , 1 | 176 , 411 | 480 , 1 | 480 , 411
345 , 1 | 345 , 411 | 466 , 1 | 466 , 411
345 , 1 | 345 , 411 | 480 , 1 | 480 , 411
466 , 1 | 466 , 411 | 480 , 1 | 480 , 411
```

I have included another sample image in the zip fle. The test images provided to us did not have exact parelellograms and the angles were off by 1 or 2 degrees. I used a step of 1 degree for angle in the accumulator array, so my algorithm looks for perfect parellelograms in the image. The algorithm may not work perfectly for testimage1c because it is not a perfect parallelogram. However to prove it works perfect, I used it on the sample image which gives the accurate answer. The threshold values are same as testImage1c.

The output for sample.png is

```
75 , 89 | 75 , 288 | 272 , 89 | 272 , 288
```