

Introduction to Visual Computing

Assignment# 10

Line Detection with the Hough Transform

April 27, 2020

Description

Following the detection of the green Lego board in an image, we want now to extract the edges of the board as lines. Once we have our four lines, finding the corners is a matter of computing their intersections.

The main algorithm to extract lines from an image is called the **Hough transform**. We will implement it this week.

Objectives

Based on the results of edge detection performed in the previous weeks, implementing a Hough transform algorithm to extract lines in still images and in a webcam stream.

Specific Challenges

Understanding a typical real-world image processing algorithm.

Preliminary steps

If needed, finish last week's assignment: it is a pre-requisite for today.

Part I

The Hough algorithm

The Hough transform is an algorithm that extracts **lines** from an image (not only edges as we did with the edge detection algorithm, but real lines, with their equations).

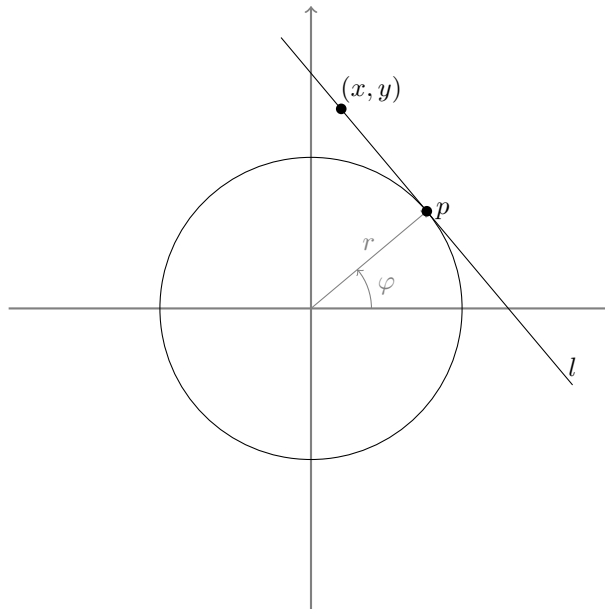
Polar representation of lines

First, a brief reminder on lines in polar coordinates:

A point (r, φ) in polar coordinates has Cartesian coordinates

$$\begin{cases} x = r \cdot \cos(\varphi) \\ y = r \cdot \sin(\varphi) \end{cases}$$

In polar coordinates, a line l is represented by (r, φ) such that l is the tangent to the circle of radius r at a point p forming an angle φ with the x -axis:



Besides, the set of lines passing through the point (x, y) can be described in polar coordinates by:

$$\begin{cases} r = x \cdot \cos(\varphi) + y \cdot \sin(\varphi) \\ \varphi, \text{ for } \varphi \in [0, \pi[\end{cases}$$

Principle of the algorithm

The key idea of the Hough transform is to compute **for each pixel** that **could** belong to a line (i.e. pixels that belong to the edges that you have detected) the **whole list of lines that pass through this pixel**. Those lines are the **candidate** lines.

So, for each pixel, we get the (infinite) set of all the lines from all possible directions, going through this point (obviously, the set is not actually infinite: it will be limited by the size of our **discretization step**).

We store all these lines for all the pixels in a buffer called an **accumulator**: everytime a candidate line (r, φ) is found, we increment the accumulator at position (r, φ) by one: `accumulator[phi * rDim + r] += 1`.

After analysing all pixels, the lines with most votes (i.e. the lines that go through the maximum of pixels) will be the more likely to correspond to real lines in the image.

Step 1 – Compute and store the polar representation of lines passing through edge pixels

The algorithm takes as input a black-and-white image. You can use the synthetic image provided on Moodle (`hough_test.bmp`) or, if you feel brave, the output image from your edge detection.



Figure 1: On the left, the Moodle synthetic image. On the right, an example of edge detection result.

Iterate over the image and for each pixel belonging to an edge, compute and store in the **accumulator** buffer the candidate lines. Use two variables `discretizationStepsPhi` and `discretizationStepsR` to set the discretization level of the line set. For each pair (r, φ) in the accumulator that received more than 50 votes, store them in a List and return them.

Page 4 of 9

```
return lines;
}
```

Step 2 – Display the accumulator

Using the following code, display the content of the accumulator.

```
PImage houghImg = createImage(rDim, phiDim, ALPHA);

for (int i = 0; i < accumulator.length; i++) {
    houghImg.pixels[i] = color(min(255, accumulator[i]));
}

// You may want to resize the accumulator to make it easier to see:
// houghImg.resize(400, 400);

houghImg.updatePixels();
```

The resized `houghImg` should be similar to the following picture.



Step 3 – Plot the lines

Write a function to plot the lines on top of the original image using a code similar to the following:

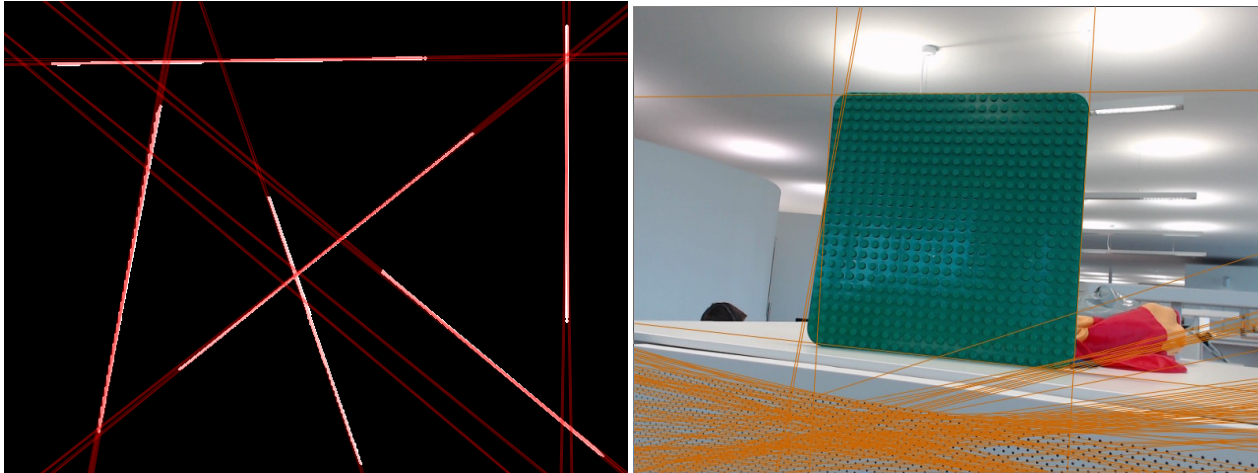
```
for (int idx = 0; idx < lines.size(); idx++) {
    PVector line=lines.get(idx);
    float r = line.x;
    float phi = line.y;

    // Cartesian equation of a line:  $y = ax + b$ 
    // in polar,  $y = (-\cos(\phi)/\sin(\phi))x + (r/\sin(\phi))$ 
    // =>  $y = 0 : x = r / \cos(\phi)$ 
    // =>  $x = 0 : y = r / \sin(\phi)$ 

    // compute the intersection of this line with the 4 borders of
    // the image
    int x0 = 0;
    int y0 = (int) (r / sin(phi));
    int x1 = (int) (r / cos(phi));
    int y1 = 0;
    int x2 = edgeImg.width;
    int y2 = (int) (-cos(phi) / sin(phi) * x2 + r / sin(phi));
    int y3 = edgeImg.width;
    int x3 = (int) (-(y3 - r / sin(phi)) * (sin(phi) / cos(phi)));

    // Finally, plot the lines
    stroke(204,102,0);
    if (y0 > 0) {
        if (x1 > 0)
            line(x0, y0, x1, y1);
        else if (y2 > 0)
            line(x0, y0, x2, y2);
        else
            line(x0, y0, x3, y3);
    }
    else {
        if (x1 > 0) {
            if (y2 > 0)
                line(x1, y1, x2, y2);
            else
                line(x1, y1, x3, y3);
        }
        else
            line(x2, y2, x3, y3);
    }
}
```

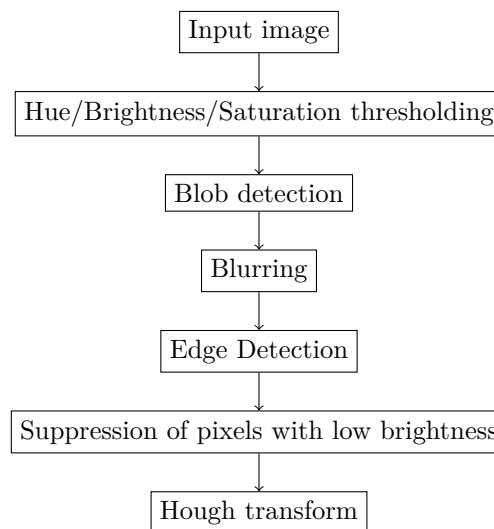
As you can see, a lot of wrong lines are detected. We will see next week how to filter them out.



Part II

Line Detection with the LEGO Board

Update last week's application to include the Hough Transform in your pipeline for image processing.



Part III

Using the Webcam

Last week, we implemented an edge detection algorithm using still images. Since our aim is to “live control” the game board by manipulating the Lego board in front of a webcam, we now want to replace the still images with the webcam stream.



Note

You will need to add the Processing video library to your installation in order to access the webcam:

- If you are using **gstreamer1.0** on your system (**Ubuntu 16.10 and later**), download the **video-2.0-beta1.zip** from <https://github.com/processing/processing-video/releases/tag/r3-v2.0-beta1> and extract it into your libraries directory, commonly found under `$HOME/sketchbook/libraries/` or `$HOME/$DOCUMENTS/Processing3/libraries/`. See <https://github.com/processing/processing/wiki/How-to-Install-a-Contributed-Library> for more details on how to install an external library.
- If you are using **gstreamer0.1** on your system (**Ubuntu 16.04 and earlier**), go to Sketch → Import Library... → Add Library, find the **Video** library there and install it.

As a first step, use the following code to open and display the image stream of your webcam:

```
import processing.video.*;

Capture cam;
PImage img;

void settings() {
    size(640, 480);
}

void setup() {
    String[] cameras = Capture.list();
    if (cameras.length == 0) {
        println("There are no cameras available for capture.");
        exit();
    } else {
        println("Available cameras:");
        for (int i = 0; i < cameras.length; i++) {
            println(cameras[i]);
        }

        //If you're using gstreamer0.1 (Ubuntu 16.04 and earlier),
```



```
//select your predefined resolution from the list:
cam = new Capture(this, cameras[21]);

//If you're using gstreamer1.0 (Ubuntu 16.10 and later),
//select your resolution manually instead:
//cam = new Capture(this, 640, 480, cameras[0]);

cam.start();
}
}

void draw() {
  if (cam.available() == true) {
    cam.read();
  }
  img = cam.get();
  image(img, 0, 0);
}
```



Note

Note the console output: several modes (different resolutions, different framerates) are possible. We recommend to use a resolution of 640x480.



Note

On some systems the auto-regulation of the camera, such as white balancing, might appear odd. You could try to change this parameters using the camera proprietary software on Windows and MAC or Gvvcview on Linux.