Introduction to Visual Computing

Assignment# 11

Line Selection and Quad Estimation with the Hough Transform

May 7, 2019

Description

Last week, we implemented the core of the Hough transform: the detection all the lines present in an image.

This week, we will filter them to keep only the best ones, and we will compute the intersections between lines to identify possible corners.

Objectives

Based on the result of the line detection performed last week, select the best lines and compute and display their intersections.

Specific Challenges

Post-processing a real-world image processing algorithm to account for the inevitable noise.

Preliminary steps

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

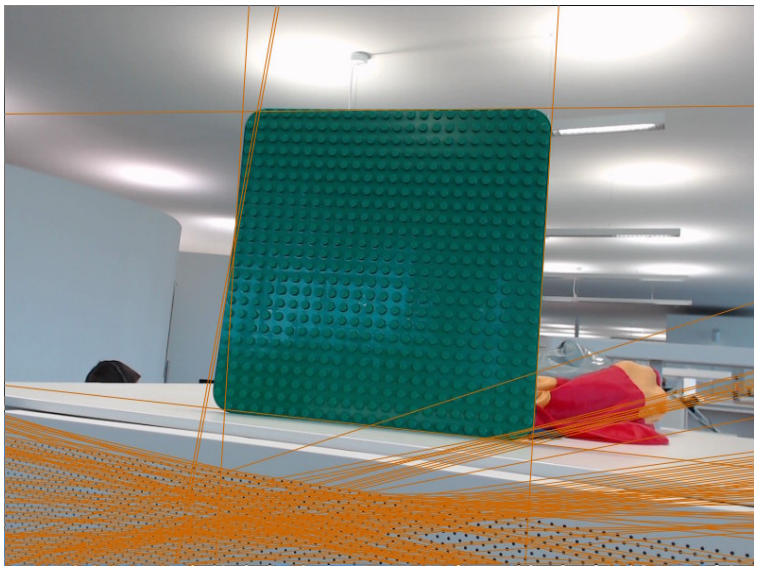
Part I

Select the best candidate lines

Note

To ease the debugging, we recommend you to first work with a static image, and then try with the webcam.

After last week's detection, you should have found lines similar to these:



As you can probably see, many lines are detected. We need to filter them out and keep only the best ones.

Step 1 – First approach: the n-best lines

Modify your Hough transform function to take a parameter `nLines` and select (and plot) only the best (i.e. the most voted) `nLines` lines.

To this end:

- create an `ArrayList<Integer>` called `bestCandidates`,

- store in this list only the candidates (i.e., their **indexes** in the accumulator) with more that `minVotes` votes (to quickly remove most of the lines, before sorting the best ones – adapt `minVotes` to your implementation),
- sort these indexes by their number of votes. To achieve this, you need a special **comparator** that compares two indexes based on the number of votes received by the two corresponding lines:

```
class HoughComparator implements java.util.Comparator<Integer> {
    int[] accumulator;
    public HoughComparator(int[] accumulator) {
      this.accumulator = accumulator;
    }
    @Override
    public int compare(Integer l1, Integer l2) {
      if (accumulator[l1] > accumulator[l2]
          || (accumulator[l1] == accumulator[l2] && l1 < l2)) return -1;
      return 1;
    }
}
```

To use this comparator:

```
        ArrayList<Integer> bestCandidates = new ArrayList<Integer>();

        //TODO fill bestCandidates ...

        Collections.sort(bestCandidates, new HoughComparator(accumulator));

        // bestCandidates is now sorted by most voted lines.
```

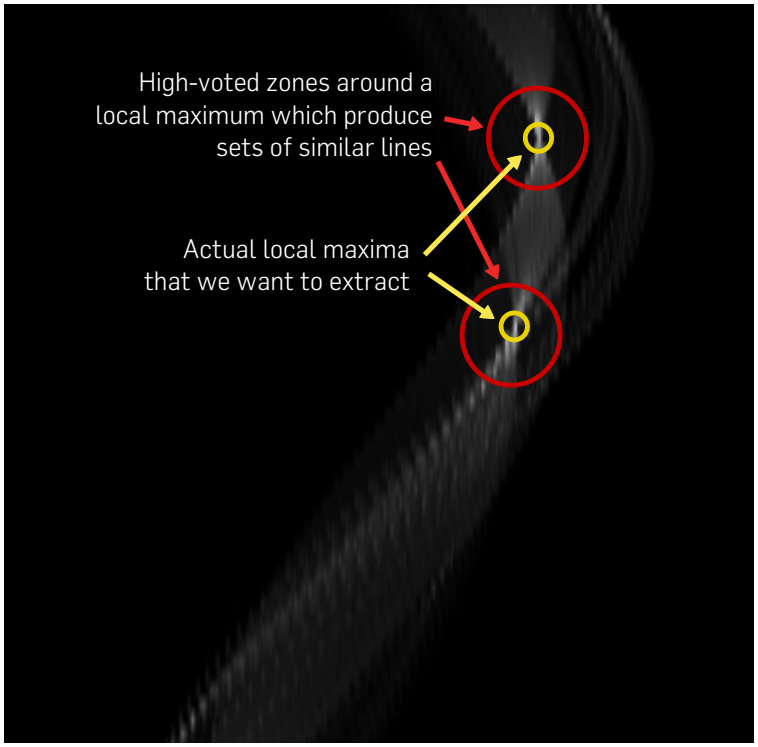
Note

If you are not familiar with the Comparator interface, have a look at `https://docs.oracle.com/javase/tutorial/collections/interfaces/order.html`

Finally, plot the `nLines` first lines in `bestCandidates`.

Step 2 – Local maxima

The n-best candidates approach works well, but tends to return groups of very similar lines, corresponding to the zones around local maxima in the accumulator:



To isolate local maxima, we can simply look for lines with more votes in a region around each candidate line in the accumulator, and only keep the line with the most vote in that region.

The algorithm to find local maxima goes as follow:

- Define the size of the region (i.e. the neighbours considered) in which you will search for local maxima, e.g 10.
- For each element of the accumulator
  - if its number of votes is higher than `minVotes`, then
    - if every neighbouring element in the accumulator has a lower number of votes, then
      - add its index in the accumulator to the `bestCandidates` list.

Using the algorthim above, implement this optimization **before** sorting the `bestCandidates` list.

Note

At this point you should have a hough method to return the detected lines as an `ArrayList<PVector>`, each vector containing a pair (r, φ)

Part II

Intersections, Corners and best quad selection

Today's last task is to detect the board corners in a reliable way. Starting from the set of lines, we will find the quads formed by the lines and select the best one according to the following heuristics:

- Non-convex quads
- Quads that are too small
- Quads that are almost flat

In order to do this you will use the class *QuadGraph* provided on Moodle and call the method `findBestQuad()`, which takes as input

- The list of lines `List<PVector> lines`
- The size of the original image, `int width`, `int height`
- The max and min area for a valid quad, `int max_quad_area`, `int min_quad_area` (tune these parameters)
- A boolean, `verbose`, to print some debugging messages

The function returns a list of four corners of the biggest valid quad or an empty list if no valid quad is found.

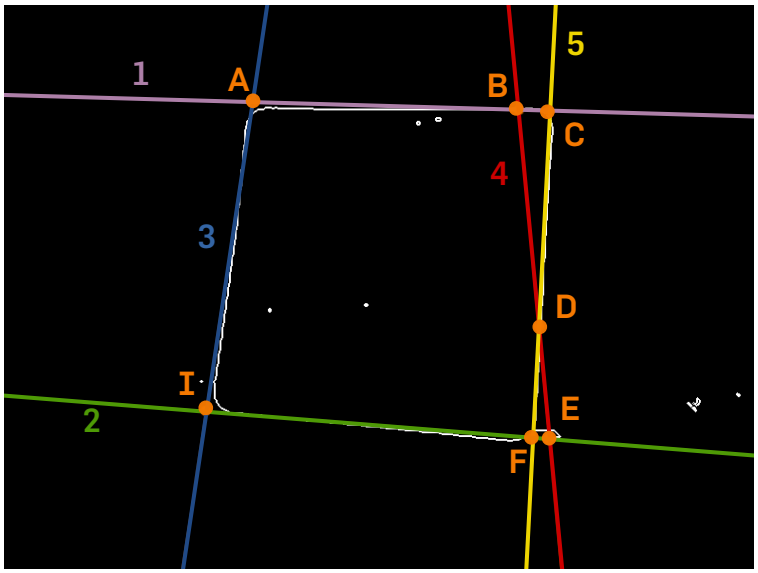
You should now have achieved a pretty robust selection of the four corners of our Lego board.

Next week, we will compute the physical orientation of the board from these four corners.

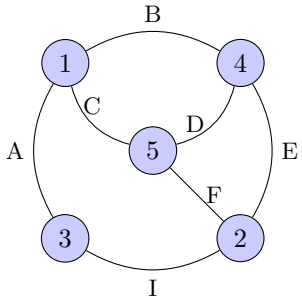
Wondering what is inside QuadGraph?

The class implements a a *Depth-first Search* algorithm to walk through the graph and detect cycles (methods `findCycles()` and `findNewCycles()`).

If you look at the next image, 5 lines and 7 intersections were detected:



This can be represented as a graph where lines are nodes and intersections are edges:



Using this representation, finding quads equates to finding cycles of length 4 in this graph. In this case, **three** quads can be extracted from the lines (lines 1,3,2,4, lines 1,5,2,4 and lines 1,3,2,5). We simply represent the graph as the list of edges. For instance, the graph above would be represented as (the order is not important here since our graph is undirected):

```
int[][] graph =
    {
        {1, 4}, {1, 5}, {1, 3},
        {2, 4}, {2, 3}, {2, 5},
        {4, 5}
    };
```

Intersections between lines are computed knowing that in polar coordinates, two lines (r_1, φ_1) and (r_2, φ_2) intersect at:

$$\begin{cases} x = \frac{r_2 \cdot sin(\varphi_1) - r_1 \cdot sin(\varphi_2)}{d} \\ y = \frac{-r_2 \cdot cos(\varphi_1) + r_1 \cdot cos(\varphi_2)}{d} \end{cases}$$

with $d = cos(\varphi_2) \cdot sin(\varphi_1) - cos(\varphi_1) \cdot sin(\varphi_2)$.

Part III

Make it Faster

This last part suggest one major optimization to get your code to run faster.

THIS PART IS OPTIONAL but recommended for a fluid game experience.

Lookup tables for trigonometric functions

In your Hough transform, the CPU spend a lot of time computing `sin` and `cos` of always the same set of φ. You can improve performances a lot by pre-computing these values:

```
// pre-compute the sin and cos values
float[] tabSin = new float[phiDim];
float[] tabCos = new float[phiDim];

float ang = 0;
float inverseR = 1.f / discretizationStepsR;

for (int accPhi = 0; accPhi < phiDim; ang += discretizationStepsPhi, accPhi++) {
    // we can also pre-multiply by (1/discretizationStepsR) since we need it in the Hough loop
    tabSin[accPhi] = (float) (Math.sin(ang) * inverseR);
    tabCos[accPhi] = (float) (Math.cos(ang) * inverseR);
}
```

Update your Hough transform to use the pre-computed values.