Random Shape Recognition
By Thien Nguyen – Boston University – CS 585

Problem definition
The goal of this project is to design and implement an algorithm that is capable of recognizing objects based on the difference in colors and determine the shape of the objects.

The following requirements were implemented:

1. Label each shape blobs according to their color.
2. Find the outermost contour of each shape blob.
3. Classify the type of border pixels (against background, against another shape, or against edge of image).
4. Recognize the shape of each shape blob (square, triangle, or circle).

Note: The original dataset was used as input data

Method and Implementation
The program’s behaviors according to the requirements are as follow:

Input: a JPEG image with a single, or multiple shapes (square, circle, triangle), with each having a different color compared to other shapes and the background color.

Output: n different images where n is the number of shape blobs detected in the original input.

Implemented requirements:

1. Each blob detected is put on a separate image by its own, with its original color and position (black background, original background color removed).
2. The border of each blob on its own image is colored as follow:
   - White: border against background
   - Green: border against another blob
   - Blue: border against the edge of the image
3. Each blob outputted has a letter on top indicating which shape it was recognized as, (S)quare, (T)riangle, or (C)ircle. In addition, each blob-separated image outputted is saved with the original name + the shape recognized. i.e. “1234_square.jpeg”
Algorithms

1. **Blob detection based on colors.**
   I realized there were a lot of noises in the original input. Thus, speaking in terms of computer vision, there were more than hundreds of “different colors”, with each having a 1-2 value difference in each color channel compared to the other pixels. To solve this problem, I **grouped all of the pixels with approximately** the same values and set all of them to the color that has the highest frequency in the group. Specifically, the program treats 2 pixels as the same color if none of their channels has a difference higher than a certain threshold, which I figured **13 is the optimal** threshold for this project.

   After grouping, the program counts the frequency of each color. At this point, there should only be 2-5 colors on the image (background and 1 to 4 blobs). The color with the **largest count is determine to be the background.** The other colors counted are determined to be blobs’ colors and their coordinates are saved in different variables.

2. **Border following algorithm.**
   Moore’s border following algorithm was implemented for this part. Briefly, in a raster scan style, find the first occurrence of the object color, follow the border and “go around” it until the first occurred pixel is seen twice.

   The input for this algorithm is a separated blob by itself, in its original color, with the background removed (black).

   I encountered **2 problems** while implementing this algorithm.

   1. After the original cleaning when separating blobs, there were still some noises left. The noises give the Moore algorithm a really hard time to find the starting point to trace the border of the object. i.e. A pixel by itself in the middle of nowhere. When the algorithm reaches this pixel, it starts the tracing process, and “successfully” returns that single pixel as the border, which is obviously not the object that we are looking for.
      
      My solution to this problem was to clean it one more time, this time, by blurring the whole image using the average value of every pixel and their 8 neighbors.

   2. When the “cursor” of the algorithm reaches the edge of the image, it still tries to go “around” the border. Essentially, it wants to continue to go out of the edge to make sure that the border does not continue out there. This causes an index error as the pixel it’s looking for is not in the image.
      
      I fixed this problem by padding the image with an extra layer of black pixels for the “cursor” to “walk” on. The pads are add before the process starts and removed afterward.
3. **Border classification.**

This algorithm takes as input a set of pixel coordinates of a border. The algorithm is rather simple. For every pixel in the border:

- If the pixel has a value of 0 or 127 in either its x or y coordinate
  - It’s a border against the edge
- If the pixel has a color that is the same with the background’s
  - It’s a border against the background
- Otherwise
  - It’s a border against another blob

4. **Shape recognition algorithm**

Input: a set of pixel coordinates of a border.

This algorithm uses an elimination method, checking for the unique attributes of the object, if false then we can eliminate the object from the set of possible results.

Border pixels that are against the edge of the image are removed before the process, as they might be mistaken for a edge square or triangle.

First, check if it is a square.

- A square must have a vertical line. In theory, ¼ of the pixels of the border of a square must have the same x values. Some time if might be chopped of by other shapes so I set it to 1/6.
  - If over 1/6 of the total number of pixels have the same x values then it’s a square.

If not a square, check if it is a triangle.

- Similar to above, a shape with a vertical line but not a square, then it must be a triangle.
  - If over 1/6 of the total number of pixels have the same y values then it’s a triangle.
  - Otherwise.
  - It’s a circle.
Results

Worked-out examples:

In these examples, everything worked out almost perfectly.
The shapes of the blobs are correctly identified.
The blobs are separated cleanly without noise.
The borders are correctly classified.
In these examples however, the results do not turn out to be quite expected. The flaws in the implementation that causes these errors are discussed in the “Discussion” section below.

Shape detection testing:

The precision of the algorithm is calculated based on the equation (= TP / (TP + FP)) with TP (true positive) being the number of times the algorithm correctly recognizes a shape, and FP where it falsely recognizes one.

Comparing output images to annotations: **83.46% accuracy**

I noticed that in some cases the blobs separation is not very successful. Thus, resulting in a faulty shape recognition process.

I tried running the algorithm directly on the given annotation images to see how it performs. For each image in the annotations folder, I run the Moore algorithm to first detect the border of the object. Then input the set of border coordinates to the shape recognition algorithm to test it separately from the blob separating algorithm.

The correctness of the test was acquired by comparing the shape recognized by the algorithm with the name of the file, which clearly states the shape it holds.

The result turned out to be quite good: **89.66% accuracy**
Discussion

There were a lot of flaws in this implementation:

1. When separating blobs, a threshold of 13 is apply to ignore noises. This threshold works for most cases. However, it also makes the algorithm unable to separate blobs which have a color difference of less than 13.

   Speculative solution: An algorithm that checks for the connectedness based on the coordinates of the pixels could potentially help to distinguish noises and actual distinctive blobs.

2. Second cleaning: when blurring the image to detect noises in the middle of nowhere, I believe, close to 100% of noises were removed, however, some pixels that were parts of the objects were mistaken to be noises and were also removed.

   i.e. The tip of a triangle, since the average of its 8 neighbors has more blacks than colors, it is determined to be noised pixel and thus removed, resulted in a rounded tip triangle/square.

   However, as we can see below, the roundness is hardly noticeable and it does not interfere with the shape recognition algorithm. In my opinion, the tradeoff is worth taking as it removes almost 100% of the noise in every case.

   Speculative solution: A mask could be applied to be more biased toward the center pixel so that the average is not too close to black even if there were more black neighbors around.
3. Shape detection: border pixels against the edge of the image cannot be taken into account. However, when checking for a triangle, we only check for the horizontal line, if that base line of a triangle happen to be at the edge of the image, it cannot be detected, subsequently, be determined as a circle.

Speculative solution: Add a check for the diagonal line to check for triangles.

Also, when a shape is overlapped by another. The algorithm is not capable of telling which edge is which. For example, when a circle is overlapped by a square

The algorithm sees a vertical edge and determine it to be a square.
Speculative solution: Check the concavity of the edge, if it’s concave then it obviously does not belong to the object and should be exclude from consideration.
Conclusion

This is my first project working with images. It gives me a taste of how complex image analysis could get. Even though, I knew ahead of time the type of inputs and exactly what I’m looking for in the image, it was difficult enough to detect them. I can only imagine the level of complexity it would take to implement an algorithm that is capable of recognizing an arbitrary object in a random image.

I’m looking into implementing the speculative solutions stated above in the “Discussion” section to further improve the accuracy of the program. This project is one that I really enjoy doing as it introduces me to a new aspect of computer science that I’ve never touched.

Credits and Bibliography

Python libraries used:

- openCV (image manipulations)
- numpy (array manipulations, mostly used to for colors’ values computation)
- collections.Counter (counting objects)

Moore’s algorithm tutorial


And lot of help from friends and TF (Yifu Hu) in CS 585