Shape Detection Report

Sean Chen

Contents

List of Figures

1 Overview

I implemented a very basic shape detector that can theoretically detect any shape given a particular structuring element. It works reasonably well with circles, but is somewhat inefficient (currently runs at about 2 frames a second). The code, when run, provides a list of all possible objects, with their image location data and size.

OpenCV contains several ways to detect circles. The canonical way seems to be using the "Hough Transform". However, there are certain downsides to this approach. For one, this only allows us to detect circles in the image. Second, after some testing with the code, it seemed to me that this method also required an infeasible amount of parameter tweaking. There was an extremely high false positive rate (almost every frame without a circle was detected to have a circle). Consequently, I tried to implement a more general detection method that reduces the

search space of objects by filtering out colors. Due to time constraints, I chose the most simple method possible. While the results, in my opinion, are much better, I realize the simplicity of my approach and urge the reader to implement a more state-of-the-art method, possibly using machine learning. Pointers to such methods are provided in the "Related Works" section of this report.

Software requirements are listed on the Wiki page. Code can be found the GitHub link provided on the same Wiki page.

1.1 Goals

The goal was to make a shape detector that could run on any hardware using a simple USB webcam without requiring training data.

2 Technical Details

2.1 `MorphDetector` class

The `MorphDetector` class provides a framework for the user to detect any kind of shape. The user needs to do two things to take advantage of the object detection:

1. Implement the constructor, which at minimum needs to take a filter_width, window_size, threshold, pyramid_scale and suppression method. The explanations can be seen in the code for the class. I will also go over them in more detail in this report.
2. Implement the filter

2.2 `create_filter`

A filter (or kernel) in my software is simply a matrix that represents a shape. In our code, the matrix has the additional constraint of being square (rows = columns). For example a cross can be represented as:

Figure 1: Filter example

```
0 0 1 1 0 0
0 0 1 1 0 0
1 1 1 1 1 1
1 1 1 1 1 1
0 0 1 1 0 0
0 0 1 1 0 0
```

The cross shape can be seen where 1s are. The `filter_width` is 6. Theoretically, any values can be used, though care should be taken in the code to account for such. More specifically, the threshold will need to be changed appropriately (more will be explained later). The filter ideally should be created in a way that allows it to be resizable. However, it is possible to simply draw such a filter in a text file and then write a parser to convert it into a fixed-size array. In such a case, the filter_size parameter must be changed in the constructor to represent the width of the square array because it is used to estimate the size of the object, as well when to stop pyramiding the image. The benefit of the filter method is that it is dead simple and extremely extensible, since all that is required is to make a square matrix. The downsides include the fact that it's a very naive algorithm, so it's not as effective as more cutting edge techniques involving data sets/machine learning or more complicated math. Ultimately, the decision was made to pursue this simple method because of the extremely short amount of time I had to implement this code.

2.3 `detect`

Broadly speaking, the detect method works by cross-correlating the provided filter with the image. The linked Wikipedia article provides an excellent summary of it, but briefly, cross-correlation allows us to score the similarity between two signals ("sliding dot product"). For example, if we have two functions f and g, where f(t) = t and g(t) = t, then the cross correlation will give us a high score because the functions are similar (they are exactly the same, actually). If f(t) = 1 and g(t) = t, the score will be lower. Consequently, an image can be viewed as a two dimensional signal. Thus, if we cross-correlate the provided filter from `create_filter` with the same sized patch, we can get a "response" which tells us exactly how similar that patch was to the filter. Therefore, if the filter is created in a way that emphasis a circular shape, cross-correlation will give us a high score for a circular looking patch in the image. This cross-correlation is applied to EVERY patch in the image to give us a score at each pixel. Borders also need to be considered because a patch may lie off the image. In this case, we need to "guess" what is off the image (or ignore it entirely). My code uses BORDER_REPLICATE, which simply extends the border as if those edge pixels continued off the image.

The circle filter used in my code has 1s for pixels in the circle, and -1s for pixels outside of the circle. The -1 effectively penalizes pixels that show up as part of the circle that are outside the circle boundary. This is necessary because using our method, one can see that a square will also show up as a circle if we were to use only 1s and 0s. For example, suppose a square that is 10x10 in size. If we were to use a circle filter with 1s in the edges to detect a circle of diameter 10, we would get the highest score of $10\pi * r^2$. The parts of the square that are not part of the circle are multipled by 0 during the cross-correlation. Thus, by setting those pixels to -1 instead of 0, the score decreases for such border pixels.

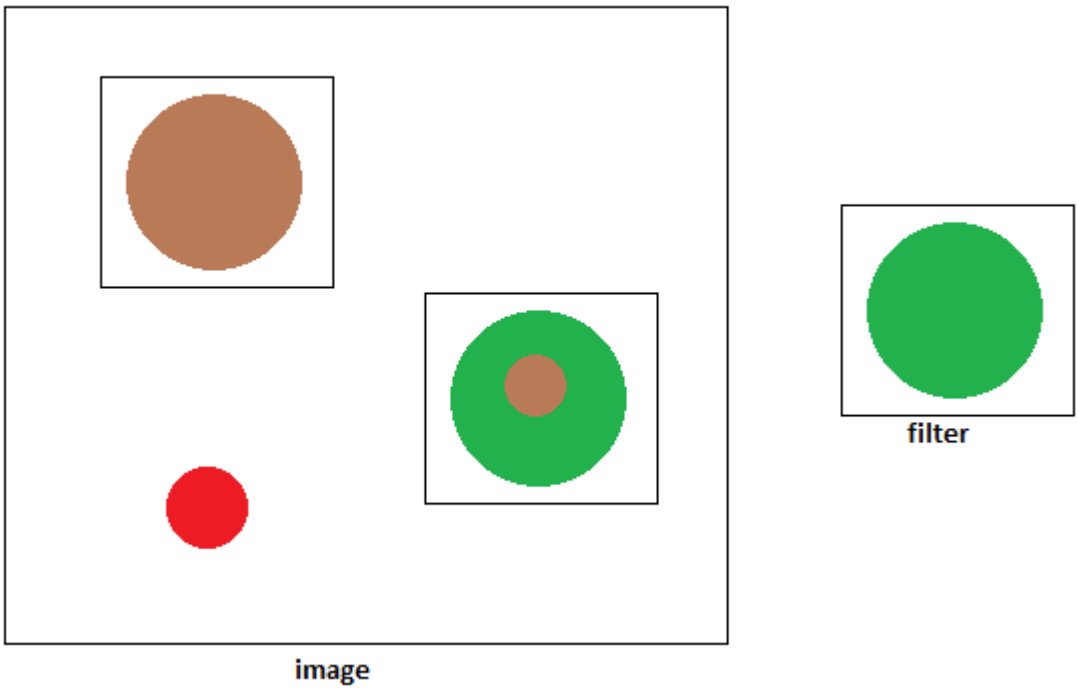
Figure 2: Picking values for filter

Here, we are detecting a red square, with the circular filter represented by the green. Note that using 0s, the square is considered to be a circle because the red area is ignored when cross-correlation multiplies it by 0. When set to -1, the red area is penalized, giving the score for the square a much lower score than a circle that was exactly the size of the filter

2.4 Pyramiding

Thus, I have explained how we can detect circles at a single image scale. What if there's a patch that looks like the filter, but it's too big? The score with thus be very low, because cross-correlation emphasizes a direct correspondence between signals. Therefore, to detect the same object at multiple sizes, we use a method called pyramiding. In essence, we iteratively decrease the size of the image by `pyramid_scale` amount (by dividing the number). Therefore, if we want to divide the image by half, we would provide `pyramid_scale` = 2. As one can see, the scale directly affects the performance of the detector. At large scales such as 2, we are skipping objects of sizes in between the size of the filter and double the size of the filter. However, of course, the downside of using a fine-grained scale such as 1.1 means a decrease in performance since we have to re-run the detection code at each resized image. Additionally, note that because of pyramiding, the smallest object we can detect is exactly that of `filter_width`.

Figure 3: Multi-scale issues



Here, we're detecting red circles. The circular filter is represented by the green circle. Cross-correlation of the objects will result in the small circles "missing" the mark in a large area, despite being a circle

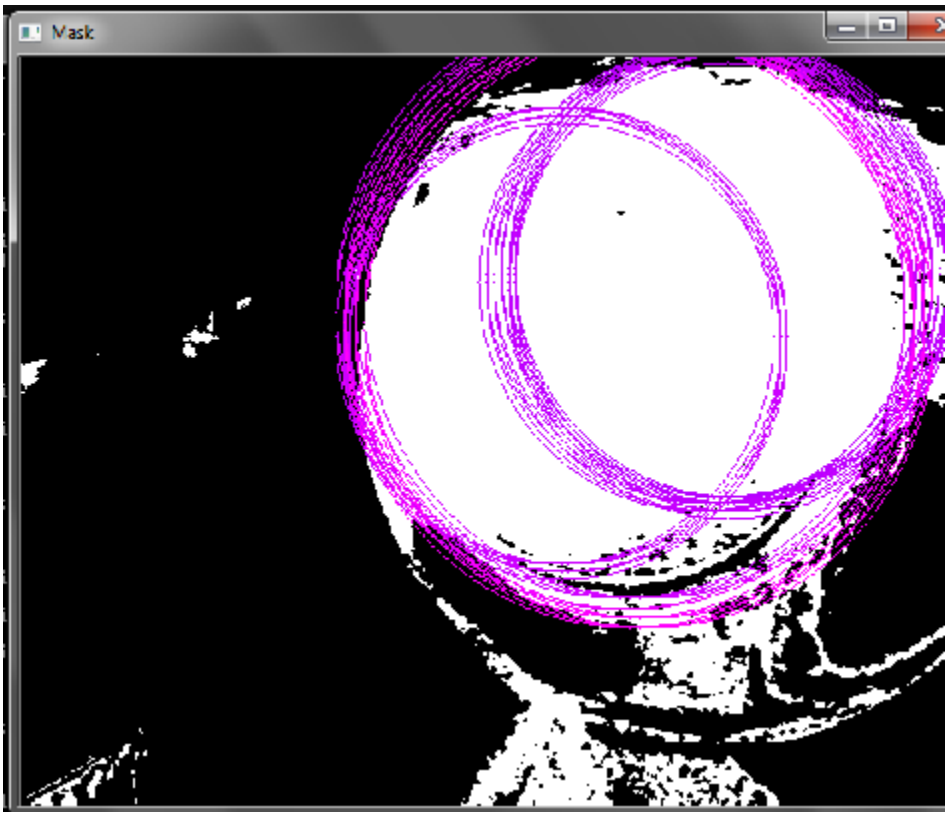
After we've scored every pixel's potential to be the object, we throw away all the pixels that have not passed `threshold` score. This allows us to get rid of some noise and false positives, but we will still be left with a lot of pixels that are definitely not circles.

2.5 Suppression

Now that we have detected a bunch of possible circles, we have yet another issue. Because we know we're not going to get an exact match with our filter, we've given a little leeway by setting our threshold to allow for, say, a 90% match. In our circle filter, a 90% match corresponds to a

threshhold of 230 if the detected circle is a little smaller than our filter. This causes something like this to occur:

Figure 4: Before Suppression



Note that there are a lot of overlapping circles because each pixel around the center is within the threshold. Thus, we find the local maximum of a given square window specified by `window_size`. Thus, all but one circle will be removed by the suppression.

However, there is one more issue. Technically, all but one circle of the same SCALE will be removed, since we detect at an individual scale. Therefore, in the above picture, the circles within the larger circle will disappear (except one). The circles that are of the larger size will all disappear (but one). The result will be two circles, with one lying inside the other.

The downside of suppression is that, with image pyramiding, it is insanely slow, relative to the rest of the code. Suppression is currently the bottleneck because it is called at every scale.

2.6 Total scale suppression

The problem as described in the previous section is solved splashing all the circles after the single-scale detect stage onto a new image. Then, a modified form of maximum suppression is run. We consider each circle (object) that we've detected. If there's another object that lies within it, we throw away the one with the smaller response. An alternative, if we can assume that one circle exists, is to run K-means clustering (which is extremely efficient because it's implemented in SciPy, which amongst some amazing algorithmc optimizations, can also take advantage of computer architecture). KMeans finds the centroid of all the detected circles, giving us an "average" which gives an approximate location of the circle. Unfortunately, I was unable to test this thoroughly because there seems to be a bug with my current version of SciPy.

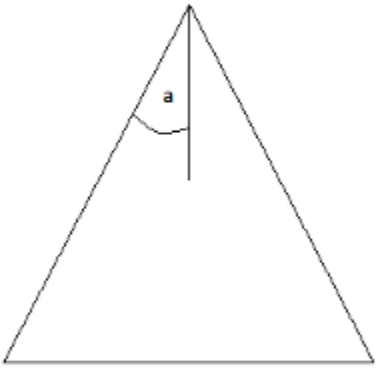
2.7 Sliding Scale Heuristic

The code as described above runs on my computer at about 1 frame per 1.5 seconds (and a little faster on the lab computers). This is pretty slow. However, using certain assumptions such as the constraint that there will only be one object per image, we can use a heuristic to dramatically increase speed. The intuition goes like this: If we know there's only one object, then we can stop pyramiding once we've detected it. However, due to noise and slight variations in movement, the scale may change a little. Thus, once we've found the object, we store the scale as the `best_scale`. The next time we run detect, we go immediately to the best scale and see if the object still exists there (and in the 2 neighboring scales one pyramid larger and one pyramid smaller). If it does, we can pyramid the scale exactly once and continue to do so. Using the 3 scales, if the object is moved very slowly back and forth, we can still benefit from the single pyramid speed improvement. However, once an object is lost, we have go through all the scales again to find it. Clearly, multiple objects will cause performance to suffer, hence the assumption that only one object will be detected at any given time. Also, we will not see any performance benefit if our object changes locations dramatically in terms of depth.

2.8 Triangle Detection

A simple triangle detector is also implemented, which does not detect the scale very well due to various reasons. A triangle is determined in my code by the angle (a) and height:

Figure 5: Triangle Filter



It is included as an example of how one would extend `MorphDetector`

3 Improvements

Again, beyond trying an entirely different method (which I highly recommend), there are some improvements that can be made to my code. In terms of run-time optimization, writing the non-maximum suppression in C is highly recommended. "Efficient Non-Maximum Suppression" by

Neubeck has some novel ideas of skipping large portions of the image, but from my experiments, python while loops are simply unusable if we want real-tie detection. Additionally, floating point arithmetic should be avoided when possible, because experimental data on my computer shows that they are much slower than using integer arithmetic. Finally, while I believe that running feature detection naively in Python will be no better than (and probably worse) than what I'm doing now (because it requires the entire image to be filtered and suppressed twice, and then cross-correlated with, whereas my code filters, cross-correlates and suppresses once), it is very likely that using OpenCVs feature detection code may yield tangible benefits due to it being written extremely efficiently in C++. However, this is one theory that hasn't been tested due to time constraints.

4 Related Work

- Robust Real-Time Detection of Multiple Balls on a Mobile Robot (Masselli, Hanten and Zell, 2013)
- A Survey of The Hough Transform (Illingworth and Kittler, 1998)
- Learning Methods for Generic Object Recognition with Invariance to Pose and Lighting (LeCun, 2004)