Using Corners to Match Images

Martin Desmery

Abstract

Due to their wide range of uses, corners are one of the most commonly tracked features in image processing. It can be used for anything from facial recognition to tracking AR objects. In this paper we will discuss the use of detected corners in two images to see if there are matching objects in both images. In order to achieve this, we will use some of the useful methods in the OpenCV library for C++.

General Terms

Feature Detection, Feature Matching

Keywords

Harris Corner, Feature, Computer Vision, Matching, Key Points

1. INTRODUCTION

Finding matching corner features between discrete images has many useful applications such as, facial recognition, motion tracking, object detection, and many more. There are also many different algorithms that can be used to detect corners. The three main algorithms are the Harris corner detector, SIFT and SUSAN. There is even a detector that also does image matching called SLAM [3]. For this paper we will be discussing the use of the Harris corner detector to match images, and implementing with the OpenCV library in C++.

2. HARRIS CORNER ALGORITHM

The Harris corner detection algorithm takes gray-scaled image data and finds edge features, then determines whether or not an area on the edge is a point of interest, or corner. An edge is determined to be a corner if there is a sharp change in direction of the edge. Since corners are seen as two dimensional structures, the sharper the angle of change, the easier the corner is to detect [4].

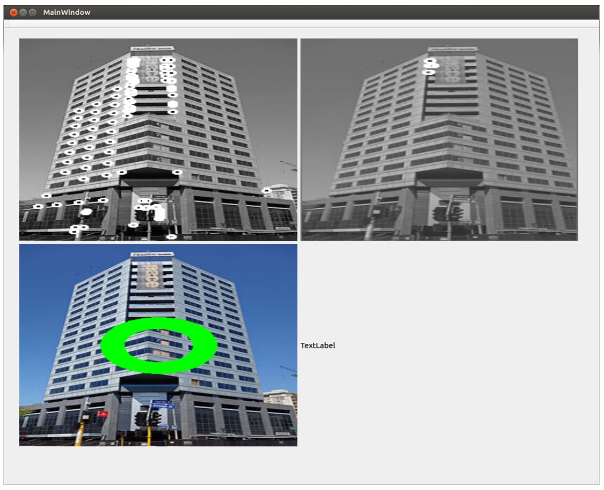

Figure 1: Successful Match

2.1 Corner Detection Math

The standard feature detection equation

$$E(u,v) = \sum_{x,y} w(x,y)[I(x+u, y+v) - I(x,y)]^2$$

is used by the Harris corner detector to find the features in an image. It then uses a unique score finding equation to determine whether the detected edges are corners.

$$R = det(M) - k(trace(M))^2$$

M is used to denote the $E(u,v)$ of the feature detection equation, in the unique scoring equation. If the score R is greater than the predetermined threshold value, than the edge region is determined to be a corner. This combination of the feature detection and scoring equations is how the OpenCV Harris corner function detects the corner features in an image[1].

2.2 Detecting Corners

Using the C++ OpenCV library in conjunction with Qt Creator, there are lots of useful methods for feature detection. By using the `cv::cornerHarris` with many other useful methods, corners can be detected more accurately. I created a `detect` method that utilized the Harris corner method. In my project I use the `cv::dilate` function in order to increase the accuracy of the corner detection. The `cv::dilate` method takes in the Mat of corners found by

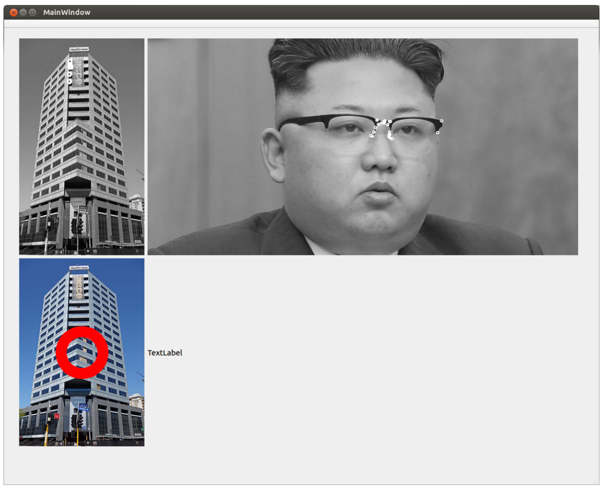

Figure 2: Unsuccessful Match

the `cv::cornerHarris`, and performs a dilation on it. The purpose of dilating the Mat is because the dilation enhances the images features. The `detect` method returns a Mat used in my `compare` method.

3. MATCHING

The best way to find matches with corners is to match by finding local features. This can be done by finding the distance between all of the detected features in both images and comparing them for similarities[5]. For this project I wrote a few methods for my `harris` class. In order to match the images I found useful methods and changed them in order to suit my purposes: a `detect` method, a `compare` method, and a `match` method[2].

3.1 Implementation

The `compare` method uses both the `detect` method and the `match` method. `compare` takes in the images to be compared, then converts them to gray-scale. After being converted to gray-scale, the images are individually passed to different `detect` methods in order to get a dilated Mat of detected corner features. Next, a `std::vector` of `cv::Points` are instantiated, then passed individually to two discrete `getCorners` functions that take the Mats of detected corner features from both images, and put their coordinates into the vector of points. Another function called `drawOnImage` is called that takes one of the images and the vector of points associated with it, and draws circles around the coordinates of the detected corners.[2] This can be seen in both Figure 1.(2.2), and Figure 2(3).

Now comes the actual matching, which requires the sizes of each image to be calculated. The vector of points from each of the images and their sizes are passed to the `match` function. The `match` method iterates through both vectors of point values, and calculates the distances between every point and adds them to a discrete array of floats for each set of points. It then iterates through the arrays and checks if there are common distances between the two of them. Every time a match is found it adds to a running total, if the total is greater than half of the smallest number of corners detected in one of the images, than the `match` function returns true. If it is less the half, than the function returns false.

Back in the `compare` function, the returned boolean from the `match` function is used to determine what color circle is drawn in the middle of the first image. If the function returned true, than the circle is drawn green, denoting the fact that the images contain a sufficient match. If the function returns false, than the circle is drawn in red, denoting that the images do not contain a sufficient number of matches. Figure 1(2.2). shows an example of a proper match using a building, and a slightly cut off on the right version of the same image. Figure 2(3). shows an example of a bad match, using the same building as before, and a picture of a very round faced man.

4. FUTURE WORK

My implementation of a corner matching algorithm is incredibly low tech. As a whole it is slow and inaccurate. It takes roughly an hour and a half to try to match two low quality images. In the future I would like to make my algorithm for matching more efficient, speeding up the computational process exponentially. A possible change would be to use a faster method of traversing the detected points when computing their distances. Right now the algorithm checks all of the computed distances and if more than half the distances found in the smaller of the two images are also found in the larger image's distances, they are considered a match. In the future I would only like to have it check necessary values, and only determine a match if all of them are found in both images. In an ideal setting, looking into the use of deep, machine learning to streamline the process would be the next step[3].

5. CONCLUSION

My algorithm does work, but incredibly slowly. With some work it can definitely be more effective. I was able to get results but unfortunately I am concerned as to the fact that because it takes so long, it pretty much unpresentable. Another difficulty with the timing of each instance of running the algorithm, is that it makes it hard to see what the limitations of what I have written is. The potential plans set aside for the future would be wholly necessary in order to check what exactly these limitations may be. For right now the only way to learn them would be to wait exorbitant amounts of time.

6. REFERENCES

[1] Harris corner detector, 2018.
[2] Robert Laganiere. *OpenCV 2 Computer Vision Application Programming Cookbook*. Packt Publ., Birmingham, 2011.
[3] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. Cambridge University. Cambridge University.
[4] Rick Szeliski. Notes on the Harris Detector, 2005.
[5] Vision Faculty UW CSE. Computer Vision: Feature detection and matching, 2009.