Computer Vision Assignment 2 - Report

Samuel Petit - 17333946 - petits@tcd.ie

1. Location of the Target

a. Using Edge Detection

I've decided to use Hough transform as the basis around my solution as it enables relatively straightforward detection of simple shapes such as lines or in this case circles. Let's go into the series of steps (or techniques) that I would use to implement target detection using edge detection, in this case using Hough transform.
Before going into edge detection, we should pre-process the image such as to prepare it for detection in the best possible conditions.

- Let's first convert the image to grayscale, in this situation we can ignore the colours and still be able to detect edges in a much more straightforward way.
    - This technique uses a colour image as an input, outputs a grayscale image. I use it in many places in this assignment, I will only explain in further details here how it works. There are many different kinds of conversions from different colour types to grayscale, however the most common is from RGB to grayscale, we use the following equation for each pixel point: $Y = 0.2126R + 0.7152G + 0.0722B$. This will effectively convert our 3 channel colour image into 1 channel grayscale.

- Adding blur: there are many different blurring methods we could use here however keep in mind that we need to preserve edges here as we will then proceed to do some edge detection. Given those restrictions and the blurring methods we've seen in class Median blur seems to be the most appropriate. It will help by reducing the noise in the image while preserving edges. We may choose to experiment with Gaussian blur too and see how the confusion matrices compare to choose which method to use, however it is likely that median blur will perform best here.
    - In this situation, the method takes a 1 channel image (gray scale image) as input, it outputs an image of the same size and type with the blurring applied. This method takes a kernel size parameter which determines the size of the kernel to use for blurring, in this scenario a relatively small size will be preferred, the goal is not to obtain a blurry image.
        - We can determine the ideal blur kernel size by running the program on a set of different sizes, examining the precision, recall and F1 metrics for each of the blur sizes, using cross-comparison we can then determine the best value to use. Let's use a size of 5x5 as a small, safe and uneducated guess.
        - Median blur is quite simple in concept, it will set the current pixel the median value of the considered area (in this situation a 5x5 range). On the other hand, gaussian blur will convolve our image using the provided range (5x5) with a Gaussian function.

We can now apply HoughTransform to our blurred image, in OpenCV it only suffices to call the method however I shall explain into a bit more detail what happens under the hood.

- Hough transform uses an accumulator per parameter to the shape that we are looking for in an image, for a circle we use 3 parameters: x,y the points for the center of a circle and r: the radius of the circle. (We need to include r as we have varying circle sizes).
- Given that we have a 3D accumulator, we need to pick an appropriate size for each of these 3 parameters, and then consider each edge point in the accumulator, increment the ones which match possible circle centers.
- We look for maximums in the accumulator.

Another thing to consider when using the HoughCircle method in OpenCV:

- In the first step, we detect the edges of an image using an Edge detection algorithm, in our case that will be Canny edge detection, this technique will take our grayscale image as input and return an edge map as the output array. It uses 2 threshold parameters which are used as part of the hypothesis thresholding which verify which edges are really edges, this works by setting anything above the high threshold to an edge, anything below to not be an edge. A value that lies in between will be set to an edge or non-edge based on its connectivity to edges that are valid edges.
- Canny is quite maths-heavy however in short it will compute the gradient and orientation of the image using its first derivative, then use the second derivative in order to remove non-maximas. We then execute the thresholding system as explained above in order to obtain our binary edge image.

Let's now see how we will using this method in order to solve our problem:

- We can now call the HoughCircle method on our image. It takes as an input our grayscale image, and outputs a set of found circles, where each circle includes 3 values: a center point (x and y position) as well as the radius.
- It uses a set of parameters, let's discuss how to set/use them:
    - Detection method: we will use CV_HOUGH_GRADIENT which is an implementation of OpenCV that is a bit more efficient than the standard 3D accumulator method.
    - A dp parameter which determines the size of the accumulator based on the image, it seems that a commonly used parameter here is 2 which will result in a kernel half the size of the provided image. We may be able to use a higher value (which will result in a smaller accumulator) as there are not many circles that we want to detect and they are fairly large in size. Once again we can experiment with changing this value and observing the changes on our confusion matrix such as to make an educated decision here.
    - A minimum distance parameter which sets the minimum distance between two detected circles, in our case we will set that to 0 given that our circles all start from the same center.
    - A threshold parameter, this is the highest threshold that is used in Canny edge detection, in our case I believe that we will be able to use a high threshold here given that the circles we are trying to detect are quite clear, by default it is 100 and the lower parameter is twice as low. Let's try a high value hee, of 200 and reduce it if needed by looking at the circles detected by our

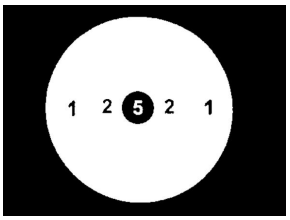
algorithm. (The OpenCV method will then set the lower threshold to be half of what we set here)
- A second threshold parameter that is passed to our accumulator threshold to determine if a circle is labelled as a circle or non circle. By default that value is 100 in OpenCV, I believe we may be ok here with taking the same approach, setting it to a high value such as 200 or 300 and decreasing as appropriate such as to find a system which reliably detects our circles, once again using our confusion matrix and performance metrics to guide our decisions.
- A minimum radius parameter, in this case we notice the smallest circle we want to detect is the circle representing a score of 5, it seems to be roughly smaller than 10% of the image, let's take a first attempt at using <Image width> * 0.07 such as to obtain 7% of the width. If we do not detect the smallest circle, we can attempt to reduce that value.
- A maximum radius parameter, in our case the maximum circle we want to detect is the 1 score area, it seems to take the majority of the image, let's use 90% of the image width as a rough approximation and similar to above we may increase it or decrease it depending on the accuracy of our detections.

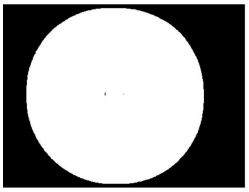
Now that we have this output, we should have a set of 3 circles (given that we have set our parameters correctly in the previous steps) which include their center point (x,y) and radius r. In the case that we have more than 3 circles, we can ignore any circle that does not have its center point near the center of our image and hopefully that will be enough to only keep the relevant circles here.
Finally, we can detect which circle corresponds to a score by computing their areas: for each circle we obtain its area by computing A = r^2 * π. The smallest area corresponds to the highest score (5) and the largest area corresponds to the lowest score (1).

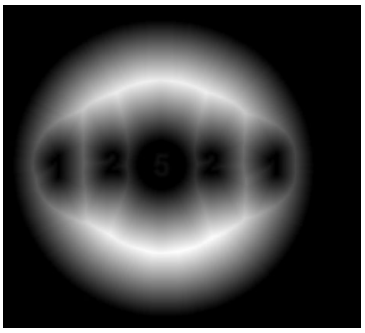
b. Using Region Detection

Let's start by removing the background of the image which we would like to ignore in this situation:

- We start by converting our image to grayscale as in this situation the use of colours will only complicate our method, the regions can be detected in grayscale space. To do so we use the conversion method from OpenCV which takes in an image and converts it to our desired output format, in this case a grayscale image, note that a more precise explanation of how this works was given in the previous section of this report.
- We apply Otsu thresholding to the grayscale image, this technique applies an adaptive thresholding where the threshold varies depending on the image's region (it uses Otsu's algorithm to determine the threshold). It takes as input our grayscale image in this situation, and outputs an image of the same dimensions and channels, however the only values in this grayscale image are now 0 or 255 (min or max values) as the threshold was applied. Since we are using Otsu here, there is no need for setting specific parameters.
    - I applied Otsu thresholding to our provided image out of curiosity to verify my approach, we obtain the following image:

- 

- We notice that this is quite good however there is a black section in the middle of the image which is not part of the background, same goes for the numbers that are still visible. We can apply some dilation on our image such as to get rid of these. Dilation takes as an input an image, in our case 1 channel binary image and outputs an image of the same kind (1 channel binary image). It uses a few parameters:
    - Kernel: the structure of the dilation element, in our case we can use a small circle (perhaps of radius 4 or 5 pixels), using one too big may expand our circle which would not be desirable
        - Dilation works by starting the provided kernel shape at the anchor point (here in the center) and scans all of the image using that kernel, it will set the maximal pixels overlapping the kernel to the max pixel value (white, true) - this will cause the white region to grow, and as part of that include the numbers from the center of the circle.
    - An anchor position: we can leave this to be the default value, which is the center of the image
    - Iterations: the number of iterations to use, I have experimented with this and found that 4 iterations gave us the result we wanted - no black bits remain within the circle. We obtain something like the following:
    - 
        - Note that this is using a rectangle 4x4 Kernel. We can also optimise this parameter using cross comparison of performance metrics over a range of kernel sizes and iterations amounts.
- We can now move on to obtain an image of the foreground to use as markers for our region detection algorithm, we do so by computing the distance transform of the image we obtained in the previous step. It computes the distance to the closest zero pixel in each of the pixels of the provided image. As input it will take our binary image and output a one channel image of the same size as the input where its intensity represents the distance to the closest non zero pixel.
    - We can provide parameters which will set how those distances are computed, for our purpose we can stick to classic values for the distance types and sizes. In order to obtain an accurate output, we can use an L2 distance type which will determine the algorithm used for calculations, and a mask of 5x5 such as to obtain accurate results.
    - Note that we may also normalise our results here in case we need them in a specific format (such as from 1 to 0).
    - We obtain the following image from this step:

- 

- We can then threshold this image, we will need to pick a threshold which keeps the structure of our different areas such as to detect all our circles (thus avoiding to lose relevant circle structure).
    - As mentioned above, threshold will take our single channel image as input and output a binary image. In order to set our threshold parameters we will need to experiment with values, taking into account what was mentioned above in order to keep the structure of our circles, I would expect a low threshold value here to work fairly well.

- We can now proceed to do some connected components processing, we can do so by using OpenCV's connected component method. This technique takes a single channel image as an input, and outputs a labelled image of the same size where each pixel is given a label which represents the region it is in. This method works by iterating over the rows of an image and essentially assigning new labels to some pixels where applicable, depending on the previous pixel labels, while keeping track of equivalent labels.
    - As for the input image, we obtain it by converting the thresholded image to an 8 bit image using the simple conversion method provided by opencv.
    - This method ignores the number within the target image given that they are zero-pixels in the previous steps

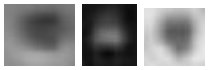
We obtain a labelled image which we can use as markers for our final step: the watershed method.

- In OpenCV we can call the watershed method which takes as input: our original colour (3 channel) image and the markers we have computed in the previous step. It returns a single channel image map of markers which are used to split the different sections of the image detected by the algorithm.
- The watershed algorithm can be implemented in different ways, put simply it will find the minimas of the image and label those as different regions, then expand regions from those minimas. We then obtain a separation line when regions meet. We often get many regions here so it is up to us to provide this algorithm with reliable marking from the previous step in order for it to word reliably.
    - From the output separation lines of the watershed algorithm, we can draw the contours of regions as such:
    - 

Given that we have the edge of circles. We can execute the same approach as discussed previously in order to detect which area belongs to what score. For each separation line we compute its area and order them by size. The smaller size is assigned to the highest score.

2. Shot detection and Scoring

Given that this is a static camera, we can run one of the above algorithms (whichever one has the best accuracy metrics) and know the positions of the regions, this will not be useful straight away however it will come in useful later in this algorithm.
So, to start detecting shots on the target, we start by running the target detection algorithm from step 1 on the first frame (assuming it is not obscured), we can then store the output for future use.
Let's now go into detecting shots on the target. I will be making use of template matching for my solution.

- The first step is to convert our image to grayscale, as we've seen previously, this step is quite straightforward and takes as input the current frame - a multiple channel colour input image and outputs a 1 channel grayscale image.
    - This will make our processing easier as we do not need colour in order to detect shots on this target.
    - Notice the change in hue in the provided image, one is blue-ish and the other one more yellow, grayscale will help to deal with this issue as well.
- We can then start to apply our template matching. In order to do so, I extract a section of a shot in grayscale:
    - 
    - These are not ideal and come from screenshots, we can obtain better images in a situation where we need to implement this. Notice that I also have added 3 different screenshots, one per background color / section of the target.
- We can then execute template matching for our template on the current frame, in OpenCV that is using the matchTemplate method. At its core it will try to match our template in every position in the image and evaluate a match criterion. It takes as input the current image (grayscale - 1 channel) and our template (also grayscale - 1 channel) and will output a map of comparison results.
    - We need to decide which template matching method to use, we may start by comparing the performance of the ones we've seen in class, including: Cross correlation, normalised cross correlation, normalised squared difference and squared difference. We can pick the method which yields the best confusion matrix metrics.
- We can then extract from the map the matching values with are higher than a set threshold. This will result in us obtaining a set of matching values which enable us to locate matches within our original image.
    - Once again here we need to set a threshold. We can start with an approximate value and by testing a large set of different values, use cross comparison on the performance metrics in order to obtain the best possible threshold for our application.

When we get here, we should have a set of locations in which we found a match with our template all we need to do is calculate the score to do so:

- Iterate over the matches, we should have a set of points (x,y) representing the top left point of our match
    - we can obtain the center of the shot by computing:
        - Center x = match x + (template width / 2)
        - Center y = match y + (template height / 2)
    - Map the position to a point amount, we can do the following
        - If the center point belongs in the score 1 area : continue the algorithm. If it does not mark the point as 0
        - If the center point belongs in the score 2 area continue the algorithm, if no mark the point as 1
        - If the center point belongs in the score 5 area mark the score as 5, if not mark is as 2
            - We can check mathematically if a point (x,y) belongs inside a circle using the following equation:
                - **(x - center_x)^2 + (y - center_y)^2 < radius^2**
    - Sum these values in order to obtain the total score

A concern worth noting is the detection of shots that are between two score points, it is possible that our algorithm may detect them already as is however if we find that it struggles to find the reliably we may include more templates of a shot between scores 1 and 2 as well as between 2 and 5 such as to detect those shots more reliably.

- This may require us to change some of the parameters (especially thresholds) to account for these new templates - we will not want to have multiple templates detect the same shot.

In terms of scoring those shots when they are detected I believe that my approach already covers such scenarios as it will take the center of the shot as a reference point for scoring the shot.

Another concern is that adding many templates may make our algorithm inefficient given that template matching attempts to match every template in every possible location. An idea to solve this could be to reduce the search area per template, for example when considering the template with gray background, only search in the 1 score area and so on. We may also ignore the shots outside of the target which would make it more efficient.

- This would require us to write some custom template matching as basic methods in OpenCV do not seem to enable us to do that directly.

However performance is not marked as critical in this assignment so I believe that my current approach is good enough given the assignment.