Universiteit Leiden
The Netherlands

Computer Science & Advanced Data Analytics

Sequential Recognition And Scoring Of Archery Shots

Raymond Parag

Supervisors:

Dr. A. J. Knobbe & Drs. M. Meeng

MASTER THESIS

Leiden Institute of Advanced Computer Science (LIACS)
`www.liacs.leidenuniv.nl` August 4, 2017

Abstract

In this master thesis, computer vision is applied to archery in order to recognize and score archery shots in realtime or videos. This research is initiated by the NHB, which is the Dutch archery federation, for the reason that currently scores of arrows are filled in manually. This creates distraction from the primary training process. Also the order of the arrows which are shot, is not taken into account, which could be useful to analyze. For this purpose, various methods and techniques using OpenCV with Java are discussed and applied to detect and score the arrows. The target is detected by a color-based approach. The perspective of the target is corrected by a homography transformation matrix and arrows are detected and scored using the Hough Lines Transform in combination with a point in contour test. The scoring system is tested on videos, which are taken from the archery training at the NHB.

Acknowledgements

In the first place I would like to thank my supervisors, Arno Knobbe and Marvin Meeng, for providing amazing support on the project. At each milestone, their critical feedback kept me sharp. I want to thank the researchers, Dirk Meijer and Arie-Willem de Leeuw, for giving me useful tips during the project. A big thanks to the NHB for initiating this project and for giving us the opportunity to take photos and videos during the practice. Without the NHB, this project did not exist at all. At last, I want to thank my family and friends for the mental support. During times when there was a lot of pressure, they managed to cheer me up and relax.

Contents

Chapter 1

Introduction

Nowadays computer vision plays an important role in different areas. Some of these areas include traffic, sports, security systems, cars, etc. Archery is one of those sports where computer vision in the form of recognition and scoring of shots could be very useful.

The NHB (Nederlandse Handboog Bond) is the Dutch archery federation that organizes training and big competitions for amateur and also professional archers [1]. The NHB uses archery targets that consist of differently colored circles, which indicate a different score (Image 1.1). At the NHB, they currently are filling in the scores of arrows manually through an application. The archer or assistant has to fill in each score after every round with the consequence that it creates distraction from the primary trainingsprocess. Because the score is manually entered after every round (after six arrows), the order of the arrows is not taken into account, which could be useful to analyze.

The NHB therefore wants to develop a scoring system, that is able to score the arrows that are shot into the target by using a camera. This scoring system needs to be mobile, relatively cheap and robust. The scoring system should be able to push the scores and location of each arrow to an online database. On this database various historical analyses can be run. From these analyses, development of potential talent might be detected and it can be used to improve the skills of the archers.

Current solutions that are on the market for archery are the Falco-eye System and Scatt Shooter Training System.

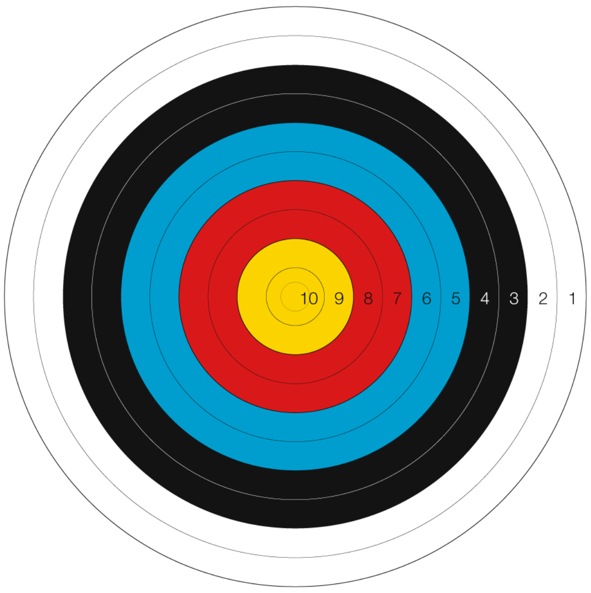
- The Falco-eye System [2] consists of a big electronic target on which the arrows are shot. This system is pretty accurate, but unfortunately it is quite expensive and far from being mobile (86 kg). Such a system is only suitable for big competitions, but less for amateur competitions.

- The Scatt Shooter Training System [3] was primary developed for guns to measure the stability during aiming. By placing the sensor on the bow, the aiming procedure can be analyzed. Unfortunately in the archery sport it is not possible to determine the score just by analyzing the aiming procedure, because there are various factors during arrow release that can influence the flight of the arrow. This solution is therefore only suitable before shooting the arrow. Besides that, it is also difficult to apply and quite expensive.

Both of these systems, as told earlier, are not suitable for the NHB. They are either too expensive or not mobile at all.

This master thesis focuses on the development of such a scoring system with the use of computer vision library OpenCV [4] and programming language Java [5]. This master thesis is a continuation of the research project. In the research project, the focus was on exploring OpenCV possibilities in scoring arrow shots based on images. The master thesis will build further on that, but instead on images, it will focus on video entirely. The aim is to use and improve the techniques that are described in the research project, to create a start in the scoring system that is proposed by the NHB.

Figure 1.1: Archery target.



Chapter 2

Related work

Most of the articles in this research area are based on images only, such as Nguyen & Lin [6] and Zin et al. [7]. The article however, that is the closest to what we want to achieve is written by Danielescu [8]. Danielescu introduces onTarget, which is an electronic scoring system that is able to score each player's shots. It keeps track of the player's scores and shows the rankings to the spectators. The set-up of onTarget consists of two cameras that are connected to a frame around the target. The cameras are about 4" in front of the target. Besides the hardware, Danielscu uses OpenCV in combination with C++ for the arrow scoring. Various computer vision techniques, such as ellipse fitting, edge detection, erosion, dilation, are used to find the arrow.

For the recognition of the target, there are several assumptions that are made. First, it is assumed that the target has a white background, so that it stands out. Second, it is assumed that there will be either a difference between the colors of the rings or edges between the rings that differentiate them. Third, it is assumed that there always will be an X in the center of the target.

To begin, onTarget asks the users to calibrate the cameras by clicking on the center of the target on screen. OnTarget uses edge detection to find the rings of the target and therefore the regions that are scored differently. After some noise removal methods, onTarget applies ellipse fitting to create a mask. This mask is then used to ignore anything except what is inside the mask. The location of the arrow is found by computing the difference between the previous and current frames based on a threshold value. To find the location of the arrowhead, onTarget uses a vertical rectangular structuring element to extract the arrow from the surrounding area. The scoring is based on the ring on which the arrowhead is shot. For each ring a mask is created. It is checked whether the arrowhead is inside or outside each mask. The score is then added to the player's score.

The similarities between Danielescu and our approach are the usage of ellipse fitting and creating a mask. Ellipse fitting is used to find an overall mask of the target. This mask is then used to ignore anything except what is inside the mask. The difference between two consecutive frames is also used in this master thesis to detect change, just like in the article. To score the arrow, Danielescu uses masks of each ring. In this master thesis the same principle is used, but instead of using masks that are found with edge detection, masks are used that are found using color segmentation of the blue, red and yellow rings.

Besides the similarities, there are also notable differences between Danielescu and our approach. In Danielescu, the cameras are placed on specific distances from the target. This will create images that are the same in terms of position. In this master thesis this is not the case, the camera can be placed freely, but it should not be perpendicular to the target. Instead of using two cameras, only one is used. In our approach, calibrating the camera does not require human interaction. In this master thesis, the only two assumptions are that the camera should not be placed perpendicular to the target and the target contains the blue, red and yellow rings.

Chapter 3

Dataset

For this master thesis project, videos and images are used, which are taken from the archery training at the NHB. A snapshot of one of these videos can be seen in Image 6.2. From these videos, a set of 'noise' videos are created in which noise is induced through video editing software. These 'noise' videos in combination with the videos described above, represent the training set on which the parameters of the methods are adjusted to. The total training dataset consists of six videos and 14 images, which can be found at [9].

Chapter 4

Detection of the target

The target that needs to be detected, can sometimes occur in a noisy area, such as Image 6.2. In order to cope with such noise around the target, features should be chosen that represent the target well. One could think of two features. The circles and the color of the circles.

4.1 Hough Circle Transform

The Hough Circle Transform is a technique in computer vision for detecting circular objects in a digital image. As can be derived from the name, it is a variant of the Hough Transform. The Hough Transform can be used to determine the parameters of a circle when a number of points that fall on the perimeter are known [10]. The number of adjustable parameters this Hough Circle Transform in OpenCV takes, are six.

The Hough Circle Transform has a number of disadvantages. Besides that it requires a lot of time tweaking the six parameters, the parameters that are eventually chosen will not work in every scenario, since they are prone to change. The way in which the NHB places the camera at different positions, the Hough Circle Transform will not work with the same parameters. Another disadvantage is that the Hough Circle Transform only works well with circles. In this case the target could appear as an ellipse, because of the positions of the camera. Therefore the Hough Circle Transform is not suitable in this case.

It should be noted that Hough Circle Transform works well in some cases, especially when the archery target is perfectly circular. Such a case, where the Hough Circle Transform works well, can be seen in the paper of Nguyen & Lin.

4.2 Color segmentation

The colors of the circles are white, black, blue, red and yellow, starting from the most outer circle to the center (Image 1.1). These colors always remain the same on every target. The intensity of these colors however can vary a bit, depending on the weather condition. Besides that, this seems to be the most convenient feature of the target. The focus here will be the blue, red and yellow colors. The black and white colors are not reliable enough to consider, since the color range of black and white could interfere with the color range of blue, red and yellow. The general procedure to find the target based on color is as follows:

1. Define color range for each color.
2. Morphological closing (to remove small noise).
3. Find contours of each color.
4. Draw convexhull around it or fit an ellipse into it.

4.2.1 Color range specification

In order to detect color, a color range for each color has to be specified. In OpenCV, detection based on color uses the HSV color space. Color conversion from an RGB image to an HSV image happens through the usage of `Imgproc.cvtColor()`. The HSV color range in OpenCV varies from the normal HSV color range, which is from 0 to 360 for hue and 0 to 100 for saturation and value. The HSV color ranges used in this master thesis, are chosen to represent a broad spectrum of the original colors. These HSV color ranges are based on color palettes of blue (Image 4.1), red (Image 4.2) and yellow (Image 4.3). This way, even light or dark versions of blue, red and yellow will be detected. The HSV color ranges are as follows:

- Blue: hue range [90, 125], saturation range [100, 255] and value range [130, 255].
- Lower Red: hue range [0, 10], saturation range [100, 255] and value range [130, 255].
- Upper Red: hue range [160, 179], saturation range [100, 255] and value range [130, 255].
- Yellow: hue range [20, 30], saturation range [30, 255] and value range [180, 255].

The colors are segmented from the image by using the `Core.inRange()` function, which creates a binary image where pixels which fall in the defined range are true and all other pixels are false. Color segmenting the color palettes results in images 4.4, 4.5 and 4.6. These show that a broad spectrum of the colors in the blue, red and yellow palettes are detected. The colors that are not detected are either too bright or too dark to such an extend that it does not represent the color well anymore. An example of color segmentation on a target can be seen in Images 4.7, 4.8 and 4.9, where Image 4.10 is the original image.

4.2.2 Morphological closing

Morphological closing is necessary to remove any noise that is visible in the blue, red and yellow area. Morphological closing makes finding contours later on more accurate. Morphological closing is a dilation followed by the erosion of the result. This will fuse narrow breaks and long thin gulfs, eliminate small holes

Figure 4.1: Blue palette.

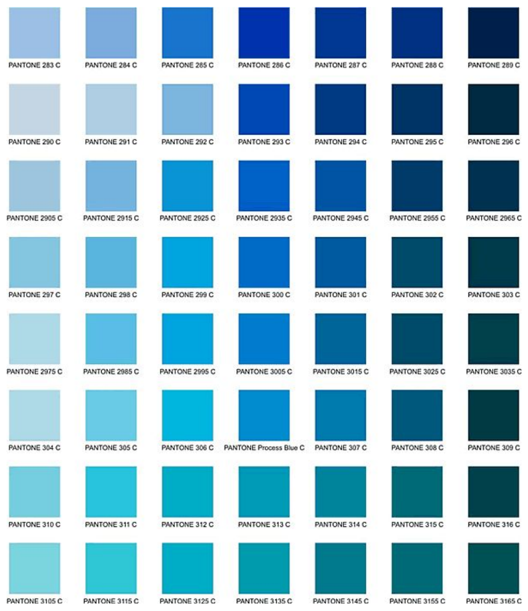


Figure 4.2: Red palette.

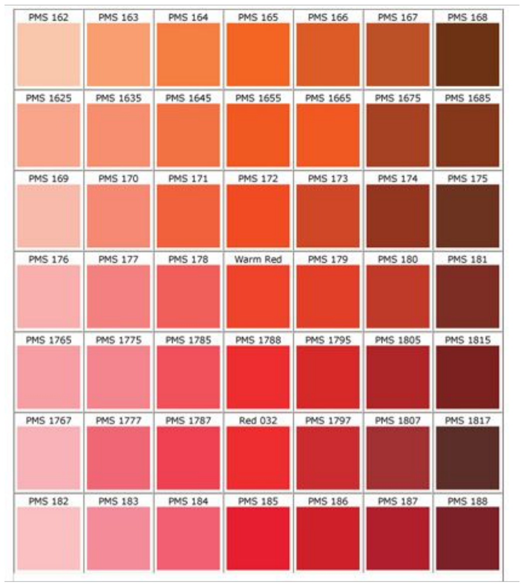


Figure 4.3: Yellow palette.

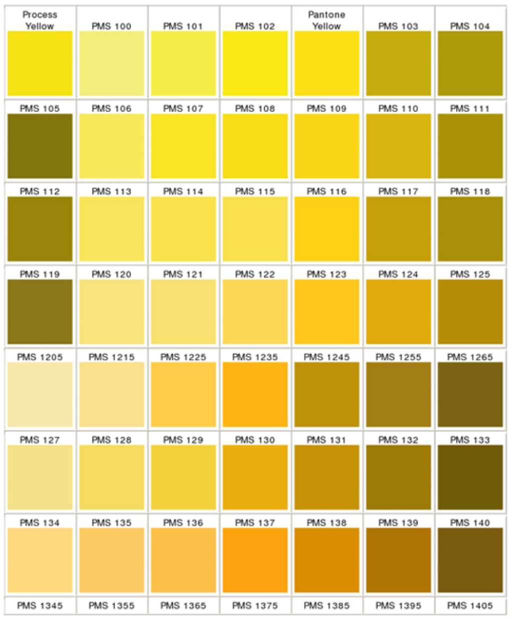


Figure 4.4: Blue spectrum.

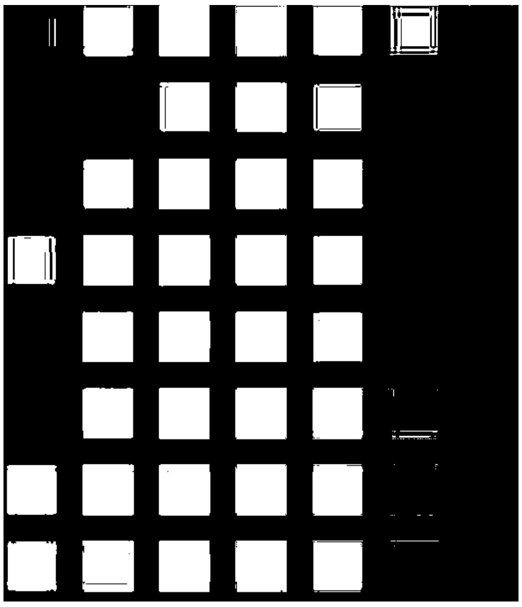


Figure 4.5: Red spectrum.

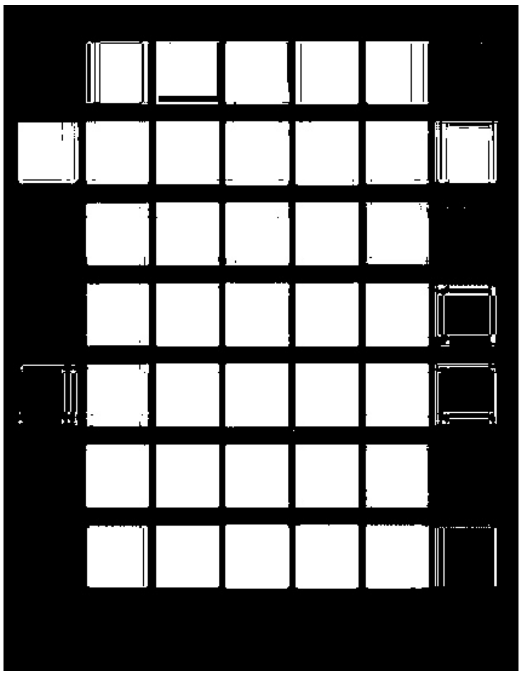


Figure 4.6: Yellow spectrum.

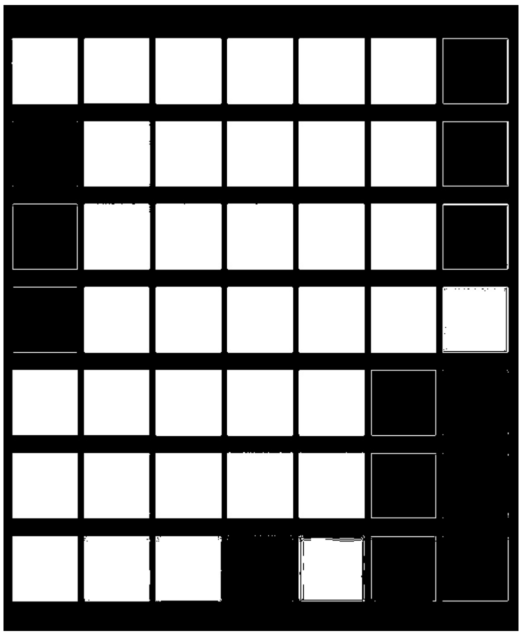


Figure 4.7: Blue segmentation.

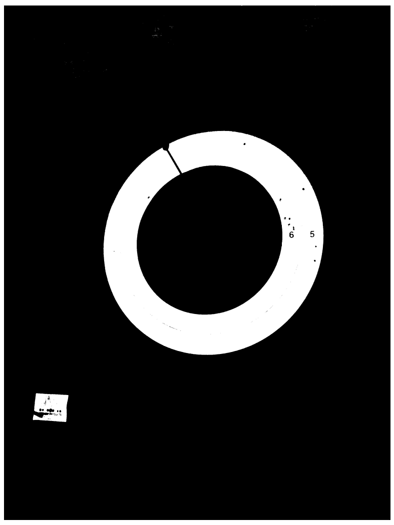


Figure 4.8: Red segmentation.

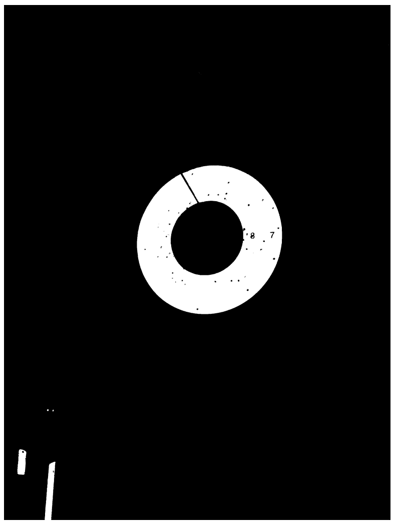
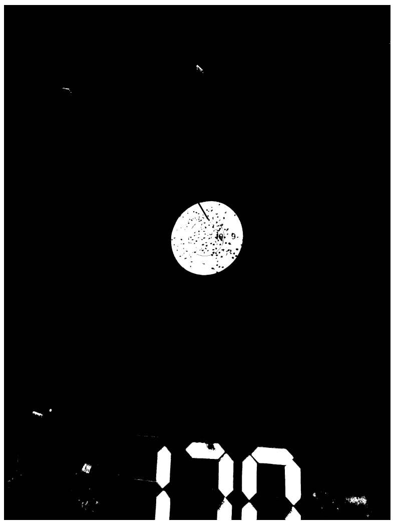


Figure 4.9: Yellow segmentation.



and fill gaps in the contour [11]. The function `Imgproc.getStructuringElement()` can be used to create an elliptical kernel `Imgproc.MORPH_ELLIPSE` with a specified size. Consider the yellow segmented image (Image 4.9). This will look as follows (Image 4.11) after applying morphological closing with an elliptical kernel size of 25x25. This kernel size is the result of a compromise between speed and effectiveness, since increasing this kernel size will result in slow morphological closing of the image. A kernel size of 25x25 is acceptable in this case, since it reduces most noise in a short amount of time.

4.2.3 Finding contours of each color

Finding the contours is the process that makes it possible to use the detected colored rings to derive information from. Without contour finding, the detected colored rings are useless. Finding the contour of each color uses the `Imgproc.findContours()` function on a binary image. What this function does, is that it saves the contour of every non-interrupted area with value true from the binary image into a list. These are visible as white parts in the binary image. Contour finding has to be implemented careful, since there could be also other objects visible around the target that are in the same color range as the rings. Such an example can be seen in Image 4.11, where the yellow score is also in the same color range as the yellow center.

Fortunately, the objects around and in the target can be filtered by size, by shape, but also by position. The initial filtering would use the `Imgproc.approxPolyDP()` function, which approximates a shape with a specified precision, which is epsilon. The epsilon parameter controls the approximation accuracy, which is the maximal distance between the original shape and its approximation. An epsilon of $0.04 \cdot$ `perimeter` works well for saving the amount of vertices a contour has. This amount can determine what shape the contour is. For example, if the contour has four vertices, this could indicate that the shape is a square, rectangle or any other shape which has four vertices. For five vertices, the shape could be a pentagon and for more than six vertices, this could indicate that the shape is close to an ellipse or circle.

The size filtering is possible by the usage of the `Imgproc.contourArea()` function. This function returns the size of the contour. The position of the contour could also provide essential information to consider it a colored ring or not. This position measure (`Imgproc.pointPolygonTest()`) works by checking if a point of the given contour is inside another contour. For the red and the yellow ring this is useful, since they are inside the blue ring.

To summarize, for the blue contour a combination of `approxPolyDP()` and `contourArea()` is useful, since the `approxPolyDP()` function filters out non-circular objects and the `contourArea()` function makes it possible to choose the biggest circular object that is left, which is the blue contour. For the red and yellow contours, the `pointPolygonTest()` in combination with `contourArea()` is enough to correctly detect the red and yellow ring. The `pointPolygonTest()` makes sure that the found contour is inside the blue contour and the `contourArea()` assures that it is the biggest found contour. Image 4.12 shows an example of the detected contours for blue, red and yellow after applying shape filtering, size filtering, position filtering and ellipse fitting.

Figure 4.10: Original image 1.

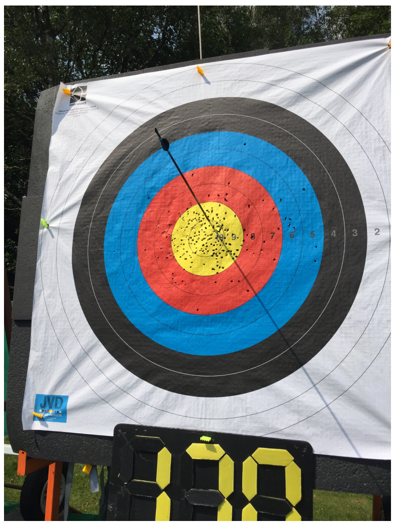
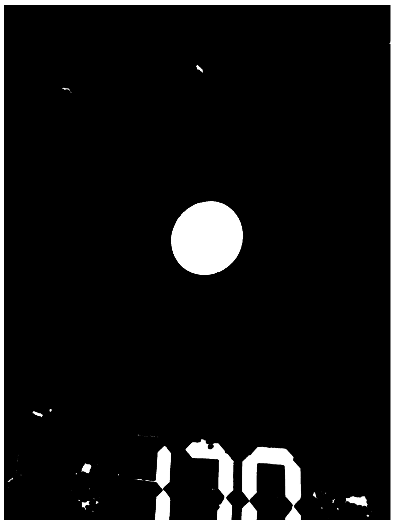


Figure 4.11: Yellow segmentation after closing.



4.2.4 Drawing contours

The detected contours are mostly not well-defined using the standard `Imgproc.drawContours()` function to draw them. It is recommended and more robust to fit an ellipse into the contour by using the `Imgproc.fitEllipse()` function. An alternative is to draw a convex hull around the contour by using the `Imgproc.convexHull()` function. This has the same effect as the `fitEllipse()` function. The convex hull would be the better choice if the target was heavily deformed and not elliptic at all. In this master thesis, the `fitEllipse()` function is used because it requires far less code. The difference between `drawContours()` and `fitEllipse()` can be seen in Image 4.13 for the blue ring.

4.3 Pseudocode target detection

The pseudocodes, 1, 2 and 3 are implemented to accomplish the target detection on video. This target detection has to be applied only once in the initial phase. Once the camera is set, the camera remains at the same position.

Algorithm 1 Blue ring detection.

```
 1: Change color of the frame to HSV color
 2: Segment the blue color with the specified color range
 3: Remove noise with morphological closing
 4: Find all blue contours within the frame
 5: for each found contour do
 6:     if the contour is circular and the contour is the biggest then
 7:         Save the contour size
 8:         Save the contour
 9:     end if
10: end for
11: if the chosen contour is not empty then
12:     Fit an ellipse into the contour
13:     Save the fitted ellipse to a contour
14:     Return the contour
15: else
16:     Return the empty contour
17: end if
```

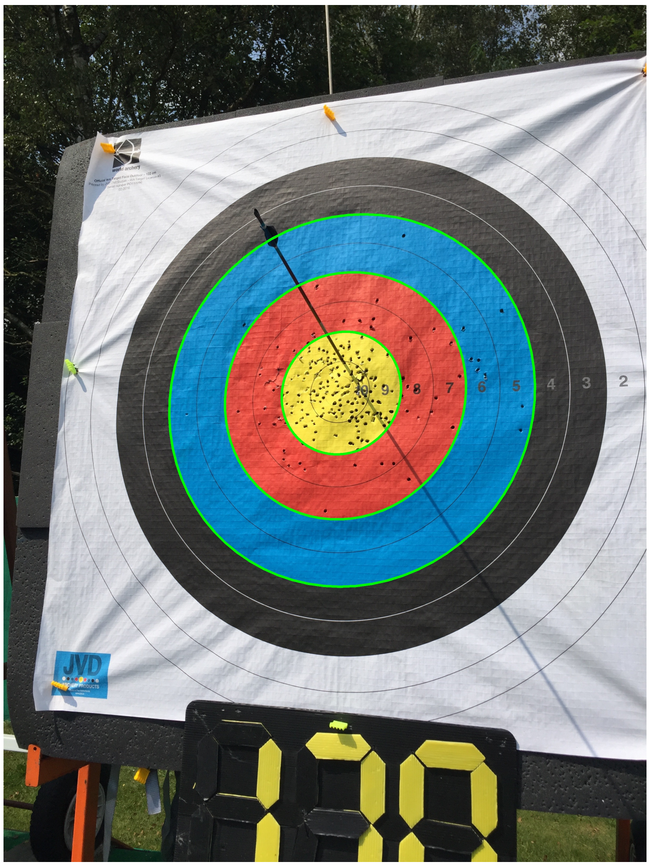


Figure 4.12: Detected contours after shape filtering, size filtering, position filtering and ellipse fitting.

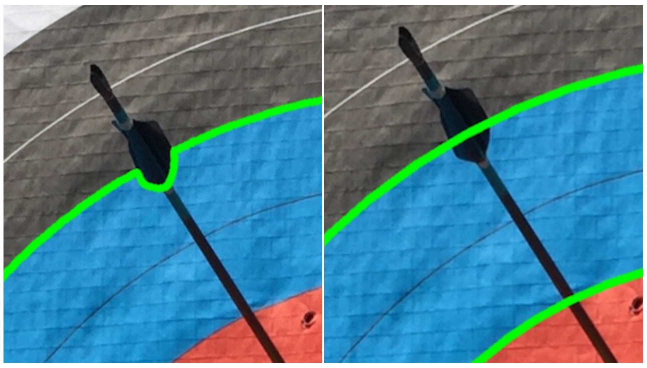


Figure 4.13: `drawContours()` vs `fitEllipse()`.

Algorithm 2 Red ring detection.

```
 1: Change the color of the frame to HSV color
 2: Segment the red color with the specified lower and upper color range
 3: Remove noise with morphological closing
 4: Find all red contours within the frame
 5: for each found contour do
 6:     if a point of the contour is inside the blue contour and the contour is the biggest then
 7:         Save the contour size
 8:         Save the contour
 9:     end if
10: end for
11: if the chosen contour is not empty then
12:     Fit an ellipse into the contour
13:     Save the fitted ellipse to a contour
14:     Return the contour
15: else
16:     Return the empty contour
17: end if
```

Algorithm 3 Yellow ring detection.

```
 1: Change the color of the frame to HSV
 2: Segment the yellow color with the specified color range
 3: Remove noise with morphological closing
 4: Find all yellow contours within the frame
 5: for each found contour do
 6:     if a point of the contour is inside the red contour and the contour is the biggest then
 7:         Save the contour size
 8:         Save the contour
 9:     end if
10: end for
11: if the chosen contour is not empty then
12:     Fit an ellipse into the contour
13:     Save the fitted ellipse to a contour
14:     Return the contour
15: else
16:     Return the empty contour
17: end if
```

Chapter 5

Perspective correction

This chapter focuses on all the aspects of calculating the perspective correction of the target. As already known, the camera can be placed almost freely across the field. By placing the camera very close with a short angle towards the target, it could visualize the target as an ellipse. During the archery competition/training, the visitors also look at the score screen and therefore it is necessary to correct the perspective of the ellipse to a more appropriate frontal view. The ideal situation would be to correct the ellipse into a perfect circle. This however is not always the case, since there are various factors that contribute to creating the 'perfect' circle from an ellipse. The perspective correction in this chapter is purely based on OpenCV and its functions.

The perspective correction works according to the following steps:

1. Retrieve the contours of the three colors: blue, red and yellow.
2. Calculate the destination circle for each retrieved contour.
3. Create the transformation matrix and warp the perspective according to it.
4. Verify the perspective and improve it by rotation.

5.1 Calculate the destination circles

Retrieving the contours of the three colors has already been discussed in Chapter 4. Those contours of the three colors are the source ellipses, which are detected (example Image 5.1). They need to be converted to preferably circles. In Section 5.2, a method is described that needs the number of source pixels of the ellipses and the number of destination pixels of the circles to be equal. Based on that it calculates the appropriate transformation matrix. Therefore the calculation of the destination circles is important, to assure that the amount of pixels of the destination circles are equal to the source ellipses.

The calculation of the destination circles are based on the amount of pixels the source ellipses have. In OpenCV, the function `Imgproc.circle()` is used to draw a circle with a given radius. This radius determines the number of pixels and thus the size the circle has. There is unfortunately no known formula in OpenCV to calculate the number of pixels given the radius of the circle, because each increment in radius results in the increase of the number of pixels by an inconsistent number. Therefore three simple approximation formulas are created by trial-and-error that approximate the radius of the destination circles given the number of pixels the source ellipses have. Drawing destination circles with these estimated radiuses results in an amount of pixels that are equal or almost equal to the amount of pixels the source ellipses have. These formulas like said earlier are created by trial-and-error, which means that for each image and video in the training dataset, the optimal destination circle radiuses are determined and the amount of pixels of those radiuses are noted. From these radiuses and amount of pixels, the following radius approximation formulas are created:

$$e\_blue = \frac{741 \cdot s\_blue}{4187} \tag{5.1}$$

Where *e_blue* is the estimated radius of the blue destination circle and *s_blue* is the number of pixels the blue source ellipse has.

$$e\_red = \frac{492 \cdot s\_red}{2777} \tag{5.2}$$

Where *e_red* is the estimated radius of the red destination circle and *s_red* is the number of pixels the red source ellipse has.

$$e\_yellow = \frac{244 \cdot s\_yellow}{1375} \tag{5.3}$$

Where *e_yellow* is the estimated radius of the yellow destination circle and *s_yellow* is the number of pixels the yellow source ellipse has.

These approximation formulas simply approximate, because in some cases the number of pixels cannot be perfectly equal to the source ellipses using the estimated radiuses of these formulas. In those cases, the number of pixels in the destination circles is either too big or too small compared to the source ellipses. Therefore bigger radiuses are chosen for the destination circles which is near the source ellipses in amount of pixels. The difference in number of pixels between the source ellipses and destination circles are subtracted from the destination circles in order to equal the number of pixels. Algorithms 4, 5 and 6 describes this process for each color. Image 5.2 shows an example of the destination circles, which are created from the source ellipses from Image 5.1. In these images, the amount of pixels are equal to each other.

5.2 Create the transformation matrix and warp the perspective

A transformation matrix is a special matrix that can describe 2D and 3D transformations [12]. They are frequently used in linear algebra and computer graphics, since transformations can be easily represented,

Figure 5.1: Source ellipses.

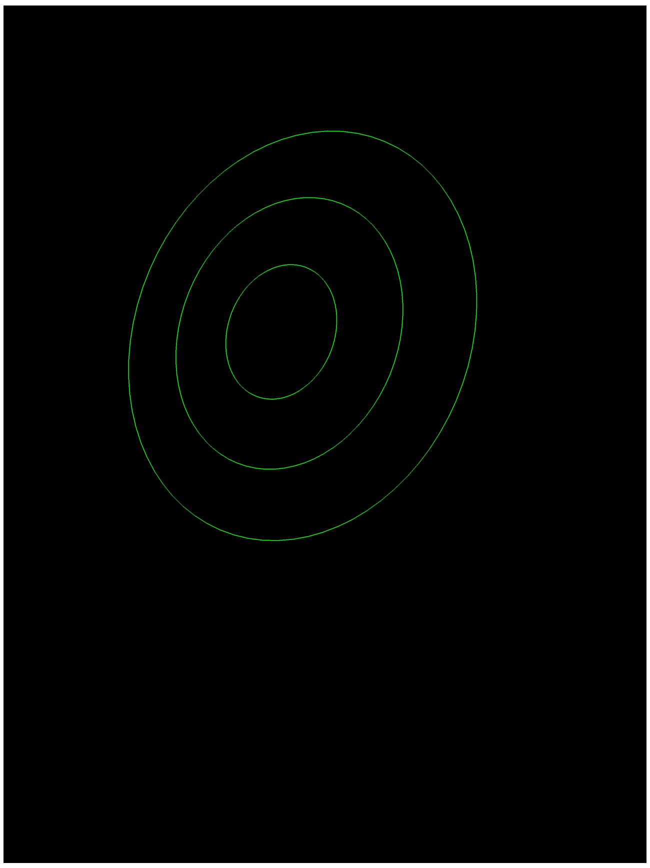
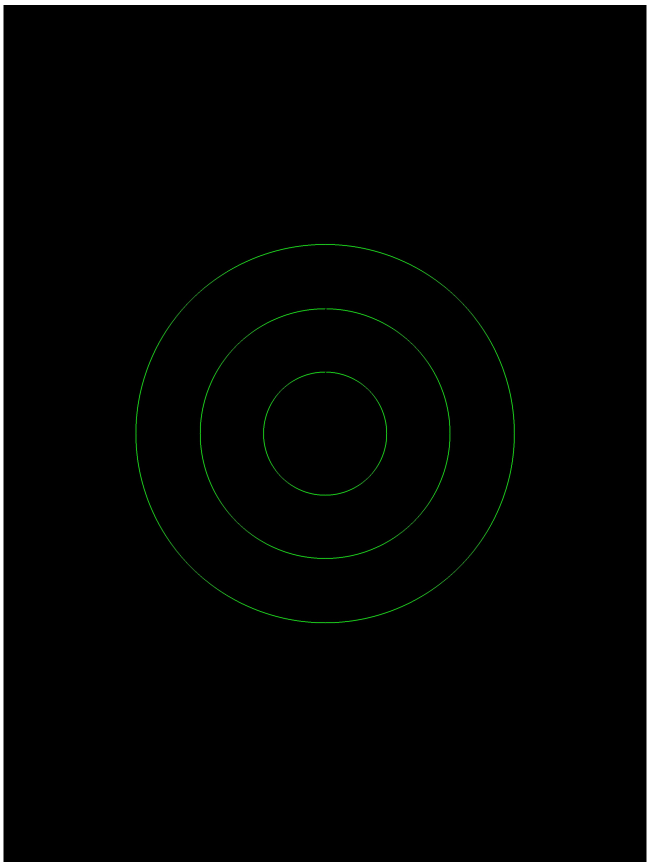


Figure 5.2: Destination circles.



Algorithm 4 Blue destination circle calculation.

```
1: $a$ = estimated radius of the destination circle using formula (5.1)
2: for $i$ = $a+1$ to $a-1$ do
3:     Draw circle with radius $i$ on the same center as the blue source ellipse
4:     Find contours to retrieve the drawed circle
5:     Calculate difference in the number of pixels between destination circle and the blue source ellipse
6:     if difference is smallest and $\geq 0$ then
7:         Save current smallest difference
8:         Save the contour
9:         if difference $== 0$ || $i == a-1$ then
10:            Save the pixels of the contour minus the difference (so that the number of pixels is equal)
11:            Break out of all loops
12:        end if
13:    else
14:        if The saved contour is empty then
15:            $i = i + 2$
16:        else
17:            Save the pixels of the contour minus the current smallest difference (so that the number of pixels is equal)
18:            Break out of all loops
19:        end if
20:    end if
21: end for
22: Return the self-made Mat, which contains the saved pixels
```

Algorithm 5 Red destination circle calculation.

```
1: $a$ = estimated radius of the destination circle using formula (5.2)
2: for $i$ = $a+1$ to $a-1$ do
3:     Draw circle with radius $i$ on the same center as the red source ellipse
4:     Find contours to retrieve the drawed circle
5:     Calculate difference in the number of pixels between destination circle and the red source ellipse
6:     if difference is smallest and $\geq 0$ then
7:         Save current smallest difference
8:         Save the contour
9:         if difference $== 0$ || $i == a-1$ then
10:            Save the pixels of the contour minus the difference (so that the number of pixels is equal)
11:            Break out of all loops
12:        end if
13:    else
14:        if The saved contour is empty then
15:            $i = i + 2$
16:        else
17:            Save the pixels of the contour minus the current smallest difference (so that the number of pixels is equal)
18:            Break out of all loops
19:        end if
20:    end if
21: end for
22: Return the self-made Mat, which contains the saved pixels
```

Algorithm 6 Yellow destination circle calculation.

```
1: $a$ = estimated radius of the destination circle using formula (5.3)
2: for $i$ = $a+1$ to $a-1$ do
3:     Draw circle with radius $i$ on the same center as the yellow source ellipse
4:     Find contours to retrieve the drawed circle
5:     Calculate difference in the number of pixels between destination circle and the yellow source ellipse
6:     if difference is smallest and $\geq 0$ then
7:         Save current smallest difference
8:         Save the contour
9:         if difference $== 0$ || $i == a-1$ then
10:            Save the pixels of the contour minus the difference (so that the number of pixels is equal)
11:            Break out of all loops
12:        end if
13:    else
14:        if The saved contour is empty then
15:            $i$ = $i+2$
16:        else
17:            Save the pixels of the contour minus the current smallest difference (so that the number of pixels is equal)
18:            Break out of all loops
19:        end if
20:    end if
21: end for
22: Return the self-made Mat, which contains the saved pixels
```

combined and computed. By multiplying the transformation matrix with the original points of the image, the transformation can be evaluated.

In OpenCV, a transformation matrix can be constructed by using the `Calib3d.findHomography()` function. This function determines the homography between two planes. It constructs a homography matrix based on the two images (source and destination) that are given as input to the function. Instead of using two images, it is also possible to use two set of points. This would be more practical in our situation, since we only know the complete source image, but do not know the complete destination image. The source ellipses and destination circles, which are discussed in sections 4.2 and 5.1, could be used as an input set of points and destination set of points respectively, because in the end circles are requested. Like said earlier, the source and destination parameters of the `findHomography()` function requires to have an equal amount of pixels.

In the `findHomography()` function, there is another parameter that controls the methods used to compute the homography transformation matrix. There are three methods namely: `Regular(default)`, `RANSAC` and `Least-Median(LMEDS)`. The `regular` method uses all points to calculate the homography transformation matrix. Both, `RANSAC` and `LMEDS`, select randomly matched points and are iterative methods [13]. `LMEDS` calculates the median of the square of the error and seeks to minimize this. `RANSAC` first randomly chooses enough matched points to compute model parameters. Secondly it checks the number of elements of the input feature point dataset which are consistent with the model just chosen. Then it repeats this two steps within a specified threshold until it finds the maximum number of elements within a model. This model is then selected and the mismatches are rejected. The method that is chosen in this case is `RANSAC`. `RANSAC` without any threshold tuning works better than `regular`, but worser than `LMEDS`. However once the appropriate threshold value is chosen for the `RANSAC` method, it shows that transformations achieved through `RANSAC` are even better and more natural than with LMEDS. An example of Image 5.6 transformed with LMEDS and RANSAC can be seen in Image 5.3 and 5.4.

For the perspective transform, the `warpPerspective()` function is used. This function applies a perspective transform to an entire image given an input image, transformation matrix (retrieved from `findHomography()`) and image size.

With the `findHomography()` function in combination with RANSAC there is some difficulty in finding the appropriate threshold value, since this value is different for every scenario. For the purpose of finding the optimal threshold value for every situation, a method is created, that searches for the optimal threshold value by evaluating how close the centers of the blue and yellow ring are to each other. If the distance is very small, the rings are almost perfectly circular. If the distance is high, there is still some skewness which makes the rings not perfectly circular. The method iterates from 1 to 20 for the threshold value and in each iteration the distance between the centers is measured. The threshold value with the lowest distance is chosen. The pseudocode of this method can be seen in Algorithm 7. An example of the importance of the threshold *i* is shown in Image 5.4 and 5.5.

Algorithm 7 Creating the optimal RANSAC transformation matrix.

```
1: Retrieve source ellipse points
2: Retrieve destination circle points
3: for $i = 1$ to 20 do
4:     Create transformation matrix with RANSAC threshold $i$
5:     Transform the perspective with the created transformation matrix
6:     if The distance between the centers is the smallest then
7:         Save the distance
8:         Save the threshold $i$
9:     end if
10: end for
11: Create transformation matrix with the best RANSAC threshold $i$
12: Return the transformation matrix
```

5.3 Verify the perspective and improve it by rotation

Even after the perspective transform with the optimal RANSAC transformation matrix, the target is still not 100% completely circular. Obtaining a center distance of 0 seems impossible, since there always some distance in centers. Therefore after each perspective correction, the target is evaluated. This means that the distance is measured between the blue and yellow contour from the left, right, top and bottom (Image 5.7). Based on the distances of left, right, top and bottom the whole image is rotated in the X and Y direction. The pseudocode of this process can be seen in Algorithm 8. The whole purpose is to incrementally rotate the image in every iteration. This rotation takes place 10 times and in every iteration it is checked if the absolute differences on the horizontal axis (distance left − distance right) and vertical axis (distance top − distance bottom) are minimal, which in the end will result in the best rotated image. Each X and Y value in every iteration is saved into a List of Points. In the end, the rotations which build up to the best rotated image are saved and the rest are discarded. Images 5.8 and 5.9 show the difference before and after the verification and improvement through

Figure 5.3: `LMEDS`.

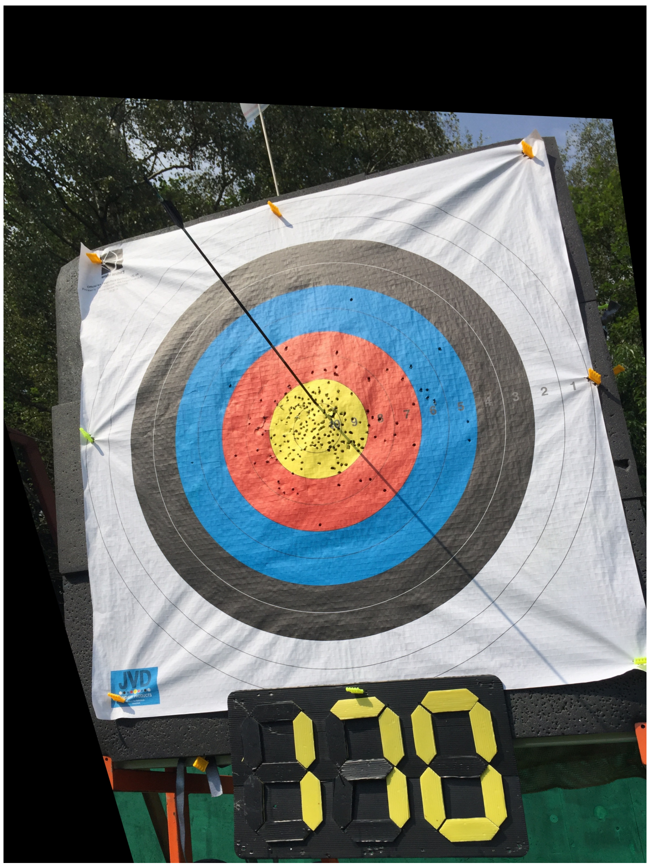


Figure 5.4: `RANSAC` with optimal threshold *i*.

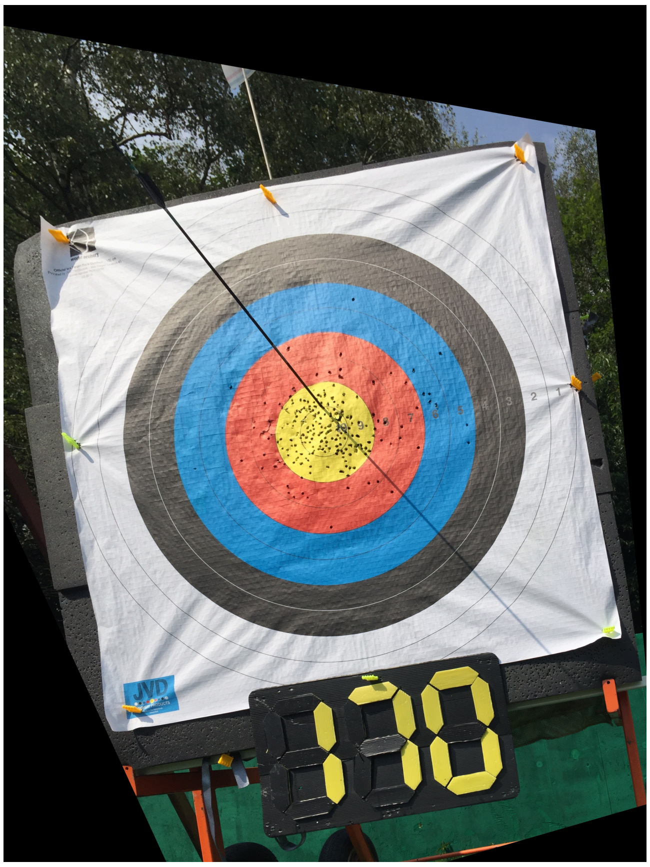


Figure 5.5: `RANSAC` with default threshold *i*.

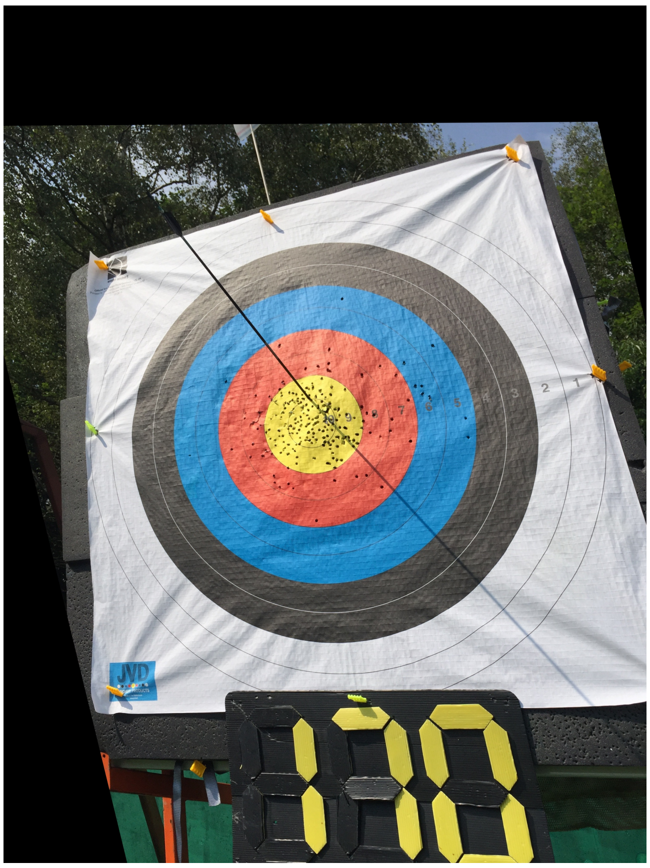
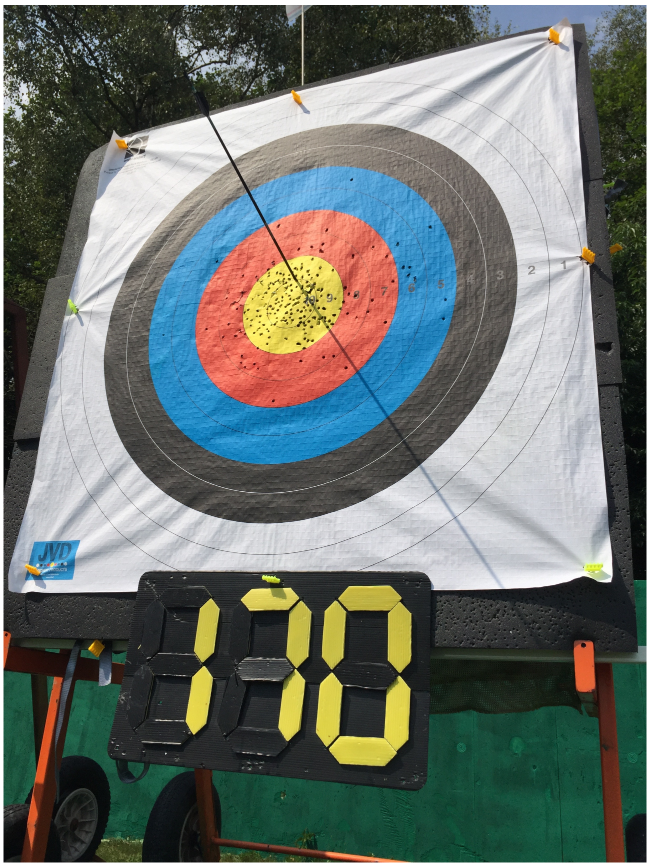


Figure 5.6: Original image 2.



rotation. In Image 5.8, it can be seen that the target is a bit more skewed towards the top. This however is corrected after the verification and improvement by rotation.

Algorithm 8 Verification and improvement of the perspective.

```
1: Create empty variable $it$
2: Initialize variables $x$, $y$ and $z$ with value 90
3: Create an empty List of Points
4: for $i = 0$ to 10 do
5:     Add Point($x$, $y$) to the List
6:     Get the distances of left, right, top and bottom
7:     Calculate the horizontal difference between left and right
8:     Calculate the vertical difference between top and bottom
9:     if Horizontal difference is the smallest and vertical difference is the smallest then
10:        Save the smallest horizontal difference
11:        Save the smallest vertical difference
12:        Save $i$ in $it$
13:    end if
14:    if Horizontal difference is smaller than 10 then
15:        Do nothing
16:    else
17:        if Distance left is bigger than distance right then
18:            Rotate the image around the x-axis with $-0.1$
19:        end if
20:        if Distance right is bigger than distance left then
21:            Rotate the image around the x-axis with $+0.1$
22:        end if
23:    end if
24:    if Vertical distance is smaller than 10 then
25:        Do nothing
26:    else
27:        if Distance top is bigger than distance bottom then
28:            Rotate the image around the y-axis with $-0.1$
29:        end if
30:        if Distance bottom is bigger than distance top then
31:            Rotate the image around the y-axis with $+0.1$
32:        end if
33:    end if
34: end for
35: for $i = Listsize - 1$ to $i >= 0$ do
36:    if $i == it$ then
37:        break
38:    else
39:        Remove object $i$ from the List
40:    end if
41: end for
42: Return the List of Points
```

5.4 Pseudocode perspective correction

The methods that are described are CPU intensive. Therefore it is not recommended to run those methods each time after x number of frames. That's why there is a sequence of methods that are run in the initial phase and another sequence of methods that are run after x number of frames, which are less CPU intensive. The

Figure 5.7: Distance measurement.

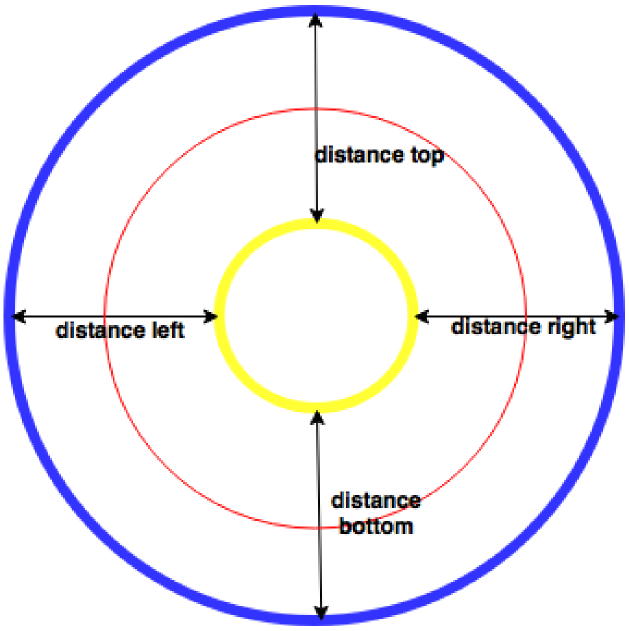


Figure 5.8: Before verification and improvement by rotation.

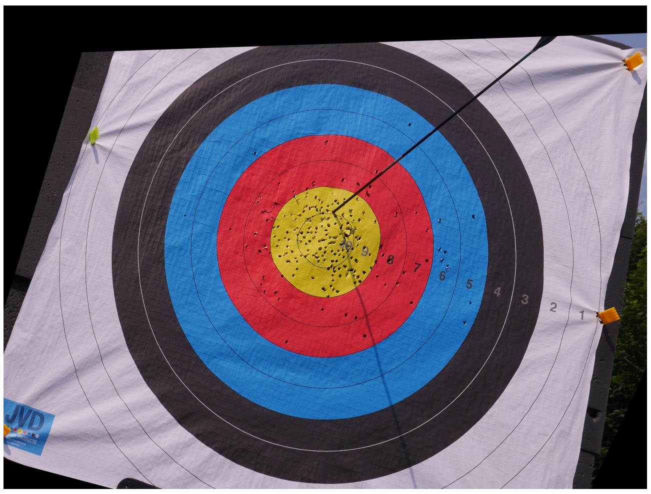
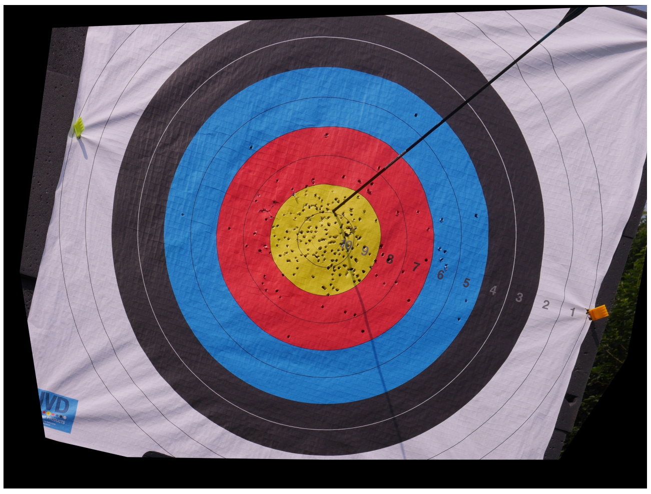


Figure 5.9: After verification and improvement by rotation.



pseudocode of the initial perspective correction is shown at Algorithm 9. This initial perspective correction is run only once at the beginning of the arrow scoring session. The pseudocode of the perspective correction that is run after x number of frames is shown at Algorithm 10.

Algorithm 9 Initial perspective correction.

```
1: Grab the $10th$ frame
2: Retrieve the blue source ellipse (Algorithm 1)
3: Retrieve the red source ellipse (Algorithm 2)
4: Retrieve the yellow source ellipse (Algorithm 3)
5: Combine the source ellipses into one Mat()
6: Calculate the blue destination circle (Algorithm 4)
7: Calculate the red destination circle (Algorithm 5)
8: Calculate the yellow destination circle (Algorithm 6)
9: Combine the destination circles into one Mat()
10: Create the transformation matrix with RANSAC optimal threshold $i$ finding (Algorithm 7)
11: Warp the perspective with the created transformation matrix
12: Create the List of Points with the best rotations, which are obtained from verification and improvement
    through rotations (Algorithm 8)
13: for each Point in the List do
14:     Rotate the frame over the $x$ and $y$ axis
15: end for
```

Algorithm 10 Perspective correction after x number of frames.

```
1: Warp the perspective with the created transformation matrix
2: for each Point in the List do
3:     Rotate the frame over the $x$ and $y$ axis
4: end for
```

5.5 Short evaluation on the distances

In this section the distances left, right, top and bottom are measured (Table 5.1). The images used in this evaluation are part of the training dataset, however specific distances were not measured before. The training set images were only evaluated by visual appearance. Based on the distances we can actually evaluate how circular the targets are after all the methods described above. The distances are measured in pixels. It can be seen that almost all images were relatively close to being perfectly circular. The only images that were the exception were Image 11 and 14. The vertical and horizontal distances were pretty big compared to the rest of the images. Since RANSAC chooses randomly matched points, it could be that for these images the chosen points are not that optimal. Let's stress the fact here that with the OpenCV methods, there is not much tampering possible. The core remains a black box in which you can only change few parameters. Most of it is handled through OpenCV though.

Table 5.1: Distance evaluation.

	distance left	distance right	difference horizontal	distance top	distance bottom	distance vertical	distance centers
image 1	783	777	6	762	769	7	2,5
image 2	312	317	5	310	316	6	3,5
image 3	478	478	0	463	458	5	2,5
image 4	497	497	0	496	498	2	2,1
image 5	487	486	1	482	476	6	5,3
image 6	387	381	6	380	390	10	4,1
image 7	345	336	9	335	341	6	1,8
image 8	380	370	10	364	365	1	5,9
image 9	45	47	2	46	46	0	0,3
image 10	469	458	9	458	452	6	6,6
image 11	783	782	1	741	804	63	27,8
image 12	783	777	6	762	769	7	2,5
image 13	796	777	19	760	766	6	3,7
image 14	731	769	38	785	786	1	19,2

Chapter 6

Sequential arrow detection and scoring

The sequential arrow detection and scoring is a difficult task. The high difficulty is in the fact that there are various noise factors around and on the target, that can result in a low accuracy of the arrow detection and scoring. These noise factors include shadows, light inconsistencies and objects such as birds that could fly in front of the target. This chapter covers the detection, scoring and the various noise factors that are dealt with.

6.1 Amount of change between frames

The initial approach was to detect the arrow, based on the amount of change. One would expect that an arrow would create some sort of a spike in the channels or the amount of change. Such a short experiment is conducted, to see if an arrow would create something that is usable and consistent. The short experiment is conducted on a video from an archery training, where arrows are shot into the target. A snapshot from this video can be seen in Image 6.2. The short experiment consisted of four parts.

The first part was conducted on the full resolution of the video by calculating the amount of difference between the current and previous frame with a frame distance of 100 frames.

The second part was conducted on the full resolution of the video by calculating the peak of every channel of the absolute differenced frame. This absolute differences frame is created by taking the absolute difference between the current and previous frame with a frame distance of 100 frames.

The third part was conducted on the ROI (which was everything inside the blue circle) of the video by calculating the amount of difference between the current and the previous frame with a frame distance of 100 frames.

The fourth experiment was conducted on the ROI of the video by calculating the peak of every channel of the absolute differenced frame. The results of these parts can be seen in figures [6.3, 6.4, 6.5, 6.6].

In the video there were six arrows shot. The arrows were shot around the following frames: 400, 1400, 2800, 4100, 5200 and 6750.

The first part (Figure 6.3) shows that there seems to be too much background noise in order to detect the arrows. This background noise could be the branches of the trees in the background that are constantly moving. This creates a lot of fake spikes in the figure.

The second part (Figure 6.4) shows, the same way as the first part, that there seems to be too much background noise. There are little spikes around the frames where the arrows are shot, however bigger spikes are appearing around other 'normal' frames.

The third part (Figure 6.5) seems not useful enough, since other spikes besides the arrows are also appearing. The expectation was that this part would be much better than the first and second part, since the background is cut out. This is however not the case.

The fourth part (Figure 6.6) actually shows something interesting. There are spikes in the first and third channel visible when the arrow is shot into the target. Unfortunately there also fake spikes visible at non-arrow frames, which makes this approach also not usable.

After evaluating the results, detecting arrows only by the amount of change of frames or channels does not seem to be very useful in this case. Even after the background is cut out, there is still some fake spike noise visible under normal circumstances. This fake spike noise could be the result of the target paper that is moving a bit due to the wind or the arrow. This fake spike noise could worsen for example if the illumination changes over time due to the weather or a bird flies by.

6.2 Background Subtractor (MOG2) vs Absolute difference

Two approaches that can be used to calculate the frame difference, are the background subtractor and absolute difference. These two methods can be used to create binary images that further can be used to detect lines on.

The absolute difference (`absdiff()`) uses two frames and calculates the absolute difference based on that. There is not much parameter tuning for this function, since it is based on basic matrix manipulations. The only parameter tuning is for the `threshold()` function, which is used to convert the absolute differenced image to a binary image. In this case, 50 for `thresh` and 255 for `maxval` is used. An example of `absdiff()` without any thresholding can be seen in Image 6.1.

Background subtraction is a major pre-processing step in many computer vision-based applications. It is a technique used for generating a foreground mask (namely, a binary image containing the pixels belonging to moving objects in the scene) by using static cameras [14]. Background subtraction calculates the foreground mask by performing a subtraction between the current frame and a background model, containing the static part of the scene or, more in general, everything that can be considered as background given the characteristics of the observed scene (Image 6.8). Background modeling consists of two main steps, background initialization

Figure 6.1: Absolute difference between two frames.

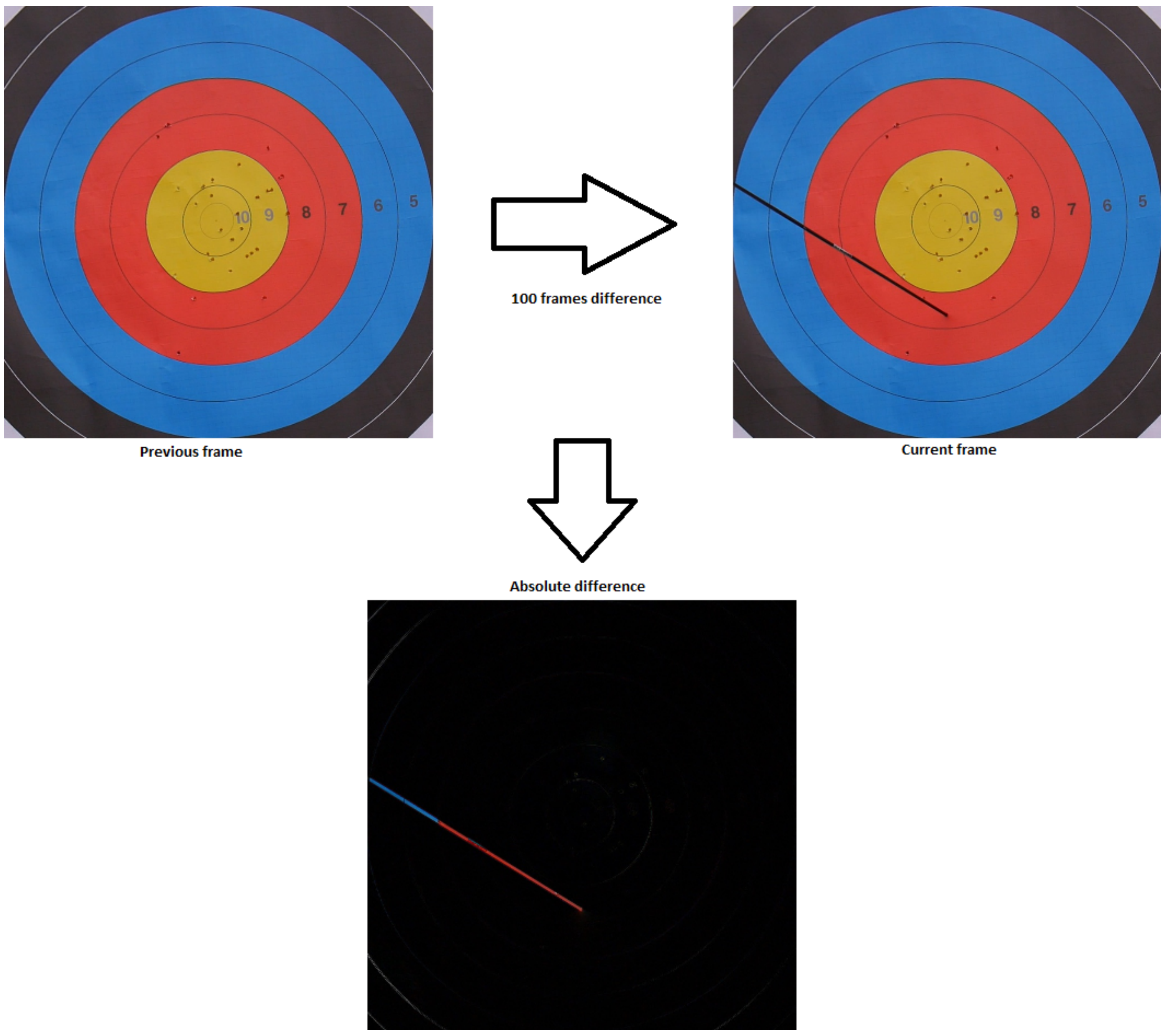


Figure 6.2: Snapshot of archery training video.

Figure 6.3: Full resolution, amount of change.

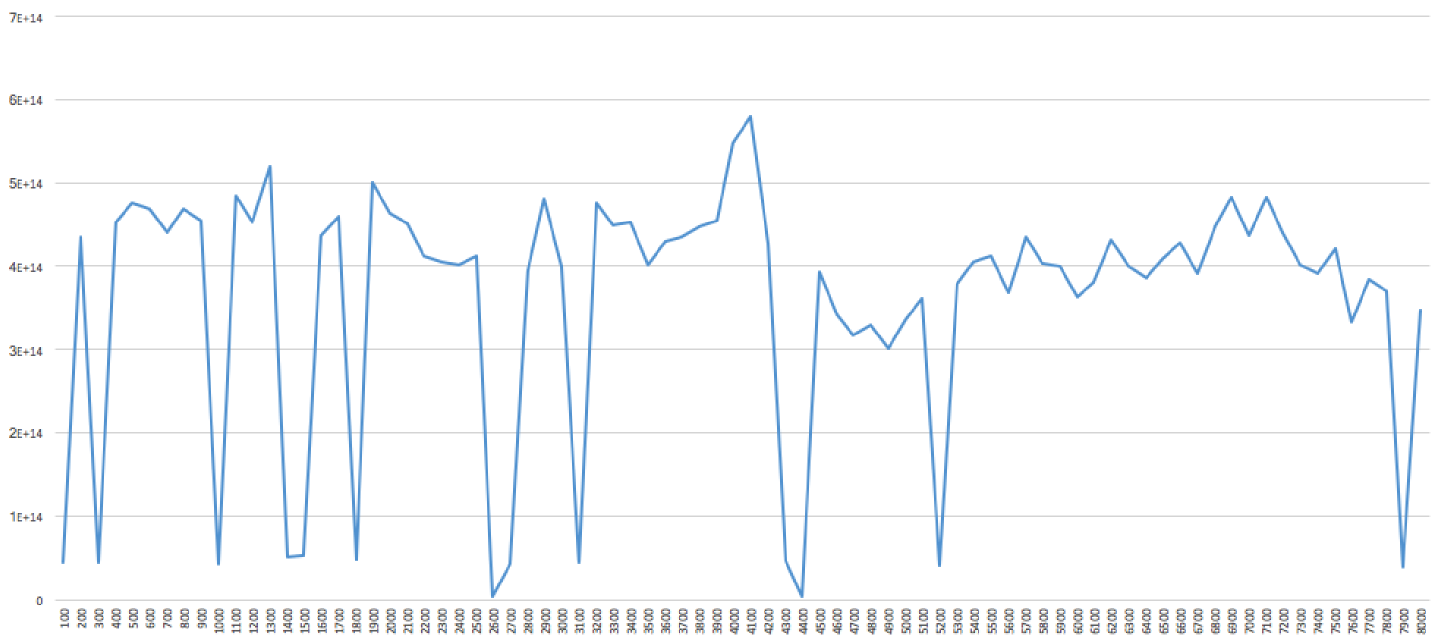


Figure 6.4: Full resolution, peak of every channel.

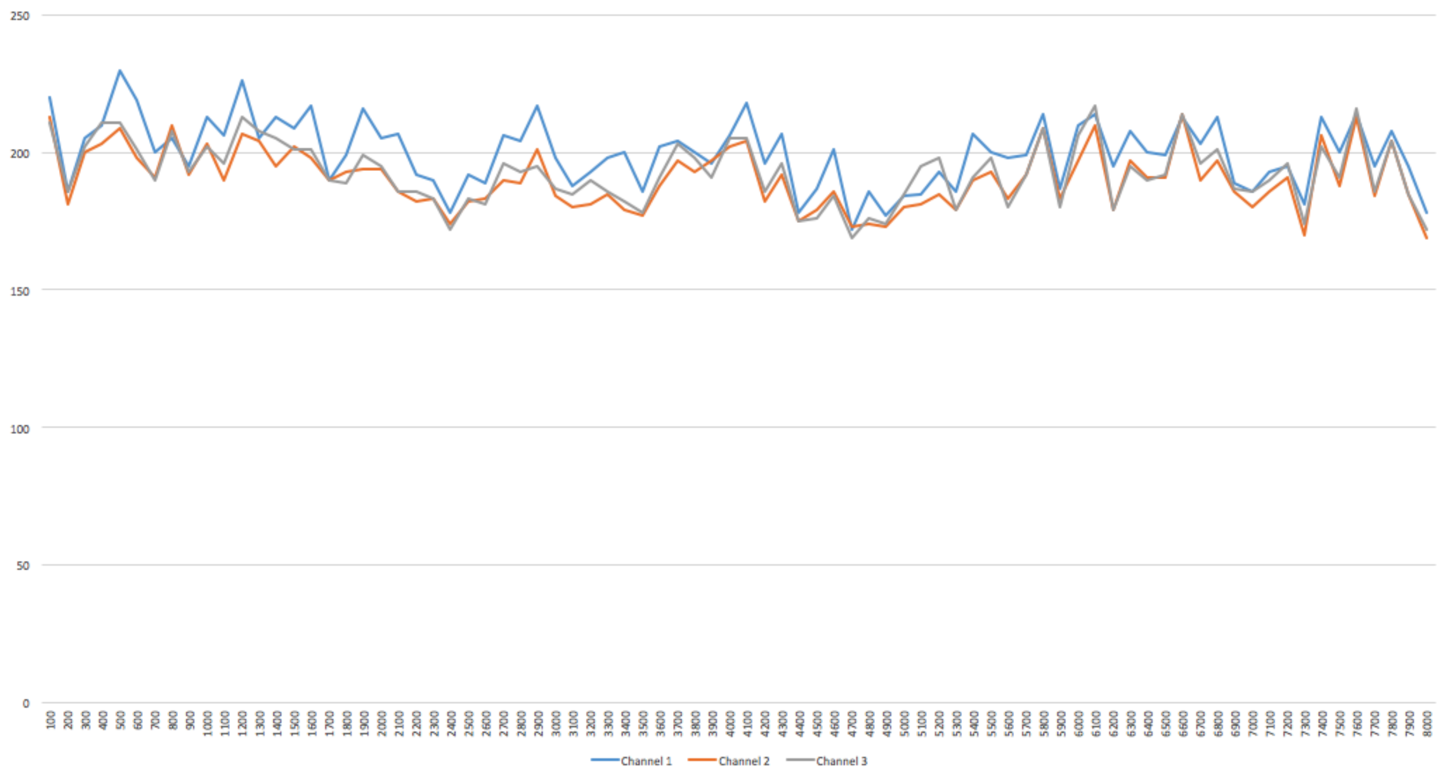


Figure 6.5: ROI, amount of difference.

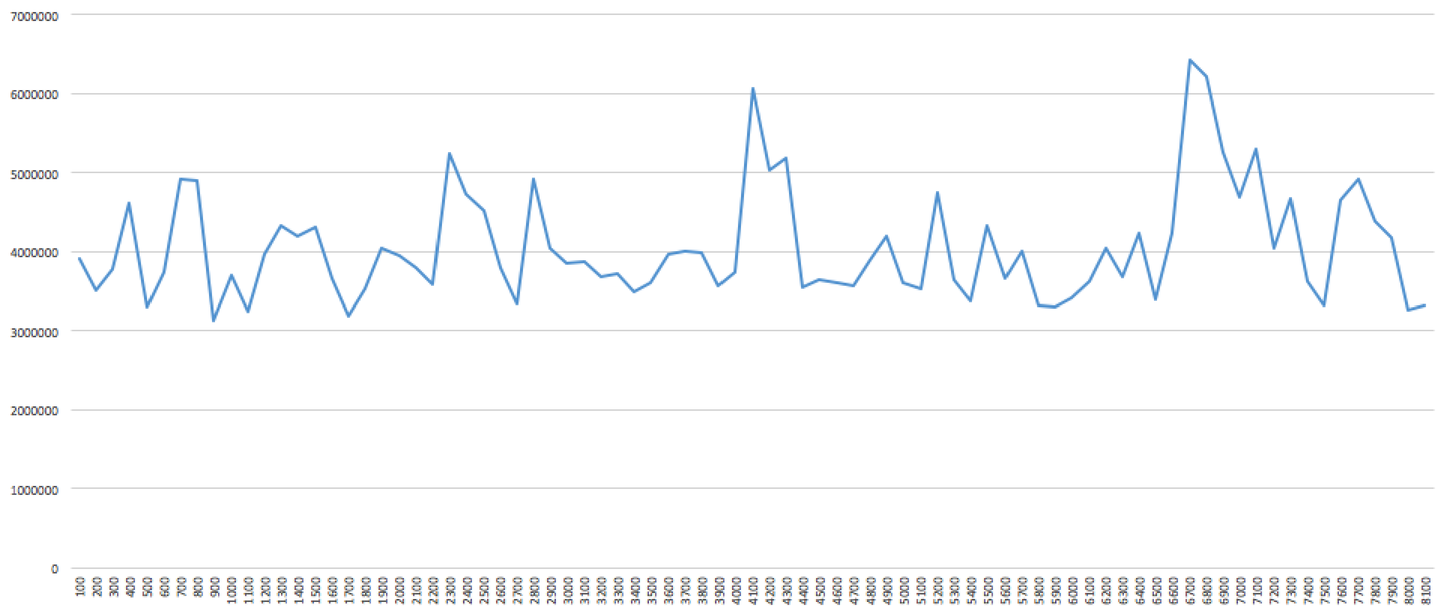
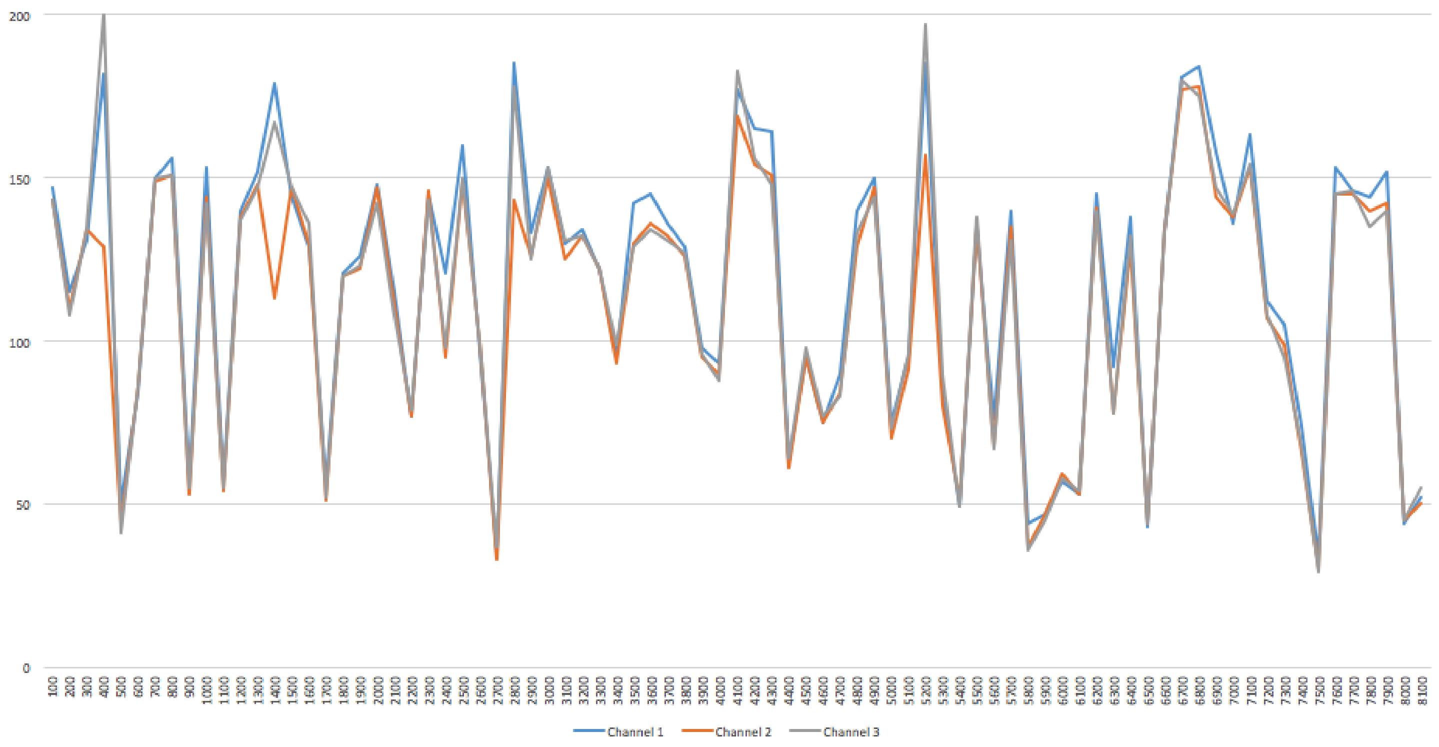


Figure 6.6: ROI, peak of every channel.



and background update. In the first step, the initial model of the background is computed, while in the second step this model is updated in order to adapt to possible changes in the scene.

The `BackgroundSubtractorMOG2()` is a gaussian mixture-based background/foreground segmentation algorithm [15]. It is based on two papers by Zivkovic [16, 17]. With `BackgroundSubtractorMOG2()`, it is possible to detect shadows as well (Image 6.7). This could be extremely useful in our case, since archery in open air can create shadows if the arrows are shot into the target. If shadow detection is enabled as parameter, it detects and marks the shadows with a grey color. It is said that it decreases the computation speed, however during implementation no such things were observed.

Besides the shadow detection parameter which is a boolean, `BackgroundSubtractorMOG2()` has three other important parameters. These are `history`, `threshold` and `learning rate`. `history` represents the length of the history, which sets the amount of time over which some 'experience' of a pixel color will last [18]. Essentially, it is the time it takes for the influence of that pixel to decay away to nothing. `threshold` determines how well a pixel is defined by the background model. For example by putting a high value, it will only detect new pixels which are extremely different from the background model. Every frame is used for both, calculating the foreground mask and updating the background model. The `learning rate` determines the rate for updating the background model. For this master thesis, 500 for `history`, 200 for `threshold` and 0.075 for `learning rate` is used.

6.2.1 Comparison

While both methods sound promising, especially `BackgroundSubtractorMOG2()`, we want to see the actual performance of using both methods. Therefore a comparison is made of these two methods, by comparing the frame differences. These frame differences are created from the training dataset videos. One set of frame differences is created with `BackgroundSubtractorMOG2()`, while the other set is created by using absolute difference. Because it is not convenient to show the comparisons between all the frames of the video, two frames are picked that are created with `BackgroundSubtractorMOG2()` (Image 6.9 and 6.10) and two frames that are created with `absdiff()` (Image 6.11 and 6.12).

First of all looking at the images, the overall noise seems to be around the same. During observations sometimes using `absdiff()` results in more noise, while other times using `BackgroundSubtractorMOG2()` results in more noise. Unfortunately the shadow detection from `BackgroundSubtractorMOG2()` does not worked well in our case. In some videos, it was able to distinguish shadows from the actual arrows (Image 6.9), while in other cases it detected almost everything as shadows including the arrow (Image 6.13). Attempts were made in adjusting the parameters, however changing these does not showed any improvement in shadow detection. We can thus conclude that the shadow detection feature of `BackgroundSubtractorMOG2()` does not worked reliable in our case. Besides the shadow detection feature, both methods show around the same performance in differencing frames. `BackgroundSubtractorMOG2()` requires more work in the sense that it has many parameters and needs some frames to learn on, while `absdiff()` can be used at any moment without any knowledge about the whole history. Because they show similar performance and `absdiff()` requires far less resources, this is chosen as method to use.

6.3 Hough Line Transform

The Hough Line Transform is a feature extraction technique that is used in image analysis, computer vision and digital image processing. The theory of the Hough Line Transform is that any point in a binary image could be part of some set of possible lines [19]. If you plot for a given point the set of lines that goes through it, you will get a sinusoid in the rho-theta space. This can be done for all points in an image and if the curves in rho-theta space of two different points intersect, that means that both points belong to the same line. So in general what the Hough Line Transform does is that it keeps track of the intersection between curves of every point in the image and if the number of intersections is above some threshold, then it declares it as a line [20].

The Hough Line Transform needs to be applied to a binary image before it can find any lines. The Hough Lines Transform in OpenCV (`Imgproc.HoughLinesP()`) consists of five adjustable parameters. These are `rho`, `theta`, `threshold`, `minLineLength` and `maxLineGap`. `rho` is the resolution of the parameter `r` in pixels (default is 1). `theta` is the resolution of the parameter θ in radians (default is $\Pi/180$). `threshold` is the minimum number of intersections to detect a line. `minLineLength` is the minimum number of points that can form a line. `maxLineGap` is the maximum gap between two points to be considered in the same line. The parameters are adjusted to the training dataset as follows: 1 for `rho`, $\Pi/180$ for `theta`, 100 for `threshold`, 50 for `minLineLength` and 20 for `maxLineGap`. The first two parameters are kept at default. The 3rd, 4th and 5th parameter values are chosen by testing on the training dataset. The `minLineLength` is kept fairly moderate. This way too short lines, which often represent noise, are discarded. The `maxLineGap` is kept very low, because if the lines are detected with interruption gaps, they often are close to each other and this makes sure that the algorithm does not stop when there are little gaps in the lines.

What often is the case with `HoughLinesP()`, is that the arrows are detected as multiple short and long lines that together build the detected arrow. Such an example can be seen in Image 6.14. To counter this problem, the lines need to be drawn with a distinctive color, like green. Since the lines are mostly connected, `findContours()` after green color segmentation can make sure that the multiple detected lines are detected as one contour (line).

Comparing the Hough Line Transform on images and videos, shows that it is much more convenient on videos. The reason for this is because with videos there are multiple frames and this way differences can be accurately spotted, such as arrow lines that suddenly appear. With images, there is not much information and therefore makes it more difficult for the Hough Line Transform to accurately detect arrows.

Figure 6.7: Shadows detected in gray with `BackGroundSubtractorMOG2()`.

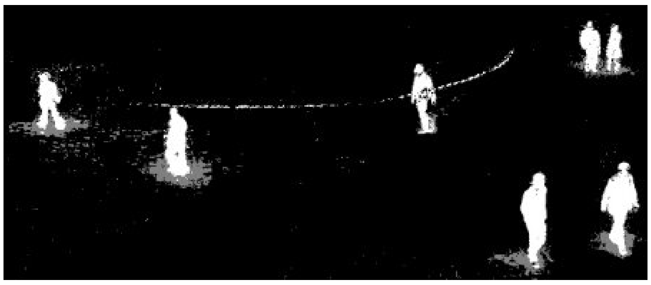


Figure 6.8: Foreground mask calculation.

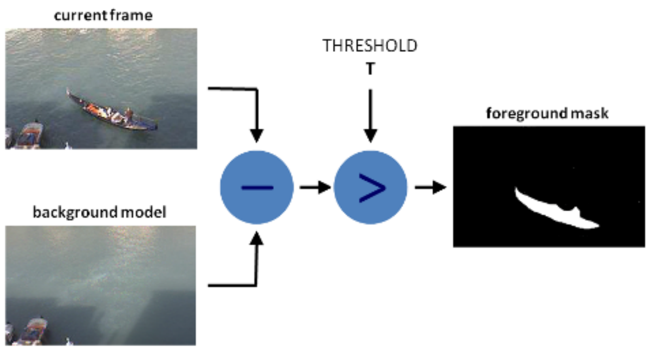


Figure 6.9: `BackgroundSubtractorMOG2()` example.

Figure 6.10: `BackgroundSubtractorMOG2()` example.

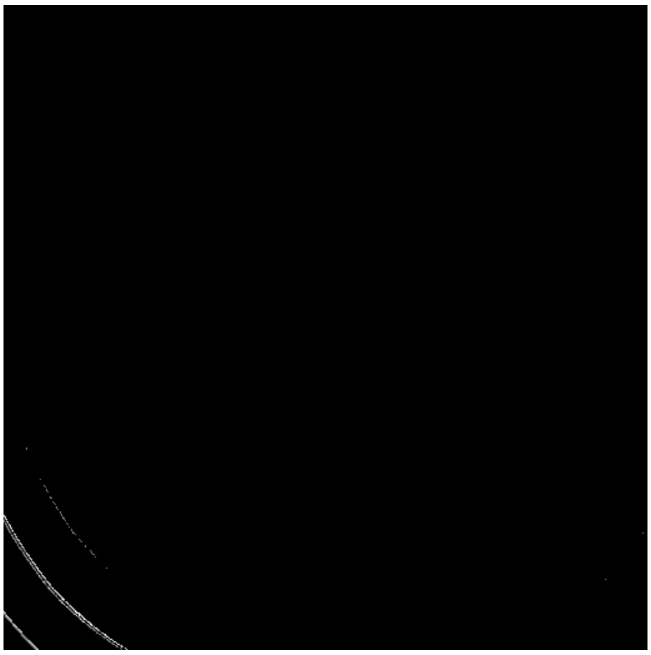


Figure 6.11: `absdiff()` example.

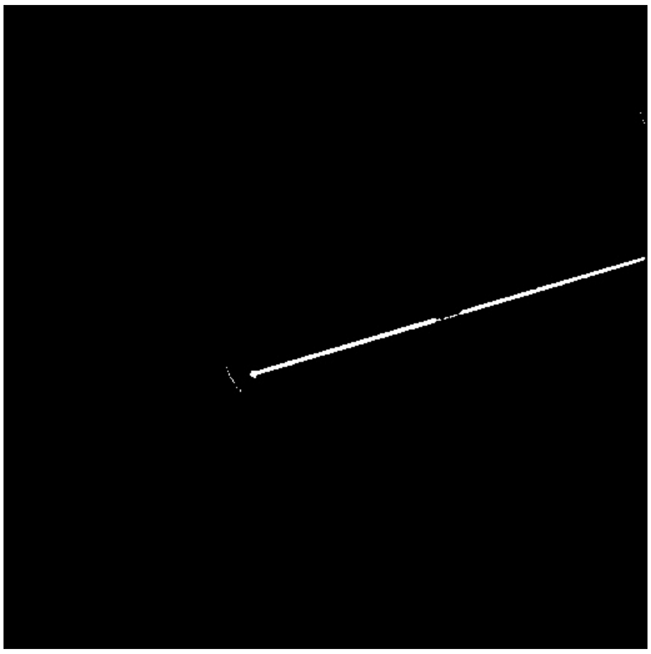


Figure 6.12: `absdiff()` example.

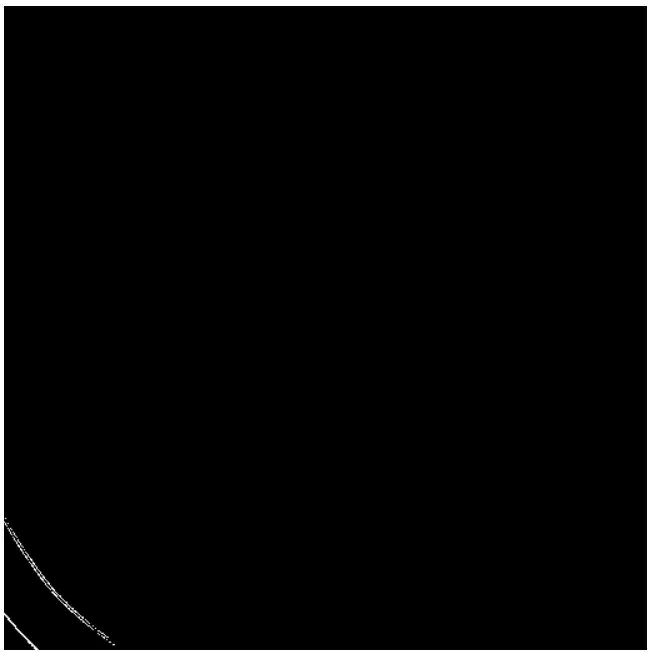
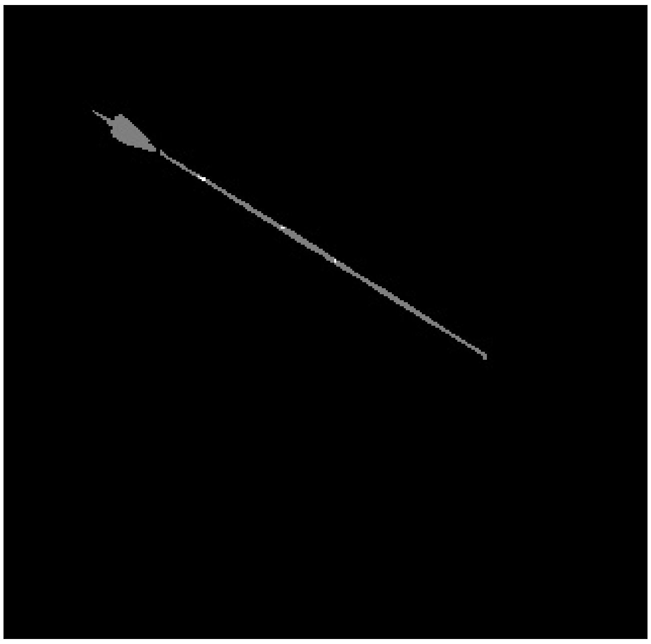


Figure 6.13: `BackgroundSubtractorMOG2()` incorrectly detecting shadows.



6.4 Light inconsistency

Light inconsistency is a problem that could take place during archery training, especially when the training is taking place outside. This light inconsistency appears in the form of change in illumination over time. Under normal circumstances such an illumination change may not appear instant, but slowly over time during the match, which will be barely noticeable. However it should be taken into account that a dark cloud in front of the sun could create such a sudden change in illumination. The result of such a sudden illumination change could impact the detection of arrows. Therefore different methods and techniques are dealt with in order to tackle this potential problem. Image 6.1 shows the absolute difference between two consecutive frames (with a distance of 100 frames), without using any special method or technique.

6.4.1 Histogram equalization

Every image has an intensity histogram. This intensity histogram is a representation of the intensity distribution of an image. It quantifies the number of pixels for each intensity value in the image. Histogram equalization is a technique that aims to improve the contrast of the image, by stretching out this intensity range [21]. A basic example of this can be seen in Image 6.15.

Histogram equalization is tested by applying it to the current and the previous frame (with a frame distance of 100 frames). The difference between the two frames is then visualized (Image 6.16). Unfortunately on the image it is seen that there is much noise visible around the rings.

Histogram equalization is also tested by applying it to the light channel of the current frame and previous frame (with a frame distance of 100 frames). This light channel can be found when the frame is converted to a YUV or LAB color space. In the YUV color space, the light channel is indicated by Y. In the LAB color space, the light channel is indicated by L. The difference between two frames using histogram equalization on the light channel can be seen in Image 6.17. Unfortunately there is also much noise visible around the rings. Such noise is not desirable, since it could create difficulties for the arrow scoring mechanism using the Hough Line Transform.

6.4.2 CLAHE

CLAHE (Contrast-limited Adaptive Histogram Equalization) is an improvement over histogram equalization, in the sense that it does not considers the global contrast of the Image [22]. The global contrast is not useful in some cases where the histogram of the image is not confined to a particular region, since this can create over-brightness of the image. With CLAHE, the image is divided into small blocks. Then each of these blocks are histogram equalized as usual. The contrast limiting part is that if any histogram blocks are above the specified contrast limit, those pixels in the block are distributed uniformly to other blocks before applying histogram equalization. After equalization, bilinear interpolation is applied to remove artifacts in the blocks borders.

Figure 6.14: Multiple lines that make up the whole arrow.

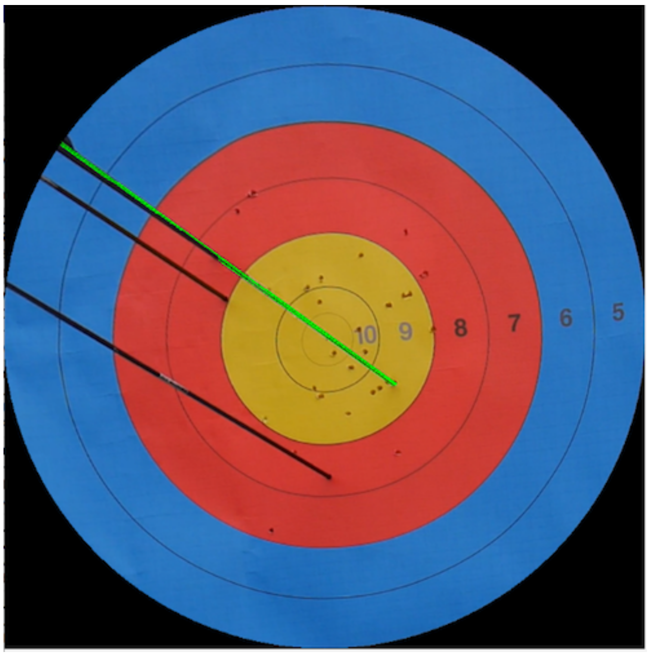


Figure 6.15: Example of histogram equalization (taken from MathWorks).

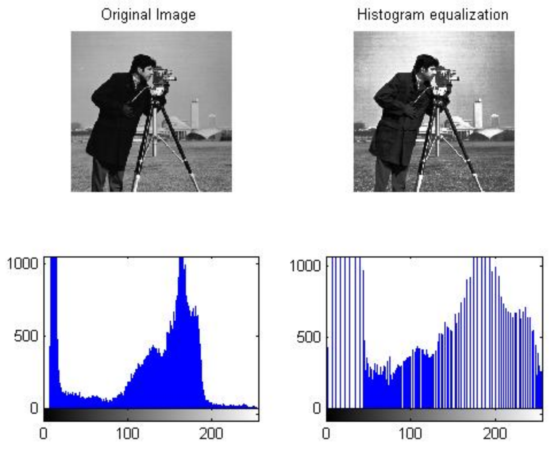


Figure 6.16: Using histogram equalization.

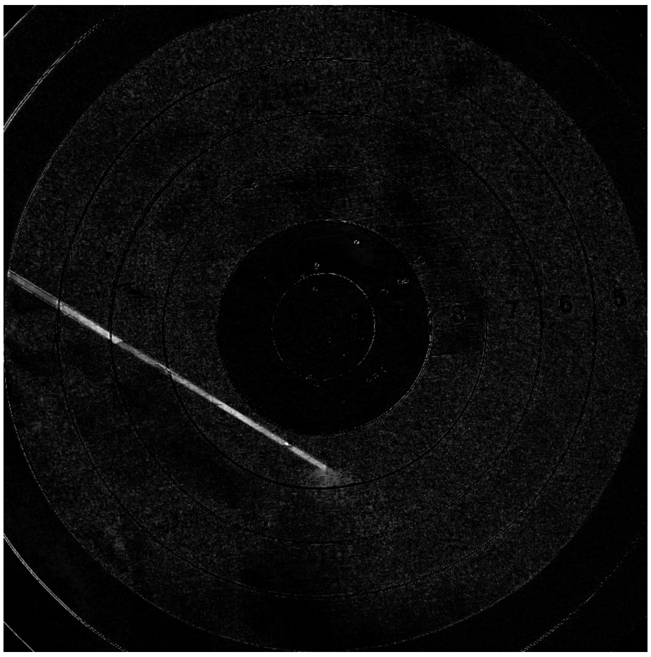
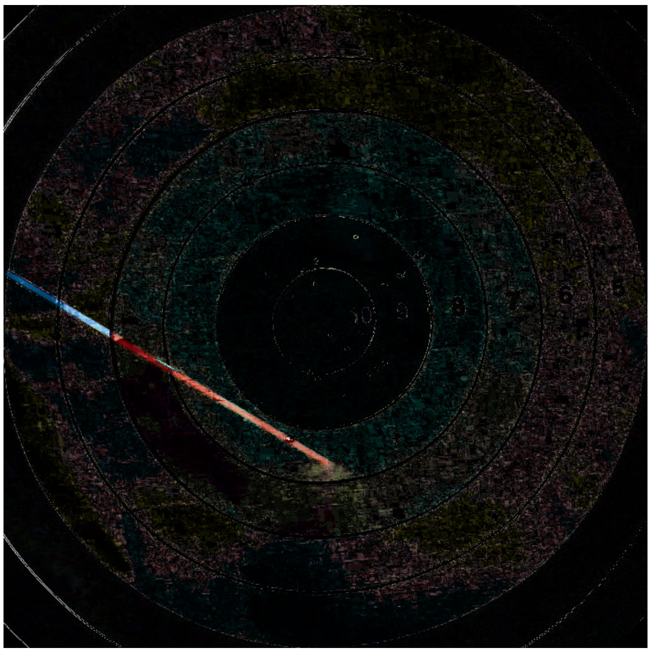


Figure 6.17: Using histogram equalization on the light channel of LAB.



CLAHE is tested by applying it to the current and the previous frame (with a frame distance of 100 frames). The difference image can be seen at Image 6.18. The noise is far less compared to histogram equalization. Unfortunately there is more noise compared to Image 6.1, especially around the center of the image.

CLAHE is also applied to the light channel of the LAB color space (Image 6.19). Unfortunately there is still more noise visible in the center compared to Image 6.1, the same as in Image 6.18.

6.4.3 Alternative method

We can conclude that using the absolute difference of two consecutive frames without using any special technique seems to be the best solution. It shows (Image 6.1) that there is almost no noise visible compared to using histogram equalization (Image 6.16 and 6.17) or CLAHE (Image 6.18 and 6.19). The current alternative to combat sudden light intensity changes, is by checking the amount of lines that are detected by the Hough Line Transform. Sudden changes in intensity, or objects/animals that suddenly appear before the target can create a huge amount of fake lines. To give an example, an arrow that is shot creates around 10 detected lines, while a sudden intensity change or object creates around 100 or more detected lines. By counting the lines against a threshold, it can be determined if the frame difference is an arrow that is shot or something else. This determination gives us te possibility to accept it as an arrow or simply ignore it.

6.5 Arrow scoring

The arrow scoring is based on the location the arrows are shot. Arrow scoring uses the contours of the three colors to determine the score of the arrow. The arrow scoring is being processed in different steps to ensure that the arrows are accurately detected.

6.5.1 Shadows

Shadows are the tricky part of the arrow scoring. The reason for this, is because sometimes shadows are being detected as arrows itself if not handled correctly. Unfortunately the shadow parameter in the `BackgroundSubtractorMOG2()` function was no option because in some cases it even classified arrows as shadows. Therefore another approach was taken into account against shadows. In the training dataset, there was one video which contained shadows. `absdiff()` with `HoughLinesP()` was not able to capture any shadows, however it would be better to built in an extra security measurement to prevent that shadows are detected as arrows.

To prevent that shadows are being detected as arrows, a limit should be put on how many arrows are shot at a time. The archers at the NHB shoot one arrow at a time, so it would be a logical decision to limit this to one arrow per time. Before arrows are taken into consideration, it is first evaluated how many lines are detected by the `HoughLinesP()` function. If this is lower than 20, it is probably an arrow. After this, all the contours that

Figure 6.18: Using CLAHE.

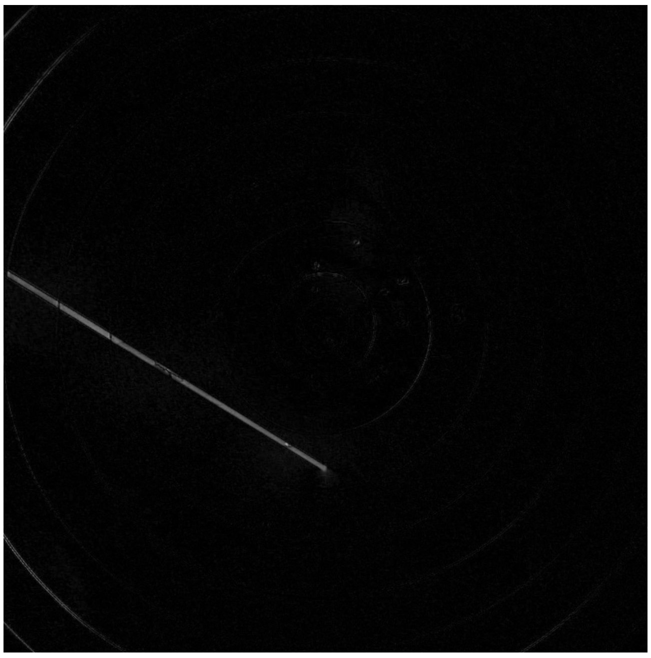
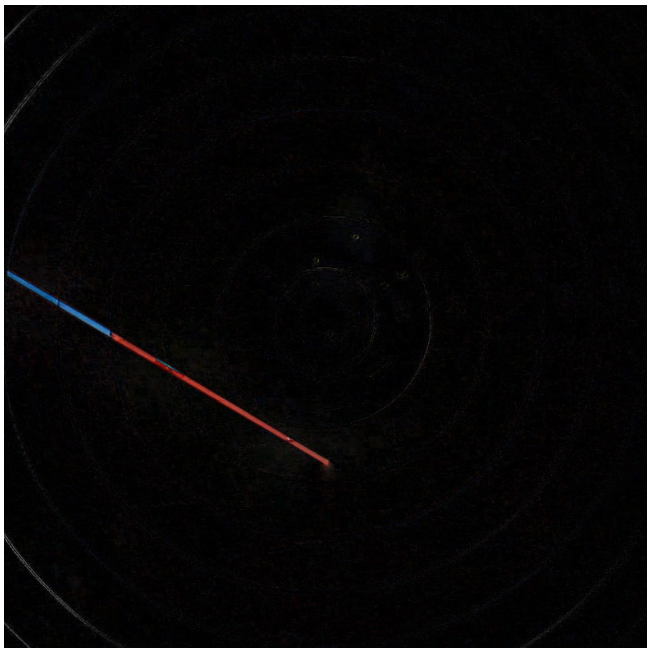


Figure 6.19: Using CLAHE on the light channel of LAB.



are detected, are analyzed, and the biggest contour is chosen as arrow. Shadows are smaller and/or less easier detected than real arrows so this approach makes sure that the real arrow is classified as arrow, and not the shadow.

6.5.2 Determining the tip of the arrow

The first step is to determine in what direction the arrow is shot. Since the arrows has two sides, it has to be determined which side represents the tip of the arrow. After an arrow is shot and passes all the 'safety' checks, the first pixel of the arrow is being saved. This first pixel is or is relatively close to a corner by which the diagonal is the end tip of the arrow. `minAreaRect()` finds the rotated rectangle of the minimum area enclosing the arrow. By drawing a `minAreaRect()` around the arrow, each corner of that rotated rectangle can be evaluated with the position of the start pixel. The evaluation happens in the form of the euclidean distance between each corner of the rotated rectangle and the start pixel. The corner that is the closest to the start pixel is the start corner and the diagonal of that corner represents the tip of the arrow.

6.5.3 Calculating the score

Once the tip of the arrow has been found, the location of that tip can be used for the scoring. The score can be determined by evaluating its location against each of the three contours. This evaluation uses the `pointPolygonTest()` function, which determines whether a given point is inside a contour, outside or lies on an edge. It returns a positive, negative of zero value respectively. If the `measureDist` parameter is set to true, it will return a signed distance between the point and the nearest contour edge, otherwise the returned value is $+1$, -1 and 0.

This `pointPolygonTest()` function is used to first detect in which region the arrow is shot. In each region, there are two scores to which it belongs. For yellow this is 10 or 9, for red this is 8 or 7 and for blue this is 6 or 5. If the arrow is inside yellow, it is checked with the euclidean distance if the point is closer to the center of the yellow circle (score 10) or closer to the yellow contour (score 9). If the arrow is inside red, it is checked with the euclidean distance if the point is closer to the yellow contour (score 8) or closer to the red contour (score 7). If the arrow is inside blue, it is checked with the euclidean distance if the point is closer to the red contour (score 6) or closer to the blue contour (score 5).

It was advised by the NHB to calculate decimal scores. For each case (arrow in yellow area, arrow in red area and arrow in blue area), the distance between the arrow and the contour is calculated, the total distance between one contour and the other is calculated and the half of that distance is also calculated. An example can be seen in Image 6.20. By using basic mathematical calculations, the position of the arrow can be converted to a decimal number inside a color region. The pseudocode of this process can be seen in Algorithm 11.

Algorithm 11 Calculation of the score.

```
1: Determine the center of yellow
2: Calculate the distance to the yellow contour (pointPolygonTest())
3: Calculate the distance to the red contour (pointPolygonTest())
4: Calculate the distance to the blue contour (pointPolygonTest())
5: if distance to blue $> 0$ and distance to red $> 0$ and distance to yellow $> 0$ then
6:     distance to center = the euclidean distance between the arrow and the yellow center
7:     total distance = distance to yellow + distance to center
8:     half distance = total distance / 2
9:     if distance to center $<$ distance to yellow then
10:        newpos = abs(distance to yellow $-$ half distance)
11:        score = 10 + (newpos / half distance)
12:    else
13:        score = 9 + (distance to yellow / half distance)
14:    end if
15: end if
16: if distance to blue $> 0$ and distance to red $> 0$ and distance to yellow $\leq 0$ then
17:     total distance = distance to red + abs(distance to yellow)
18:     half distance = total distance / 2
19:     if abs(distance to yellow) $<$ distance to red then
20:        newpos = abs(distance to red $-$ half distance)
21:        score = 8 + (newpos / half distance)
22:    else
23:        score = 7 + (distance to red / half distance)
24:    end if
25: end if
26: if distance to blue $> 0$ and distance to red $\leq 0$ and distance to yellow $\leq 0$ then
27:     total distance = distance to blue + abs(distance to red)
28:     half distance = total distance / 2
29:     if abs(distance to red) $<$ distance to blue then
30:        newpos = abs(distance to blue $-$ half distance)
31:        score = 6 + (newpos / half distance)
32:    else
33:        score = 5 + (distance to blue / half distance)
34:    end if
35: end if
36: return score
```

6.5.4 Converting the coordinate system

The current coordinate system of the arrows, are not that informative to the archers. This is because it uses the resolution of the video, which is dependent of the resolution of the camera. To evaluate competitions or trainings, it is more convenient to use a coordinate system that is consistent across all trainings. The creation of such a coordinate system splits the resolution into four parts with zero in the middle. It looks as follows (Image 6.21). The coordinates of the arrow are converted to the proposed coordinate system by the following formulas 6.1, 6.2, 6.3 and 6.4.

$$new\_arrow\_start\_x = -1 + \frac{(arrow\_start\_x \cdot 2)}{video\_width} \tag{6.1}$$

$$new\_arrow\_start\_y = 1 - \frac{(arrow\_start\_y \cdot 2)}{video\_height} \tag{6.2}$$

$$new\_arrow\_end\_x = -1 + \frac{(arrow\_end\_x \cdot 2)}{video\_width} \tag{6.3}$$

$$new\_arrow\_end\_y = 1 - \frac{(arrow\_end\_y \cdot 2)}{video\_height} \tag{6.4}$$

In these formulas, *arrow_start* and *arrow_end* are the coordinates of the arrow in the old coordinate system, where *arrow_start* is the tail of the arrow and *arrow_end* is the tip. Respectively *new_arrow_start* and *new_arrow_end* are the coordinates in the new coordinate system. The left corner on the x-axis starts with -1 and the left corner on the y-axis starts with 1. The *arrow_start* and *arrow_end* are being multiplied by 2 and divided by the width and height of the old coordinate system, because the total horizontal and vertical distance in the new coordinate system is 2.

6.5.5 Pseudocode of the arrow scoring

The pseudocode of all the methods and techniques described in Section 6.5 can be seen in Algorithm 12. There is a cool down after an arrow is shot. Mostly after an arrow hits the target, the tail swings a bit. In the worst case, the swinging of the tail can be detected as a new arrow. Therefore this cool down ensures that after an arrow hits the target, in the next 100 frames nothing can be detected. This cool down is evaluated at the beginning of the arrow scoring, before any other methods takes place.

Figure 6.20: Total distance and half distance visualized.

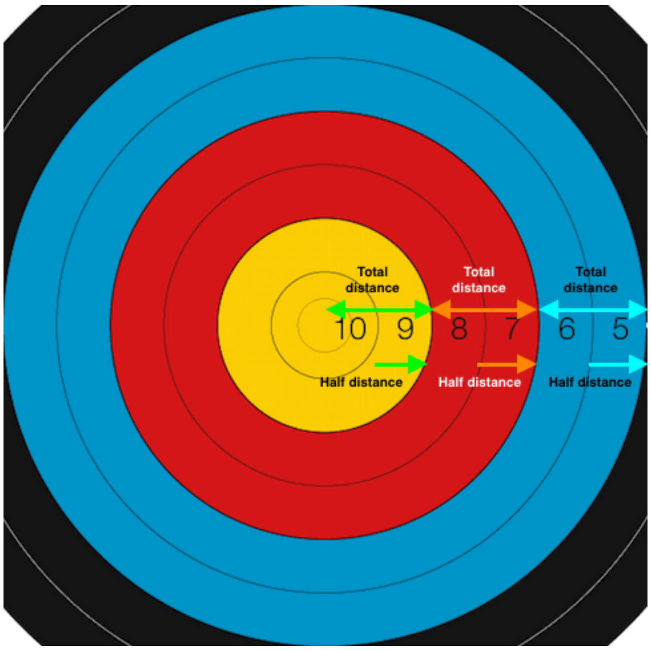
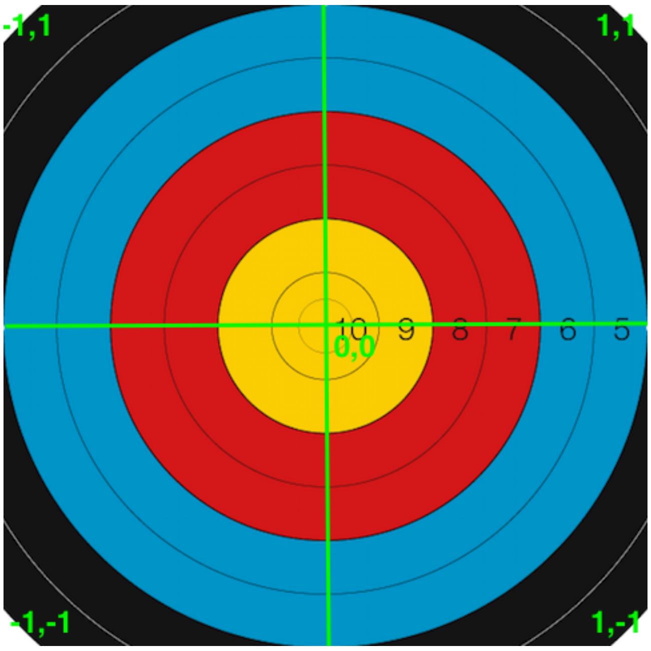


Figure 6.21: Coordinate system.



Algorithm 12 Pseudocode of the arrow scoring.

```
1: if cool down is not active then
2:     if number of detected lines $> 0$ and $\leq 20$ then
3:         Mark multiple detected lines on the same location as one, by using findContours()
4:         Calculate the location of the center of yellow
5:         if the number of found contours $\geq 1$ then
6:             Save the biggest found contour
7:             Save the start coordinates of the contour
8:             Draw a minAreaRect() around the contour and save that to a RotatedRect
9:             Calculate the euclidean distance from each corner of the RotatedRect to the start coordinates
10:            Find out which corner is the closest to the start coordinates
11:            Save that corner as start coordinates and the diagonal of that corner as end coordinates
12:            Calculate the score for the end coordinates using Algorithm 11
13:            Calculate the new start and end coordinates in the new coordinate system using formulas 6.1,
               6.2, 6.3 and 6.4
14:            Calculate the euclidean distance between the end coordinates and the yellow center
15:            Save the score, coordinates and distance to center
16:            Activate cool down for the next 100 frames
17:        end if
18:    end if
19: end if
```

Chapter 7

Experiments

The data used in the experiments are somewhat different than in most other papers, since they are videos. Unfortunately due to poor availability of new test data, we are forced to test our methods on the training dataset. The new test data would be obtained from the NHB, but because of various factors including distance, it was not easily obtainable. In fact, having test data would actually show the quality of how well the program would work or not, but since this is not available we used the training data instead. The results of this experiments may not seem that useful, however it still can be interesting to see if there are any occasions where arrows are not correctly detected, and what is causing that. It is true that we did adjust the parameters based on the training dataset, however there were still some odd moments during parameter tuning. This chapter, thus would be a good moment to explain such cases.

For this experiment, we used in total six training set videos (which was the max number of videos we had). Four of those videos were unedited while in two videos noise and/or light inconsistencies were induced through video editing software. Two of the four videos were taken with the Logitech C920 PRO HD, while the other two were taken with an unidentified camera. The data set of the six videos can be found at [9]. In this experiment, it is checked how many times the target is correctly detected and how many times the arrows are correctly scored for each video. The results are shown in Table 7.1.

From the experiments some important things can be noted. The target detection works pretty solid. We did not test this on the training dataset only, but we also pointed the Logitech C920 PRO HD at target images, in which it was able to detect the target. It is also expected that the target detection should work pretty well, since it supports a great range of color intensities for each color, which can be seen by looking at the color palettes (Image 4.4, 4.5 and 4.6).

The scoring works well, in the sense that the program can correctly score in each region (10, 9, 7, 6 and 5). However once we look closer at the decimal calculation of the score, we can see that these are sometimes a bit off (Image 7.5). In the image, it can be seen that the decimal score of arrows 3 and 4 actually should be in the $>$ .90 range instead of .85 and .82 respectively. These little decimal differences are also happening at other videos. This is something we discussed with the NHB, and according to them, we should score from the inside

Table 7.1: Detection of the target and scoring of the arrows on the training dataset.

	target detected	number of arrows	correctly scored
Vid_1	yes	7	7
Vid_2	yes	6	6
Noise	yes	2	1
Darkening	yes	5	5
Logitech_C920_1	yes	6	6
Logitech_C920_2	yes	6	4

of the arrow (Image 9.1). Fortunately the main scores are correct, but only the decimal scores could use some more refinement. More about this will be discussed in the future work (Chapter 9).

Something notable, is that the videos that are taken with the undefined camera (Vid_1 and Vid_2) are far more sharper than with the Logitech C920 (Logitech_C920_1 and Logitech_C920_2). A comparison of the video quality can be seen in Image 7.2 and 7.3. In the experiments, we did not see any big differences in detection between these two cameras, however it is logical to think that the arrow scoring would be more accurate on higher resolution quality cameras.

The experiments furthermore show that at the Noise and Logitech_C920_2 videos, some arrows were missed or incorrectly scored. In the Noise video, the arrow was shot into the target during a huge brightness shift. This huge brightness shift caused a lot of detected lines. Because the number of detected lines was bigger than 20 (Section 6.5), the arrow was rejected all together with the high brightness shift. At the Logitech_C920_2 video, one arrow was missed and another was wrongly scored. The arrow that was missed can be seen in Image 7.1. This arrows appears to be too short to be detected. That is the risk of choosing the appropriate value for the length of the line in the `HoughLinesP()` function. Making the length too short and fake noise arrows may appear, but making the length too long and a lot of arrows will not be detected at all. The arrow that was wrongly scored can be seen in Image 7.4. It can be seen that arrow 3 should have score $10.xx$, but instead it has the score 9.85 and also gets detected in the wrong region. The reason for this, is because unfortunately when the arrow hit the target it moved the paper a bit at the hole's position. Because there is movement detected at the hole and the distances to each other are short, the `HoughLinesP()` function prolongs the arrow. The distance for dots to connect is already put very low, so nothing much can be done software-based, however the paper of the target could be more tightened to prevent such cases.

Figure 7.1: Missing the short arrow.

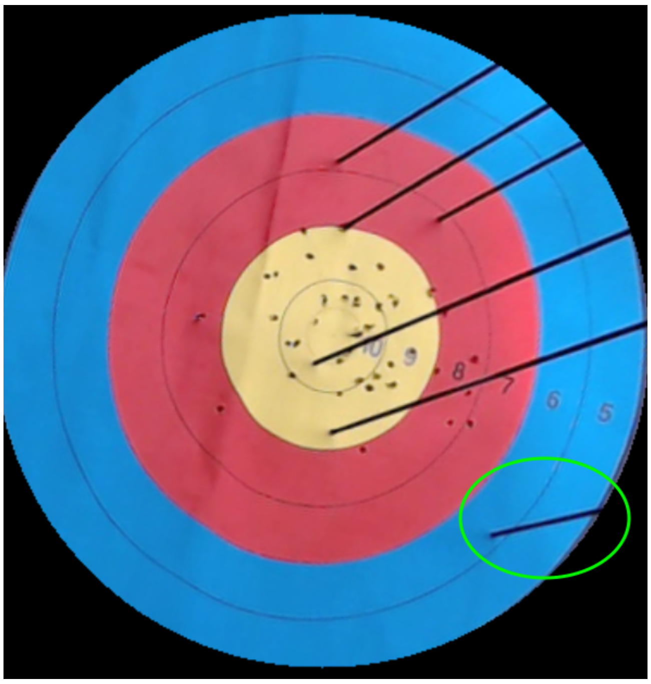


Figure 7.2: Quality of the target (*Vid*_1).

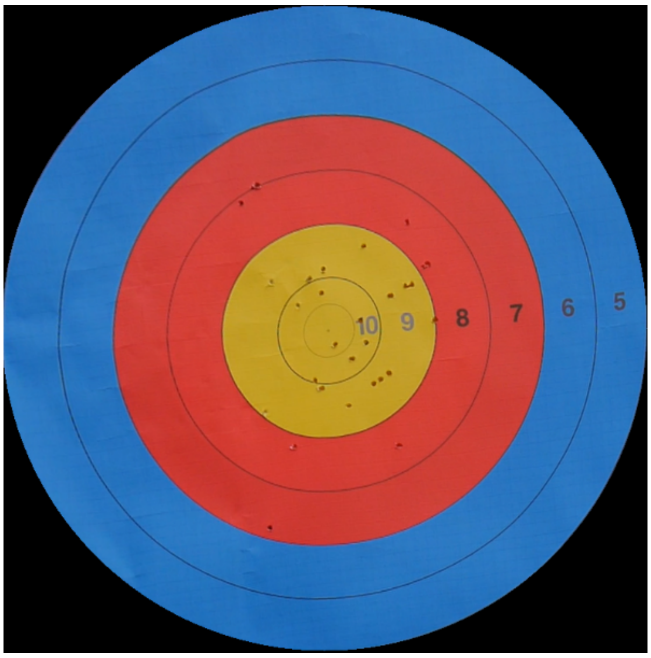


Figure 7.3: Quality of the target (*Logitech_C920_1*).

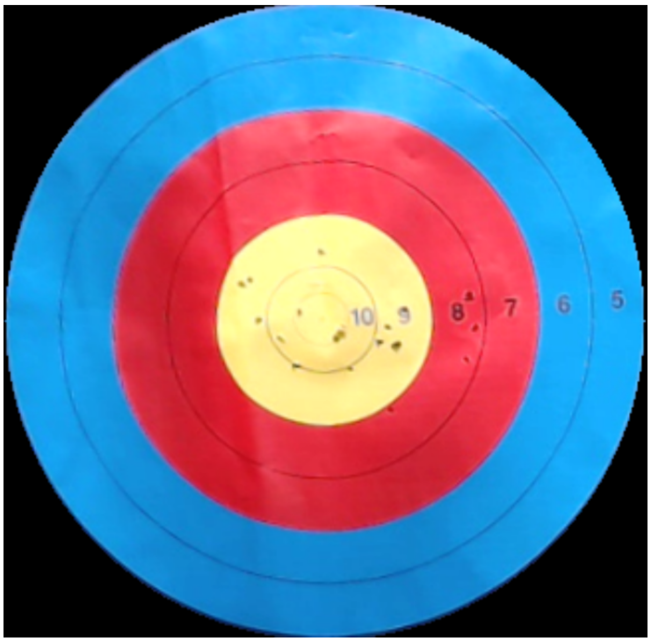
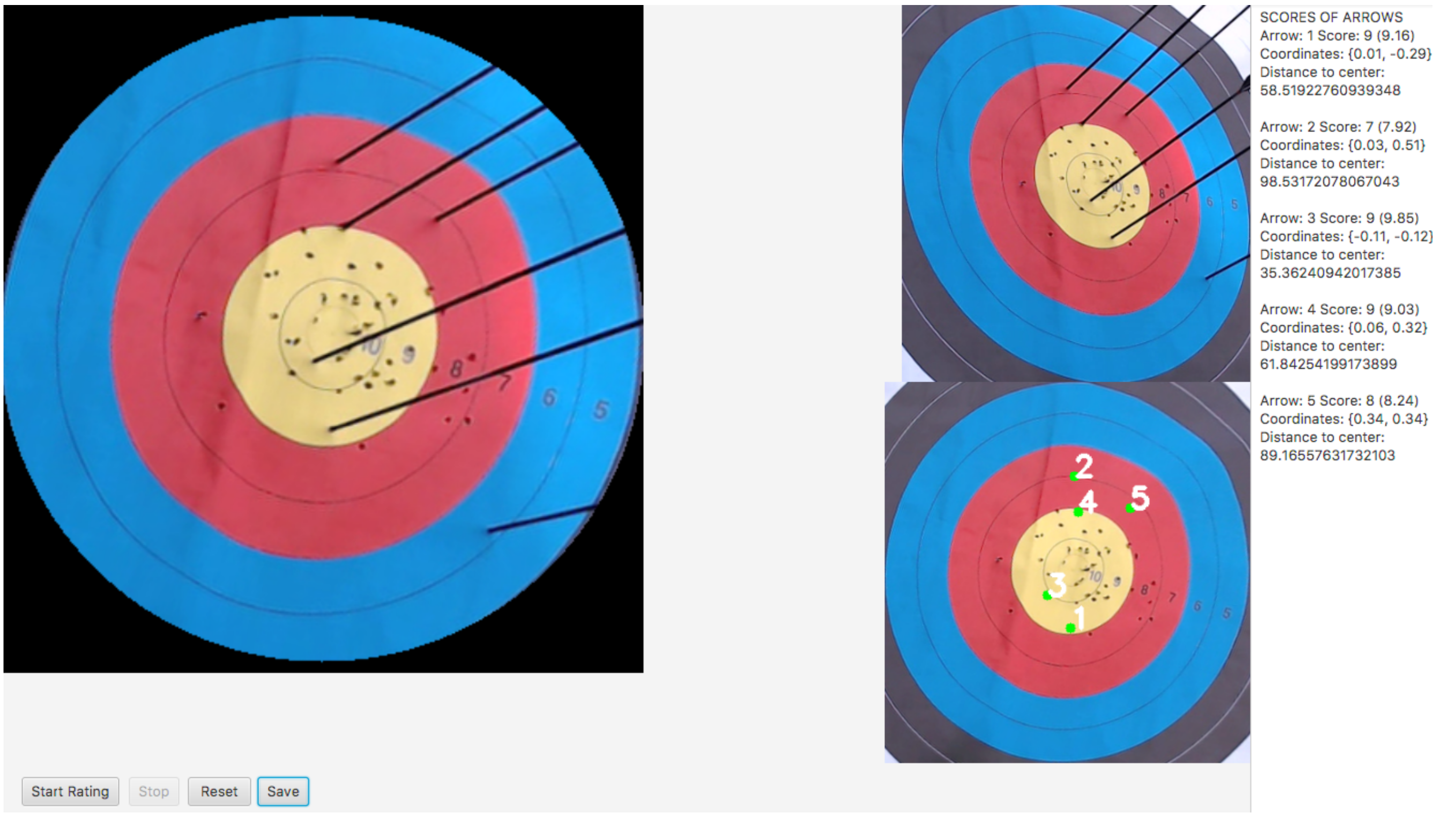
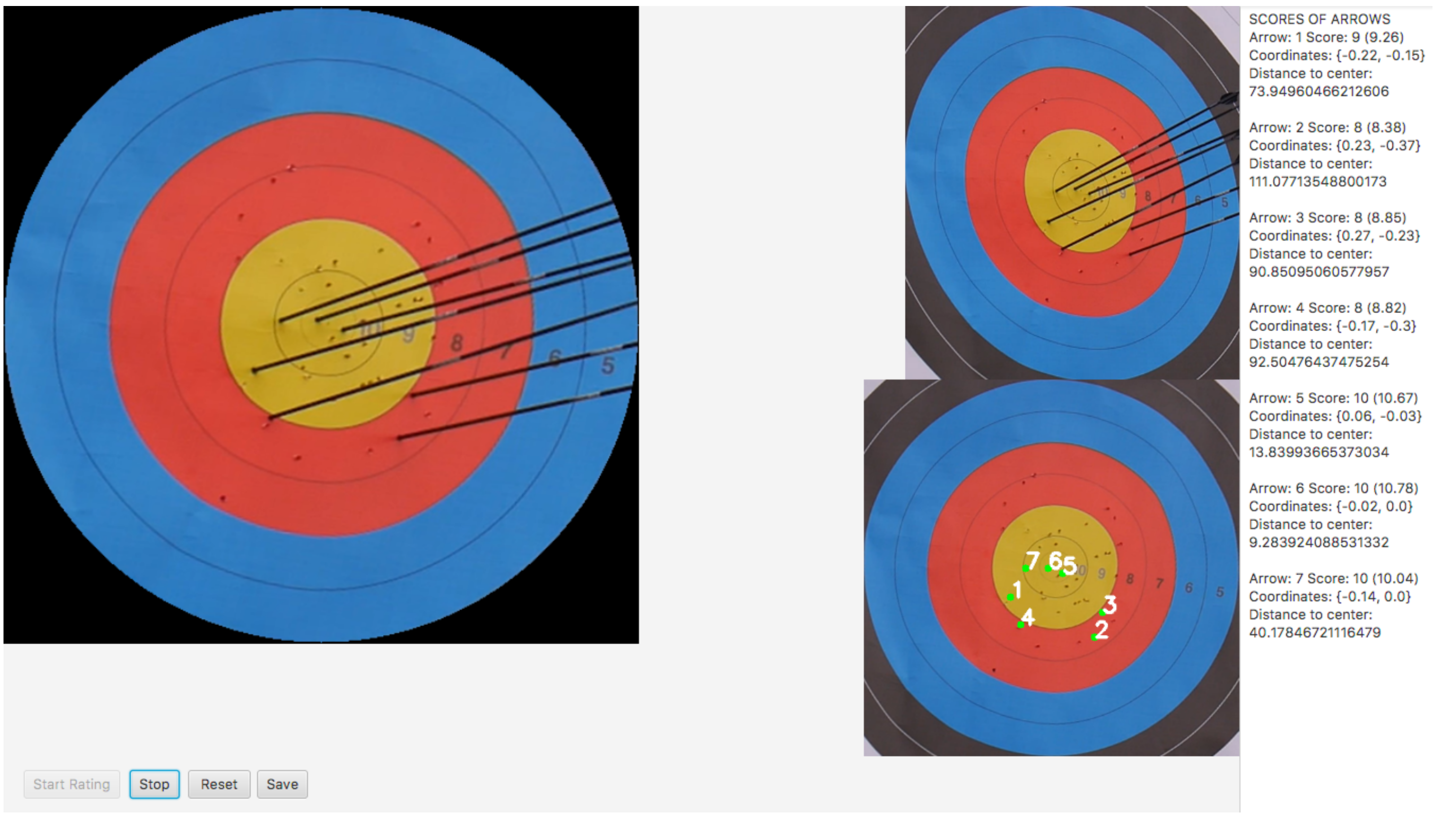


Figure 7.4: End result of *Logitech_C920_2*.



```
SCORES OF ARROWS
Arrow: 1 Score: 9 (9.16)
Coordinates: {0.01, -0.29}
Distance to center:
58.51922760939348

Arrow: 2 Score: 7 (7.92)
Coordinates: {0.03, 0.51}
Distance to center:
98.53172078067043

Arrow: 3 Score: 9 (9.85)
Coordinates: {-0.11, -0.12}
Distance to center:
35.36240942017385

Arrow: 4 Score: 9 (9.03)
Coordinates: {0.06, 0.32}
Distance to center:
61.84254199173899

Arrow: 5 Score: 8 (8.24)
Coordinates: {0.34, 0.34}
Distance to center:
89.16557631732103
```

Figure 7.5: End result of *Vid_2*.



```
SCORES OF ARROWS
Arrow: 1 Score: 9 (9.26)
Coordinates: {-0.22, -0.15}
Distance to center:
73.94960466212606

Arrow: 2 Score: 8 (8.38)
Coordinates: {0.23, -0.37}
Distance to center:
111.07713548800173

Arrow: 3 Score: 8 (8.85)
Coordinates: {0.27, -0.23}
Distance to center:
90.85095060577957

Arrow: 4 Score: 8 (8.82)
Coordinates: {-0.17, -0.3}
Distance to center:
92.50476437475254

Arrow: 5 Score: 10 (10.67)
Coordinates: {0.06, -0.03}
Distance to center:
13.83993665373034

Arrow: 6 Score: 10 (10.78)
Coordinates: {-0.02, 0.0}
Distance to center:
9.283924088531332

Arrow: 7 Score: 10 (10.04)
Coordinates: {-0.14, 0.0}
Distance to center:
40.17846721116479
```

Chapter 8

Conclusion

This master thesis project was a continuation of the research project. This project was initiated by NHB to develop a cheap and mobile system that can score arrows during competitions. The focus in the research project was on creating a basic application that recognizes and scores archery shots on images. The main goal of this master project was to build further on that, by creating an application that is able to recognize and score arrows on video/realtime, based on a camera that is placed close to the target. This main focus was split into different parts in order to cope with the scope. The parts were: detection of the target, perspective correction of the target and sequential arrow detection and scoring. For this project, the programming language Java was used in combination with the computer vision library OpenCV. For the detection of the target, a color-based segmentation approach was used, which was able to detect the blue, red and yellow rings of the target. For the perspective correction, the homography matrix was calculated by determining the source and destination pixels of the blue, red and yellow rings. The perspective correction was done by applying this calculated homography matrix to the frame. Afterwards the perspective of the frame was evaluated and according to the evaluation, rotated in the X and Y direction to improve the perspective correction. For the sequential arrow detection and scoring, the absolute differenced frame between frames was calculated. Based on the absolute differenced frame, arrows could be detected using the Hough Line Transform. Before scoring any arrows, several security measurements were making sure that the detected object is in fact an arrow and not some random noise or light inconsistency. The scoring of the arrows were based on a point in contour test, which checks whether a given point is inside a contour or not. At last, the position of the arrow was converted to another coordinate system, in which the coordinates made more sense and were easily readable by the archers and audience.

The functions of OpenCV used in this master thesis were adjusted according to the training dataset. This training dataset consisted of six videos and 14 images. Unfortunately due to various factors, including distance, we were not able to obtain any new videos for the experiments on time. Therefore we conducted the experiments on the training dataset videos instead. It should be noted that because of this, the results in Table 7.1 should be interpreted with a grain of salt. Even though we used the training dataset in the experiments, we still managed to observe some important things. The target detection worked solid. This was tested on the training dataset, but we also tested this by pointing the Logitech C920 at target images. The scoring in each region worked well. The main scores were correct, however the decimal calculated scores could be more accurate. This was a matter of always scoring the arrow from the inside instead of the outside. A high resolution quality camera could contribute towards more accuracy of the arrow position. In the experiments, we had one case where an arrow was not detected because it was too short. Not much could be done about this, since the length parameter in `HoughLinesP()` was already set very low. Furthermore we observed that arrows that hit the target during a huge brightness shift were not detected due to a safety threshold, which was essential. At last it was observed that when an arrow hits the target, the paper moves a bit. In one of our videos this movement of paper was so big, that it extended the arrow (since the program connects pixels that are close to each other). Unfortunately, software-based not much could be done, however the target paper could be more tightened to ensure that it does not move much.

To conclude, looking at other papers we can say that in this topic area there is not much work done and there is yet so much more to discover. Most papers have very restricting assumptions about arrow detection in archery or use very expensive hardware. In this project, we tried to keep the assumptions as low as possible with relatively cheap components. The only assumptions we have is that the camera should not be perpendicular to the target and the target should have colored rings. It should be noted that the application that we are presenting in this master project is not feature complete. There are still some features and some improvements that can be implemented to make it more complete and robust, which will be discussed in future work (Chapter 9). The application can be thought of as a solid prototype, in which the basis works and various features and improvements can be build upon.

Chapter 9

Future work

The current state of the application can be described as an initial product for scoring archery shots on video. The program shows what is possible with OpenCV and also shows that relatively good results can be obtained by appropriate tweaking of parameters. In some areas there is still room for improvements, which can improve the initial product into a more complete product.

9.1 White and black rings

Besides the blue, red and yellow rings, there are also the white and black rings. Unfortunately because the color ranges of these colors are overlapping with background noise and/or the other colors, they have not been chosen in this master thesis to further process. This is of course something that can make the program more complete in the sense that all scores can be given to arrows. Currently there are no ideas in how to solve this problem. Most probably an approach based on colors would not work.

9.2 Rotation of the target

After all the methods, the numbers on the targets are not horizontal on the right side, which means that the target is not straight and is rotated. The rotation of the target is somewhat important because the rotation can create consistency across all videos, so that each coordinate system in every situation has the same orientation. This problem however is not that easily solvable. The only reference points inside the target that can be used to determine the orientation, are the numbers. Unfortunately the intensities of these numbers correspond to the intensities of the shadows and the holes, so any color/intensity-based approach would be very difficult if not possible at all.

To try to solve this problem, we tried two approaches. The first approach was a template matching-based approach. We tried to template match between numbers of different fonts and the target. This however did not detected the numbers on the target. Another try was to cut the numbers from the target and use that to template match on other images. Unfortunately, this also did not detect any numbers except on the image it was cut from. Template matching thus is not a suitable approach, since little rotations of the target, noise around the numbers or different background colors of the numbers, could result in a failure.

The second approach was to dive a bit deeper into OCR (Optical Character Recognition). The tesseract framework was used for this approach. Tesseract is an open-source OCR engine. It can be used in code but also through the command line. Tesseract contained a functionality that seemed pretty useful in our case, namely the detection of the orientation of the text (which were the numbers in our case). The initial approach was first to try it through the command line rather than coding it, before spending too much time on it. We tested it on Image 1.1, in which the numbers are visible, without any noise holes. This however did not return any numbers or orientation of the numbers. When we tried this test on text images, it detected the text and orientation without any problems. We tried to tweak the parameters of the tesseract command line program, but unfortunately it wasn't able to detect anything at all on Image 1.1 and other target images. This could be due to the fact that the numbers are on different colored backgrounds.

This rotation of the target is an interesting but difficult problem. Unfortunately due to time restrictions, not much time has been spent on this problem.

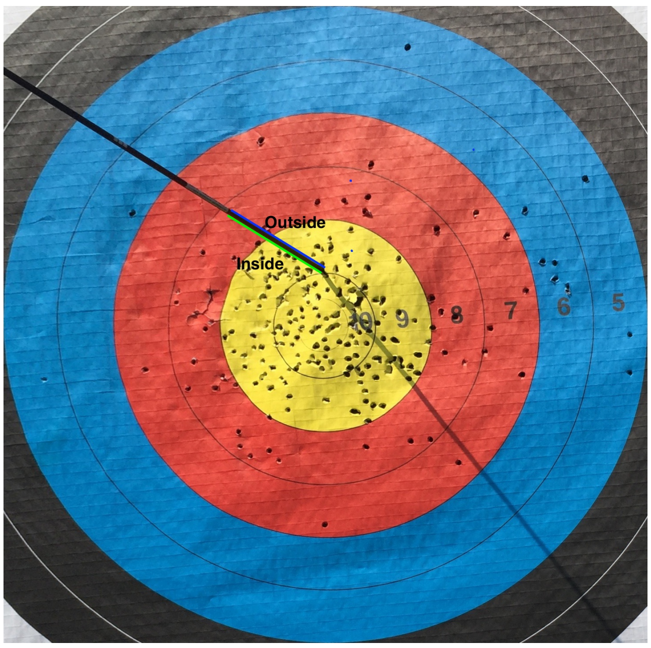
9.3 More accuracy for the arrow scoring

The accuracy of the arrow scoring is another thing that needs some improvement. Currently when an arrow is shot, the HoughLine represents the arrow and thus the score. During some initial testings at the NHB, we came to the conclusion that instead of looking at the outside of the arrow, we should look at the inside of it. Such an example can be seen in Image 9.1. In most cases, it does not matter if you look at the outside or the inside of the arrow, however when there are score differences, the NHB tends to look at the inside of the arrow. Due to time constraints there was no time spent in finding a solution. However one can easily think of one. For example, determine the position of the arrow on the target. Score the arrow on the side that is the opposite to the location it has been found on the target. So if the arrow for example is found at top right, you score the bottom left side of the arrow.

9.4 Saving the scores in a database

To make the program more complete, it would make sense to push the scores to a database. Currently the scores can already be saved to a file, however it would be much more convenient to push these scores to a database, where some statistical analyses can be run.

Figure 9.1: Inside vs outside of the arrow.



Bibliography

[1] Handboogsport nederland. https://www.handboogsport.nl. [Online; accessed 25-July-2017].

[2] Falco-eye, the future of archery. http://www.falco-eye.com/en/. [Online; accessed 25-July-2017].

[3] Scatt shooter training system. http://www.scatt.com. [Online; accessed 25-July-2017].

[4] Opencv. http://opencv.org. [Online; accessed 25-July-2017].

[5] Java. https://www.java.com/. [Online; accessed 25-July-2017].

[6] C. Nguyen and I. Lin, "Arrowsmith: Automatic archery scorer," *Department of Computer Science, Stanford University*.

[7] T. T. Zin, I. Oka, T. Sasayama, S. Ata, H. Watanabe, and H. Sasano, "Image processing approach to automatic scoring system for archery targets," in *Intelligent Information Hiding and Multimedia Signal Processing, 2013 Ninth International Conference on*. IEEE, 2013, pp. 259–262.

[8] A. Danielescu, "Ontarget: An electronic archery scoring system," *ISU Math/Stat REU Summer*, 2009.

[9] Training dataset. https://drive.google.com/file/d/0B1Ab6_kHl4OKWXFhZ2pZNUdmejg/view?usp=sharing. [Online; accessed 25-July-2017].

[10] H. Rhody, "Lecture 10: Hough circle transform," *Chester F. Carlson Center for Imaging Science, Rochester Institute of Technology*, 2005.

[11] R. C. Gonzalez and R. E. Woods, *Digital Image Processing (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.

[12] Transformation matrix. http://mathforum.org/mathimages/index.php/Transformation_Matrix. [Online; accessed 25-July-2017].

[13] Z. Li and D. R. Selviah, "Comparison of image alignment algorithms."

[14] How to use background subtraction methods. http://docs.opencv.org/trunk/d1/dc5/tutorial_background_subtraction.html. [Online; accessed 25-July-2017].

[15] Background subtraction. http://docs.opencv.org/trunk/db/d5c/tutorial_py_bg_subtraction.html. [Online; accessed 25-July-2017].

[16] Z. Zivkovic, "Improved adaptive gaussian mixture model for background subtraction," in *Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on*, vol. 2. IEEE, 2004, pp. 28–31.

[17] Z. Zivkovic and F. Van Der Heijden, "Efficient adaptive density estimation per image pixel for the task of background subtraction," *Pattern recognition letters*, vol. 27, no. 7, pp. 773–780, 2006.

[18] A. Kaehler and G. Bradski, *Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library*. " O'Reilly Media, Inc.", 2016.

[19] G. Bradski and A. Kaehler, "Learning opencv [m]," *Yu Shiqi, Liu Ruizhen. Beijing: Tsinghua University*, vol. 10, 2009.

[20] Hough line transform. http://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/hough_lines/hough_lines.html. [Online; accessed 25-July-2017].

[21] Histogram equalization. http://docs.opencv.org/2.4/doc/tutorials/imgproc/histograms/histogram_equalization/histogram_equalization.html. [Online; accessed 25-July-2017].

[22] Clahe. http://docs.opencv.org/3.1.0/d5/daf/tutorial_py_histogram_equalization.html. [Online; accessed 25-July-2017].

Appendix A

Hardware & software set-up

There are many possibilities and choices for the usage of cameras in this master thesis project. There are seven categories of cameras, DSLR cameras, photo cameras, camcorders, action cameras, webcams and IP cameras. The worst choice is to use a DSLR camera, since they are not made to stream for long durations because of the heat and are quite expensive. The camcorders, photo cameras and action cameras usually do not support streaming over USB. This way a capture card has to be bought, which would be put between the camera and the computer to grab the HDMI output. According to the NHB, the best camera choice for this project is to use IP or HD webcam cameras. The reason for those two is because they usually support streaming over USB connections and the IP cameras can be connected through a network cable. This way when the weather is undesirable, the computer can stay inside a building.

For this project, the C920 PRO HD Logitech webcam is used. This webcam provides full HD streaming with various other useful features like automatic low-light correction, automatic focus and movement detection. This webcam is connected to a laptop using the Universal Serial Bus (USB). Connecting the camera to OpenCV is done by calling the `VideoCapture()` function. The frames from the camera can be grabbed by calling the `read()` function.