Third year project

Real-time Hand Tracking

Izunna Aneke

Bsc (Hons) Computer Science and Mathematics

Supervised by
Dr. Aphrodite Galata

May 5, 2020

Abstract

In this report we will be exploring an implementation of a real-time hand tracking system using OpenCV for Python. This system will use a standard webcam and will not us other hardware like a depth sensor. The methodologies we will explore are inspired largely from [1]. As such, we will consider our system as comprising of two main parts: hand detection and hand tracking. We will explore both of these in detail. The hand detection is accomplished primarily by classifying pixels using a naïve Bayes classifier. Once we have our set of skin-coloured pixels, we implement tracking by fitting an ellipse to the skin pixels. The full details of these will be explored in the ensuing chapters. As it turns out, our method for detecting hands can also be used to detect the subject's face. As a result, our hand tracker will also be able to track faces. For brevity, we may simply refer to our system as a hand tracker as opposed to a hand and face tracker.

Acknowledgements

This was a challenging yet extremely interesting and fun project for me. For this reason I would like to start by thanking my supervisor, Dr. Aphrodite Galata. I would also like to extend a special thanks to my mathematics academic advisor and matrix analysis lecturer, Prof. Francoise Tisseur, who helped me greatly appreciate and understand the singular value decomposition of a matrix. Many thanks to my wonderful friends, both on and off my course, who helped me with gathering training examples and giving me some extremely useful insights. Thanks in particular to Karan, Dominic, Jonathan and George. Special thanks to my Mother, Father and Brothers who have supported me throughout my development. Finally, a big thanks to Hayley for always believing in me.

Impact of COVID-19

The COVID-19 pandemic caused significant disruption to my project and studies as a whole. In particular, I am not able to study or carry out work at home due to inadequate environmental conditions. As a result once it was announced that University would close, I had to address these factors before I could properly continue to work my project and outstanding coursework. As a result, I was unable to implement code to address occlusion for my project. I have the necessary knowledge to implement this, but unfortunately I did not have enough time in the end due to the impact of COVID-19. I have explored my method for addressing occlusion in section 5.2.3 along with diagrams that I have drawn. For the same reason, I was not able to further improve the runtime speed of the hand tracker with Cython to enable it to run in real-time. Instead it runs a few seconds behind real-time (see section 5.2.1 for more on this).

Contents

Chapter 1

Context

1.1 Description of hand tracking

Before we proceed we will first clearly define what hand tracking is. Doing so will allow us to define clear objectives for which our hand tracker will achieve. This motivates the following definition.

Definition 1.1.1. Hand tracker
A hand tracker is a system that detects the position of the user's hands and follows its movements.

Remark 1.1.1. Hand trackers are typically implemented using a combination of specialised hardware and software systems. We will explore some examples in section 1.2.

1.2 Current state of the art

Hand tracking and more specifically hand gesture recognition, allows for more intuitive human computer interaction systems to be built. This can used to improve the level of immersion experienced in virtual reality (VR) simulations.

In addition, hand tracking can also facilitate greater accessibility for physically impaired individuals who are unable to use current input device solutions such as a mouse and keyboard [3]. Whilst the system we will be discussing specifically achieves hand tracking, it is worth noting that an immediate extension from here is to interpret different hand gestures based on the detected and tracked hands. We will consider this type of extension to our proposed system in section 5.2.2.

A key requirement for a hand tracker is for it to be robust under occlusion. The paper, [5], explores tracking the hands of the driver of a vehicle. This is a particular situation that is often prone to occlusion due to the cluttered nature of a car's control system and the way in which drivers typical interact with these control systems.

1.3 Objectives

We now summarise the objectives that we will achieve in this report. We are building a system capable of tracking a subject's hands and face in real-time. As a result, there are two key functionalities that we need to develop. These are as follows.

1. Hand detection.
2. Hand tracking.

These are the key features that we will make up our hand tracker. In particular, the system we will be discussing in this report will fulfil definition 1.1.1. In particular, our system will be able to detect the location of the subject's hands and face. Additionally, our system will be able to follow the movements of the subject's hands and face.

Now that we have set out our objectives for our hand tracker, we will now move on to discussing the details of our implementation.

Chapter 2

Hand detection

2.1 Introduction

Before we can begin to track hands in a given video sequence, we must first detect them. We will accomplish this by taking a given frame of a video sequence and classify all pixels as skin-coloured and not skin-coloured. The details of this process are presented in the remainder of this chapter.

2.2 Colour space

When looking to classify any type of feature in a video, i.e. a sequence of image frames, we need a way to achieve this in a robust way under different lighting or illumination conditions. To help mitigate the effect that varying illumination has on our skin colour classifier, we need to careful choose our colour space. The familiar RGB colour space unfortunately does not help mitigate this as the effect of varying illumination plays a key role in the intensities of each of the three colour channels.

It turns out that an ideal colour space is one that separates the chrominance component (or colour information) from the luminance component (light intensity information) [1]. As a result, we will be using the YUV colour space. The Y component encodes luminance whilst U and V encode chrominance information for blue and red respectively.

Now that we have decided on the colour space that we will be using, we can now proceed to classifying skin-coloured pixels.

2.3 Skin colour classifier

For a given image frame from a video sequence, we will detect the subject's hands and face using a simple yet effective naïve Bayes skin colour classifier. This will be a supervised learning algorithm and as such, we will need a labelled

dataset. The main advantage for using skin colour is that this will make our model robust in situations where the camera is moving. Additionally, unlike with methods that rely on geometric models, our use of skin colour makes our model invariant under scale transformations by design. We will have an offline training phase in which we train our skin colour classsifier model by computing the probability of a pixel with particular YUV values being skin-coloured or not skin-coloured.

2.3.1 Offline training

We will now discuss how we trained our skin colour classifier offline. As already discussed, we will need a labelled dataset. We may be able to use a suitable dataset that we may find online such as the skin segmentation dataset from the UCI Machine Learning Repository [2]. Unfortunately, after experimenting with precisely this dataset, it turned out to be highly skewed towards lighter skin tones. As a result, and depending on the skin tones of the expected subjects, it is best to create our own dataset.

In order to accomplish this let us consider the following image.

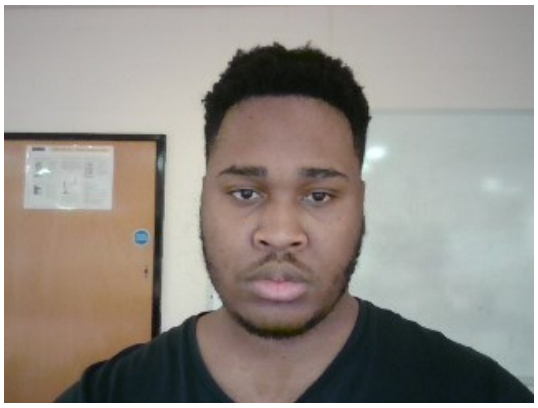

Figure 2.3.1: Training example for skin colour classifier.

The image above is one of the real training examples that we used. This image has various pixel colours. We need to manually extract each pixel and label them as skin-coloured or not skin-coloured. This idea is summarised expressed below.

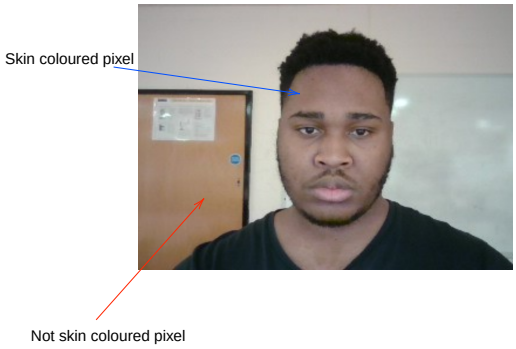

Figure 2.3.2: Labelled training example for skin colour classifier.

Our first step to doing this was to manually segment the foreground from the background. In our case, the foreground is simply the pixels that make up the skin of the subject in figure 2.3.1, whereas the background is everything else. To see this in action, let us look at the following figure.

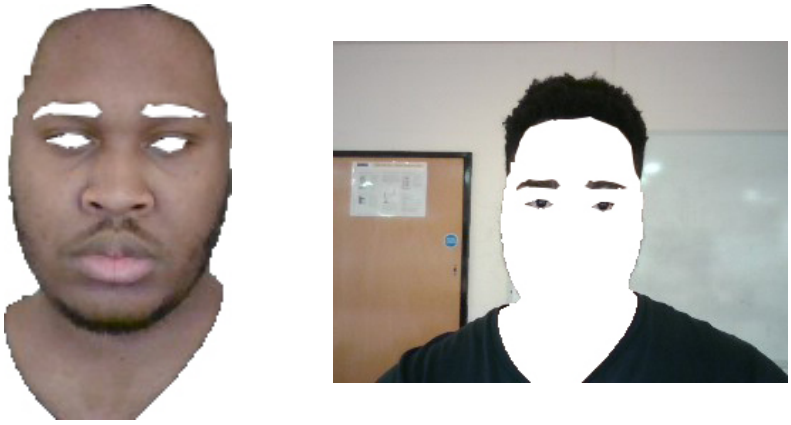

Figure 2.3.3: Skin-coloured pixels (left) and not skin-coloured pixels (right).

We have segmented figure 2.3.1 manually using Adobe Photoshop so that we are left with only skin-coloured pixels and not skin-coloured pixels as we see in figure 2.3.3.

Once we have our skin colour and background images, we now need to extract the pixels and label them as skin-coloured and not skin-coloured. In order to automate this, we have written a Python script to do this.

```python
import sys
import cv2

if (len(sys.argv) < 3):
    print("Error. Usage: python3 pixels_to_data [file_name] [class_label]")
else:
    fileName = sys.argv[1]
    classLabel = sys.argv[2]

    # Read image
    frame = cv2.imread(fileName, cv2.IMREAD_COLOR)

    # Convert image to YUV
    tmp = frame
    frame = cv2.cvtColor(tmp, cv2.COLOR_BGR2YUV)

    maxWidth = len(frame[0])
    maxHeight = len(frame)

    # open file for appending. Create file if it doesn't exist
    f = open("training_data.txt", "a+")

    for width in range(maxWidth):
        for height in range(maxHeight):
            if ((frame[height][width][0] != 255) and (frame[height][width][1] != 255) and (frame[height][width][2] != 255)):
                f.write(str(frame[height][width][0]) + "\t" + str(frame[height][width][1]) + "\t" + str(frame[height][width][2]) + "\t" + str(classLabel) + "\n")
```

Figure 2.3.4: Script for automatically labelling skin-coloured pixels.

The script shown in figure 2.3.4 takes two arguments: `file_name` and `class_label`. The `file_name` is expected to be an image file. The script reads this image file and writes each pixel line-by-line line to an output text file, `training_data.txt`, along with the `class_label` value. We will use 1 and 0 to label skin-coloured and not skin-coloured pixels respectively.

After running the script from figure 2.3.4 on the segmented images as in figure 2.3.3, we obtain a text file of the following format.

⋮	⋮
254 128 129 1	160 126 127 0
254 128 127 1	160 127 127 0
254 128 129 1	159 127 127 0
254 128 128 1	158 127 127 0
⋮	⋮

Figure 2.3.5: Offline labelled training dataset for skin-coloured (left) and not skin-coloured (right) pixels

The training data in figure 2.3.5 is a snippet from a much larger dataset. It shows the YUV pixel values followed by the class label for the pixels from our offline training dataset.

Before we begin training our model we first read the `training_data.txt` and store it in a numpy array data structure. We are using numpy arrays instead of the in-built Python lists in order to take advantage of the vast optimised mathematical operations that numpy implements. The code responsible for parsing `training_data.txt` into a numpy array is shown below.

```
import cv2
import numpy as np
import queue
import time
import math
import copy
from pathlib import Path

# Read textfile of skin coloured pixels and non-skin coloured pixels into a matrix and return the matrix
def readData():
     # Open the data textfile for reading
    with open("training_data.txt", "r") as dataFile:
        # Determine the number of rows of the matrix by checking the number of lines in the data textfile
        numOfRows = len(dataFile.readlines())
        numOfCols = 4

        dataFile.seek(0)

        # Create a numOfRows x numOfColumns matrix
        dataMatrix = np.zeros(shape=(numOfRows, numOfCols))

        # Read the first line in dataFile
        currentLine = dataFile.readline()

        currentRow = 0
        while currentLine:
            currentLineAsArray = currentLine.split("\t")

            # Replace the "currentRow" row of dataMatrix with a 1x4 vector containing B, G, R and class label values
            dataMatrix[currentRow] = [currentLineAsArray[0], currentLineAsArray[1], currentLineAsArray[2], currentLineAsArray[3]]

            currentRow += 1
            currentLine = dataFile.readline()

    return dataMatrix
```

Figure 2.3.6: Algorithm used to read `training_data.txt` into a numpy array.

The algorithm shown in figure 2.3.6 results in a numpy array of the following

form.

Array index	Y	U	V	class label
318889	254	128	129	1
318890	254	128	128	1
318891	254	128	129	1
318892	160	126	127	0
318893	160	127	127	0
318894	159	127	127	0

Figure 2.3.7: A snippet to show the structure of the `dataMatrix` numpy array.

Now that we have our labelled dataset, we now to proceed to building our skin classifier model using our `dataMatrix`. As discussed previously, our skin classifier will implement a naïve Bayes classifier to calculate the probability that a given pixel is skin-coloured or not skin-coloured. However, we must completely ignore the Y values when building our model. This is because Y represents the luminance value of a given pixel. Therefore in order to make our model not sensitive to varying illumination, we must make sure not to involve luminance in our model. Hence, the only features we will be learning are the U and V values for each pixel.

The naïve Bayes classifier computes the probability of a pixel being skin-coloured or not skin-coloured using Bayes' theorem which we restate below.

Theorem 2.3.1. *Bayes' theorem*

$P(A|B) = P(B|A)P(A)\ /\ P(B)$,
where A and B are events and $P(B) \neq 0$. *Moreover,*

- $P(A|B)$ *is a conditional probability and denotes the likelihood of event A occuring given that B is true.*
- $P(B|A)$ *is also a conditional probability and denotes the likelihood of event B occuring given that A is true.*
- *P(A) and P(B) are the probabilities of A and B occuring respectively.*

We now apply theorem 2.3.1 to allow us to classify skin colour. As a result, we arrive at the following equations for our specific use case.

$$P(\text{skin-coloured}) = \frac{P(\text{U value is skin-coloured}) \times P(\text{V value is skin-coloured})}{P(\text{U, V values are skin-coloured}) + P(\text{U, V values are not skin-coloured})} \tag{2.1}$$

$$P(\text{not skin-coloured}) = \frac{P(\text{U value is not skin-coloured}) \times P(\text{V value is not skin-coloured})}{P(\text{U, V values are skin-coloured}) + P(\text{U, V values are not skin-coloured})} \tag{2.2}$$

The denominators for equations (2.1) and (2.2) are what we use to normalise the the computed probabilities. We have used the sum, P(U, V values are skin-coloured)+ P(U, V values are not skin-coloured), because this allows us to normalise the probabilities such that P(skin-coloured) and P(not skin-coloured) sum to 1. Other normalisers exist and are also valid.

Notice that equations (2.1) and (2.2) tell us how to compute the probability that a particular pixel is either skin-coloured or not skin-coloured. These calculations rely on probabilities that we will calculate from `dataMatrix` in figure 2.3.7.

In particular, we will use `dataMatrix` to construct a matrix of statistical (MSI) information for the U and V components of each pixel. This MSI will be a 256×4 numpy array of the following form.

Array index	# skin-coloured	# Not skin-coloured	P(skin-coloured)	P(not skin-coloured)
0	*	*	*	*
1	*	*	*	*
⋮				
253	*	*	*	*
254	*	*	*	*
255	*	*	*	*

Here, #, is shorthand for 'the number of...' and, *, denotes an arbitrary value.

Figure 2.3.8: A snippet to show the structure of a matrix of statistical information constructed from `dataMatrix` in figure 2.3.7. We maintain a separate MSI for U and V components of pixels from our dataset.

Each row in figure 2.3.8 represents statistical information for a particular pixel value. Moreover, each component of YUV range from 0 to 255 and so we use a 256×4 numpy array.

The code that is responsible for calculating the MSI in figure 2.3.8 is shown below.

```
def computeStatisticalInfo(featureCol):
    skinColouredCol = 0
    nonSkinColouredCol = 1
    classCol = 3

    # Define a 256 x 4 matrix to store statistical information
    about each pixel value of a given feature

    # Each row in range [0, 255] represents a pixel value.

    # The first column represents the number of times a given pixel
     value occurs in dataset as skin coloured

    # The second column represents the number of times a given
    pixel value occurs in dataset as non skin coloured

    # The third column represents the probability that a given
    pixel value occurs in dataset as skin coloured (first column /
    total skin coloured in dataset)

    # The fourth column represnts the probability that a given
    pixel value occurs in dataset as non skin coloured (second
    column / total non skin coloured in dataset)
    matOfStatisticalInfo = np.zeros(shape=(256, 4))

    # Loop through dataMatrix and count how many times a given
    pixel value in the dataset occurs as skin coloured and non skin
     coloured
    for row in range(len(dataMatrix)):
        currentClassLabel = dataMatrix[row][classCol]
        currentFeatureValue = int(dataMatrix[row][featureCol])

        # if skin coloured
        if currentClassLabel == SKIN_COLOURED_CLASS_LABEL:
            matOfStatisticalInfo[currentFeatureValue][
    skinColouredCol] += 1

        # else if non skin coloured
        elif currentClassLabel == NON_SKIN_COLOURED_CLASS_LABEL:
            matOfStatisticalInfo[currentFeatureValue][
    nonSkinColouredCol] += 1

    return matOfStatisticalInfo
```

Figure 2.3.9: Algorithm used to compute a matrix of statistical information from `dataMatrix` in figure 2.3.7.

We run the algorithm in figure 2.3.9 in order to compute an MSI for each feature. That is, we construct a matrix of statistical information for the U and V pixel values. We then save each MSI in separate text files. This allows us to load our trained model ready for online classification without having to undergo offline training again.

2.3.2 Online classifier

We now move on and consider how we can classify a pixel from a given image frame as skin-coloured or not skin-coloured. At this stage we have computed our MSI for U and V values as we see in figure 2.3.8. Let us now suppose we are looking at an image for which we want to classify pixels as either skin-coloured or not skin-coloured. We can represent the pixels that make up such as image as image points on the xy-plane as follows.

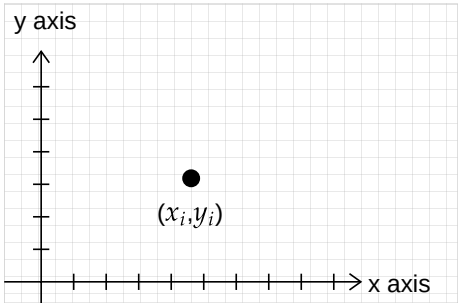

Figure 2.3.10: Image point in xy-plane.

Here (x_i, y_i) denotes the coordinates of the i-th image point. We iterate through the set of all image points and compute the probability, P(skin-coloured), of a particular image point being skin-coloured, and also the probability, P(not skin-coloured), of this image point being not skin-coloured. We compute these probabilities using equations (2.1) and (2.2) respectively.

Following this, we classify an image point as being not skin-coloured if P(not skin-coloured) $>$ P(skin-coloured). Otherwise, we will classify an image point as being skin-coloured if P(skin-coloured) $>$ P(not skin-coloured) and P(skin-coloured) $> T$ for some constant T. That is, we classify an image point as being skin-coloured if its probability of being skin-coloured exceeds its probability of being not skin-coloured, in addition to its probability of being skin-coloured exceeding a predefined threshold, T.

The purpose of T is so that we only classify an image point as skin-coloured if its probability of being skin-coloured is sufficiently high. That is, we do not want to classify image points as skin-coloured if its probability is extremely small, say 0.01, even if its probability of being not skin-coloured is even smaller than this, say 0.0001. It turns out that a good choice for T is 0.5 [1].

The code we implemented for classifying image points as described is shown below.

```
def naiveBayes:
    # P(U | skin coloured)
    probOfFeature2GivenSkinColoured = matOfStatisticalInfoFeature2[
    currentPixelFeature2, probOfSkinColouredCol]

    # P(V | skin coloured)
    probOfFeature3GivenSkinColoured = matOfStatisticalInfoFeature3[
    currentPixelFeature3, probOfSkinColouredCol]

    probOfBeingSkinColoured =  probOfFeature2GivenSkinColoured *
    probOfFeature3GivenSkinColoured * probOfSkinColoured

    # P(U | not skin coloured)
    probOfFeature2GivenNonSkinColoured =
    matOfStatisticalInfoFeature2[currentPixelFeature2,
    probOfNonSkinColouredCol]

    # P(V | not skin coloured)
    probOfFeature3GivenNonSkinColoured =
    matOfStatisticalInfoFeature3[currentPixelFeature3,
    probOfNonSkinColouredCol]

    probOfBeingNonSkinColoured = probOfFeature2GivenNonSkinColoured
     * probOfFeature3GivenNonSkinColoured * probOfNonSkinColoured

    # Compute probability normaliser
    probFeature1 = (matOfStatisticalInfoFeature1[
    currentPixelFeature1, numSkinColouredCol] +
    matOfStatisticalInfoFeature1[currentPixelFeature1,
    numNonSkinColouredCol]) / totalNumOfPixels
    probFeature2 = (matOfStatisticalInfoFeature2[
    currentPixelFeature2, numSkinColouredCol] +
    matOfStatisticalInfoFeature2[currentPixelFeature2,
    numNonSkinColouredCol]) / totalNumOfPixels
    probFeature3 = (matOfStatisticalInfoFeature3[
    currentPixelFeature3, numSkinColouredCol] +
    matOfStatisticalInfoFeature3[currentPixelFeature3,
    numNonSkinColouredCol]) / totalNumOfPixels

    probNormaliser = probOfBeingSkinColoured +
    probOfBeingNonSkinColoured #probFeature1 * probFeature2 *
    probFeature3

    # if probNormaliser = 0, then probOfBeingSkinColoured +
    probOfBeingNonSkinColoured = 0.
    # Hence just set probNormaliser = 1 to avoid division by zero
    since numerator is either probOfBeingSkinColoured or
    probOfBeingNonSkinColoured
    if (probNormaliser == 0): probNormaliser = 1

    probOfBeingSkinColoured = probOfBeingSkinColoured /
    probNormaliser
    probOfBeingNonSkinColoured = probOfBeingNonSkinColoured /
    probNormaliser

    return probOfBeingSkinColoured, probOfBeingNonSkinColoured
```

Figure 2.3.11: Naïve Bayes classifier implementation.

Notice that when calculating P(skin-coloured) and P(not skin-coloured) we are using statistical information for U and V values from our training data. As a result, what if we are trying to classify an image point whose U or V values never occurred in our training data. In this case, we will have no data to use in our classifier. This is the famous zero frequency problem for naïve Bayes.

One approach that addresses this problem is that in the event of observing a value that did not occur in our dataset, we would increment the number of occurrences for each value for U and V in our MSIs (see figure 2.3.8) by one . Following this, we would recalculate P(skin-coloured) and P(not skin-coloured) for the MSI for U and V. Finally, we would repeat the classification using our `naiveBayes` script as in figure 2.3.11.

As you can imagine, the above method for resolving the zero frequency problem has quite a bit of overhead. Since we are aiming for classification to be performed in real-time, we need to avoid or mitigate any overhead like this. In the end we simply allow P(skin-coloured) and P(not skin-coloured) to be zero in the event that we observe a pixel that was not in our dataset. We found that this did not greatly affect the efficacy of our hand tracker as we see in chapter 4.

At this stage we have successfully classified the image points of a given image as skin-coloured or not skin-coloured. However, we still have more work to do as far as hand and face detection goes. In particular, we need to perform more image processing on the classified image in order to isolate the hands and face of the subject in preparation for tracking.

2.4 Processing our classified image

In this section we will perform further image processing on the image we would have obtained from carrying out the processes discussed in section 2.3.2. In doing so, we will remove from consideration small regions classified as skin-coloured that are likely due to noise. We will then label each connected set of skin-coloured pixels as a blob. Finally, we will remove from consideration blobs of skin-coloured pixels that are under a certain size. Once all this is done, we should be left with at most three blobs that represent the subject's hands and face and we will have successfully achieved hand and face detection.

2.4.1 Hysteresis thresholding

Recall that at the end of section 2.3.2, we applied a threshold, T, that was used to only classify a given image point as skin-coloured providing its probability is greater than 0.5. As it turns out, this single threshold is not enough to achieve robust detection of skin-coloured pixels [1]. As a result, we will use a second threshold to achieve this. The use of two thresholds is known as hysteresis thresholding. This particular practice is very popular and produces good results in edge detection.

For clarity let us rewrite the threshold, T, from earlier as $T_{\max}$. Let us now denote another threshold value by $T_{\min}$ such that $T_{\max} > T_{\min}$. Now, to perform hysteresis thresholding we will iterate through all the skin-coloured pixels in an image and look at their immediate neighbours that has been classified as not skin-coloured. For such a neighbour, if its probability of being skin-coloured is greater than $T_{\min}$ then we will also classify this neighbour as skin-coloured. It turns out that a good value for our smaller threshold is $T_{\min} = 0.15$ [1]. Also recall that $T_{\max} = 0.5$.

As already discussed, hysteresis thresholding allows us to more robustly detect skin-coloured pixels. Notice that this process will grow skin-coloured regions. As a result, the idea of carrying it out is that if a particular image point has been classified as not skin-coloured then we should re-classify it as skin-coloured if this image point is a direct neighbour of a skin-coloured pixel and its probability of being skin-coloured was sufficiently high. That is, skin-coloured pixels are likely to be next to other skin-coloured pixels.

The code we wrote to carry out hysteresis thresholding is shown below.

```
def hysteresisThresholding(arrayOfSkinColouredPixels, arrayOfPixels):
    # Look at neighbouring pixels and classify as skin coloured if probability > Tmin
    for i in range(len(arrayOfSkinColouredPixels)):
        hPos = arrayOfSkinColouredPixels[i][0]
        wPos = arrayOfSkinColouredPixels[i][1]

        for w in range(wPos - 1, wPos + 2):
            if (w >= 0) and (w < maxWidth):
                for h in range(hPos - 1, hPos + 2):
                    if (h >= 0) and (h < maxHeight):
                        if  (arrayOfPixels[h, w] == NON_SKIN_COLOURED_CLASS_LABEL) and (arrayOfProb[h, w] > Tmin):
                            arrayOfPixels[h, w] = SKIN_COLOURED_CLASS_LABEL

                            arrayOfSkinColouredPixels.append([h, w])
```

Figure 2.4.1: Algorithm used to carry out hysteresis thresholding.

As already mentioned, hysteresis thresholding grows the regions of skin-coloured pixels. We can see this in action in the following.

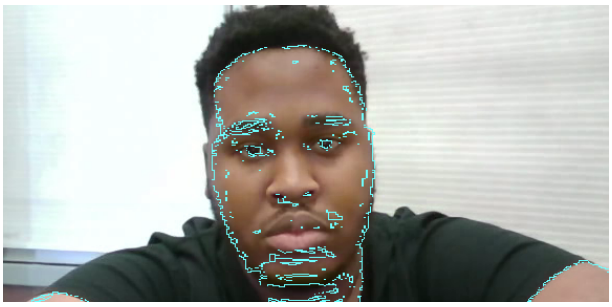

Figure 2.4.2: Hysteresis thresholding used on an image.

The blue pixels in figure 2.4.2 are pixels that have been reclassified as being skin-coloured after we have run hysteresis thresholding. As we can see, this has helped detect various skin-coloured pixels that clearly should have been initially classified as being skin-coloured. We can see this in particular on the sides of the subject's face.

Now that we have detected the skin-coloured pixels in our image, let us move on to labelling connected skin-coloured pixels are blobs.

2.4.2 Connected components labelling

Connected components labelling (CCL) is a graph theoretic process that involves relating pixels based on whether or not there exists a path that connects them. It is instructive to see this in action. First consider the following.

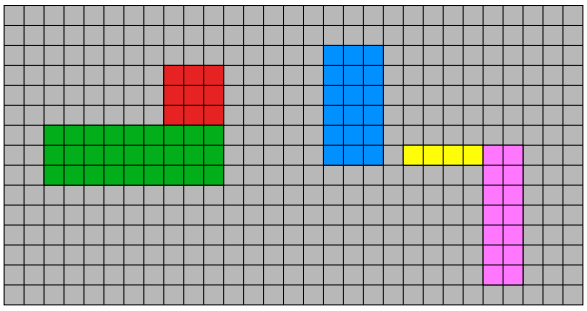

Figure 2.4.3: A grid of pixels belonging to different objects.

Figure 2.4.3 shows a grid of pixels belonging to different objects. Each object is distinguished by colour. At the moment we have five different objects. Now suppose we run a CCL algorithm on this grid of pixels to group connected objects and label them accordingly. The type of algorithm we would use is an exhaustive graph traversal. We start at a pixel in one of the objects and move

to an adjacent pixel. Each time we move to a neighbouring pixel, we add both the previously visited pixel and the pixel that we are currently on, to the same group. If we were to fully carry out this type of CCL algorithm on the grid of pixels in figure 2.4.3, we would obtain the following result.

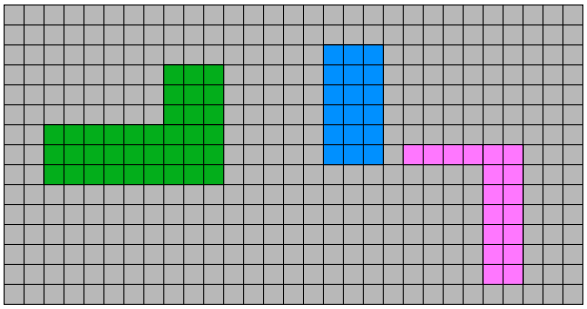

Figure 2.4.4: The grid of pixels from figure 2.4.3 after running a CCL algorithm.

Observe that we now have three groups as opposed to five as in figure 2.4.3. We now carry out this CCL process on the skin-coloured pixels that we have detected as described in section 2.4.1. By doing this, we obtain blobs of skin-coloured pixels as shown below.

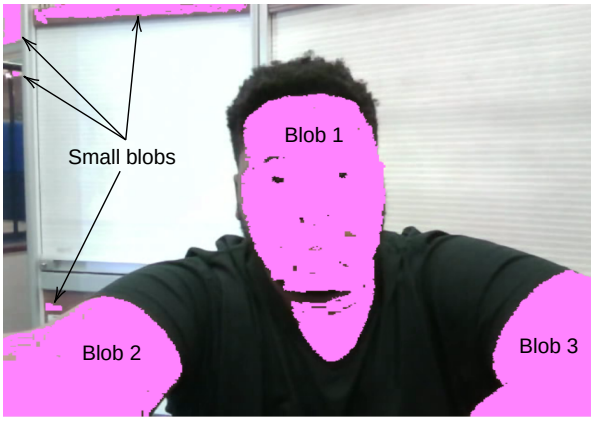

Figure 2.4.5: The result of running a CCL algorithm on our skin-coloured pixels.

We implemented an exhaustive breadth-first search algorithm to carry out CCL which we show below.

```
def connectedComponentsLabelling(arrayOfSkinColouredPixels,
    arrayOfPixels):
    arrayOfBlobs = []

    visitedFlag = np.zeros(shape=(maxHeight, maxWidth))
    index = 0
    for item in arrayOfSkinColouredPixels:
        Q = queue.Queue()

        h_item = item[0]
        w_item = item[1]

        if visitedFlag[h_item, w_item] == 0:
            Q.put(item)
            visitedFlag[h_item, w_item] = 1

            blob = []
            blob.append((h_item, w_item))

            arrayOfBlobs.append([])
            while not Q.empty():
                currentNode = Q.get()

                h_currentNode = currentNode[0]
                w_currentNode = currentNode[1]

                # Look at neighbours
                for w in range(w_currentNode - 1, w_currentNode +
    2):
                    if (w >= 0) and (w < maxWidth):
                        for h in range(h_currentNode - 1,
    h_currentNode + 2):
                            if (h >= 0) and (h < maxHeight) and (
    arrayOfPixels[h, w] == SKIN_COLOURED_CLASS_LABEL):
                                neighbourNode = [h, w]
                                if visitedFlag[h, w] == 0:
                                    visitedFlag[h, w] = 1
                                    Q.put(neighbourNode)
                                    blob.append(neighbourNode)

            arrayOfBlobs[index] = blob
            index += 1
    return arrayOfBlobs
```

Figure 2.4.6: Implementation of breadth-first search algorithm to perform connected components labelling.

Now, in figure 2.4.5, we have labelled each blob of skin-coloured pixels. In particular, we have three large blobs labelled as blob 1, 2 and 3, but we also have some relatively small blobs that we have also labelled. These small blobs are undesirable and they will often occur due to noise in the image but can also occur due to skin-coloured background objects. This leads us to the next process of size filtering in which we will remove these small blobs.

2.4.3 Size filtering

Size filtering is the process of removing the blobs of skin-coloured pixels that are too small for our purpose. In particular we will be removing blobs that are smaller than the subject's hands and face in a typical setting.

We perform size filtering by iterating through each blob. If the number of skin pixels in a particular blob is less than a certain amount, we will reclassify such pixels as not skin-coloured. In doing so, we will remove the entire blob from consideration. Experimentally we found that an appropriate threshold for this is 3,000 or 5,000. However, this value can be tuned based on how far from the camera we expect the subject to stand.

The code that performs size filtering is shown below.

```
1 def sizeFiltering(blobArray):
2     blobArray_copy = blobArray.copy()
3 
4     for i in range(len(blobArray)):
5         if len(blobArray[i]) < 5000:
6             blobArray_copy.remove(blobArray[i])
7 
8     return blobArray_copy
```

Figure 2.4.7: Implementation of size filtering.

We now run `sizeFiltering` on the image in figure 2.4.5. The result of this is shown below.

Figure 2.4.8: Size filtering the image from figure 2.4.5.

After implementing size filtering, we have now completed our implementation of hand and face detection.

2.5 Conclusion

This chapter has discussed the details of our implementation of our hand and face detection using a skin colour classifier. We also discussed various image processing techniques used to more robustly ensure we are detecting the desired skin-coloured features. Assuming a sufficiently trained skin classifier model, the method described performs rather well at detecting our desired features (see chapter 4 for more on this).

Now that we can reliably detect the subject's hands and face, we move on to tracking the movement of these features. The details of this will be discussed in the next chapter.

Chapter 3

Hand tracking

3.1 Introduction

In this chapter we will discuss how we follow the movements of the subject's hands and face in a given video sequence. Our video sequences are simply a temporal sequence of still images. Therefore a given feature may change in position over time. In addition, features in our video sequence obey spatial coherence. That is, if we take for example a hand, this feature will not move in unexpected ways. In particular, a hand moving across the scene will follow a predictable trajectory and will not spontaneously teleport to arbitrary locations.

These notions of association of time with features in an image and spatial coherence are key assumptions we make for our hand tracker. We will now make additional observations that relate to the precise method we will be using to track the subject's hands and face.

Observation 3.1.1. *(Observations for hand tracking)*

1. *The shape of the subject's hands and face can be roughly approximated by an ellipse.*
2. *An ellipse must be created to track a given feature.*
3. *An ellipse must be updated to follow the movement of the feature it is tracking.*
4. *An ellipse must be deleted for example when the feature it is tracking leaves the scene.*

Notice that observation 3.1.1 (1) is valid largely due to the convex shape that hand palms and faces for our intended subjects typically have. We will now define some mathematical objects that will be useful when exploring the details of observation 3.1.1 (2)-(4).

Definition 3.1.1. (Detected blobs)
Assume that at a time, t, there are N_b blobs detected as described in chapter

2. Now let each blob, b_j with $1 \leq j \leq N_b$, correspond to a set of skin-coloured image points that have been detected.

Definition 3.1.2. (Detected skin-coloured objects)
Let N_s be the number of skin-coloured objects detected in the scene at a time t. Also, let o_i, with $1 \leq i \leq N_s$, be the set of skin-coloured pixels that make up the i-th skin-coloured object.

Observe that the distinction here is that definition 3.1.1 describes the number of skin-coloured blobs in the scene whilst definition 3.1.2 describes the number of skin-coloured objects (or features) in the scene. Therefore if two hands crossover (i.e. occlude), there would be one blob but two skin-coloured objects. This distinction is important for making our hand tracker robust under occlusion (see section 5.2.3 for more on this).

Definition 3.1.3. (Ellipse object hypothesis)
Let $h_i = h_i(c_{x_i}, c_{y_i}, \alpha_i, \beta_i, \theta_i)$ denote the ellipse that models the i-th skin-coloured object. Here (c_{x_i}, c_{y_i}) denotes the centroid of the ellipse whilst α_i and β_i denote the major and minor axes of the ellipse respectively. Finally, θ_i denotes the angle or orientation of the ellipse.

Remark 3.1.1. Let $B = \bigcup_{j=1}^{N_b} b_j$, $O = \bigcup_{i=1}^{N_s} o_i$ and $H = \bigcup_{i=1}^{N_s} h_i$.

Now that we have set up these key definitions and observations, let us now move onto ellipse generation.

3.2 Ellipse generation

Ellipse generation is the process of creating an ellipse which roughly approximates the shape of the subject's hand palms and face. Suppose we want to fit an ellipse, $h_i = h_i(c_{x_i}, c_{y_i}, \alpha_i, \beta_i, \theta_i)$, to a skin-coloured object, o_i. We can achieve this by calculating appropriate ellipse parameters, $c_{x_i}, c_{y_i}, \alpha_i, \beta_i, \theta_i$, that will best approximate the shape of o_i.

3.2.1 Calculating the centroid of the ellipse

Calculating the centroid of h_i is relatively straightforward. Recall that from observation 3.1.1 (1), hand palms and faces are roughly convex-shaped. As a result, we can find the midpoint of such a shape by drawing a bounding box around it and calculating the midpoint of this bounding box. To see this more clearly consider the following.

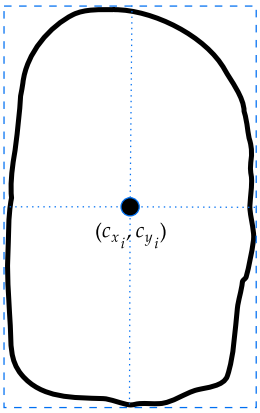

Figure 3.2.1: Calculating centroid using a bounding box.

Figure 3.2.1 shows us taking an irregular object (shown in black) and computing its centre by drawing a bounding box for the shape and using this to calculate the centroid of the shape. We employ this method for computing the centroid of our ellipse. In particular, we calculate the centre of the bounding box for a skin-coloured object. This centre becomes the centroid of the ellipse for this skin-coloured object. We omit the exact code implemented for this due to it being inherently quite straightforward.

3.2.2 Calculating the axes and angle of the ellipse

We now turn our attention to calculating the major and minor axes, α_i and β_i respectively, for our ellipse, h_i. Let us first consider the following definitions.

Definition 3.2.1. (Covariance)
Let $x = (x_1, \ldots, x_n)^\top$ and $y = (y_1, \ldots, y_n)^\top$. Then the covariance between x and y is given by,

$$cov(x, y) = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{n}.$$

where $\bar{x}$ and $\bar{y}$ denote the mean of x and y respectively.

Remark 3.2.1. Covariance is a measure of the strength between two or more variates.

Definition 3.2.2. (Covariance matrix)
Let $x = (x_1, \ldots, x_n)^\top$ and $y = (y_1, \ldots, y_n)^\top$. Then the covariance matrix for x and y is given by,

$$\Sigma = \begin{bmatrix} cov(x,x) & cov(x,y) \\ cov(y,x) & cov(y,y) \end{bmatrix} = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} \\ \sigma_{yx} & \sigma_{yy} \end{bmatrix}$$

Remark 3.2.2. Observe that $\sigma_{xy} = \sigma_{yx}$.

It turns out that if we calculate the covariance matrix of skin-coloured image points, we can use this matrix to calculate the corresponding ellipse parameters that would fit the image points [1, 7].

The idea is that the eigenvectors for the covariance matrix gives us the directions of the major and minor axes for the ellipse, whilst the corresponding eigenvalues gives us the length of the major and minor axes. Therefore by solving for the eigenvectors and associated eigenvalues for the covariance matrix, we obtain the major and minor axes for our ellipse.

It is straightforward to calculate the covariance matrix as we simply apply definitions 3.2.1 and 3.2.2. We use the `cov()` function from the `numpy` library to calculate the covariance matrix for our skin-coloured image points because `numpy` functions are optimised to run faster than native Python code.

We can calculate the eigenvalues of our covariance matrix, $\sum$, as follows.

$$\lambda_1 = \frac{\sigma_{xx} + \sigma_{yy} + \Lambda}{2}, \lambda_2 = \frac{\sigma_{xx} - \sigma_{yy} + \Lambda}{2}, \Lambda = \sqrt{(\sigma_{xx} - \sigma_{yy})^2 + 4\sigma_{xy}^2}. \quad (3.1)$$

A more general way to calculate the eigenvalues of a matrix is to perform singular value decomposition on that matrix [9]. It follows that we can calculate our ellipse parameters using the following equations.

$$\alpha_i = \sqrt{\lambda_1}, \beta_i = \sqrt{\lambda_2}, \theta_i = \tan^{-1}\left(\frac{-\sigma_{xy}}{\lambda_1 - \sigma_{yy}}\right). \quad (3.2)$$

We draw our ellipse using the `ellipse` function built into OpenCV [6]. The arguments for `ellipse` that encode the ellipse parameters are obtained from equations (3.1) and (3.2).

This concludes our discussion on ellipse generation i.e. the creation of an ellipse that roughly approximates the subject's hand palms and face. Notice that we would generate an ellipse for each skin-coloured object detected.

3.3 Ellipse tracking

Ellipse tracking is the process of following the movements of a skin-coloured object by moving its corresponding ellipse to the new position of the skin-coloured object. We achieve this by recalculating the centroid of the skin-coloured pixels along with the other ellipse parameters. The result of this is that for each successive frame, the ellipse will track the skin-coloured object. See section 5.2.3 for improvements that can be made to ellipse tracking.

3.4 Ellipse deletion

There are situations whereby we will need to delete an ellipse that was generated as discussed in section 3.2. Such a situation is when the a skin-coloured object, such as the subject's hand, leaves the scene entirely.

Notice that the case whereby we must delete an ellipse is precisely when the number of ellipses is greater than the number of skin-coloured objects in a given frame. In this case we delete an ellipse for which there are no skin-coloured pixels contained inside of it. However, we must take care not to immediately delete an ellipse once it does not contain any skin-coloured pixels. We should allow the ellipse to persist for a few frames before deletion. Doing this allows for situations of possibly poor skin colour detection.

3.5 Conclusion

We have now discussed the implementation of a working hand and face tracker using the concepts discussed in chapter 2 and 3. We will now evaluate and test the system we have implemented in order to deduce how effectively it detects and tracks skin-coloured objects.

Chapter 4

Evaluation and testing

4.1 Introduction

In this chapter we will evaluate and test the hand and face tracker that we have developed. We are particularly interested in two things.

1. How accurately a skin pixel is classified as skin-coloured.
2. How well the movement of skin-coloured objects is tracked by its corresponding ellipse.

These two points form the basis behind our testing methodology as we will discuss in the rest of this chapter.

4.2 Running the hand tracker

We ran our hand and face tracker to see how well it appears to work. The resolution of the video used was 320×240. The results are shown below

Figure 4.2.1: Snapshots from online run of hand tracker.

The subject is initially in the scene with the only skin-coloured object in the scene being his face. The face is detected, as shown by the light blue ellipse, and so he proceeds to introducing his left hand into the scene. After enough of his left hand enters the scene, it is detected. Notice that the hand could have been detected at an earlier point by relaxing the size filtering threshold used by the algorithm (see section 2.4.3).

The subject then introduces his right hand and sure enough the algorithm detects it and is able to track it. The subject removes his hands from the scene and moves his head in order to demonstrate the ellipse transforming to the shape of the face. Following this, the subject exits the scene which demonstrates ellipse deletion. Finally the subject reintroduces only his right hand into the scene which further demonstrates ellipse generation and additionally shows ellipse tracking as the ellipse once again transforms to follow the movements of the hand.

Whilst the above figure showed that the hand tracker appears to work quite well, we will need to establish more empirically, the accuracy of detection and tracking.

4.3 Accuracy of skin colour detection

In order to calculate how well the skin colour detection performs, we will need to establish our ground truth data. Our ground truth for detection will be a sequence of images in which we will manually classify and annotate where the skin-coloured pixels are. We will then run our skin colour classifier on the ground truth so that we can calculate how accurately the algorithm detects skin-coloured pixels.

The ground truth is shown below.

Figure 4.3.1: Snapshots from ground truth.

The ground truth was captured in a different environment to where the hand tracker was initially trained. This was done so that we can truly test the robustness of the system we have built. The results from running our skin colour classifier on the ground truth is shown below.

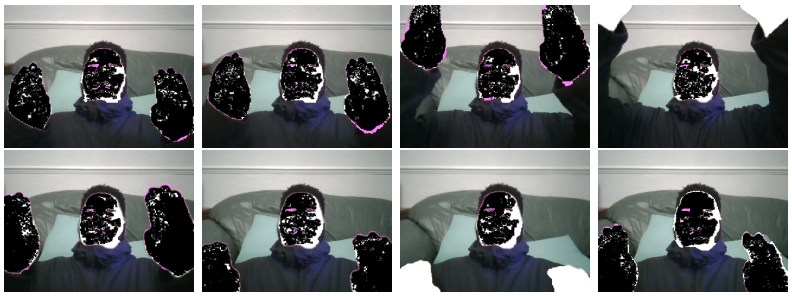

Figure 4.3.2: Snapshots showing results of detection on ground truth.

In figure 4.3.2, the white pixels represent the ground truth, the pink pixels represent the pixels that our classifier labelled as skin-coloured and the black pixels represents the intersection of former and the latter. Hence, the black pixels show us the regions where the classifier correctly detects skin-coloured pixels. The average accuracy across the frames in figure 4.3.2 is 85%, with the first few frames scoring 93% accuracy. This accuracy demonstrates the robustness of the hand and face detector as the ground truth was taken in a different environment to where our model was initially trained.

Observe that in frames 4 and 7 of figure 4.3.2, the subject's hands are not detected. This is because the number of skin-coloured pixels that make up the hands in these frames are less than 3000. This is less than our threshold for size filtering and so our detection algorithm does not detect the hands in these frames. We omitted the hand pixels from the calculation of accuracy but left them coloured white for consistency.

4.4 Accuracy of skin-coloured object tracking

We will now evaluate the performance of tracking for our ground truth data.

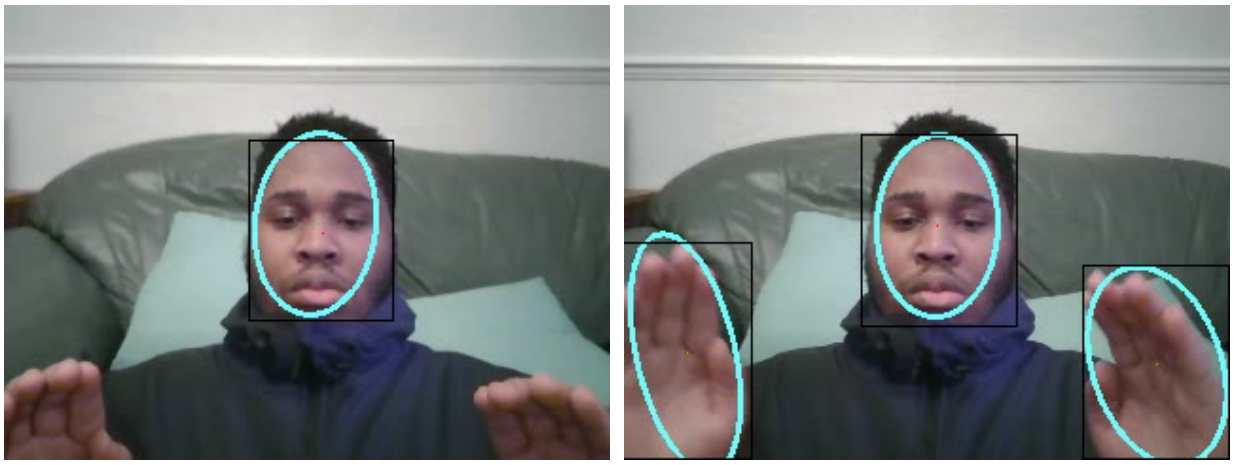

Figure 4.4.1: Snapshots showing results of skin object tracking.

In the above figure we have manually drawn the bounding box (with black pixels) for the ground truth's skin-coloured pixels (see figure 4.3.2). We have drawn the ground truth centroid for the black bounding box with a green pixel. The red pixel is the centroid drawn by our ellipse generation algorithm and the cyan is the ellipse itself. Please zoom in to see the red and green centroids more clearly. Since the red and green pixels are approximately in the same location, we can conclude that the tracker is performing as expected.

Notice that, in particular with the face, the red pixel (ellipse centroid) is farther away from the green pixel (ground truth centroid) compared to what we see in the hands. This is to due to the skin detection not classifying the entirety of the face (see figure 4.3.2). However, the difference in position is small so we can conclude that tracking is performing exactly as we expect.

This concludes our evaluation of the performance of our hand and face tracker.

Chapter 5

Conclusion

5.1 Personal reflection

Overall this project was successful. I was able to design and implement a working hand and face tracker that satisfied definition 1.1.1 along with the objectives set out in section 1.3. This project was developed using the OpenCV API for Python and achieved an 85% detection accuracy (see section 4.3). There are a lot of things I have learned throughout this project and these resulted in the successes achieved with this project. I can now take a step back to evaluate the project as a whole and comment on the effectiveness of some of the decisions I made.

To begin with, during my first meeting with my supervisor regarding this project, she advised me that I could implement the hand tracker in C++, MATLAB or any other language. At the time I had used both C++ and MATLAB for different modules in my second year as well as during my industrial placement year. In the end I decided to implement the project using Python as I had never worked on Python beforehand. My rationale was to learn as much as I could with this project. Everything was completely new to me, from computer vision as a whole to the OpenCV API and of course the Python language.

Having now completed this project and knowing what I know now, if I was to do this all again I would definitely opt for a faster programming language such as C++ instead of Python. This is because native Python is significantly slower than C++, therefore implementing the hand tracker in C++ would have allowed it to run in real-time as opposed to a few seconds behind. It turns out that the reason for this is largely down to Python being dynamically typed (see section 5.2.1 for a solution to this).

I learned that each time we try to perform an operation on a native Python variable, the interpreter must perform some checks to make sure that the operation we are trying to perform is compatible with variable in question. This becomes a big problem for us particularly because our implementation performs various pixel-by-pixel operations which often require nested `for` loops and so

the amount of checks that the interpreter carries out, increases exponentially. There are lots of benefits and conveniences pertaining to using Python, instead of say C++, for computer vision. Fortunately there is a way to improve the speed of native Python code. This is discussed in section 5.2.1.

I am very pleased with what I have achieved with this project and I am very grateful for having the opportunity work on such an incredibly rewarding project. I have thoroughly enjoyed creating this hand tracker from the ground-up and I have acquired a lot of skills and knowledge that will help me as I pursue further computer vision projects in the future.

5.2 Future work

This project is still an open research field. Furthermore, there are aspects of the hand tracker that need to be improved. Let us take a look at some of these.

5.2.1 Cython

A very effective way to speed up native Python code is by using the Python C static compiler known as Cython [4]. Cython allows us to write typing information for variables in Python. We can then compile this using Cython to convert the Python code to C code. Cython then creates a shared object file for this C code. This allows one to `import` the shared object file to any native Python project so that we can run optimised statically-typed code directly within Python. What makes Cython particularly interesting is that some tests have shown Cython to be more than 100 times faster than native Python [8].

We did not use Cython in this project, however there is a lot of potential for its power. As a result we should seek to implement Cython as a further future work for this project. Doing this would allow our hand tracker to run in real-time whilst keeping the conveniences that the Python language offers over more low-level languages like C++.

5.2.2 Hand gesture recognition

In this project we were only able to implement a hand tracker. However, we could extend this project to recognise hand gestures that could translate to different mouse operations for a computer. This could provide individuals with physical impairments a more intuitive way to interact with their computer compared to a mouse and keyboard.

A somewhat straightforward way to achieve this is by using histogram back-projection [3]. This involves us segmenting the histogram corresponding to the region of where the hand is located. From this, we can obtain peaks that correspond the the subject's fingers, assuming that their fingers are fanned out. We could then detect a *point* gesture for example if the histogram shows only one peak. A *clenched-fist* can also be detected by checking whether the major axis of the ellipse tracker is roughly halved relative to the previous frame.

5.2.3 Occlusion

Another aspect of the hand tracker that should be addressed is occlusion. This is when two distinct skin-coloured objects move such that one obscures the other from view. Suppose for example two hands across each other so as to overlap. In this situation we will have detected only one blob whereas there are actually two skin-coloured objects which should each have its own separate associated ellipse tracking it. We can see this situation below.

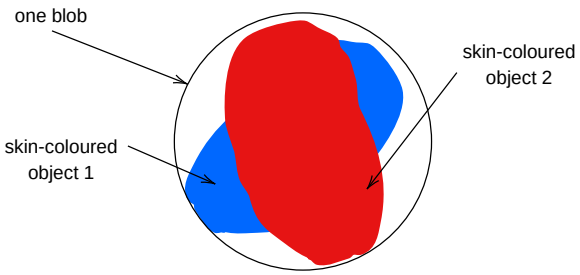

Figure 5.2.1: Two skin-coloured objects under occlusion.

To address this issue, consider the following sequence of images.

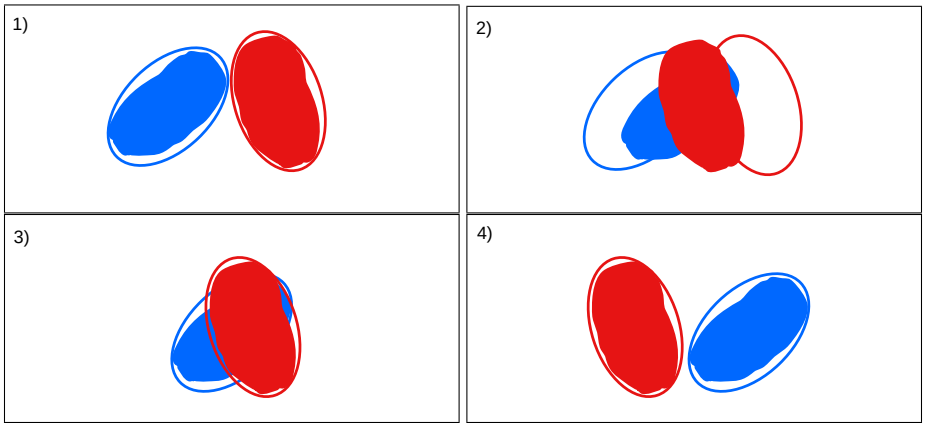

Figure 5.2.2: Addressing two skin-coloured objects under occlusion.

Occlusion can be taken into account by updating each ellipse such that it contains skin-coloured pixels that are closest to it. This can be seen in figure 5.2.2 (2)-(3). Notice that if there are skin-coloured pixels from different skin-coloured objects that are added to an ellipse, then the ellipse should be associated with only the skin-coloured object with the most amount of skin-coloured pixels contained by the ellipse. This results in tracking that is robust under occlusion.

This concludes our discussion on future work for this project. Morever, this brings us to the end of this report. We hope that you have found the results that we have presented useful and interesting.

Bibliography

[1] Antonis A. Argyros and Manolis I. A. Lourakis. "Real-Time Tracking of Multiple Skin-Colored Objects with a Possibly Moving Camera". In: *Computer Vision - ECCV 2004*. Ed. by Tomáš Pajdla and Jiří Matas. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 368–379. ISBN: 978-3-540-24672-5. DOI: https://doi.org/10.1007/978-3-540-24672-5_29. URL: https://link.springer.com/chapter/10.1007/978-3-540-24672-5_29.

[2] Rajen Bhatt and Abhinav Dhall. *Skin Segmentation Dataset*. URL: http://archive.ics.uci.edu/ml/datasets/Skin+Segmentation. (accessed: 11.10.2019).

[3] cvsisterteam. *Real Time Hand Posture/Gesture Recognition with OpenCV*. URL: https://www.youtube.com/watch?time_continue=3&v=kQxiFaZbOfA&feature=emb_logo. (accessed: 11.02.2020).

[4] Cython. *About Cython*. URL: https://cython.org/. (accessed: 11.02.2020).

[5] Akshay Rangesh, Eshed Ohn-Bar, and Mohan M. Trivedi. "Hidden Hands: Tracking Hands With an Occlusion Aware Tracker". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. June 2016.

[6] Open CV API reference. *Drawing Functions*. URL: https://docs.opencv.org/2.4/modules/core/doc/drawing_functions.html. (accessed: 11.12.2019).

[7] Cookie Robotics. *How to Draw Ellipse of Covariance Matrix*. URL: https://cookierobotics.com/007/. (accessed: 11.12.2019).

[8] Adrian Rosebrock. *Fast, optimized for pixel loops with OpenCV and Python*. URL: https://www.pyimagesearch.com/2017/08/28/fast-optimized-for-pixel-loops-with-opencv-and-python/. (accessed: 11.02.2020).

[9] Cornell University. *The Singular Value Decomposition*. URL: http://www.cs.cornell.edu/courses/cs322/2008sp/stuff/TrefethenBau_Lec4_SVD.pdf. (accessed: 11.12.2019).

Appendix A

Acronyms and abbreviations

- VR: Virtual reality.
- MSI: Matrix of statistical information.
- #: The number of a particular item.
- CCL: Connected components labelling.