Tracking of Dynamic Hand Gestures on a Mobile Platform

by

Robert Prior
B.Eng., University of Victoria, 2015

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

© Robert Prior, 2017
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

Tracking of Dynamic Hand Gestures on a Mobile Platform

by

Robert Prior
B.Eng., University of Victoria, 2015

Supervisory Committee

Dr. David Capson, Supervisor
(Department of Electrical and Computer Engineering)

Dr. Alexandra Branzan Albu, Departmental Member
(Department of Electrical and Computer Engineering)

Supervisory Committee

Dr. David Capson, Supervisor
(Department of Electrical and Computer Engineering)

Dr. Alexandra Branzan Albu, Departmental Member
(Department of Electrical and Computer Engineering)

ABSTRACT

Hand gesture recognition is an expansive and evolving field. Previous work addresses methods for tracking hand gestures primarily with specialty gaming/desktop environments in real time. The method proposed here focuses on enhancing performance for mobile GPU platforms with restricted resources by limiting memory use/transfers and by reducing the need for code branches. An encoding scheme has been designed to allow contour processing typically used for finding fingertips to occur efficiently on a GPU for non-touch, remote manipulation of on-screen images. Results show high resolution video frames can be processed in real time on a modern mobile consumer device, allowing for fine grained hand movements to be detected and tracked.

Contents

Supervisory Committee **ii**

Abstract **iii**

Table of Contents **iv**

List of Tables **vii**

List of Figures **viii**

List of Code Snippets **x**

Acronyms **xi**

Glossary **xii**

1 Introduction **1**

2 Survey of Related Work **5**

2.1 Computer Vision Algorithms on GPU or Mobile Platforms 5

2.1.1 GPU 5

2.1.2 Mobile 6

2.1.3 Other 8

2.1.4 Summary 8

2.2 Hand Tracking Algorithms 9

2.2.1 3D model-based approaches 10

2.2.2 Appearance-based approaches 11

2.2.3 Summary 15

3 A New Approach For Mobile GPU Architectures **17**

3.1 General Overview 17

3.2 Codebook-Based Background Subtraction 19
3.2.1 Overview 19
3.2.2 Codebook Generation 20
3.2.3 Codebook Background Subtraction 24
3.2.4 Parameters and Variations 25
3.2.5 Training Data 30
3.3 Image Cleaning 31
3.3.1 Overview 31
3.3.2 Basic Concepts in Mathematical Morphology 31
3.3.3 Compound Operations 32
3.3.4 Parameters 33
3.4 Hand Localization 35
3.5 Contour Generation 36
3.5.1 Overview 36
3.5.2 Contour Generation 36
3.5.3 Contour Thinning 37
3.6 Fingertip Detection 42
3.6.1 Neighbourhood Encoding 42
3.6.2 Determination of Fingertips from Angle 44
3.6.3 Data 46
3.6.4 Parameters 47
3.7 Fingertip Refinement 47
3.7.1 Overview 47
3.7.2 K-Means 49
3.7.3 K-Means Correction 50
3.7.4 Parameters 52
3.8 Gesture Recognition 53
3.8.1 Overview 53
3.8.2 Implementation 54

4 Evaluation **57**
4.1 Accuracy 57
4.1.1 Quantitative 57
4.1.2 Qualitative Discussion 64
4.2 Compute Performance 71

4.2.1 Overview . . . 71
4.2.2 Individual Step Compute Time . . . 73
4.2.3 CPU vs GPU . . . 77

5 Conclusions **79**

A Appendix **82**
A.1 Appendix A: Comparison of CPU and GPU . . . 82
A.2 Appendix B: GPGPU Library Selection and Terminology . . . 83
A.2.1 OpenCL . . . 84
A.2.2 Renderscript . . . 84
A.2.3 OpenGL ES . . . 84
A.2.4 Choice of Library . . . 85
A.2.5 GPGPU and OpenGL ES Terminology . . . 85
A.3 Appendix C: Hardware Architecture . . . 87
A.4 Appendix D: Parameter Summary . . . 88
A.5 Codebook Parameters . . . 88
A.6 Hand Localization . . . 90
A.7 Fingertip Detection . . . 91
A.8 Fingertip Refinement . . . 92
A.9 Summary . . . 94

List of Tables

3.1 Parameters of the Codebook Step 25
3.2 Comparison of Morphology Compute Time (ms) by Structuring Element Shape and Size 34
3.3 Neighbourhood Labeling 37
3.4 Parameters for Fingertip Detection 47
3.5 Parameters of the Fingertip Refinement Step 53

4.1 Accuracy of Each Finger With Candidate Points - Good Lighting . . 60
4.2 Accuracy of Each Finger With Candidate Points - Poor Lighting . . 62
4.3 Runtime of Each Step 71
4.4 Codebook Sub-steps 73
4.5 Moment Sub-steps 74
4.6 Contour Creation Steps 74
4.7 Fingertip Detection Sub-steps 75
4.8 K-Means Sub-steps 76
4.9 Gesture Detection Sub-steps 77

A.1 Parameters of the Codebook Step 89
A.2 Parameters of the Codebook Step 91
A.3 Parameters for Fingertip Detection 92
A.4 Parameters of the Fingertip Refinement Step 93

List of Figures

1.1 Proposed Method Example Rotation Gesture (map image credit and copyright [1]) . . . 3

3.1 Algorithm Overview . . . 18
3.2 Codebook Conceptual Representation . . . 19
3.3 Codebook Background Subtraction Simple Case . . . 24
3.4 Codebook Background Subtraction More Complicated . . . 24
3.5 Codebook Background Subtraction With Background Elements Visible in Foreground . . . 25
3.6 Codebook Background Subtraction with no Increment . . . 27
3.7 Codebook Background Subtraction using RGB Colour Space . . . 28
3.8 Performance Cost of Replacement Methods . . . 29
3.9 Performance Cost of Least Used with Differing Number of Code Elements . . . 30
3.10 Cross Structuring Element . . . 32
3.11 Example Cleaning Using Morphology . . . 33
3.12 Alternate Structuring Elements . . . 34
3.13 Output of Structuring Elements . . . 35
3.14 Comparison of Contour Generation and Thinning Methods. No thinning results (a) to (c), Zhang-Suen[38] in (d) to (f), the Kwon et al. [37] (g) to (i), Kwon et al. [37] using only the third iteration in (j) to (l) . . . 40
3.15 Example Encoding . . . 43
3.16 Example of colour coded output for the fingertip detection step . . . 45
3.17 Angle Data . . . 46
3.18 K-Means Initial Positions . . . 50
3.19 K-Means Example . . . 52
3.20 Example Move Gesture . . . 54

3.21 Example Zoom Gesture 55
3.22 Example Scroll Gesture 55
3.23 Example Rotation Gesture 56

4.1 Example Frame from Good Lighting Video 59
4.2 Good Lighting Example Frames 61
4.3 Example Frame from Poor Lighting Video 62
4.4 Poor Lighting Example Frames 63
4.5 Example Background Subtraction 64
4.6 Lowering Fingers Example 65
4.7 Recovery From Fast Motion 66
4.8 Reduced Candidate fingertip Points 67
4.9 Candidate Fingertip Point Correct 68
4.10 Incorrect Wrist Detection 69
4.11 Non-thin Contour Example 69
4.12 Fingertip Position Error versus Gesture Error 70
4.13 Timing Scale 72

List of Code Snippets

3.1 RGB Conversion to YCbCr 20
3.2 Codebook Generation 22
3.3 Finding Next Neighbour 44
3.4 K-Means Data Structure 49

Acronyms

CoG Center of Gravity

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

FPS Frames Per Second

(GP)GPU (General Purpose) Graphics Processing Unit

OpenCL Open Compute Library

OpenGL ES Open Graphics Library Embedded Systems

SIMD Single Instruction Multiple Data

SVM Support Vector Machine

Glossary

Semaphore abstract data type used to ensure only a certain number of threads can access a piece of code at a time.

OpenGL Texture an OpenGL data type which is a 1D array of pixels. Textures can have multiple formats but the one used most in this work is RGBA32F. Each pixel has 4 channels red, green, blue and alpha; each channel is a 32bit floating point number.

Thread processing thread, a sequence of instructions to be executed

Warp group of GPU cores which all execute the same instruction

Colour spaces

HSL Hue Saturation Luminosity

HSV Hue Saturation Value

RGB(A) Red Green Blue (Alpha)

YCbCr colour space with a luminance component Y, blue minus luminance component Cb and red minus luminance Cr

YUV Y luminance UV chroma

Chapter 1

Introduction

Desire for innovative ways to interact with computers has lead to the development of specialty gaming/desktop devices available to consumers, such as the Microsoft Kinect™ or Leap Motion™ sensor. These devices rely on depth sensors to cleanly and accurately segment hand regions from input images, and to detect a variety of hand gestures. However, further work is required to enable ubiquitous mobile devices such as cellphones, and tablets to perform similar actions at the same accuracy without using input from depth sensors.

Modern cellphones and tablets in addition to having high resolution cameras, have a graphics processing unit (GPU). GPUs are hardware specifically designed for displaying graphics. They have hardware support for operations necessary for displaying graphics. This hardware can include cores optimized for things like tessellation (adding or removing detail from a polygonal shape) or shading. Despite being made for drawing graphics to a screen, GPUs can be used for non-graphics. This is known as general purpose GPU (GPGPU). GPUs are a single instruction multiple data (SIMD) architecture meaning they can issue a single instruction to many threads. GPUs have a large number of cores compared to CPUs (the device used in this work had 256 GPU cores with 4 CPU cores) however each individual core has a lower clock speed and cannot execute instructions independently of other cores. GPU cores are broken into warps where every core in a warp must execute the same instruction. GPUs are well suited to tasks that are amenable to parallelization; if an algorithm is designed in such a way to make use of all the cores, a GPU implementation would likely be faster than a CPU implementation. GPUs are well suited to tasks like some image processing operations where each pixel is processed independently.

Until recent years hardware vendor support for libraries like OpenGL ES 3.1

which allowed for GPGPU programing was limited. Previously, to perform arbitrary operations GPUs data and programs needed to be structured in a way to fit into a graphics pipeline. The only programmable parts of the OpenGL graphic pipeline previously were vertex and fragment shaders. Shaders are programs which run on GPUs but vertex and fragment shaders do not support arbitrary operations; there are format requirements as these shaders are meant to be used for manipulating vertex data and outputting colour data. While it was possible to do GPGPU with these restrictions it was difficult. Modern devices, have hardware support for new libraries which allow for GPGPU programming to be done such as compute shaders in OpenGL as well as more general purpose libraries like the Open Compute Library (OpenCL). Compute shaders allow for code to be written which does not conform to the structure limitations of fragment and vertex shaders.

Since cellphones and tablets are so prevalent, it would be convenient to use their cameras to track hand gestures rather than use specialty hardware. The GPUs in these device would allow for input videos to be processed in real time on the same device used to capture the input. The motivation of this work was to create a hand gesture recognition method specifically for mobile devices.

The approach described herein provides a new input method for dynamic hand gestures, which allow a user to remotely manipulate images and documents. Using a mobile device placed on a stationary surface, and without the need for specialized sensors such as depth cameras, control is implemented via a touch-free interface running in real-time. The contributions include:

- a memory management scheme for the background subtraction method
- a novel encoding scheme used to efficiently follow a hand contour in parallel
- use of a parallel algorithm to constrain candidate fingertip points to individual fingertip points

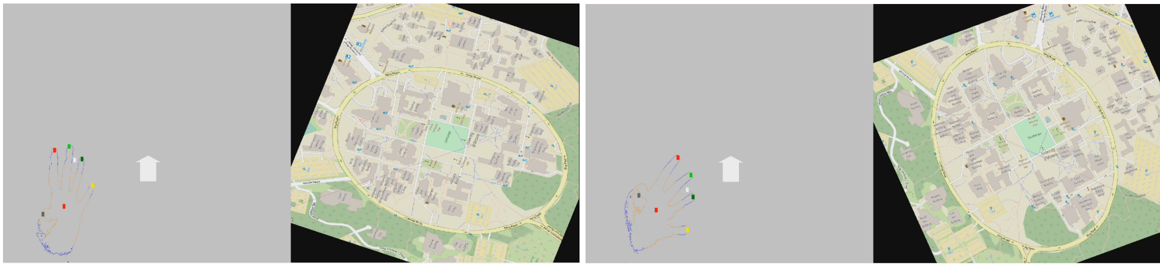


Figure 1.1: Proposed Method Example Rotation Gesture (map image credit and copyright [1])

The gestures in this work are based on how many fingers are held up. These are inspired from gestures available when interacting with laptop touch pads and touchscreens. For example, a single finger acts as a move command while two fingers act as a zoom command (used commonly in touchscreen inputs such as pinch to zoom). The gestures detected are as follows:

1 Finger Move: an on screen pointer is updated to match the finger's position

2 Fingers Zoom: an image is zoomed depending on the change in distance between the 2 fingers

3 Fingers Scroll: the average position of the fingertips is calculated and the change in location of this average position controls translational movement of an image

4 Fingers Rotate: the change in angle of the fingertips around the center of the hand is found between frames and this angle is then used to rotate an image to match hand rotation.

An illustrative example of rotating an image is shown in Figure 1.1. Different coloured boxes are drawn over the fingers showing the detected fingertip locations. The user rotates their hand between the two moments when the images in the figure are captured. The figure shows that the map images rotate by the same amount (rotation angle) as the user's hand.

The thesis is organized as follows. Chapter 2 gives a survey of recent work in the field. Chapter 3 details the new approach. Chapter 4 provides an analysis of the experimental evaluation and reports the found accuracy and compute performance. Chapter 5 concludes the thesis and gives suggestions for future work. The Appendices cover the differences between CPUs and GPUs (section A.1), terminology and libraries used in general purpose GPU programming (section A.2), a description of

the hardware used for evaluation of the proposed method (section A.3), and finally a recap of the parameters used to control the work (section A.4).

Chapter 2

Survey of Related Work

This work focused on a fingertip detection algorithm which was implemented on a mobile GPU. This related work chapter is broken down into two sections. Section 2.1 details recent work done in computer vision which has similar techniques used in this work such as detecting skin tones or making use of mobile or GPU platforms (but do not necessarily cover hand tracking specifically). Section 2.2 covers recent work on hand tracking algorithms specifically but are not necessarily implemented on mobile or GPU platforms.

2.1 Computer Vision Algorithms on GPU or Mobile Platforms

2.1.1 GPU

Szkudlarek and Pietruszka [2] implemented a head tracking system which uses both GPU and CPU to quickly process frames. Colour is used to form a degree of membership score for each pixel. The score is calculated as a dot product between the RGB pixel values and a colour filter vector. The authors note RGB colour space was selected over other spaces with more separable skin/non skin classes since no additional time to convert to other formats is needed. The use of RGB colour space to filter skin tones is interesting as other computer vision works (examples of some are presented in this chapter) determined that RGB had less separable skin tones compared to other colour spaces such as HSV or YCbCr. The score calculation involved a highly parallelizable step where each pixel had its score processed individually. These

scores were then used to determine a global centroid for the head.

Konrad [3] designed a system for combing both CPU and GPU for tracking artificial reality (AR) markers (similar to QR codes). Shape is used to detect candidate AR markers. Each marker has a black border along with a 5x5 box of squares encoding a binary pattern. Areas in the image with a clear quadrilateral shape are treated as candidate markers which are cropped and warped to be confirmed/identified. All the markers used were perfect squares so the warping undoes perspective changes to shape. The ID of the markers is found by calculating a grayscale histogram of the original marker in the original image. Otsu's method [4] (an adaptive threshold which minimizes intra-class variance in a histogram to separate it into different classes) is used to find the black/white pattern of the marker. To perform the detection and warping of the markers on a GPU, the grayscale histograms were computed and stored in an OpenGL texture.

2.1.2 Mobile

Li and Wang [5] developed a method for real-time head tracking which uses three detectors trained with local binary patterns. The authors tried to specifically deal with some of the problems inherent to mobile platforms like large pose variations due to the camera not being fixed, as well as limited processing power. The algorithm follows the detect-then-track methodology where the expansive detection algorithm is used as an initialization to a quick tracking module. The detectors operate on 18x18 windows giving a 324-dimensional feature vector with one LBP value for each pixel in the window. The system was run on a Nokia N95 which has a 322MHz CPU; the detection phase took at most 0.5s to find a face while the tracking (where fewer features are used) was 20 fps for a single face.

BulBul et al. [6] tried to create a real-time face tracking algorithm to use head motion as an input to a mobile application. The authors made use of multiple colour spaces (HSL and RGB) to separate face from background which were used in a series of separators. HSL's light values are used to find image areas which are homogeneous over time to mark pixels as background. The hue value is used to find the most common values (this assumes face takes up majority of camera frame). RGB is used in multiple discriminators such as the difference between red and green channels of skin which typically lies with in the 1:1-3:1 range. RGB second order derivatives are also used to find homogeneous regions as well. The centroid of any remaining pixel

is considered to be the face position and was used in an application to pan images.

Wang, Rister and Cavallaro [7][8] tested the capabilities of OpenCL on mobile devices by implementing the popular computer vision algorithm scale invariant feature transform [9]. The OpenCL implementation made use of the hardware using Image2D data type which utilizes the GPU's high-performance texturing hardware meant for graphics. The authors also packed 8 bit grayscale 2x2 values into a single 32 bit RGBA pixel reducing memory accesses. This packing process alone resulted in 40% reduction in processing time. Packing values other than colour values into default data types can be used to improve the performance of GPU applications given that read operations are expensive. The authors also made a OpenGL ES version. This comes with the benefit of out of bounds checks being unnecessary. In this version the authors avoided the expensive conditionals by generating the shader kernel at run time. A set of kernels is generated dependent on the size of the input image and which level of the pyramid is being processed. These kernels use un-rolled loops which, along with OpenGL ES handling boundary checks, eliminate branches. These optimizations resulted in GPU code which ran 6x faster than a CPU version.

Cheng et al. [10] performed a comparison of different implementations of computer vision algorithms running on mobile application processors (APs). Mobile AP is a general term for mobile platforms which generally consist of a multi-core CPU along with a multimedia subsystem that can include GPUs, video accelerators, digital signal processors etc. Mobile platforms have cache sizes much smaller than desktops; modern desktop processors can have 20MB available in L3 caches where on mobile higher level caches are typically in the 1-2MB range. Data would need to be rearranged from row order to fit all of a sliding window into a mobile cache. Mobile APs also suffer a larger branch penalty compared to desktop. The authors took these limitations into account and made an altered version of the Speeded Up Robust Features (SURF) algorithm [11]. The sliding window was replaced with one that used tiling and branches were removed via two methods. First a look-up table was tested which stores correlation between orientations and their corresponding histogram bins. The table removes the need for conditional expressions and does not in anyway change the functionality. The second method tested replaced the gradient histogram method with one that used gradient moments for orientation calculations. These optimizations had a 6-8x speed up compared to a naive implementation of SURF on a mobile AP.

Hassan et al. [12] implemented a face detection algorithm specifically to run on mobile GPUs using Multi-Block Local Binary Pattern. This works by generating, then

scanning, integral images for faces. An integral image allows for quick calculations of sums over rectangular areas in a grid. It is calculated by applying parallel prefix sum operation over columns in the input image then rows of the input image. The same parallel sum is computed on the output to produce the integral image. A classifier was trained offline with faces of size 24x24. The authors optimized for their hardware and noticed significant speed increases when properly aligning data and un-rolling loops or otherwise rewriting programs to avoid conditionals.

2.1.3 Other

Hemdan, Karungaru, and Terada [13] used a skin colour to find candidate face regions then look for pupils, nostrils and lip corners to track over multiple frames. In their work they cover how different colour spaces performed. The authors note that: RGB is too sensitive to lighting changes (however is significantly improved via normalization); HSL, while it has separation of luminance and chrominance making it insensitive to ambient light, is expensive; YCbCr provides a middle ground between these luminance and chrominance are separable and it is easy to convert from RGB to YCbCr. Using the YCbCr colour space, a gray-scale likelihood image is generated where the intensity of each pixel represent the likelihood of that pixel being a skin pixel. Face templates are then matched to this likelihood map and a face is considered detected if the match quality is above a threshold. This was implemented on a desktop CPU.

Vadakkepat et al. [14] tested different colour spaces for face detection. The authors state skin colours fall within a small range on the YUV colour space. Unlike RGB, changes in illumination do not drastically change the range in the UV plane where skin tones lie. YUV is not perfect for segmenting skin under particular types of lights like fluorescent which can cause flicker. YCbCr has many of the same advantages / disadvantages as YUV. The authors found the YCbCr space for skin can be bounded simply with 4 linear equations 2 of which are only dependent on a single variable.

2.1.4 Summary

This section covers a few different head tracking algorithms. As with hand tracking algorithms, a skin-tone foreground needs to be separated from background. Multiple authors state that how easily separable skin tones are from background depends on the colour space used. There is disagreement as to whether RGB or YCbCr is optimal

for skin-tones, but there is a noticeable difference between the two when performing background subtraction.

Also included are techniques used to better utilize mobile hardware. Common themes include reducing memory requirements and reducing the number of conditional operations needed by the algorithms. Both of these improve the compute performance on mobile devices. Of those mentioned Wang, Rister and Cavallaro[7] as well as Konrad [3] used a graphics library for general purpose computing. They made use of the data types provided by packing grayscale values into a data type meant for colour values. This provided inspiration for the contour encoding used in this work; arbitrary data can be packed into OpenGL data types meant for pixels.

2.2 Hand Tracking Algorithms

Hasan and Kareem[15] gave an overview of techniques worked on recently in the field of vision based gesture recognition (vision as opposed to physical sensors like accelerometers). They segmented into two broad categories: dynamic and static (this is done by other authors as well [16]). Static gestures involve no motion; examples include counting via fingers, cyclist hand signals and an "ok" sign. In these examples the final position the hand is held in matters more than the motion. Dynamic gestures involve movement and can be further broken down into many subcategories. Adapters are unconscious actions taken unintentionally by the speaker (for example hands shaking when nervous). Conscious actions are divided further and include specific terminology for different kinds of hand motion taken during speech (how hands are moved to emphasize a point).

These gestures are represented programmatically using two broad approaches: 3D model and appearance based. This breakdown is cited by other authors as well [17]. In a 3D model, a complete description of the human hand is generated. These typically include movement restrictions that a human hand does. The complete transition between hand states is tracked and the model is updated very precisely. These techniques are typically more accurate but much more computationally expensive. The 3D methods differ in how the captured 2D image is orientated to the 3D model. Geometric and skeleton models focus more on hand shape and velocities of individual joints where volumetric approaches include detailed skin information. Appearance based 2D models differ in how the gesture is recognized from the 2D input image. Colour based approaches use markers drawn on the body to aid tracking. Silhouette

models look at the shape of the entire hand and extract information like bounding box, convexity, centroid etc. to detect gestures. Deformable gabarit focuses on the hand contour. Motion-based approaches derives gestures from motion across a sequence using optical flow and other local motion techniques.

2.2.1 3D model-based approaches

Most 3D hand model-based approaches use depth cameras to construct the 3D shape of the hand. While these methods use depth cameras, which are not generally built into mobile devices, they utilize similar techniques to hand tracking algorithms which use traditional cameras. Depth cameras utilize multiple cameras to extract depth from a scene; output of the camera includes depth information for each pixel allowing for another feature to be used in tracking hands or detecting gestures.

Song et al. [18] tracked both upper body and hands. The body is separated from the background using a codebook background subtraction approach followed by a depth-cut method. The codebook method uses the colour image to try and classify foreground / background pixels. This produces a much higher resolution mask than what is capable by using the low resolution depth images. 320 x 240 depth images were captured at 20 FPS. Histogram of Gaussian (HOG) features were extracted and used as a feature vector for a multi-class support vector machine (a supervised machine learning algorithm used for classifying data) which distinguished between 4 different gestures. These gestures consisted of having 1 arm extend with either thumbs up or thumbs down as well as raising both hands above head with open and closed palms. The 1 arm raised gestures were static gestures with the 2 arm above head gestures also looked for motion.

Marin et al. [19] used a Leap Motion™ combined with a Kinect™ depth camera. The Leap Motion™ sensor provides few but accurate key points in a small view (key point consist of number of detected fingers, their position as well as the position and orientation of the palm). The Kinect™ covers a wider view and provides depth values for an entire scene but is less accurate compared to the Leap Motion™. The authors combined the key points from the Leap Motion with features from the Kinect™ such as the curvature of the hand contour as well as the distance to each point. These combined features were fed into a SVM and trained and test on a subset (10 gestures) from the American Sign Language.

Lai et al. [20] used a Kinect™ sensor to extract a hand contour. Both colour and

depth thresholds were used to extract a hand contour. Discrete curve evolution was used to simplify the contour. Fingertips are detected with turning angle thresholding.

Despite having depth data available, both Marin et al. [19] and Lai et al. [20], utilize the contour of the hand to track fingertips. Analyzing the hand contour can make fingers more distinguishable compared to strictly using depth data. Also finding fingertips by utilizing angles across the contour is a common technique as shown in the following Section 2.2.2.

2.2.2 Appearance-based approaches

Genç et al.[21] is an example of a colour-based appearance approach. The user is required to wear uniform coloured gloves (any colour not in the background) which is used during a training process to better segment the hand. The system recognized relatively unique types of queries including: spatial, motion trajectory, temporal relation and camera motion queries. Users' hands represented objects for those queries. A decision tree classifier is used to determine which gesture is being performed. The descriptors from the hand region used to determine their state were compactness, axis rotation, convexity and rectangularity. These were used to determine open / closed hands.

Mariappan et al. [22] worked to develop a hand gesture recognition system for mobile phones. A trained Cascaded Haar Classifier (CHC) performs the tracking. The CHC is trained using 3000 positive frames of clenched fists at various lighting conditions / distances as well as 3000 negative samples without a hand in them. The CHC, after training, is supplied with contrast enhanced grayscale images from a video stream and returns a vector of detected objects. On a Texas Instruments OMAP4430 Blaze Development Platform the CHC ran in 200ms.

Rautara and Agrawal[23] used HSV colour segmentation to separate hand and background. The user was required to hold their hand steady for gesture recognition to happen. The contour of the hand along with the convex hull was found. Gesture detection was then done by finding the number and direction of places where the contour was concave. The paper tested 4 gestures: only thumb out to the left/right as well as 2 and 3 fingers extended.

Pan et al.[24] tried to improve accuracy of contemporary hand gesture recognition techniques by using adaptive skin segmentation and using velocity weighted feature detection. The colour segmentation uses the YCbCr colour space and is trained to

generate skin and not skin histograms. The algorithm looks at the contour of the segmentation and finds areas with large curvatures. Each point on the contour in this method has two properties: the cosine value formed with its neighbours some K distance away, and the direction of the curve. Theses values are then thresholded to find fingertips.

Chaudhary et al. [25] worked on a method to calculate how far bent fingers were on a hand. The idea was to eventually use the tracking to control a robot hand which has human like joints / same number of degrees of freedom. As with many algorithms, captured images were converted to HSV, filtered, smoothed, binarized and all but the largest blob were removed creating a mask for the hand. Histograms of the binary image were used to further segment the image. 4 histograms were generated, each corresponding to a scan direction(left to right, up-to-down and the reverse). Wrists can be found by looking where there is a sharp inclination along the scanning direction. After the wrist is detected, finger tip detection is performed by scaling pixels based on their distance to the wrist. This is done along 1 pixel wide lines and the scaling is based on how many non zero pixels are within the line. Finger tips are generally far from the wrist and have thin scan accumulations.

Bhandari et al. [26] combined a Haar classifier and colour filtering to try and isolate a hand. Once the hand was separated from the background, the center of the palm was found. Gestures were then based on if fingers occupied a particular region around the palm. This method could distinguish which fingers were extended while the hand was moved controlling a mouse pointer.

Ahuja and Singh [17] worked on a system for quickly determining a hand gesture using principal component analysis (PCA). Fixed YCbCr colour space thresholds are used to find regions of interest. Otsu thresholding is used to minimize in-class variance within the background and skin classes and create a binary image. This image is then matched with templates created using PCA. PCA separates a set of correlated values into a smaller set of uncorrelated values (from which linear combinations can form the original values). These templates are matched with input images by calculating a weight vector for the input image then comparing it to pre-existing weight vectors from the known gestures. If the distance is less than some threshold, it is considered that gesture.

Liao, Su and Chen [27], worked on a system for gesture recognition specifically for complicated environments. The YCbCr image is thresholded and morphological open and close operators are applied to remove noise and fill in holes in the detected regions.

Component labeling was also used to remove noise by grouping pixels together. The mean and standard deviation of the candidate hand region were computed for both Cb and Cr. Only pixels which lie between two standard deviations of the mean values were kept. This separates hand and background but also creates many holes which were filled using morphological operations. A polar hand image is calculated to try and detect how many fingers are raised on the hand. This is down by taking the input image and subtracting the input image with an erosion operation applied. Then, the distance from the skin centroid of each remaining point is computed and graphed. The number of raised fingers corresponds to the thin peaks of the polar image (any wide peaks are assumed to come from the arm rather than the hand). This system was able to recognize gestures held for half a second (input video of 320x240 at 20 frames per second gesture displayed every 10 frames).

Bhame et al. [28] designed a system to perform hand gesture recognition on Indian Sign Language (ISL) digits. Here, counting the number of extended fingers is not sufficient to differentiate between gestures as for example the gestures for 2 and 7 both use 2 extended fingers (gesture for 2 uses index and middle where as the gesture for 7 uses pinkie and ring fingers). The authors segment the hand region based on maximum / minimum skin probabilities on the RGB image. As with other designs, morphological operators eliminated holes and reduce noise. After binarization of the hand image the authors compute an edge image of the hand. After finding the centroid of the remaining pixels, pixels close to the center are eliminated leaving only finger tips. By using the assumption that the palm always faces the camera, fingers are distinguished using their position relative to the centroid. The system ran on a desktop PC at six 360x280 frames per second.

Mazumdar et al. [16] developed a system for hand tracking which used the background segmentation and moments to track hand position. The user was required to wear a coloured glove to improve results. HSV with thresholds on hue were used to separate the hand from background. Frames were thrown out if the hue segmentation failed (over included background) or if the resulting binary image could not be adequately cleaned. The remaining binary image had the center of the palm located by calculating the center of mass from image moments.

Ahmed et al. [29] used 3 somewhat complicated RGB thresholds (for example $\frac{3*B*R^2}{(R+G+B)^3}$) which looked to segment face and hand regions from backgrounds. Once this was done the three larges regions were found (face and two hands). Centers of mass were found for each of these regions individually as well as the center of all

regions combined. The hand and face position relative to the combined center of mass was used as a feature vector. This was used in a discrete time warping algorithm to distinguish between 24 Indian sign language gestures.

Barros et al. [30] took a hand contour and were able to detect a set of gestures in real time. To do so the Douglas-Peucker[31] algorithm was used to minimize the contour to a simpler polygon. The convex hull is used to further reduce this polygon into a set of points classified as interior (points that correspond to space in between fingers) and exterior (points on fingertips). These points are used to train a predictor. A hidden Markov model predictor as well as a predictor based on the dynamic time warping method were tested. Seven gestures were distinguished consisting of a combination of finger extensions and waving (example gestures include index finger extension, index finger waving, 5 finger extension, 5 finger waving etc.). The system with a dynamic time warping predictor was able to process frames in roughly 65 milliseconds with a 95% class prediction accuracy.

Oyndrila et al. [32] used HSV background subtraction to segment skin regions. The convex hull of the resulting hand region was found and codified. Each point on the hull was assigned a value based on location of the next neighbour (space was separated into 8 regions based on cardinal and primary inter-cardinal directions such as north, north-east etc.). These code strings were used to differentiate between 9, 1-hand static gestures (gestures were based on static number of finger held extended).

Maqueda et. al. [33] created a new feature descriptor they dubbed temporal pyramid matching of local binary sub-patterns. The goal of this descriptor was to have low dimensionality while maintaining temporal information. Local binary patterns take a gray scale image and find for every pixel the difference between it and its neighbours. These values go through a threshold to produce a set of 8 bits (1 for each neighbour) forming the local binary pattern. This process would result in 256 (2^8) possible patterns. The authors split these local patterns into upper and lower sub-patterns each 4 bits. This results in 32 ($2^4 + 2^4$) possible patterns. The temporal portion of the descriptor averaged these patterns over a time sequence. These descriptors were used with one versus all support vector machines trained on an American sign language dataset.

Wang et al. [34] used a codebook model which allows for an efficient background subtraction. The background model consists of a collection of blobs. In each frame, if a pixel's YCbCr colour value is similar to the pixel's value at the same location on a previous frame, then it is treated as a transform on that colour; otherwise, it

is treated as a new group. The codebook consists of M code words (1 per pixel) each with N code elements made up of: 2 thresholds used for learning, 2 thresholds used during segmentation, as well as 2 variables for tracking when the code word was updated. During a training stage, every pixel is checked against the existing code words. If the pixel does not lie within the existing thresholds, a new code element is added with thresholds equal to the pixel values plus/minus a constant. Then, the thresholds for the code element are updated based on the pixel's colour value. The hand pose is calculated by determining the center of gravity as well as the orientation of the palm. The contour is also extracted and is used to locate fingertips on the hand. This is done by comparing each point on the contour to its neighbours located some distance away along the contour in both directions. Two vectors are created between the original point and these neighbouring points. If the angle between these vectors is small, then it is considered a fingertip.

2.2.3 Summary

Many hand tracking algorithms presented use colour-based segmentation to separate the hand from background. The HSV and YCbCr colour spaces seem to be the most popular, though RGB is still used. After segmentation, the works presented here fall into two broad categories, methods which use the hand contour and those which examine properties of the hand region. These algorithms vary in how specifically they are implemented but share some concepts. Pan et al. [24] and Wang et al. [34] directly use the contour where as others simplify the contour to a simpler polygon or use the convex hull in combination with the contour to find fingertips. In most of these cases the direction of the contour as well as the angle between contour points are used to distinguish fingertip regions on the contour.

In region-based approaches properties of the foreground are found and used to distinguish fingertips. These properties include convexity of regions, point density, as well as locations of the regions relative to the center of gravity of the foreground. For either of these categories, some region/contour properties are used directly to find fingertips (for example angle along the contour) or are used as a feature vector to train a machine learning algorithm.

This work in this thesis utilizes concepts from previous work including using different colour spaces for skin segmentation, extracting a contour of a hand and utilizing properties of the hand contour to find fingertips. This work specifically adopts the

codebook-based background segmentation method from Wang et al. [34] because it performs well on complex backgrounds, and is are well suited for GPU implementation since it operates on a per-pixel basis. In addition, while Wang et al. [34] used the YCbCr colour space in their work, the codebook model is colour space agnostic allowing for it to be used with any colour space.

The proposed mobile GPU implementation performs three key modifications on past work. First, for the codebook model adapted from Wang et al. [34], memory limitations on mobile devices require each pixel to have the same, bounded, number of code elements. Thus, a replacement scheme for code elements, which is used during the codebook training phase was designed. Second, the fingertip detection and contour generation are adapted to run in parallel. A novel contour encoding scheme which makes use of OpenGL ES[35] texture structure is created to allow for quick contour traversal to aid detection of fingertips in parallel. Third, a parallel implementation of K-means reduces the number of candidate fingertip locations. The gestures detected by this method are different as well. The focus is on tracking subtle movements (for example, slight changes in angle of hand) as opposed to the work of others which detect and classify larger movements (for example, a wave).

Chapter 3

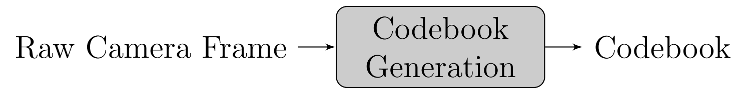
A New Approach For Mobile GPU Architectures

3.1 General Overview

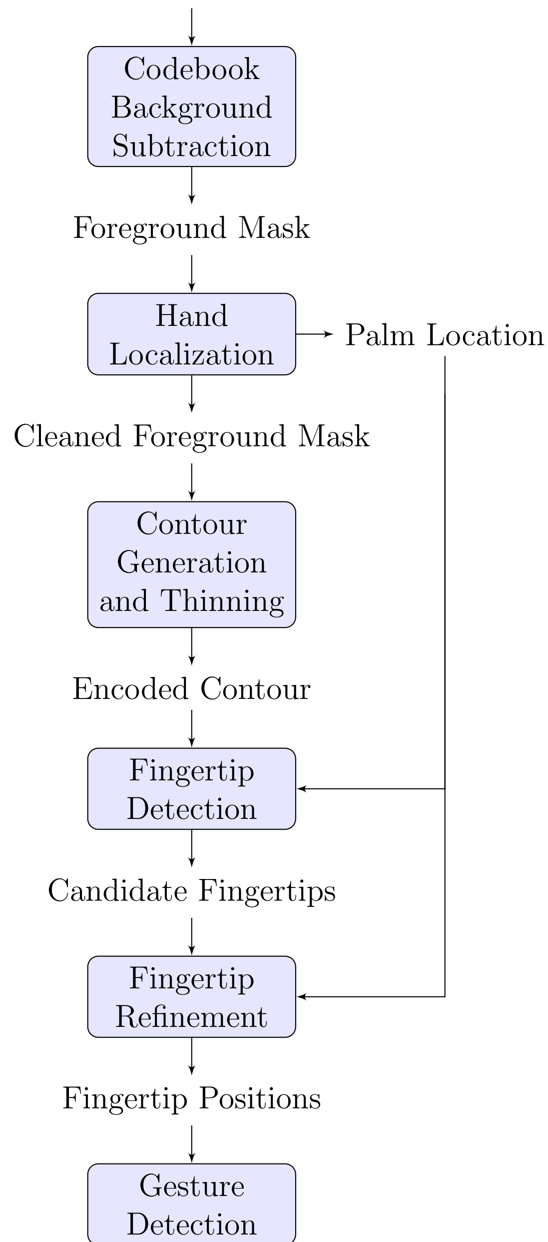
This chapter covers in detail the method and how it was implemented. Figure 3.1 shows the flowchart of the proposed approach, which involves one offline training step (the codebook generation, Figure 3.1(a)), and a computational pipeline of online steps (Figure 3.1(b)). All computations in the online pipeline are performed on a frame by frame basis. The only information conveyed from one frame to the next is the location of the detected fingertips from the previous frame; all other data is recalculated using no temporal information to reduce memory usage and improve processing speed. The only operation performed on the CPU is the final gesture detection step.

Only doing work on the GPU avoids expensive memory transfers between the CPU and GPU but requires the method to be parallel to benefit from the architecture. The majority of the inputs and outputs shown in Figure 3.1(b) are 1920x1080 images or large buffers which are infeasible to transfer every frame. Keeping the data in this format allows the GPU architecture to be utilized however as operations happen on a per-pixel basis (every thread does the same work but operate on different pixels in the image). The steps were designed in such a way so that each thread operated separately to avoid the need to synchronize threads and were designed to have few conditional operations (which are expensive on GPUs).

In the background subtraction method used each pixel is handled independently of one another. The contour generation and thinning also only uses information in



(a) Offline Steps



(b) Online Steps

Figure 3.1: Algorithm Overview

neighbourhoods around each pixel. The contour was also encoded allowing for a reduced number of conditionals during contour traversal in the fingertip detection step. The fingertip refinement step uses a clustering algorithm which can be performed in parallel. Every step was designed to exploit the hardware to allow for high resolution frames to be processed in real time.

3.2 Codebook-Based Background Subtraction

3.2.1 Overview

The first step in the algorithm is to separate the hand from the rest of the input image. Segmentation is done based on the colours in the input image. Colour segmentation has traditionally been done using thresholds on the colour space. For example, accept all pixels whose R values in a range as foreground (part of the hand). These thresholds have a minimum and maximum value which are determined empirically. Any pixel which has colour values within these threshold ranges are accepted.

The purpose of the Codebook approach is to extend these simple thresholds to handle more complicated backgrounds. For each pixel in the input, a number of code elements are trained for a particular background. Each code element essentially is a simple threshold which excludes pixels which lie within a certain range of colour values. The difference is there are multiple code elements per pixel which are determined through a training phase. Put simply, after training, the Codebook can be viewed as a number-line with certain ranges marked as background. Each individual code element post training is simply a range (just a minimum and maximum value) which marks pixels as background. There are separate code elements for each pixel in the codebook (a different number line for each pixel).

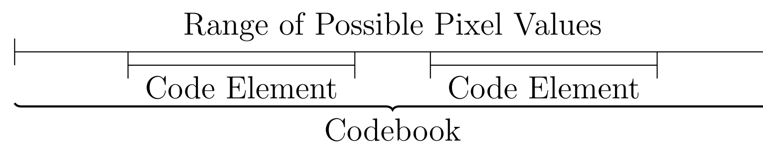


Figure 3.2: Codebook Conceptual Representation

In the training phase code elements are constructed based on the input image. For each pixel, multiple code elements are created and have their ranges set. After this is done, during the live phase, input pixels are compared to the code elements

created during training. If an input pixel has a colour value which lies outside of the range of every code element it is considered to be part of the foreground.

3.2.2 Codebook Generation

The core of each code element uses the following measure (henceforth referred to as *I value*):

$$I\ value = \sqrt{Y^2 + Cb^2 + Cr^2} \tag{3.1}$$

The magnitude of the YCbCr vector is taken for each pixel. This is used both to generate the Codebook during training and used during online processing to compare input pixels to the Codebook. First a simple shader takes converts the input from the camera (provided as an RGB OpenGL texture) and converts it into the YCbCr colour space. The conversion processes from RGB to the YCbCr colour space is a simple linear combination of the RGB values.

```
// acquire pixel from the camera texture
vec4 cameraColor =
  texture2D(camTexture, v_CamTexCoordinate);

// conv RGB in range 0-1 to
// YCbCr range Y:16-235 CbCr:16-240
float y  =  16.0 +
             cameraColor.r *  65.535 +
             cameraColor.g * 128.52  +
             cameraColor.b *  24.99;
float cb = 128.0 +
            cameraColor.r * -37.74  +
            cameraColor.g * -47.205 +
            cameraColor.b * 111.945;
float cr = 128.0 +
            cameraColor.r * 111.945 +
            cameraColor.g * -93.84  +
            cameraColor.b * -18.105;

// rescale to range 0-1 and store the new pixel value
gl_FragColor =
  vec4(y/235.0, cb/240.0, cr/240.0, 1.0);
```

Code Snippet 3.1: RGB Conversion to YCbCr

During training / codebook generation, in each frame the *I value* of each pixel is computed and then compared to the existing code elements for the pixel. If the pixel lies within the range of an existing code element, that code element is updated. Otherwise, a new one is created. Each code element records the following 5 values (each 4 byte floating point numbers):

I_High is the upper bound during code element generation

I_Low is the lower bound during code element generation. During training an input pixel is considered to lie within range of a code element if it's *I value* is greater than *I_Low* and less than *I_High*.

min the lowest *I value* recorded by this code element

max the highest *I value* recorded by this code element

tLast is a value used to determine how many frames it has been since the code element was last updated

When a code element is updated, the *I value* of the input pixel replaces the recorded minimum or maximum if the *I value* is lower/higher. Additionally, if the *I value* is close to one of the bounds (*I_High* or *I_Low*), that bound will be extended by a fixed amount.

```
// take input pixel I value and compare it to all
// code elements for this pixel
for (i = uint(0); i<numCE; ++i) {
    // if I lies within a code element update that element
    // (extends ranges adds to usage metric)
    if (codeBook.data[offset+i].Ilow < I && I <
        codeBook.data[offset+i].Ihigh) {
        updateCodeElement(offset + i, I);
        updatedCE = true;
        // guaranteed to always be smaller than any
        // actual values
        minTlast = -1.0;
    }
    // check if codebook still has empty entries
    // if no get rid of the least used one
    if (codeBook.data[offset+i].tLast == 0.0) {
        // guaranteed to always be smaller than any
        // actual values
        minTlast = -1.0;
        minIdx = i;
    } else if (codeBook.data[offset+i].tLast  < minTlast) {
        minTlast = codeBook.data[offset+i].tLast;
        minIdx = i;
    }
}
// No CodeElement matched need to add a new one
if(!updatedCE) {
    createCodeElement(offset+minIdx, I);
}
```

Code Snippet 3.2: Codebook Generation

This method differs from the method of Wang et al.[34] discussed in the related work chapter in a few significant ways. Unlike the method of Wang et al.[34], a hard limit is imposed on the number of code elements per pixel. On mobile the memory limits are much stricter. The Android OS in particular limits the amount of memory each application can use by default which is set to a certain percentage of the device's RAM. Additionally when the number of code elements increases, the space and time requirements go up. Each additional code element adds 20 bytes (5, 4 byte floating point values) per pixel as well as requiring 2 more comparisons/code branches during substitution. In this work the number of code elements used was 3

per pixel at 1920x1080 resolution. Adding more elements degraded frame rate without significantly increasing background subtraction performance.

During training, the number of stored code elements is restricted; once the limit is reached, new code elements cannot be created freely. Various replacement options were tested for when a new code element should be created but cannot due to space restrictions. One option is to do nothing and ignore new *I value*s after the maximum number of code elements is reached. This comes with the advantage of having little computation cost but this might not lead to a codebook which encapsulates the background well. Otherwise, an old code element code be replaced by a new one that uses the new *I value*. Three replacement schemes were tested:

- *least used*
- *least recently used*
- *largest negative run*

In *least used*, the code element which had the fewest pixels lie within its range is deleted and replaced. In *least recently used* the element which was updated the longest time ago (in number of frames) is replaced. Finally in *largest negative run*, the code element that went the longest time without being updated is replaced. All three visually performed equally well and showed an improvement over no replacement however, the *least used* replacement schemed had the best compute performance.

To try and reduce the space requirements, all overlapping code elements were merged every frame. If *I_Low* of one code element was in between another code element's *I_Low* and *I_High* the elements were merged. A singular code element was made using the minimum *I_Low* and *min* of the two elements as well as the maximum *I_High* and *max* of the elements. Note that this does in fact change how the algorithm functions. Two code elements could have overlapping *I_High*/*I_Low* values (which are used during training) but not have overlapping *min* and *max* values (which are used during live substitution). In most cases there is not a perceptual visual difference between scenes using merging and those without as in many cases code elements with overlapping *I_High*/*I_Low* values have overlapping *min* and *max* values. However, it is worth mentioning that adding this merging behaviour does cause loss of granularity and is not strictly correct compared to an algorithm without a merging step.

3.2.3 Codebook Background Subtraction

Once the codebook is trained on the background it remains constant and no longer changes. Background removal is done by comparing frames from the camera against the codebook. Each pixel is compared to the code elements for that particular pixel. In this stage, the pixel's *I value* is compared against the *min* and *max* of each code element (instead *I_Low*/*I_High* which were used in training). If the pixel's *I value* does not lie within the range of any of the code elements it is considered foreground. Otherwise, it is considered background and removed from further computation.

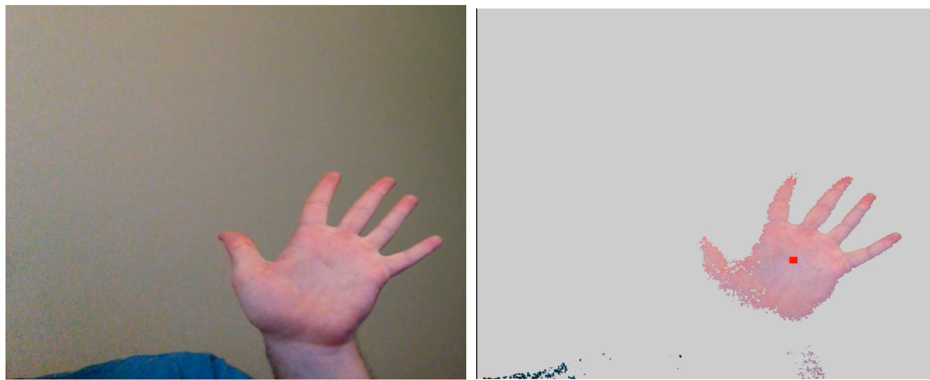


Figure 3.3: Codebook Background Subtraction Simple Case

The algorithm performs reasonable well in both simple and somewhat complicated backgrounds.

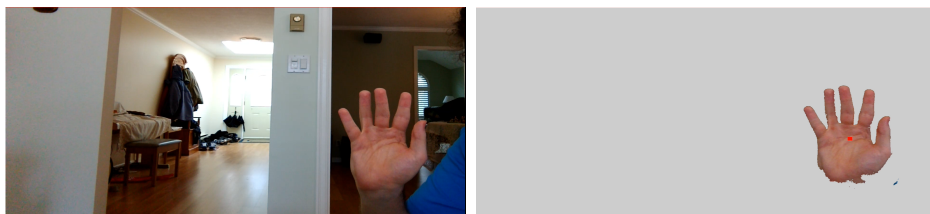


Figure 3.4: Codebook Background Subtraction More Complicated

Because each pixel has its own code elements, some structure of the background can be seen when performing background subtraction. In Figure 3.5, parts of the arm visible in frame have an *I value* close to code elements making up the window frame and blind. These pixels are pruned.

Figure 3.5: Codebook Background Subtraction With Background Elements Visible in Foreground

3.2.4 Parameters and Variations

This process uses many configurable parameters which affect how the codebook is generated. These parameters affect the width of the code element ranges and how quickly the ranges grow. They need to be optimized per background and per *I value* computation. The same parameters can be used for multiple backgrounds however this is suboptimal. In the following Table 3.1, parameter values used in this work are provided in the third column. These are for YCbCr values scaled between 0 and 1 and would need to be changed accordingly with other ranges for the colour space (for example using 0 to 255 for colour values). These values were determined empirically from testing.

Table 3.1: Parameters of the Codebook Step

Parameter Name	Description	Default Value
bounds	initial width of each code element. When a new *I value* comes in during training, the initial *I_Low* and *I_High* of the new code element are set to be *I* +/- *bounds*.	0.02

I_Inc	amount the range of a code element is extended during training when an update occurs. If the new pixel's *I value* lies within *bounds* distance to *I_Low*, then *I_Low* is decreased by this amount. Similarly, if the new pixel's *I value* lies within *bounds* distance to *I_High*, then *I_High* is increased.	0.01
minMod	during background substitution each code elements' *min* value is decreased by a small static amount	0.01
maxMod	like the minimum value, the maximum value is raised by a set same amount. These parameters widen the range of values the code elements will mark as background. This is done to try and prune as much background as possible at the cost of over pruning some foreground.	0.01
I value	the basis of the code elements. Other methods could be used to calculate this value.	$\sqrt{Y^2 + Cb^2 + Cr^2}$
Number of Code Elements	the number of code elements can also be configured. As stated previously higher values quickly degrade the compute performance at little benefit to segmentation.	3
Replacement	determines what happens when there are more unique *I values* than space to create code elements for all of them.	*least used*
Merging	Boolean value which indicates whether or not overlapping code elements should be merged	true

Figure 3.6: Codebook Background Subtraction with no Increment

These parameters individually change how the subtraction performs in subtle ways. For example Figure 3.5 was captured from the same scene as in Figure 3.3 with only the *I_Inc* parameter changed to 0 (all other parameters are the same as in Table 3.1). This change causes the code elements' ranges to not grow during training meaning more background does not get removed during live runs. Without *I_Inc* more small patches of background are included.

Figure 3.7: Codebook Background Subtraction using RGB Colour Space

Figure 3.7 shows the same scene as Figure 3.3. This image was generated by switching the *I value* to be calculated as the magnitude of the RGB colour vector instead of the YCbCr colour vector. In this colour space it is very hard to add the missing spots in the hand and arm regions without adding incorrect foreground. This figure has all parameters relating to the width of the code elements' range increased as much as possible without losing the entire arm region. Even with large code element ranges, incorrect foreground like the spots on the left side of the figure still appear. The parameter values used in this figure were:

- $minMod = 0.35$
- $maxMod = 0.35$
- $bounds = 0.50$
- $I_Inc = 0.20$

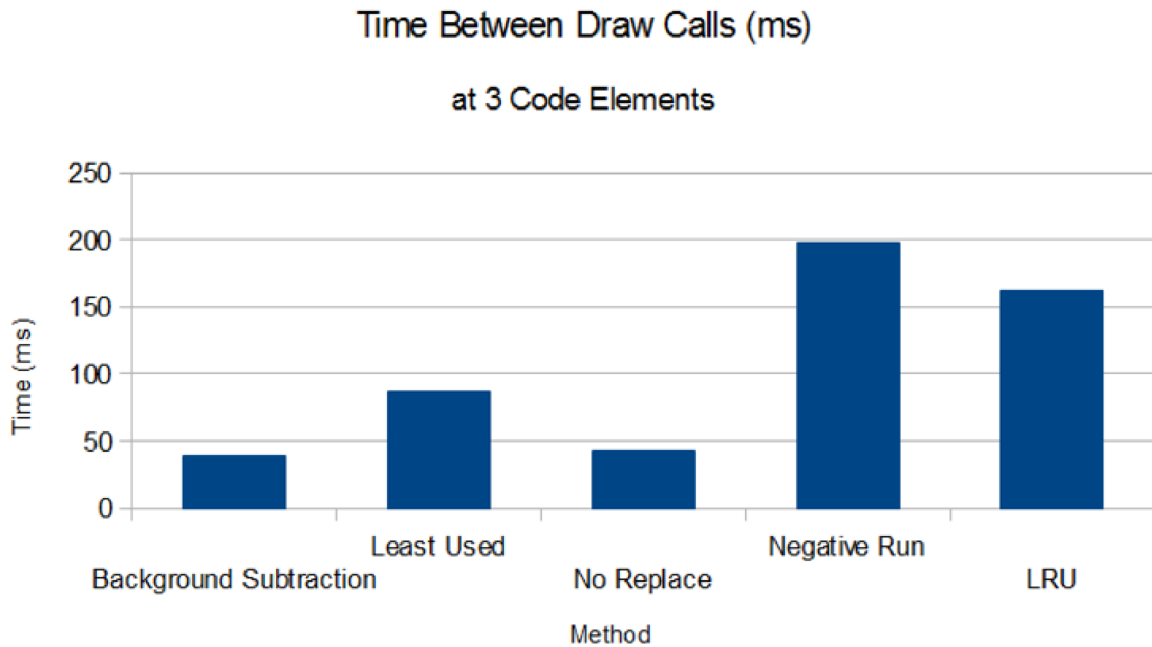


Figure 3.8: Performance Cost of Replacement Methods

The replacement scheme parameter was chosen to be *least used*. It had a noticeable improvement over performing *no replacement* however, it is comparable to the other replacement methods tested. It was chosen because it had a significantly smaller performance impact. Figure 3.8 shows the time difference between these methods. The first bar, background subtraction, is used by all training methods. The other bars show the other replacement schemes. Time is given in delay between frames so it includes all processing necessary for generating the codebook using the labeled replacement scheme (including for example getting the frame from the camera and converting it to YCbCr).

Generally speaking, the number of frames required to train a background is low (majority of testing was done using sub 100 frames to create the codebook). The performance cost of training does still matter however, as beyond a certain point the computation becomes untenable. In practice the tablet used for testing generates a lot of heat and all UI becomes unresponsive once the computation cost becomes high. For this reason, given the visual difference between the replacement methods is low (after excluding *no replacement*), the least demanding algorithm was used.

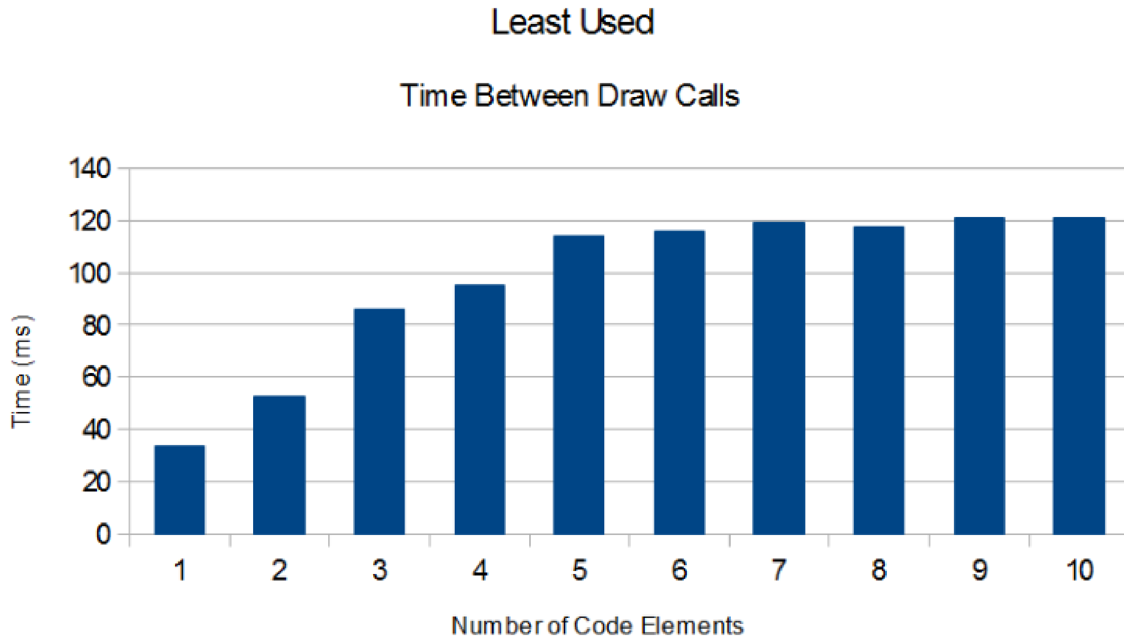


Figure 3.9: Performance Cost of Least Used with Differing Number of Code Elements

Performance cost also factored into the decision on how many code elements to use. Using more than 6 code elements the device becomes unresponsive as it did with other replacement schemes. An increased number of code elements beyond 3 code elements also did not visually improve the algorithm.

3.2.5 Training Data

Training in this work refers to a codebook being generated for a particular background. The trained backgrounds are not generalizable to other backgrounds; a codebook is generated for a specific background. The method works by training a background for a few seconds (hand not in frame) before the other, online steps start. During training code elements are created and grow to cover the range of *I values* present in the background. The codebook method works when the range of *I values* in the foreground lies outside of the range of the code elements generated.

Two environments with complicated backgrounds were evaluated (Section 4.1) one with good lighting (light sources in front of the hand) and one with poor lighting (light sources behind the hand). The codebook method can fail when the *I value* of the foreground matches the *I value* of the background. In the good lighting case, despite the background having similar appearance to the foreground (Caucasian skin

tones in front of a light wood bookcase), the background subtraction cleanly extracts the hand. In the poor lighting case the hand is closer in colour to objects in the background causing the subtraction method to struggle. Again, the resulting code elements generated for one background are not usable for another background. Each of these test environments had a separate codebook generated.

3.3 Image Cleaning

3.3.1 Overview

At this point the algorithm has produced a binary mask image generated by the codebook based on background subtraction (all background is marked as 0 all foreground marked as 1 and there are no other states). Morphology is used to clean the mask by removing elements of the background which should have not been marked as foreground as well as adding areas of the hand which were incorrectly marked as background. The simple assumption made here is that most regions incorrectly marked as foreground are small, separate, and isolated regions. Morphology can not remove large noisy regions without impacting the correct foreground. Likewise filling holes in the foreground will expand any extraneous regions separated from the hand.

3.3.2 Basic Concepts in Mathematical Morphology

Morphology works on a simple binary shape called a structuring element. Each structuring element has a center pixel which becomes relevant for morphological operations. The following figure 3.10 shows a simple cross shape structuring element. The structuring element is used in conjunction with the input binary image to produce an output binary image. The simple base operations in morphology are erosion and dilation which, when used in conjunction with each other, can clean noise from an image. Both take the structuring element and overlay the center pixel on every existing non zero pixel (equivalent to foreground pixel) in the input image. The output image is generated based off of properties of the structuring element.

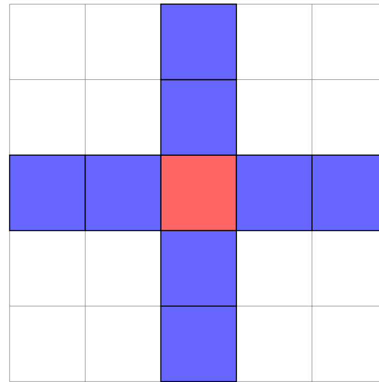


Figure 3.10: Cross Structuring Element

Dilation will add new elements to the output image based on the structuring element. For every non-zero pixel in the input image, it as well as its neighbors will be added to the output image. The neighbourhood size and shape is determined by the structuring element. With the cross structuring element, the red center pixel is overlaid over every foreground pixel. The red pixels (all input image foreground pixels) as well as the blue pixels are set to one in the output image. Any single isolated pixels (no non-zero neighbors) will become the shape of the structuring element in the output image. This is why it is important when dilating that there are very few extraneous error regions; all of these regions will increase in size after a dilation.

Erosion will decrease the number of non-zero pixels in the output image compared to the input image. Each non zero input pixel, will only be added to the output image if and only if all of its neighbours are also non-zero (as with dilation neighbourhood is determined by structuring element). Any small regions which cannot fill a structuring element are pruned. Erosion will also shrink large regions however, if the region is large enough this shrinking should be recoverable.

3.3.3 Compound Operations

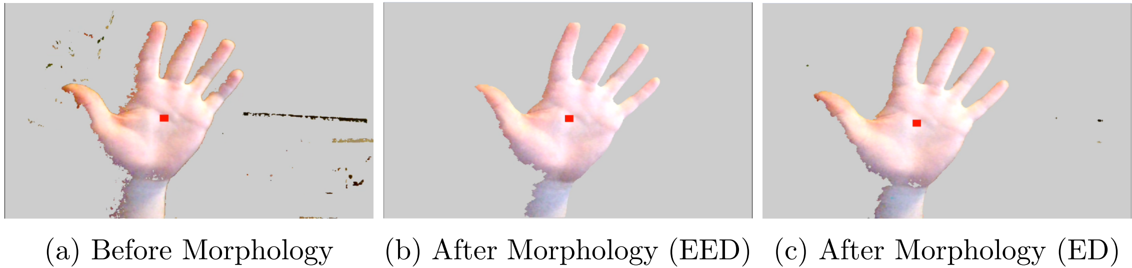
Erosion and dilation can be combined into compound operations which aim to eliminate small noisy regions while maintaining and perfecting the correct foreground region. Opening is the process of taking an input image, performing erosion with a structuring element then, performing dilation with the same structuring element. The erosion reduces the amount of small noise while shrinking large foreground regions. The following dilation recovers parts of the foreground lost by erosion.

Closing is the compound operation doing the basic operations in the reverse or-

der; dilation is performed before erosion. Closing fills in holes in structures without significantly altering their shape. During dilation if a hole in a region is filled, it will not be re-added to the shape on the subsequent erosion. The erosion will only remove border pixels that were added during the dilation.

3.3.4 Parameters

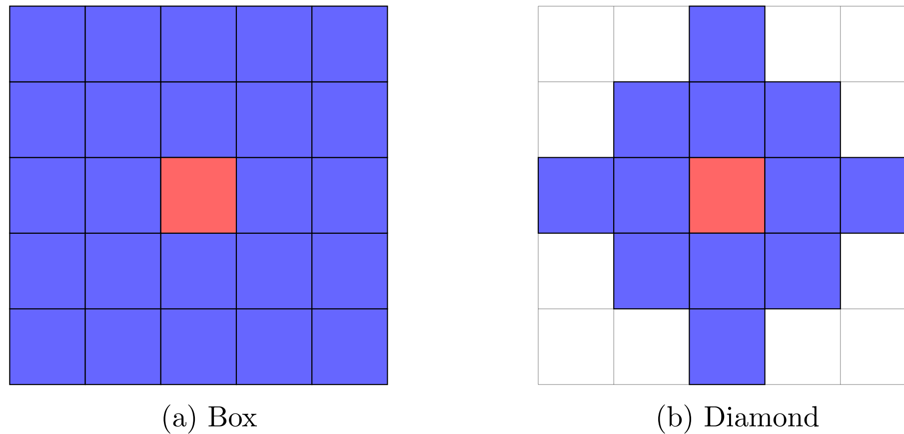
For this application an erosion followed by a opening produced good results. The majority of the noise encountered were small regions separated from the hand. This noise pattern was dependent on the codebook parameters discussed in section 3.2; it was possible to alter these parameters to produce no false foreground noise at the cost of large holes in the hand. This was not used however as, it was more difficult to recover the hand compared to removing the small separate noise.



(a) Before Morphology (b) After Morphology (EED) (c) After Morphology (ED)

Figure 3.11: Example Cleaning Using Morphology

Figure 3.11 gives an example of why two erosions followed by a dilation was used. The first image shows the original input image, the middle two erosions followed by a dilation and finally the right shows only a single erosion followed by a dilation. The right image has more noise separate from the hand. The noise in this image shows how the structuring element as well as the noise has a cross shape.



(a) Box (b) Diamond

Figure 3.12: Alternate Structuring Elements

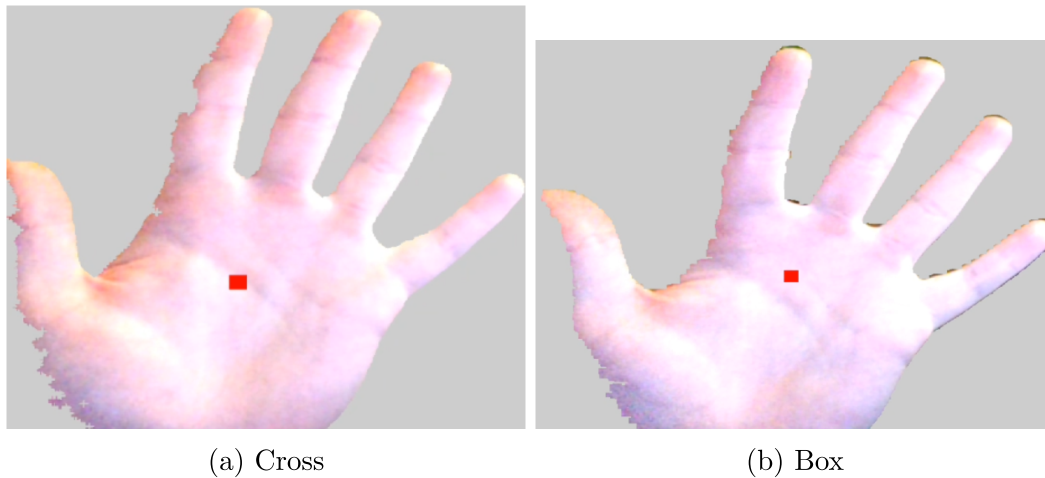
Three parameters were used to configure the morphology: the shape and size of the structuring element, as well as the combination of the erosion and dilation operations. Three shapes were tested for their compute and visual performance, the cross shape shown previously, a box shape and a diamond (shown in Figure 3.12). As with the codebook background subtraction, these morphological operations are amenable to parallelization.

Each pixel is independent (they do not depend on the result of any other pixel) and each receives the same input image. No synchronization or other concurrency mechanisms are needed. The performance impact comes from the number of memory reads/writes. For a 5x5 box a cross shape will require 9 writes (5 per line with 1 overlapping pixel in the center), a box will require 25 writes and a diamond will require 13.

	5x5	7x7	9x9
Cross	14.433 (2.04)	14.761 (1.95)	14.797 (1.74)
Box	17.877 (2.31)	19.864 (1.65)	24.102 (3.89)
Diamond	14.963 (2.33)	16.896 (2.07)	20.648 (5.56)

Table 3.2: Comparison of Morphology Compute Time (ms) by Structuring Element Shape and Size

The average compute time differences between structuring elements (along with the standard deviations in brackets) are shown in Table 3.2. All reported times are for two erosions followed by a dilation. The number of writes needed is a strong determining factor for how much time is needed to do the cleaning.



(a) Cross (b) Box

Figure 3.13: Output of Structuring Elements

For larger structuring elements, the box and diamond shape had artifacts after cleaning. Figure 3.13 shows the box structuring element adds parts of the background around the fingertips as well as in between fingers close to the palm. The side of the fingers also have jagged edges compared to the output from the cross structuring element.

A 5x5 cross shape is used to perform two erosion operations followed by a dilation. Not only is the compute time lower compared to other structuring elements, it produces smoother segmentations while still removing noise. This proved to be sufficient to remove most extraneous noise while keeping the hand shape. Again, this was determined empirically; the best parameters to use in this morphology step are heavily dependent on the parameters used during background subtraction.

3.4 Hand Localization

Cartesian image moments are used to find the center of mass of the remaining foreground pixels which corresponds to the center of the palm. Subsequent steps use the location of the palm for gesture detection. Cartesian image moments have the following form where x and y are pixel co-ordinates and $Intensity(x, y)$ is the image intensity for the pixel at x, y. In this case, $Intensity(x, y)$ is either 0 or 1 (background or foreground).

$$M_{i,j} = \sum_x \sum_y x^i y^j Intensity(x, y) \tag{3.2}$$

$M_{0,0}$ for a binary image is the number of foreground pixels in the image. Using $M_{1,0}$ and $M_{0,1}$ combined with $M_{0,0}$ can be used to find the center of gravity (CoG) or average co-ordinates of the foreground pixels. The average x position is given as $\bar{x} = \frac{M_{1,0}}{M_{0,0}}$ and the average y position as $\bar{y} = \frac{M_{0,1}}{M_{0,0}}$. These give an accurate estimate for where the center of the palm of the hand is in the image. This average is heavily skewed by foreground pixels located far from the hand as well as the arm. The position of the palm/CoG is used in subsequent for initial guesses of where fingertips are located in the image as well as performing additional image cleaning steps. To further reduce any noise in the image, all foreground pixels farther than a certain distance away from the CoG were removed.

3.5 Contour Generation

3.5.1 Overview

At this stage the mask has been cleaned both using morphology and by eliminating foreground far from the average position of the foreground pixels. To find the fingertips the boundary of the hand is traced. The goal of this step is to provide a contour usable by to the fingertip detection step with the objective of creating a single pixel width contour. In an ideal case, the contour will be exactly 1 pixel wide and fully connected (no breaks). The contour tracing creates a neighbourhood map which can fail if the contour is not 1-pixel thin or if the contour has breaks. This will be discussed further in section 3.6. This section covers the methods tested to generate and further thin the contour. It should be noted generating a thin border in parallel is not trivial. OpenCV (open computer vision), a popular computer vision library, uses an older algorithm which relies on raster scan / border following which cannot be easily parallelized.

3.5.2 Contour Generation

Simple

This method simply checks each foreground pixel for the number of pixels in the neighbourhood for pixels which are also foreground. Two variations were tested: one which tested the V8 neighbourhood (all surrounding pixels including diagonals) and one which tested the V4 neighbourhood (orthogonal directions only excluding diag-

onals). The V4 neighbourhood test will eliminates neighbours which do not have 2 or 3 foreground neighbours. Pixels which have a single foreground neighbour are isolated and cannot be part of a continuous contour and pixels which have 4 foreground neighbours are interior pixels not on the border. Similarly, the 8 neighbourhood test will eliminate any pixels which do not have between 2 and 7 (inclusive) foreground neighbours. Again, this is meant to eliminate internal and isolated parts of the mask.

Leite[36]

Morphological operations can also be used to generate a contour border. The intersection of the dilated mask with the mask's complement also provides a border. First, a complement operation creates a copy of the mask where all foreground and background pixels are switched (all 1s become 0s in the complement mask as do all 0s become 1s). Then, the original mask is dilated with a 3x3 cross structuring element. The final contour output has foreground only where both the complement mask and the dilated mask are both foreground.

3.5.3 Contour Thinning

All of the contour thinning methods use similar terminology. The goal of this step was to thin the contour in parallel. This means many techniques operate only on a per pixel neighbourhood. Each pixel in the contour has its neighbourhood examined for certain properties which determine if it will remain in the final contour. The following Table 3.3 shows the labeling for each pixel in the surrounding 8 pixels (P is the center pixel and P1 through P8 are the surrounding pixels starting form top left and going around in clockwise order).

P1	P2	P3
P8	P	P4
P7	P6	P5

Table 3.3: Neighbourhood Labeling

Apart from this neighbourhood labeling, the following terms are also utilized.

AP starting from P1 and continuing clockwise, this is the number of transitions from 0 to 1

BP number of non zero neighbours

The thinning algorithms discussed in subsequent sections are described utilizing these terms.

Kwon, Woong and Kang Thinning [37]

A thinning method typically used for generating skeletons was implemented to attempt to thin the contour to be 1-pixel width without creating holes. The method is done on a per pixel basis; each point in the foreground is discarded based on properties of its V8 neighbourhood. No global information is needed making this approach well suited for GPU implementations.

The method requires three passes over the entire image to complete. For each of these passes, every non-zero pixel's V8 neighbourhood is examined requiring 27 texture reads $((1 + 8) * 3)$. The first pass uses the following conditions:

- $2 \leq BP \leq 6$
- $AP = 1$
- Any of P2, P4, or P6 is 0
- Any of P4, P6, or P8 is 0

If all conditions are true, the pixel is deleted from future computation. The second iteration is very similar to the first and uses the following criteria:

- $3 \leq BP \leq 6$
- $AP = 1$
- Any of P2, P4, or P8 is 0
- Any of P2, P6, or P8 is 0

Again, if all of the above are true the pixel is removed. These two passes should ensure the skeleton is no more than 2-pixel wide. The second pass aims to remove redundant pixels so all remaining pixels would break connectivity if deleted. The third and final pass uses the following conditions, if any are true then the pixel is deleted.

- P1 $\wedge$ P8 $\wedge$ P6 $\wedge$ $\neg$ P3
- P3 $\wedge$ P4 $\wedge$ P6 $\wedge$ $\neg$ P1
- P5 $\wedge$ P6 $\wedge$ P8 $\wedge$ $\neg$ P3
- P4 $\wedge$ P6 $\wedge$ P7 $\wedge$ $\neg$ P1

The first condition for example, checks if there is a line through the top-left, left and bottom pixel (P1,P8,P6 respectively). If there is such a line, the center pixel is redundant if there is not an off shoot in the top right (P3). The other conditions are permutations of this same pattern.

Other Methods

An earlier work by Zhang-Suen[38] similar to the method of Kwon et al. [37] was also tested. This only uses the first 2 passes with a minor change to the second iteration (first condition is $2 \leq BP \leq 6$). Additionally, a slightly altered version of the Kwon et al. [37] algorithm was tested which only used the third iteration.

Comparison

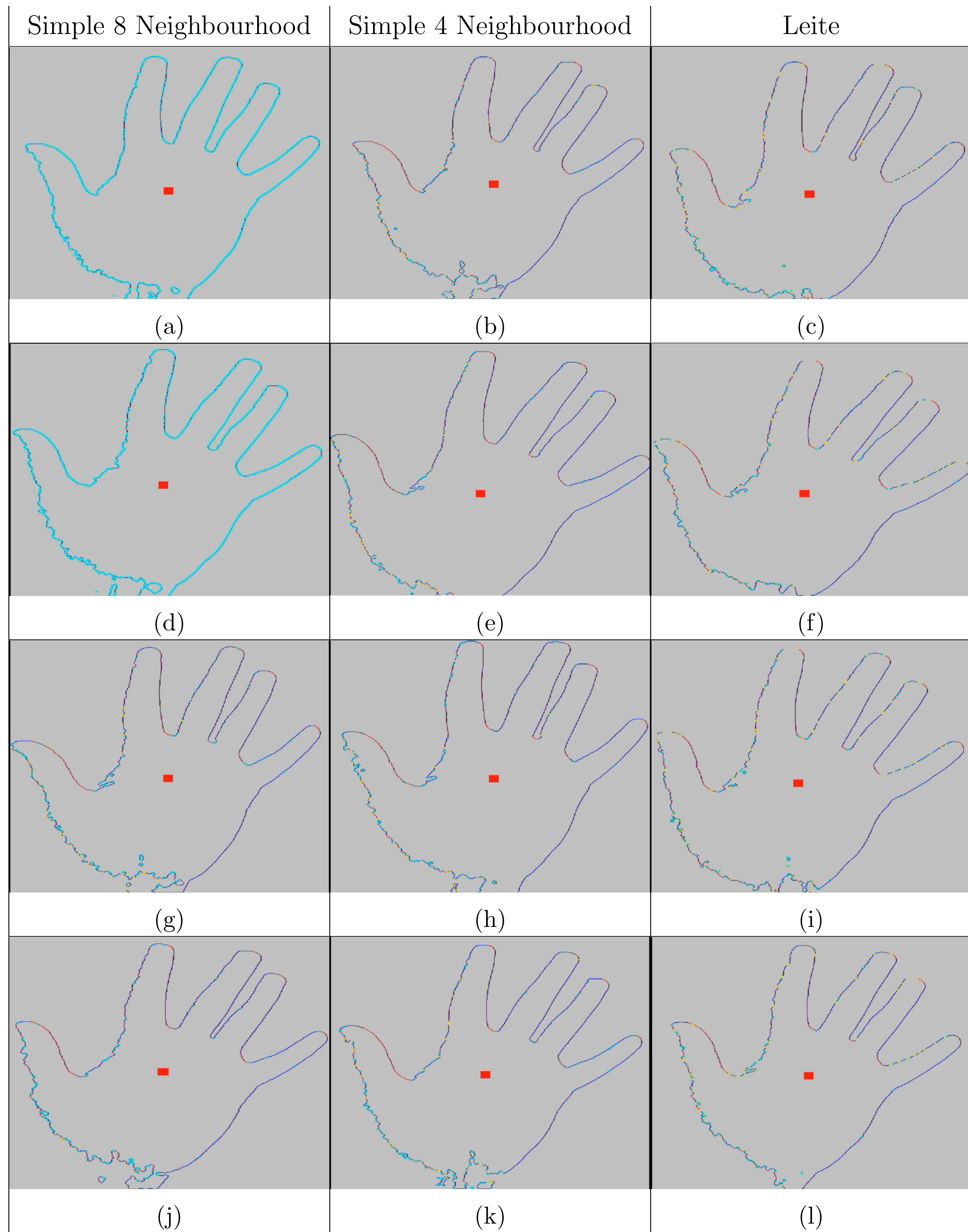


Figure 3.14: Comparison of Contour Generation and Thinning Methods. No thinning results (a) to (c), Zhang-Suen[38] in (d) to (f), the Kwon et al. [37] (g) to (i), Kwon et al. [37] using only the third iteration in (j) to (l)

In summary, three contour generation methods with four contour thinning methods were tested. The images shown in Figure 3.14 show a comparison where each row is a different thinning method and each column is a different contour generation method.

The important colours in the image are the cyan and yellow points. These designate spots on the contour where it is either not 1-pixel wide (a point has more than two neighbours/BP) or the contour is not continuous (a point has less than two neighbours/BP) respectively. The yellow or non-continuous points are a more severe problem than the cyan points; while both are not ideal properties of the desired contour, the non-continuous points are guaranteed to cause errors whereas the non-thin points only have the possibility to. Because the fingertip detection relies on tracing the contour, a break causes tracing to stop and therefore fingertip detection fails. With a non-thin point, there are multiple possible routes the tracing could take (some of which will allow for tracing to continue others may loop). Also worth mentioning in either case, these error points have a larger impact when close to the fingertips. The rationale behind the error points will be discussed further in the subsequent section which details how the fingertips are detected from the contour.

The 8 neighbourhood contour generation without thinning (Figure 3.14 (a)) produces a thick contour; nearly every point along the contour has more than two neighbors. The other generation methods (shown without thinning in (b) and (c)) produce contours with with fewer thick regions concentrated on the edge of the palm. In either case, while the number of thick regions decrease significantly, contour breaks are introduced. The second row shows the effect of the Zhang-Suen [38]. Here while thinning occurs, it is less pronounced than the other methods. The third row shows the results of using the full Kwon et al. [37] algorithm. The contour is thinned significantly however some contour breaks appear. Particularly in (g) the contour generation method without thinning (a) has no breaks where as here some are introduced on the sides of the index finger. Because of the contour breaks from other methods, only the third pass of the Kwon algorithm is used in conjunction with the 8 neighbourhood generation method (shown in (j)). This method combination produces the least amount of thick contour points while maintaining a continuous contour and is therefore used for subsequent steps in this work.

3.6 Fingertip Detection

3.6.1 Neighbourhood Encoding

A pre-processing step is used to reduce the number of memory reads required to traverse the contour before detecting fingertips. At this stage, the contour is a binary image stored in an OpenGL ES RGBA32F texture. A texture is a large 1D array of pixels. In this case, each pixel has a red, green, blue, and alpha value each of which is a 32 bit floating point number. The pre-processing step encodes this texture with a neighbourhood map for each contour point. For each pixel, the location of each of its neighbours is encoded into the texture, with the red and green channels being used for the X and Y location of 1 neighbour and the blue and alpha channels used for the other. Each neighbour can be in 1 of 8 possible positions with X and Y position ranging between -1 and 1. For example, a pixel with neighbours in the diagonally bottom-left direction as well as the straight right direction would have the encoding -1,-1,1,0. Note that this scheme uses more memory than is necessary. Packing could be done to fit the contour map into a smaller type of texture as only $\binom{8}{2} = 28$ possible values are needed per pixel. This would require more computation time for packing/unpacking the values however.

Notably this encoding scheme fails in cases where the contour is either not continuous or is not 1-pixel wide. For points with more than 2 neighbours, only the first 2 neighbours are encoded. The included neighbours may lead to a dead end, cause contour traversal to loop, or be correct and allow full traversal. However, a non-continuous contour is a non-recoverable error as a break stops traversal from being possible. This affected which contour generation/thinning method was used, since extra neighbours may produce correct results but a non continuous contour is guaranteed to fail. The location of where these error points appear on the contour is also important. Error points along the bottom or side of the palm do not affect fingertip detection, only error points near the tips of the fingers do.

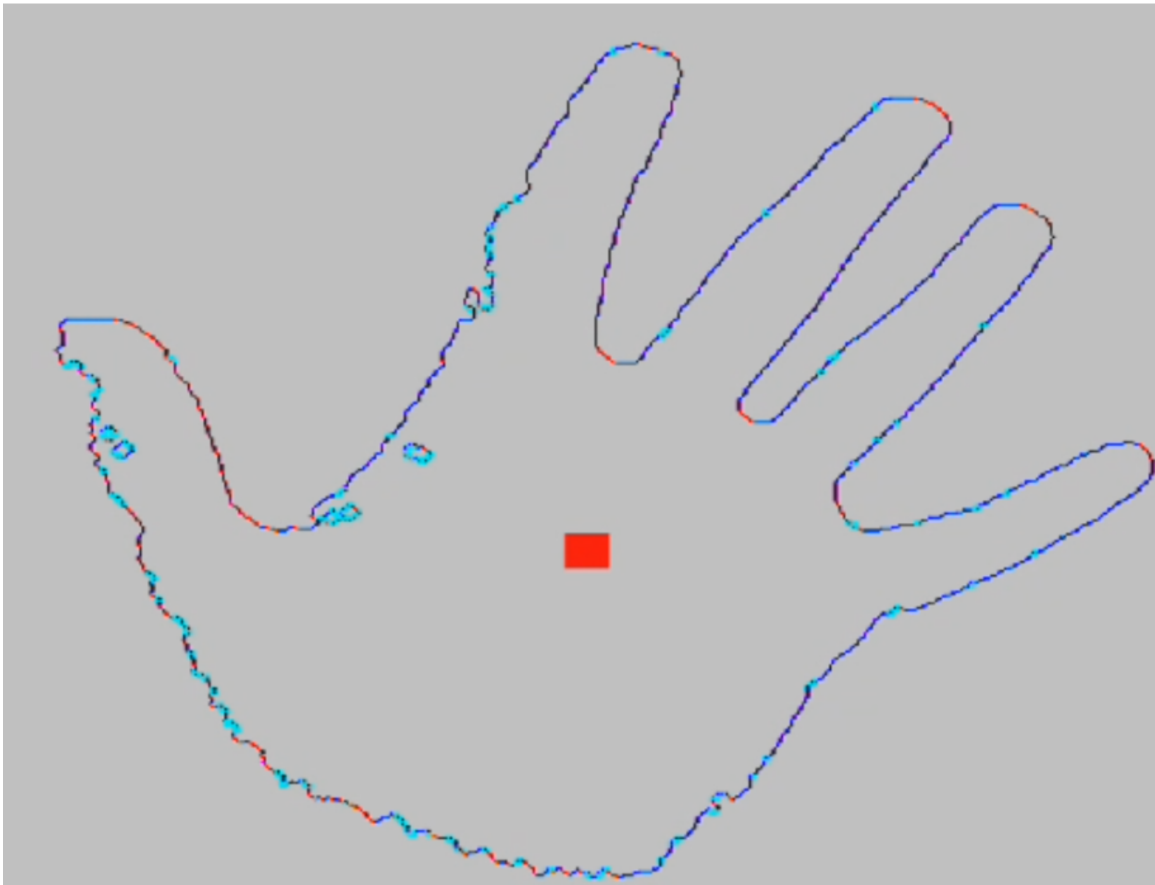


Figure 3.15: Example Encoding

Figure 3.15 shows an example of this encoding scheme. As it is stored in an image texture, it is possible to draw the output of the encoding. The drawing excludes negative values commonly found in the encoding as well as any value in the alpha channel however. The order of the neighbourhood scan determines what the colours the encoding will be. The method uses the order: left to right and bottom to top. The first found neighbour is encoded into the red and green channels. There is no green in the image because the first found neighbour in this order is likely either below the central pixel or has the same height. In either case the green channel does not get a drawable encoding; it is either -1 or 0 neither of which show in the image. The most common colours in the figure are varying shades of red and blue. Generally, the diagonal bottom-left to top right lines are blue given the encoding is -1,-1,1,1 (bottom-left pixel is found first then top right). Similarly the top-left to bottom-right lines are red given the encoding 1,-1,-1,1. In the figure

Apart from these colours, cyan 3x3 boxes are drawn in locations where the neigh-

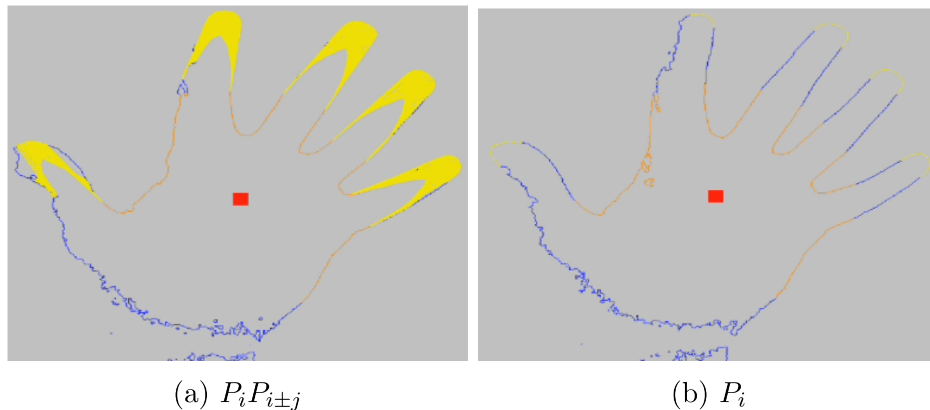
bourhood contains more than two neighbours. These points are not handled differently from the points with exactly two neighbours, only the first two neighbours encountered are encoded. The order in which the neighbourhood is traversed changes the encoding and affects how contour traversal happens with more than two neighbours.

3.6.2 Determination of Fingertips from Angle

```
void findNextNeighbour( in ivec2 globalPos, inout vec2 lastPos,
                        inout vec2 compoundDir) {
    // read the jth neighbour based on
    // current displacement from the original point
    vec4 nextNeighbour = imageLoad(inputBuffer,
            globalPos + ivec2(int(compoundDir.x), int(compoundDir.y)));
    // need to check if next position is stored in xy or zw
    // try xy and see if we go backwards
    vec2 canditate = nextNeighbour.xy + compoundDir;
    // if we did go backwards use zw instead of xy (2,3 v 0,1)
    int idx = all(equal(lastPos, canditate)) ? 2 : 0;
    lastPos = compoundDir;
    compoundDir += vec2(nextNeighbour[0 + idx], nextNeighbour[1 + idx]);
}
```

Code Snippet 3.3: Finding Next Neighbour

The encoding scheme allows the contour to be traversed in parallel. Starting from any point on the contour, every other point can be reached by reading the pixel and finding where its neighbours are. After encoding, for each point on the contour a certain distance from the center of the hand, an angle is calculated between the point and two points some distance along the contour in either direction. For a given point P_i, the neighbourhood map contains the relative location of points directly neighbouring P_i. These points are designated as P_{i+1} and P_{i-1}. The values at $P_{i\pm1}$ have information to find $P_{i\pm2}$. This process repeats to find points further and further along the contour. A running total of the displacements is saved in both directions. When reading a new neighbour, 1 of the 2 encoded values will lead backwards to an earlier point in the path. Thus for each step, the reads of P_{i+j} and P_{i-j} occur. For each of these, a conditional is necessary to ensure the next iteration reads P_{i+j+1}/P_{i-j-1} instead of P_{i+j-1}/P_{i-j+1}. After j steps, an angle is calculated as shown in equation 3.3 where $P_iP_{i\pm j}$ are vectors between points P_i and either P_{i-j} or P_{i+j}.



(a) $P_iP_{i\pm j}$ (b) P_i

Figure 3.16: Example of colour coded output for the fingertip detection step

Part of this process is shown in Code Snippet 3.3. The function is run in a loop and called once per direction each iteration. The parameters to the function are the location of the original contour point (P_i), the previous displacement, and the current displacement ($P_{i\pm j}$ after j iterations). Line 5/6 reads the next neighbour's encoding which contains the information to find the subsequent neighbour as well as information leading back to the last displacement. Line 9 and 11 show how these displacements are distinguished. Finding the next neighbour would be complicated without the encoding scheme described in the previous section; information for the next neighbour lies in one of two possible locations both stored in the same pixel. The neighbourhood does not need to be examined during traversal.

$$cos\theta = \frac{P_iP_{i-j} \cdot P_iP_{i+j}}{\left\|P_iP_{i-j}\right\| \cdot \left\|P_iP_{i+j}\right\|} \tag{3.3}$$

After finding the jth neighbours on the contour an angle is calculated between P_i and $P_{i\pm j}$ (equation 3.3). If this angle is lower than a threshold, the point P_i is marked as a possible fingertip location. Note the OpenGL inverse cosine function only returns angles between zero and pi. Points on the side of the hand or side of fingers have large angles between neighbours. Points on the contour located in between fingers however do, like fingertips, have a small angle between them and their neighbours. Requiring points to have both a small angle as well as be a minimum distance from the center of the hand eliminates all non fingertip points.

Figure 3.16 shows an example output of this step. Blue points on the contour show points which are a sufficient distance from the center of the palm (shown as the red square) but do not have a low enough angle, orange points on the contour

are too close to the center of the palm and finally yellow points are the candidate fingertip locations. Figure 3.16(b) shows just the points on the contour which meet these criterion where as Figure 3.16(a) has lines drawn between these points and their neighbours j distance away on the contour.

3.6.3 Data

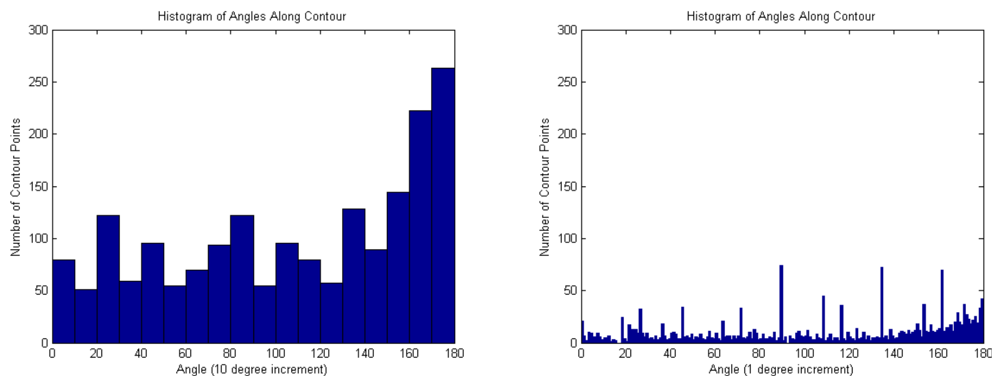


Figure 3.17: Angle Data

Solely using a single maximum angle threshold was not sufficient to segregate fingertips. An example of the angles found across the contour are show in Figure 3.17. Note that all angles lie within the range 0 to 180 degrees as the inverse cosine function available in OpenGL only returns angles in this range (specifically between 0 and π radians). A large number of points along the contour have obtuse angles as expected (many points along sides of hands). There were a number of points with very low angles between 0 and 20 degrees with a high number exactly 0. These points are generated from errors in the contour and are removed as candidate fingertip points. Additionally the encoding scheme can fail in cases where a point has more than two neighbours. Here following the contour may loop or lead to a dead end. This causes a wide variation in angle due to $P_{i\pm j}$ being close to P_i (traversal could end early due to reaching a dead end or a loop which brings the traversal . In total to be considered a fingertip the contour point requires:

- angle between neighbours is not close to 0 but also low
- sufficient distance from center of palm
- lines $P_iP_{i\pm j}$ must be long

3.6.4 Parameters

Table 3.4: Parameters for Fingertip Detection

Parameter Name	Description	Default Value
Number of Neighbours	how far along the contour the fingertip detection travels	100 neighbours
Minimum Angle Threshold	angle between $P_i P_{i\pm j}$ must be larger than this	10 deg
Maximum Angle Threshold	angle between $P_i P_{i\pm j}$ must be smaller than this	75 deg
Minimum Line Length	the lines $P_i P_{i\pm j}$ must be a certain length on top of forming a small angle	60 pixels
Remove Lines with Breaks	Boolean representing whether contour traversal which stopped due to a break should be removed	true
Minimum Distance Threshold	points on the contour must be a minimum distance away from the center of the palm to be considered fingertips	175 pixels

3.7 Fingertip Refinement

3.7.1 Overview

At this stage multiple candidate fingertip points have been found for each finger. The goal of this step is to reduce these to a single point per finger and track their locations for use during the subsequent gesture recognition phase. The candidate finger points are reduced using k-Means. k-Means is an unsupervised clustering algorithm which works by finding and iteratively updating means of clusters of an input data set. For general use, the initial position of each of the K means are set randomly. For each data point, a label corresponding to the closest mean (1 out of the possible k-Means) is set. Each mean is then updated to be the average of all data points with a particular label. This process repeats until the means are stable and do not move

during successive iterations. k-Means is a general algorithm applicable to any ordered data of arbitrary dimensions. Here, the data consists of the X and Y position of the finger tip points.

k-Means was used because it exhibits two behaviours which make it ideal for this environment. The first behaviour relates to the fact that the initial positions of the means influence both the final locations of the means and the number of iterations required to reach them. The output of the algorithm can change depending on the initial positions of the means even with the same data set. Generally, if the means start close to their final positions, the number of iterations needed to reach a solution is small (thus requiring less compute time). Secondly, k-Means performs well when the actual number of clusters in the data set is less than K. For example, if there are only four clusters in the data but the algorithm is run with K = 5, as long as the initial positions are close to the actual solution, one center is assigned no points and the other four are positioned at the center of each cluster. If one fingertip is lost (either from error or from lowering the finger), the other fingertips are not lost.

In addition to the k-Means algorithm, a simple tracking method is used if a final fingertip location cannot be found.

3.7.2 K-Means

```
struct KMeanStruct {
  // current position
    float x;
    float y;
    // number of labels which are closest to this mean
    float numPts;
    // label's positions are accumulated in these during the update
   process
    float tempX;
    float tempY;
    // elements for the simple tracking
    // semaphore lock for updating subsequent elements
    uint  sem;
    // distance to, and location of the closest point on the contour
    float closestDist;
    float closestContourX;
    float closestContourY;
};
```

Code Snippet 3.4: K-Means Data Structure

The data stored for each of the K means is shown in Code Snippet 3.4. In total, 9, 4-byte elements are stored for each of the 5 means. The means are initialized to positions that resemble fingertip locations for an open hand as shown in Figure 3.18 (thumb closer to center of hand compared to middle finger etc.). This open hand shape uses the palm position calculated previously as well as the maximum distance threshold for foreground pixels; the initial locations track and scale to these parameters. The means are initialized every frame until the fingertips are found where the means position is kept between frames.

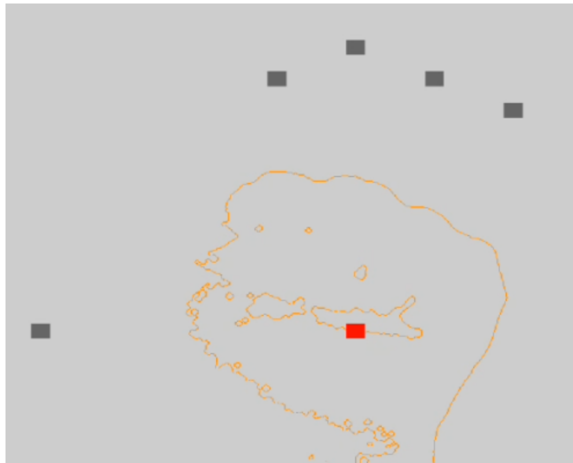


Figure 3.18: K-Means Initial Positions

A shader storage buffer (large array) with 1 floating point element per pixel is used to store the label of each data point. This is a sparse array with only the contour points from the previous steps being non zero. The candidate fingertip points have a positive numeric identifier while the other contour points have a negative value designating if the point lies within the minimum distance threshold.

The candidate fingertips' labels are updated to have an identifier designating which mean has the smallest Euclidean distance to them. The means accumulation values are then reset to zero. For each candidate fingertip, the X and Y positions are added to the accumulation value of the closest mean using atomic addition (guarantees all threads see every writes given many threads are concurrently writing to one memory location). Afterwards, if the number of points which are close to a mean are higher than a threshold, the new position of the mean is updated to be the average position of the contributing points (X and Y accumulation divided by the number of points). These steps, except for the initializations of the mean and labels, then repeat a set number of times (rather than until convergence in the general k-Means algorithm).

3.7.3 K-Means Correction

Any mean which, at the end of the K-Means process, has few contributing candidate fingertips is considered lost. The lost means are then corrected to be the closest contour point to that mean. For each contour point, the distance between it and each mean which was lost is calculated. If the distance is lower than the current minimum, the location of the point is recorded. This update process involves many

writers (a thread per contour point) needing to share access the mean data. Without synchronization, the incorrect data could be written into the means.

This synchronization is accomplished using the the semaphores each mean has. Semaphores act as locks; only a single thread at a time can acquire a semaphore, and only threads which have acquired a semaphore can overwrite data. OpenGL ES provides an atomic compare and swap function which guarantees only a single thread can change the value of the semaphores. The function takes checks if the semaphore is equal to 0 (unlocked) if so, it is set to 1 (locked) and the update proceeds. Otherwise, the thread will repeatedly try to acquire the semaphore again until successful.

After this update process, if the closest contour point lies within the minimum distance threshold, the mean is marked inactive and is not used for gesture recognition. The mean is then moved to this contour point if the distance between its current position and the contour point is lower than a threshold. Forcing the means to only move small distances in cases where the fingertip points are lost prevents the mean from switching to a different finger during hand movement.

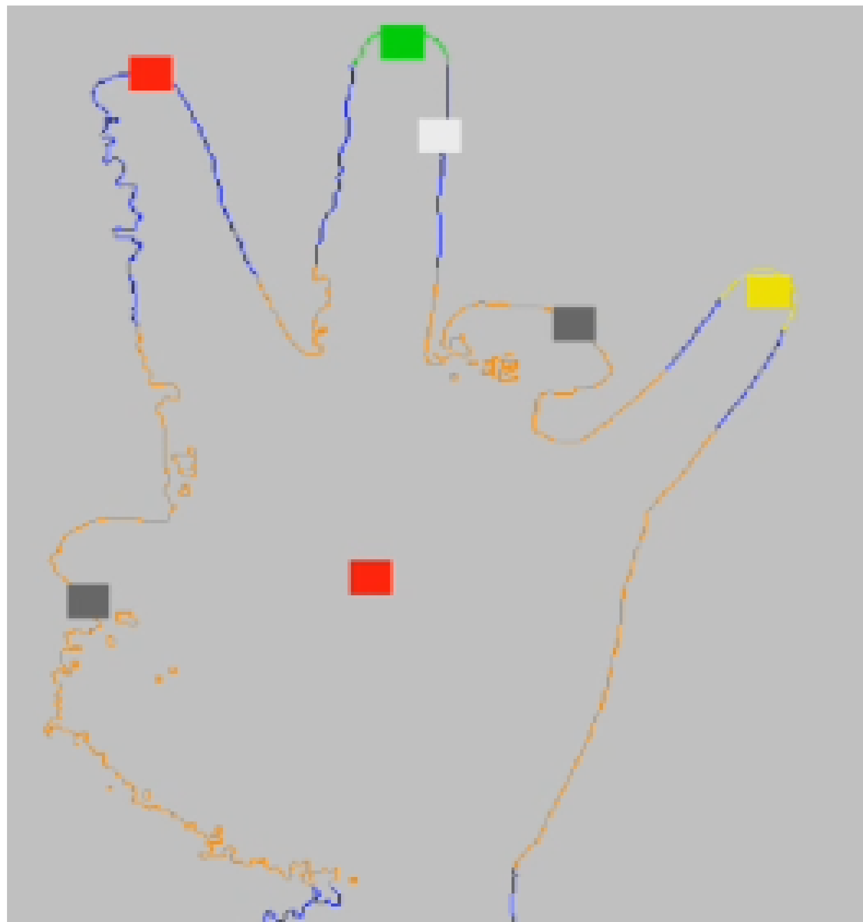


Figure 3.19: K-Means Example

An example output of this step is shown in Figure 3.19. The large coloured squares mark the locations of the means. The candidate fingertip points are colour coded to match which mean they are closest to (seen on the middle and little fingertips). The index finger in this example does not have any candidate fingertip points on the contour. The mean marked by the red square on the index finger was set to the closest contour point. The thumb and ring finger means show the result of tracking fingers lowering. Starting from an open hand, the means followed the contour as the fingers retracted. Once the closest contour points were within the minimum distance threshold the finger was deemed lowered and marked in-active designated by being coloured gray. Finally, the white square shows the average position of the active means and is used during gesture detection.

3.7.4 Parameters

Table 3.5: Parameters of the Fingertip Refinement Step

Parameter Name	Description	Default Value
Reset Mean	Boolean representing if the mean positions should be reset	true until acquisition
Iterations	number of K-Means iterations performed	5
Reposition Threshold	limit on how far the correction step can move a mean	50 pixels
Number of Labels Threshold	number of candidate points necessary for a mean not to be considered lost	5

3.8 Gesture Recognition

3.8.1 Overview

The final fingertip positions (means from previous step) are used as input to the gesture detector. Any fingertips which lie too close to the center of the hand were considered to be lowered/inactive (like the thumb and ring finger in figure 3.19). Gesture detection is based on tracking the movement of the fingertips relative to the mean position of the fingertips. The mode of operation switches depending on how many active fingertips are detected. Fingertips become inactive as the means follow the contour. As a finger is lowered, the candidate fingertip points disappear as small angles across the contour are no longer present. The mean then will follow the closest contour point as the finger merges with the hand contour. The finger is considered inactive once the mean lies within the minimum distance threshold used in finding candidate points. The gestures are as follows:

1 Finger Move: an on screen pointer is updated to match the raised finger's position

2 Fingers Zoom: an image is zoomed depending on the change in distance between the 2 fingertips relative to the mean position

3 Fingers Scroll: the change in position of the mean is used to translate an image

4 Fingers Rotate: for each fingertip, the change in angle around the center of the hand from the previous frame is found. The average of these angles is used to

rotate an image.

For this work it does not matter which fingers are used to perform the gesture. While doing the gestures is easier with certain pairings (e.g, zoom operation with thumb and index finger), it is possible to use other combinations of fingers.

3.8.2 Implementation

The first step in the gesture detection process is finding the mean location of the active fingertip positions hereby designated as fingertip center of gravity (CoG). On the first frame after changing the number of active fingers no gesture detection occurs. Instead an initialization step happens where the fingertip CoG as well as the fingertip's average distance to the CoG are recorded. On subsequent frames with the same number of active fingertips, gesture detection occurs. In all examples shown the image being rotated/scaled is a map of the University of Victoria from Open Street Map (map image credit and copyright [1]).

Move

This is the simplest gesture; the location of a pointer is simply set to be the location of the fingertip CoG.

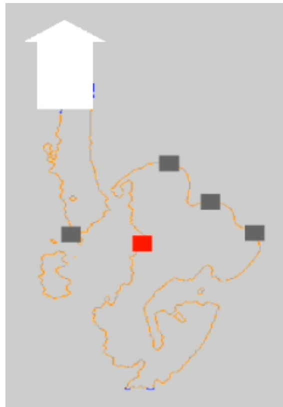


Figure 3.20: Example Move Gesture

Zoom

When the number of active fingertips first changes to 2, the average distance of the fingertips to the fingertip CoG (which is the midpoint between the two fingers) is

saved. The average distance controls how quickly the map scales. The ratio between this saved distance and the average distance frame to frame is calculated and used to scale the map. If the distance between the fingertips and the fingertip CoG halves, the size of the map will also be reduced by half. This allows the user to specify sensitivity; if the fingertips start close together, small movements will have a larger effect on the scale of the map compared to the fingertips starting far apart.

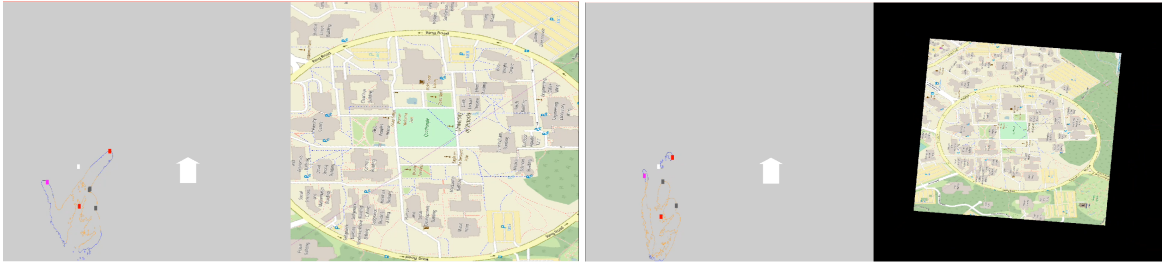


Figure 3.21: Example Zoom Gesture

Scroll

The change in position of the fingertip CoG frame to frame is used to translate the map.

Figure 3.22: Example Scroll Gesture

Rotate

Every frame the X and Y distance from each mean to the fingertip CoG is computed and stored. A rotation angle, is computed using these XY distance vectors between frames. The calculation of the rotation angle is shown in the following Equation 3.4 where D_{n_t} is the XY distance vector of the nth fingertip at frame t. The distance vector of each active finger are averaged to improve accuracy.

$$\theta = \frac{1}{4}\sum_{n=1}^{4} cos^{-1}\left(\frac{D_{n_{t-1}} \cdot D_{n_t}}{\left\|D_{n_{t-1}}\right\|\left\|D_{n_t}\right\|}\right) \tag{3.4}$$

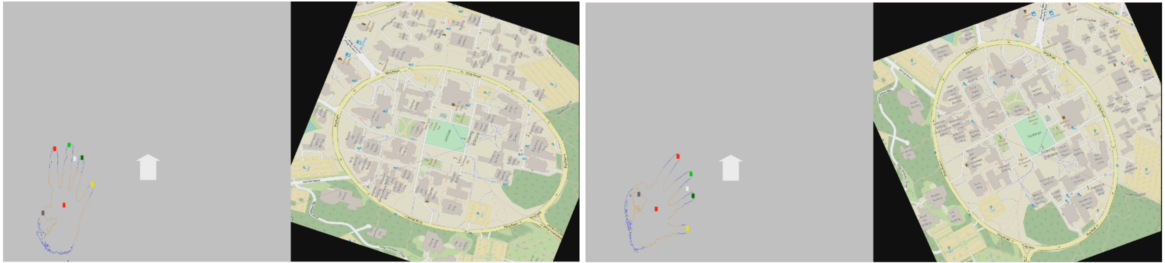


Figure 3.23: Example Rotation Gesture

Chapter 4

Evaluation

This chapter covers the analysis of both the accuracy and compute time of the method. All experiments were performed on a Google Pixel C tablet which uses a NVIDIA Tegra X1 (more information on the device is available in Appendix A.3). Both the input video resolution retrieved from the camera and the output of the method is a 1920x1080 video stream. Section 4.1 focuses on evaluating the correctness of the method (percent of frames where fingertip points were placed correctly) in a couple different environments. Section 4.2 gives a detailed examination of the frame rate achieved by the implementation of the method on the test device.

4.1 Accuracy

4.1.1 Quantitative

Videos were analyzed frame by frame to find when the fingertip points (the K-Means locations) deduced by the algorithm were incorrect. Two 1000 frame videos were analyzed one with favorable lighting / background resulting in a clean background subtraction and one in a challenging environment. Each video has a hand perform every gesture as well as some frames tracking when all five fingers are raised. The videos are also contiguous; frames where the hand is switching between gestures are also included.

Fingertip points were deemed to be correct if they lay anywhere within the half circle formed by the candidate fingertip Points. Losing a portion or all of these points causes the fingertip location to drift but this does not impact gesture detection. It is important to note that points which drifted partially down the finger while it was

extended were considered errors. For the gestures implemented in this work, if the algorithm incorrectly labeled a fingertip location in the middle of the actual finger, the gestures could still be recognized. The rotation gesture, for example, only relied on average change in angle around the center of the palm and would not be negatively impacted by such an error.

For each frame in the video sequences two values were recorded for each finger: one designating if the fingertip position was placed correctly and one signifying whether candidate fingertip points appeared for that finger. These values have three possible states:

1 representing fingertip position was correctly placed or candidate points appeared when the finger was raised

0 representing fingertip position was incorrectly placed (whether finger was raised or not) or candidate points did not appear when the finger was raised

blank representing the finger was lowered and the fingertip position, correctly, was marked in active and no candidate points appeared.

The accuracy of each finger (both for positions and candidate point) is then reported as the sum over all frames divided by the number of non blank entries. Adding this blank state was done for a few reasons. First, the gestures in this work do not equally utilize fingers. The little finger for example is only raised in the evaluation videos for no gesture (all five fingers raised) and for the rotation gesture (only thumb lowered). For all other frames the little finger remained lowered. Tracking the closest contour point when a finger is lowered is an easier problem compared to finding a raised finger given the fingertip detection with does not need to be performed. Marking all these frames as correct would raise the accuracy percent but would not represent how well the algorithm found and tracked fingertips.

In addition to these ten accuracies (position and candidate points for each finger reported as a percent), two total accuracies were recorded (marked “All”). These represent the percent of frames in which all fingers are correctly tracked. These total accuracies are not averages of the individual finger accuracies; if a single finger was incorrect (or the candidate points for one of the fingers were not present) then the frame was recorded as having an error. These total accuracies are the percent of frames in which no error is present. Also the average error time is reported in frames. When an error occurred, the duration of the error was recorded and these durations

were averaged. The error duration was calculated as the difference between the first frame an error occurred (a finger with a 1 or blank state switched to a 0) to the first frame a non error reoccurred (0 switched back to a 1 or a blank state).

Good Lighting Evaluation

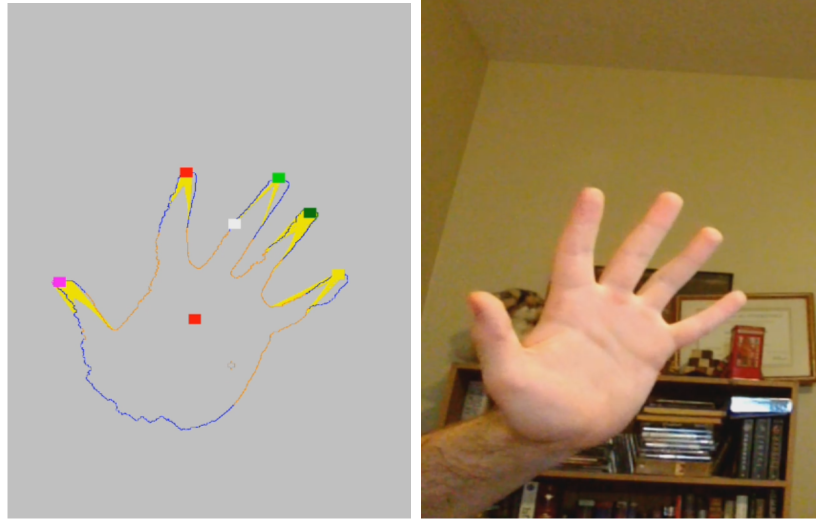


Figure 4.1: Example Frame from Good Lighting Video

In this video sequence, the camera was facing the corner of a room with a bookshelf in the background. There were three source of light: a large window, a desk light and overhead light all behind the camera facing the hand. This arrangement allowed the background subtraction step to remove all background and the arm while keeping the hand intact. The accuracies of this video sequence are report in Table 4.1. The algorithm placed every fingertip point correctly in 77.8% of the frames despite the number of frames where all raised fingers have candidate points being much lower (15%). Following the closest point on the contour in cases where the candidate points are not found is sufficient for tracking in most cases. When following the contour between frames, fingertip points begin to drift down the contour. As long as every few frames the candidate points are found, the detected fingertip point will remain correctly in place (an example of this process is given in the following Section 4.1.2). The average error time for candidate points on individual fingers are all below 10 frames and the total average error time for all candidate points is close to 10 frames (11.21). Given the method performs at roughly 10 FPS (see Section 4.2), the algorithm regains candidate points in about 1 second on average. The method recovers

from incorrectly placed fingertips quickly as well with individual fingertips corrected within 1 second and all fingertips corrected within 1.5 seconds (14.00 frames)

The thumb and little finger have a lower candidate point accuracy compared to the other three fingers (14% and 26% compared to accuracies above 50%) meaning these fingers follow the closest contour point instead of using the candidate fingertips more often. The discrepancy in the thumb comes from it requiring a larger angle threshold for candidate points to be detected relative to the other fingers. The thumb is both shorter and thicker resulting in angles across the thumb being wider than those across the other fingers. Angle detection needs to detect the wide angles on the thumb without detecting wide angles on the side of the hand. The angle threshold need to be restricted to prevent incorrect detection on the side of the hand resulting in a lower accuracy for the thumb. The little finger is difficult to explain as it shares similar properties to the other non-thumb fingers. One possible explanation is the little finger is thinner compared to other other fingers resulting in both fewer possible candidate fingertip points (actual width of the fingertip is smaller) as well as contour traversal during fingertip detection going to far. Consider Figure 4.2 (a). The candidate points for the little finger have neighbours which extend to the base of the finger and are beginning to travel up the ring finger. The candidate points for the index and ring fingers have neighbours which extend about halfway down the finger. This suggests that contour traversal on the little goes to far though reducing the distance traversed may impact the accuracy of the index and middle fingers (if candidate point neighbours were close together they would have a wide angle between them).

Table 4.1: Accuracy of Each Finger With Candidate Points - Good Lighting

Finger	% Correct	Error (frames)	% Correct (Candidate)	Error (frames)
Thumb	91.70	9.50	14.51	7.86
Index	97.08	3.89	56.88	2.59
Middle	94.64	2.19	72.73	2.34
Ring	88.04	4.63	64.96	2.40
Little	89.01	7.09	26.25	4.62
All	77.83	14.00	15.00	11.21

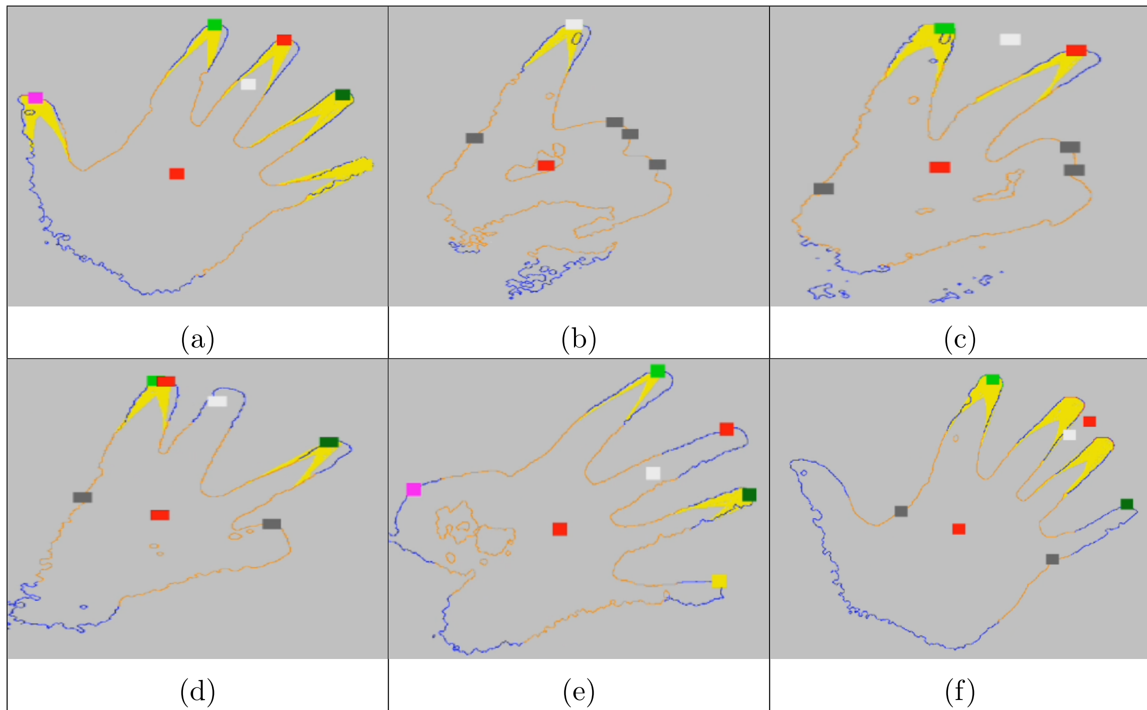


Figure 4.2: Good Lighting Example Frames

Example frames from the evaluation are shown in Figure 4.2 with the first row showing correct detection and the second row showing errors. Figure 4.2 (a) through (c) show the ideal case; for each raised finger, a position is correctly placed and the finger has candidate points. The second row shows the different types of failures which can occur. In Figure 4.2 (d) the index finger incorrectly has two fingertip positions placed on it while the middle finger has no fingertip positions placed. In this case both the index and middle finger positions were marked incorrect (0) while the candidate points were only marked incorrect for the middle finger. Figure 4.2 (e) shows an example of minimum distance threshold failing to remove the thumb when attempting to perform a rotation gesture (all other fingers are correct but the middle and little finger are missing candidate points). Finally Figure 4.2 (f) shows a few errors. First the thumb does not have a fingertip point nor does it have any candidate points despite being raised. Also, while both the middle and ring fingers have candidate points, neither has a fingertip position placed on it. In this frame, while the red square (not in the center of the hand) is close to the middle finger, it is incorrectly using candidate points from both fingers.

Poor Lighting

Figure 4.3: Example Frame from Poor Lighting Video

In this video sequence, the device was placed in the corner of a room an example input/output is shown in Figure 4.3. All light sources are behind the hand and include the lamp in the left of (a) as well as light from a skylight in the hallway visible through the door. In this environment, the background subtraction method fails to remove all background as seen in the bottom part of Figure 4.3 (b). In addition, both the chair, grid decoration on the glass door, and the bright spot above the hand have similar *I values* to that of the hand causing the contour to be lost.

Table 4.2: Accuracy of Each Finger With Candidate Points - Poor Lighting

Finger	% Correct	Error (frames)	% Correct (Candidate)	Error (frames)
Thumb	37.42	22.11	5.97	19.93
Index	79.96	7.14	42.79	3.61
Middle	74.97	7.25	51.08	2.92
Ring	66.80	7.34	40.13	5.06
Little	61.42	8.72	38.08	5.72
All	45.70	18.72	15.10	13.06

Table 4.2 gives the accuracy of this video which is lower compared to the good lighting example. Here the algorithm failed due to the segmentation removing too much of the foreground resulting in either no fingertips being detected, or fingertips being incorrectly detected. The time to recover from errors is also increased compared to the good lighting results. Figure 4.4 shows some example frames from this sequence where the top row has examples of correct detections and the bottom row

has errors. Even in cases where the algorithm correctly places fingertips, the contour is broken. In Figure 4.4 (a) the index finger is an isolated contour and in (c) every finger has separate contour regions. In these cases detection is harder because the contour following in the detection step is less likely to be able travel the full neighbourhood size. If these isolated regions are small enough, the contour following can loop resulting in inconsistent angles.

Figure 4.4 (d) and (e) show the worst error case for the method, fingertips detected outside of the hand region. Once these false positives are detected it is hard for the algorithm to recover. Incorrect detection moves the fingertip positions far from where the actual fingers are in the video. The K-Means step relies on the positions to remain in the proximity of the fingers. Otherwise, the means will not be assigned any candidate points and therefore will not be moved except for following the closest contour point. Once these false detection errors occurred, the algorithm took on average, approximately 30-40 frames to recover. Other errors in this video sequence are the result too much of the foreground being removed resulting in unusable contour. Figure 4.4 (f) shows an example frame where it is difficult to identify the hand structure of the contour.

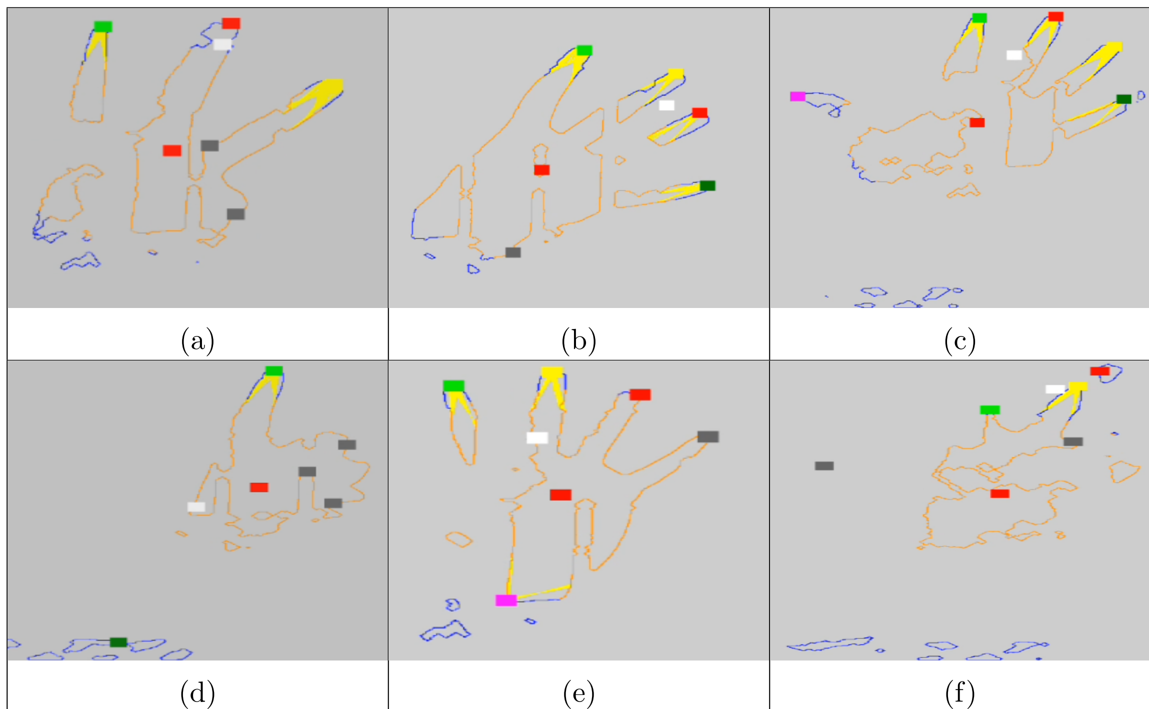


Figure 4.4: Poor Lighting Example Frames

4.1.2 Qualitative Discussion

Figure 4.5: Example Background Subtraction

The algorithm fails when the colour distribution in the hand region (foreground) resembles the background too closely. A particularly challenging combination of ambient conditions involves florescent lighting, and a pale-skin hand moving in front of a white wall. In these cases, the background segmentation does not perform well resulting in poor contours for the subsequent steps.

The best possible scenario for this approach involves a distinctly different background such as a green screen. However, this is not necessary for achieving correct segmentation; Figure 4.5 shows an example of a successful segmentation for a complex background. One should note that the errors in segmentation for the case in Figure 4.5 do not impact subsequent steps such such as fingertip detection and gesture tracking.

Here the algorithm includes few pixels not part of the hand but incorrectly removes parts of the thumb. In these regions it is possible to see elements of the background in the segmentation such as an outline of the window in the tip of the thumb or where there is a bright reflection behind the side of the thumb. The angle of the arm here makes it much darker and, as it matches the background more than the palm/fingers, more is removed.

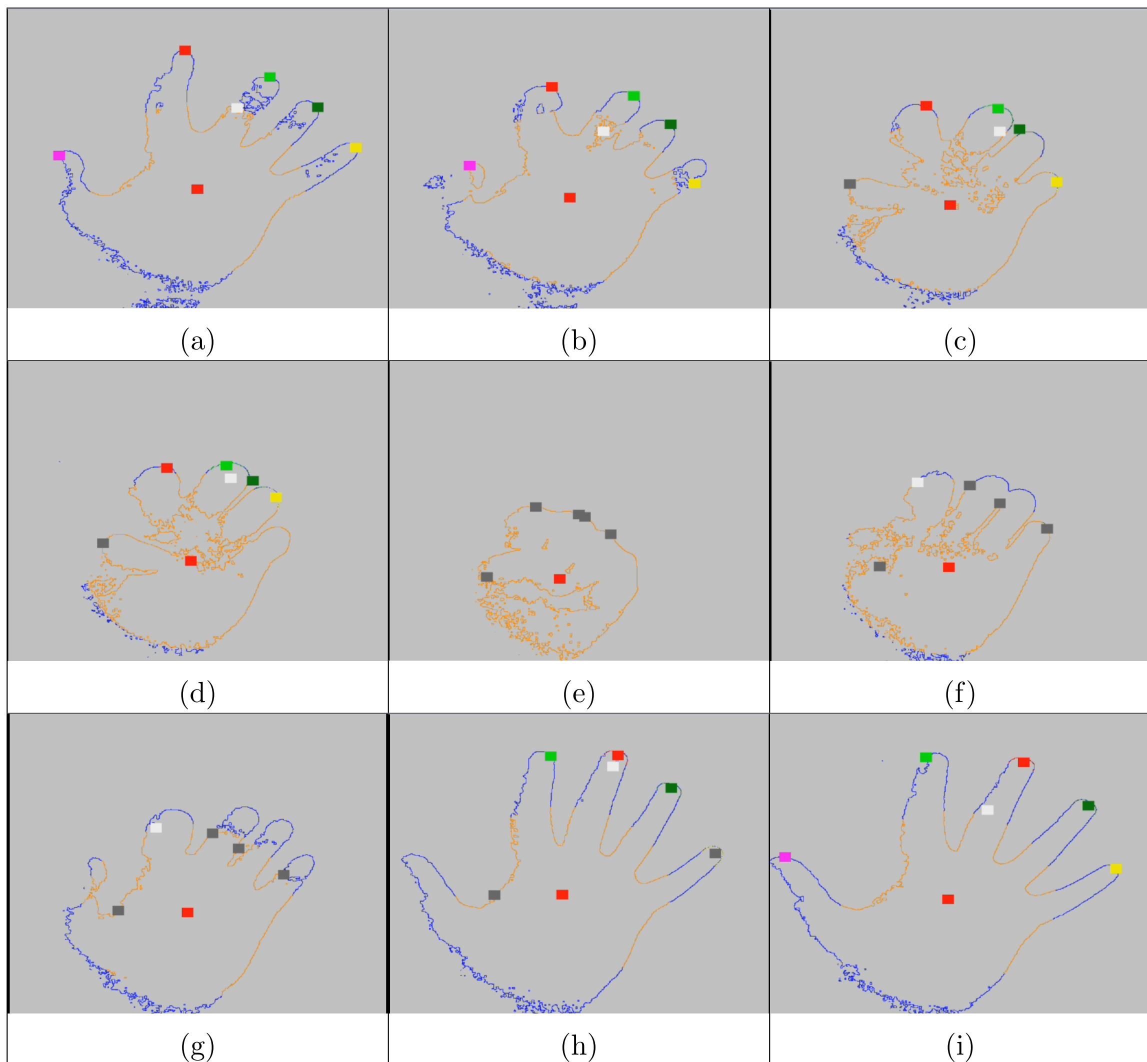


Figure 4.6: Lowering Fingers Example

The algorithm prefers fingertips found via the low angle between neighbours method discussed over following the contour which occurs if the fingertip detection method unsuccessfully finds fingertips. Preferring the candidate fingertip points over just following the contour is what makes tracking the fingertips when fingers are lowered possible; following the contour as it becomes distorted when fingertips are closed is done to keep the fingertip points close to where the fingertips will reappear. When a finger is raised, the fingertip locations will quickly snap back in position once a group of low angle between neighbours is found on the contour. An example of this process is shown in Figure 4.6. Figure 4.6 (a) through (e) show the tracking as a hand closes. Then, the hand is re-opened in Figure 4.6 (f) through (h) and, after a

few frames, all fingertips are reacquired as shown in Figure 4.6 (i).

The fingertip detection step will stop finding candidate fingertip points around Figure 4.6 (c) in the figure. The points will then track the contour as the hand is closed then opened again. Figure 4.6 (h) is the first frame where the candidate fingertip point re-appear on 4 of the 5 fingers (all except the thumb). The fingertip locations, by following the contour, are close enough to their final positions to be successfully moved to the correct location by the k-Means step. In the recorded sequence, the candidate fingertip points on the thumb took a few frames to be found which is shown in Figure 4.6 (i). It is important that the algorithm can handle the lowering and raising of fingertips as the number of raised fingertips switch the mode of operation.

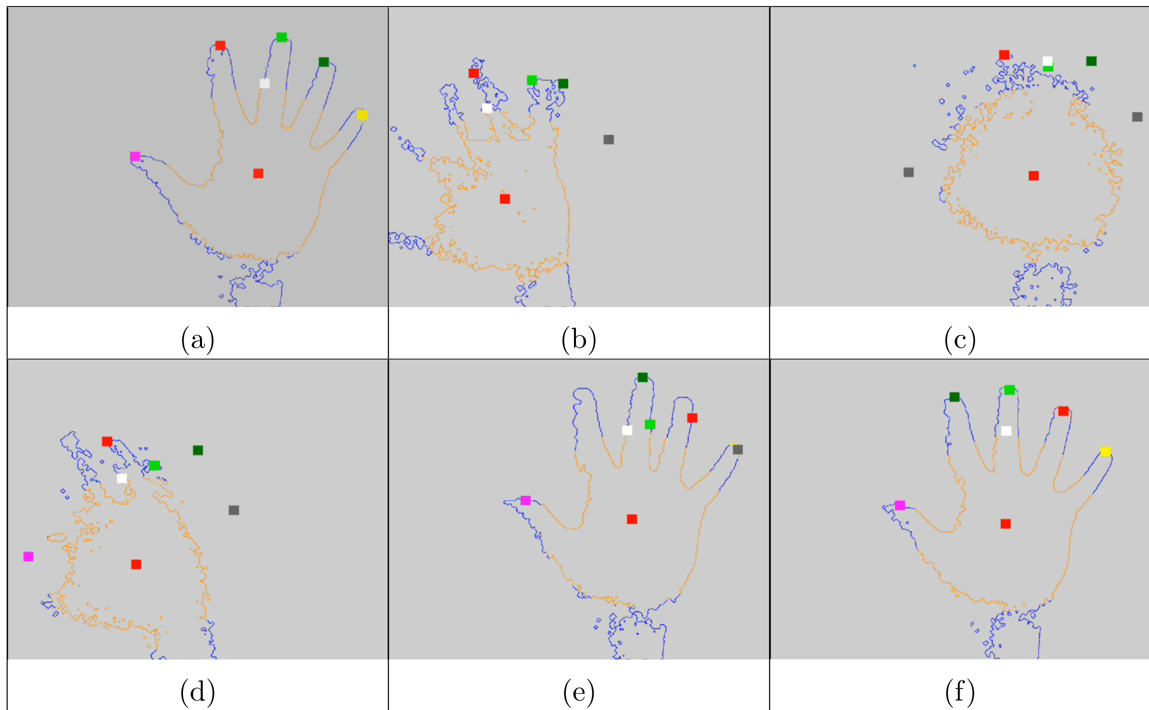


Figure 4.7: Recovery From Fast Motion

Figure 4.7 shows another example motion where candidate fingertip points are lost. Here, the hand is waved back and forth quickly enough for motion blur to appear which causes the background subtraction and contour generation to lose most of the finger shape. Figure 4.7 (a) shows the starting location, Figure 4.7 (b) through (d) show some frames during the wave motion and Figure 4.7 (e) and (f) show the hand at rest after the motion.

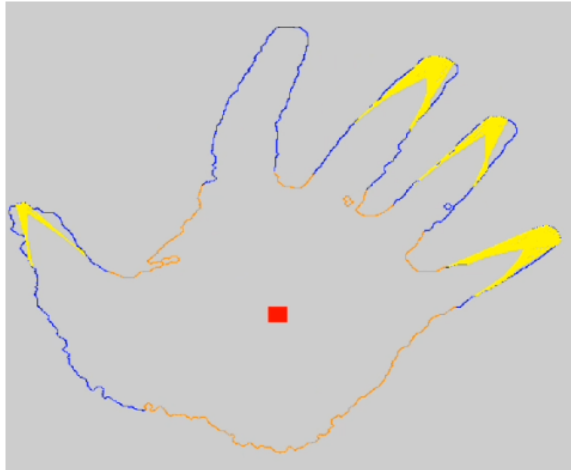


Figure 4.8: Reduced Candidate fingertip Points

Figure 4.7 (e) is the first frame where motion stops and the contour becomes undistorted. Within 10 frames (1 second at 10FPS) all fingertip locations are correctly placed (Figure 4.7 (f)). The detected fingertip locations change which real finger they are closest to (the red fingertip location corresponds to the index finger in Figure 4.7 (a) but the ring finger in (f)). Apart from the default locations of these points, the fingertip locations are not handled differently from one another (thumb detection functions exactly the same as index finger detection etc.). The gesture detection after Figure 4.7 (f) would perform the same as the gesture detection before the shaking occurred. However, during this sequence gesture detection is not stopped and some fingertip points switch between being active or not (they move closer/farther from the hand center). In Figure 4.7 (c) there are three active fingertips which would invoke the scroll gesture; in Figure 4.7 (d) there are 4 active fingertips invoking the rotate gesture. While the fingertip locations are correctly recovered, the gesture detection step still occurs even on frames where the number of active fingertips change rapidly. In both of these example motions, the algorithm relies on fingertip locations following the contour to be close to where the candidate fingertip points are expected to reappear.

The candidate fingertip positions are assumed to be reliable; there is no limit on how far a fingertip point can move if it is assigned candidate points during the k-Means step (unlike finding the closest contour point). The candidate fingertip points do not have to be present every frame. Following the contour has sufficient accuracy for slow motion especially when the fingertip position is occasionally corrected using the candidate points. An example frame without every fingertip having candidate

points is shown in Figure 4.8. The index finger has no candidate points while the thumb less than the other three with points. This would not cause the fingertips to be lost due to the contour being so close to where the candidate points were before being lost. Figure 4.9 shows an example sequence of a fingertip point without candidate points moving down the finger before being corrected. The ring finger point (dark green square) moves further down the finger between Figure 4.9 (a) and (b). Once the candidate points become available again in Figure 4.9 (c), the point returns to the top of the finger.

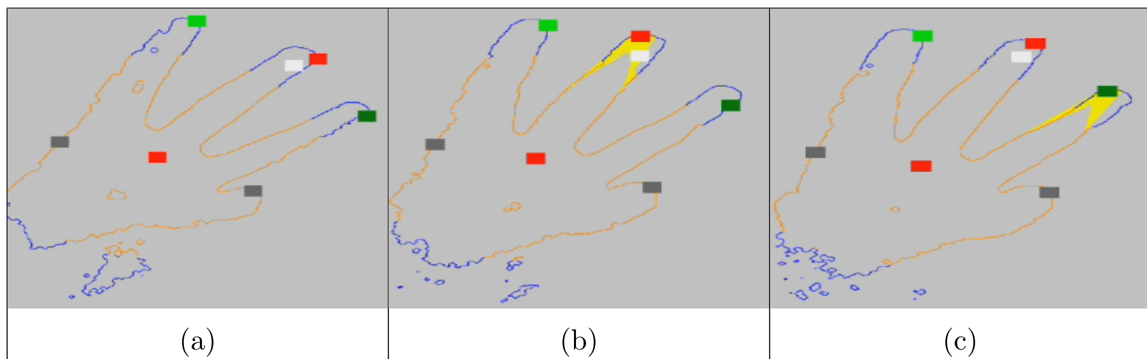


Figure 4.9: Candidate Fingertip Point Correct

Inaccurate fingertip points can cause the algorithm to fail however. If there are a sufficient number of inaccurate points it affects the positions determined during the k-Means. Once a fingertip location is moved far away from the correct position it is hard to correct. An example erroneous detection is shown in figure 4.10. Here, one of the fingertip points would be moved to the the candidate points found in the wrist. It is possible for a user to see the incorrect fingertip locations and position their fingers close to the incorrect positions however the algorithm will not recover naturally without such help. Once moved far from the correct positions, the fingertip points will not be assigned any candidate points during the K-Means step and will therefore be stuck following the contour.

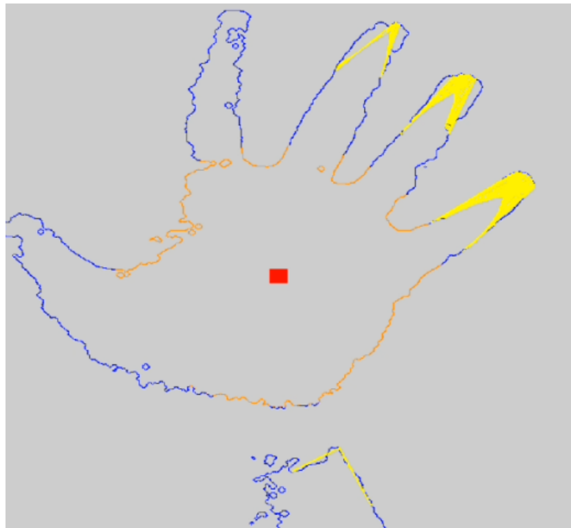


Figure 4.10: Incorrect Wrist Detection

These cases stem from a combination of incorrect background subtraction and the fact that the contour generation / thinning method used here is not sufficient to generate perfect 1-pixel wide contours. The fundamental problem in generating a thin, continuous contour only relying on information in local neighbourhoods is a hard problem. Consider the contour in the following figure 4.11.

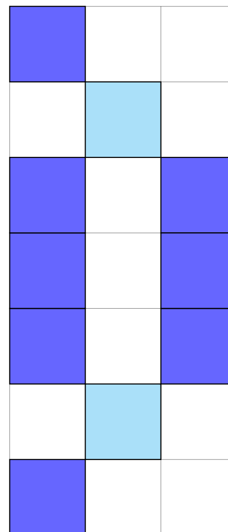


Figure 4.11: Non-thin Contour Example

In this case, ideally only one of the two paths connecting the cyan points. Both of the cyan, non-thin points would have the exact same 3x3 neighbourhood and neither can be removed without causing a break in the contour. Increasing the window size

would allow both paths to be seen by a single thread allowing for only 1 path to be included in the final contour. Increasing the window however increases the number of memory reads required as each point on the contour reads its neighbourhood. Even then, this would only solve the problem for split paths which are 3 pixels. Finding 4 pixel long paths would require an even larger window size to be searched. There are known methods for generating high quality contours sequentially but transferring the contour from the GPU to the CPU and back is prohibitively expensive (see the following section 4.2.2). Finding a good quality contour using global information (sequentially following the border) is a solved problem but it is unknown whether this is possible in a parallel algorithm using only local information.

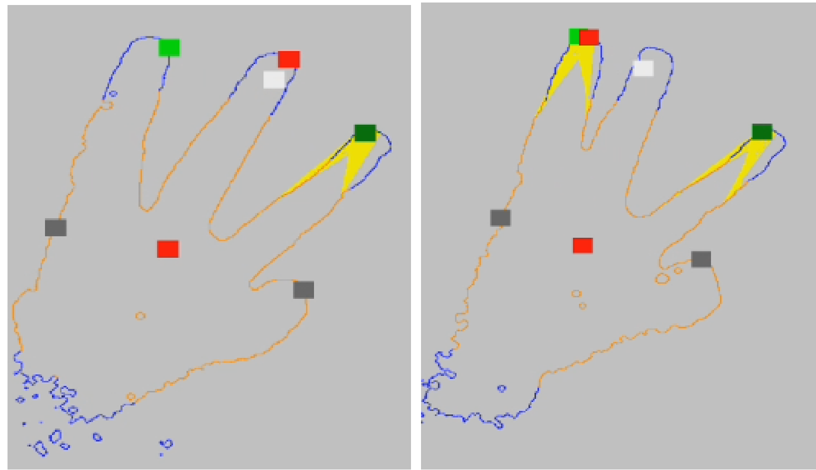


Figure 4.12: Fingertip Position Error versus Gesture Error

Quantitative accuracies of the gesture detection were not given in this work due to the gesture tracking relying solely on the fingertip detection. If there are no errors in the fingertip detection, then there are also no errors in the gesture tracking. The gesture detection uses a small percentage of the overall computation time (see the following Section 4.2) and are linear functions of the fingertip positions. For example the zoom gesture finds the distance between the two raised fingers and converts it to a scale factor. The scale is the current distance divided by the distance between the two fingers when they first were raised (first frame there were two active fingers). No new errors can be introduced during the gesture detection. However, the magnitude of errors in fingertip detection is separate from the magnitude in error in the gesture detection.

Consider Figure 4.12 which shows fingertip detection failing and placing two fin-

gertips on one actual finger. This is a larger error for the fingertip detection; one of the means is assigned points from another finger and incorrectly moves to the wrong finger. The effect on the detected gesture would be small however. The scroll gesture (detected when there are 3 active fingers as there are in this figure) uses the change in the average fingertip position (white square) to control movement of an image. Due to the white square moving a small amount, the error would probably be hard to notice when performing the gesture especially if the hand was in motion (image would shift by the same amount as the white square). Fingertip detection errors which change the number of active fingertips (e.g. finger is considered raised when it is lowered) change which gesture is detected and would have a larger impact on gesture tracking compared to the example error in Figure 4.12.

4.2 Compute Performance

4.2.1 Overview

Table 4.3: Runtime of Each Step

Operation	Time (ms)	Standard Deviation
Codebook Generation	92.078	3.934
Codebook Background Subtraction	29.495	2.025
Morphology	16.451	7.066
Hand Location	6.681	1.772
Contour Generation	5.347	0.967
Contour Thinning	7.693	1.046
Fingertip Detection	4.356	0.694
Fingertip Refinement	33.330	1.163
Gesture Recognition	1.819	0.721

Every step of the algorithm except the codebook generation is recomputed on every frame. The time required for each step in milliseconds is shown in Table 4.3. The algorithm is able to achieve approximately 10FPS including the time necessary to draw all graphical elements such as buttons and sliders used to experimentally set thresholds, display of the hand contour, and the white arrow and map shown in Figure 1.1. Note that this does not include drawing the lines representing the candidate

fingertip points (hand with the coloured squares but not the yellow lines) as the drawing of these lines is expensive and not necessary for correct operation (drops frame rate to roughly 8FPS).

The results indicate that, of the online steps, K-Means clustering and background subtraction (including morphological cleaning) are the most expensive by a large margin. Improvements in these steps could lead to higher frame rates. Gesture recognition in particular is the least time consuming step so more analysis could be done here to improve tracking of the gestures.

It is important to note that these timings were computed using similar hand scales an example of which is shown in figure 4.13. The number of foreground pixels impacts the runtime of some of these steps so a consistent amount was used during the timing process. The figure shows the result of doing background segmentation for a hand roughly 2-3 feet from the camera.

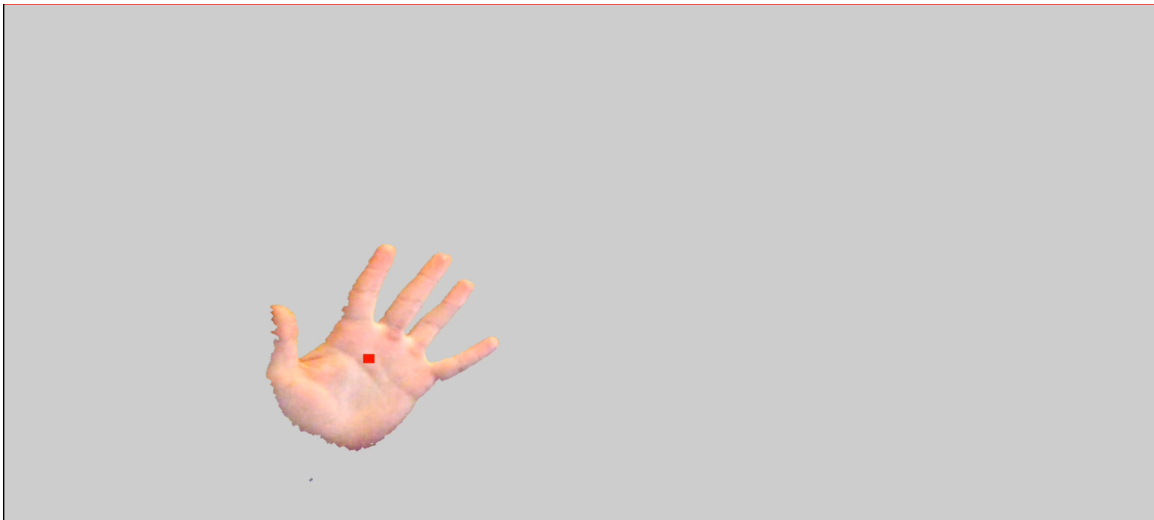


Figure 4.13: Timing Scale

In aggregate, the relationship between frame rate and foreground scale is not one to one; doubling the scale of the hand by moving it closer to the camera causes the frame rate to drop to approximately 8.5 FPS. However, some individual steps are heavily impacted. For example, the moment accumulation sub-step of hand location (detailed in the following section) at the stated scale takes approximately 2.5 milliseconds to process. Doubling the size of the hand in the input image causes this to raise to roughly 10 milliseconds. All timings were computed using the following sequence of steps:

- glFinish is called which, forces the CPU to wait for all OpenGL processes to finish
- start time is recorded using System.nanoTime which is a monotonically increasing time since some arbitrary point (and is therefore not based on wall clock time).
- all OpenGL compute shaders necessary to perform the operation are run
- glFInish is once again called
- end time is recorded with System.nanoTime

4.2.2 Individual Step Compute Time

This section gives a more detailed view of the runtime of each individual step compared to the coarser steps detailed above. These were timed differently compared to the previous section. The overview aimed to give the runtime as the implementation currently exists. It includes error checking, logging as well as sending values from the various UI sliders to the GPU. When performing the evaluation of the method, good default parameters were found and continually used rather than updating the parameters for each environment. The start and stop timer functions here were around only the launch compute shader method rather than around everything occurring in the application. Overall, the totals from the following tables have a sub ten percent difference to the overview table.

Codebook

Table 4.4: Codebook Sub-steps

Operation	Time (ms)	Standard Deviation
Convert to YCbCr	2.399	0.734
Codebook Background Subtraction	27.161	1.369

The primary cost of the codebook background subtraction is the actual reading through the codebook. The conversion to YCbCr requires no conditionals and roughly 10 multiplications per pixel. The conversion time taken, unlike most steps, is not affected by the input image; all pixels must be converted to YCbCr to be compared

against the codebook. The codebook background subtraction requires multiple comparisons and is dependent on the input image. Each foreground pixel is compared to each element of the codebook. With background pixels, computation stops on the first matching code element. This step is the second most expensive step behind the codebook generation.

Hand Location

Table 4.5: Moment Sub-steps

Operation	Time (ms)	Standard Deviation
Moment Initialization	2.192	1.297
Moment Accumulation	2.300	0.573
Moment Find Mean	0.840	0.516
Max Distance Threshold	2.246	0.839

The moment steps here are repeated twice per frame. The accumulation involves many values (every foreground pixel location) being added to a single memory locations using atomic addition. As such, the run time is highly dependent on the number of foreground pixels. Running the accumulation on an all foreground image is roughly ten times more expensive (roughly 20ms instead of 2ms). In the find mean step a single thread on the GPU divides the X and Y accumulation values by the number of contributing points.

Contour

Table 4.6: Contour Creation Steps

Operation	Time (ms)	Standard Deviation
Contour Generation	5.347	0.967
Contour Copy	4.680	0.807
Contour Thinning	2.238	0.491
Contour Read/Write Buffer	123.901	8.753

A large cost of the contour thinning method is the time required to copy the contour. The thinning method takes a contour and removes points based on properties of their

neighbourhood. This cannot be done in parallel, in place as the neighbourhoods would change mid-update creating race conditions or inaccurate results. As such, the contour is copied so the thinning step has two identical contours one of which is used as input to determine which points to remove from the output contour. The copy process unfortunately requires more time than the actual thinning. Additionally, tests were done to determine the cost of reading the contour buffer. The cost is higher than every other steps combined given its size (one value per pixel but it is sparse). Generating a contour sequentially on the CPU would produce 1-pixel wide, continuous contours but would require sacrificing more than half of the current frame rate.

Fingertip Detection

Table 4.7: Fingertip Detection Sub-steps

Operation	Time (ms)	Standard Deviation
Fingertip Detection Preprocessing	2.173	0.519
Fingertip Detection	2.365	0.485
Fingertip Detection Without Preprocessing	17.154	2.083

The preprocessing allows the fingertip detection to be performed in roughly a quarter of the time required without it. With the preprocessing step, contour traversal entails reading each contour point encoding to determine the location of the next neighbour. Without preprocessing, each of the eight neighbours need to be read to find the next contour point. As such, eight reads are required per iteration of the contour traversal instead of the one read necessary when using the preprocessing step. The detection without the neighbourhood encoding is not roughly eight times slower despite requiring eight times the number of memory reads.

This is due to the fact that the fingertip detection utilizes a minimum distance threshold; only contour points sufficiently far from the palm center are checked for having a low angle between them and their neighbours. During the detection step there is no difference between detection with or without the encoding due to the minimum distance threshold; points inside the minimum distance are only examined if they lie close to fingertips. A point in the space between fingers would need to be examined during a contour traversal starting on the side of a finger but a point along the bottom or side of the palm would likely not need to be examined. No contour

traversal would ever use the encoding at these locations as they are far from contour points outside of the minimum distance threshold. The preprocessing step does not determine if contour points within the minimum distance threshold are close to points outside it. In the preprocessing step, every contour point neighbourhood is examined and encoded regardless of the location of the point. In the fingertip detection without a preprocessing step, while it examines the same neighbourhoods multiple times, it does not examine every neighbourhood.

Fingertip Refinement

Table 4.8: K-Means Sub-steps

Operation	Time (ms)	Standard Deviation
K-Means Label Initialization	4.239	0.769
K-Means Means Initialization	0.872	0.303
K-Means Label Update	1.378	0.418
K-Means Means Reset	0.768	0.437
K-Means Means Update	1.398	0.322
K-Means Means Update Finalize	0.898	0.408
Find Closest Contour Point	4.129	0.723
Correct Lost Fingertips	0.786	0.186
K-Means Read/Write Buffer	0.602	0.700

The K-Means operations have a few characteristic properties. First all steps in between the double lines are those which repeat for each iteration. By default these steps occur five times per frame. Also the steps *reset*, *update finalize* and *correct lost fingertips* are all sequential methods; they do not benefit from parallelism by being run on the GPU. All operate on the means buffer (shown in Code Snippet 3.4) which, unlike the labels buffer, is not one buffer element per pixel. In all of these steps, each mean is updated once. In reset the accumulation values are set to zero, in update finalize the position is conditionally set based on the accumulation values and number of contributing points, and the correct lost fingertips sets the position based on the closest contour point.

These steps are performed on the GPU but do not utilize the architecture. These could be done on the CPU instead with some performance benefit if the cost of

transferring data between the CPU and GPU is low enough. The cost to read and write back the K-Means is given in Table 4.8 as *read/write buffer*. This simply reads the data from the GPU, and writes back the data unconditionally. The runtime is lower than the largely sequential methods however the results become less stable as the standard deviation is higher. The potential improvements may be out weighed by the inconsistency. In either case, these steps are not bottlenecks as both the label initialization and finding closest contour points have comparable run times and all of the other loop steps (label and means update) take more time. Of these, some steps offer little room for improvement such as the label initialization. This step sets the entire buffer to one value unconditionally.

Gesture Detection

Table 4.9: Gesture Detection Sub-steps

Operation	Time (ms)	Standard Deviation
Gesture Preprocessing	0.8279	0.508
Gesture Detection	0.002	0.007

The most significant part of the gesture detection step is the preprocessing where the center of the fingertips, the distance to the center from each fingertip and the average change in angle around the center are computed. The detection step is all done in Java and consists of a switch on the number of active fingertips where each case at most has a few multiplications/divisions.

4.2.3 CPU vs GPU

The algorithm presented here is designed specifically to take advantage of a GPU. A CPU version would need to be altered in a few ways to make comparisons between the implementations meaningful. Many steps presented in this work operate on large but sparse buffers (foreground image, contour buffer, K-Means label buffer etc.). Processing is done on a per pixel basis; each step is designed around investigating the local neighbourhood. The majority of threads in these steps do nothing after reading a zero in the buffer. Only threads which read non zero values do anything. Time would be wasted sequentially iterating over these large structures.

In addition, some steps were added due to processing only examining local windows rather than storing global data. For the contour thinning/generation for example, each pixel was included in the final contour based only on the surrounding eight neighbours. This produced contours which are imperfect, containing thick points or holes. Generating contours via border following on the CPU, while likely slower than a GPU implementation, would produce better quality contours potentially eliminating the need for later corrective steps.

During the fingertip detection step, each thread does not have information like how many candidate fingertips were already found, their locations etc. It is likely that these candidate fingertip points could be reduced to one point per finger during the detection step instead of utilizing a clustering algorithm. Contour points close to each other form similar angles with their distance neighbours (angle used to determine fingertip points). The algorithm described in this work does not make use of this fact; every contour point is examined separately to avoid the need for sharing data between threads or using concurrency locks. If implemented on the CPU, this would involve two nested iterations, one for every contour point and another for the neighbourhood size. A proper CPU implementation would not need this inner loop over the neighbourhood size. Instead, after the neighbours of the first contour point are found, they are stored and contour traversal happens for the point and its two neighbours. The angle at each contour point could be determined without traversing the contour each step to find a given point's neighbours. Fingertips would be then be the local minima of angle across the contour. Doing so would reduce the computational complexity from O(N*M) where N is the number of contour points and M is the neighbourhood size to just O(N+M). The GPU approach described in this work could not use this optimization as it relies on sequential processing. The approach does redundant processing (contour traversal which start close together examine mostly the same points) but it allows each thread to operate independently capitalizing on the strengths of a GPU.

If this algorithm was implemented as stated on a CPU it would likely be many times slower than the measured GPU implementation. However, this is not a fair comparison as a CPU utilization should make use of global data rather than the sole use of isolated neighbourhoods detailed in this work. Many steps in this approach perform work which could be avoided in a sequential approach. On a GPU the extra work performed by each thread is offset by the fact many threads can work at the same time.

Chapter 5

Conclusions

The method presented in this work introduced a memory management scheme for the codebook background subtraction method, a novel encoding scheme used to speed hand contour traversal, use of a GPU implementation of the K-Means algorithm to reduce potential points to individual fingertip points. It was designed specifically to capitalize on the mobile GPU hardware. Steps were designed in a way to maximize the amount of work which could be done in parallel. Overall the algorithm works well in cases where the background is sufficiently different from the foreground. The main strength of the proposed approach is its ability to accurately track fine hand movements. Given the high input resolution (1920x1080) and the fact that the algorithm runs in real-time (at 10FPS), small displacements and angle changes can be tracked even with complicated backgrounds.

All fingertips can be tracked correctly with good accuracy (77.83% of frames in the good lighting case) and when there are errors, the algorithm on average corrects within 1.5 seconds (14 frames). The gesture tracking works well given the performance of the finger tracking. The size and location of on-screen images could be remotely control with fine accuracy. When the fingertip detection failed the gestures were incorrectly tracked during these frames but these errors did not persist. In this work the gestures tracked were simple; computing the gestures from the fingertip locations took less than one percent of total compute time. The method focused on tracking fingertip locations and could be used for other gestures or, the current ones could be improved by smoothing the rate of change of the controlled images for example.

Future work will address several key areas. First, there currently are a large number of parameters which govern how the application performs. These include things such as in training how quickly the code elements grow, the number of code

elements, as well as parameters used in the live phase such as how many iterations of K-Means to perform, the angle threshold used to classify fingertips and the distance traversed on the contour when finding fingertips. The required distance to travel on the contour for example is dependent on the scale of the hand in the image. Hands closer to the camera needed a higher value compared to hands farther away. These parameters could be automatically configured given the input image (e.g. distance traveled should be some function on the scale of the hand). A summary of these parameters in given in Appendix A.4.

Second, the method may have an increased performance benefit from splitting work between CPU and GPU. All steps except the final gesture recognition run on the GPU. The time required to transfer data was examined, but there may be a more optimal partition of work. The algorithm also could be further optimized. When finger detection fails the fingertip locations track to the contour. Any improvements to frame rate will make this tracking better as currently, this happens on a frame by frame basis. All steps except the final gesture recognition ran exclusively on the GPU. There may be merit to splitting some steps between the GPU and CPU (splitting work between the two would induce a memory transfer cost so any improvements would need to be faster than this transfer speed to be worthwhile).

Third, a contour generation algorithm which runs on the GPU in parallel and produces a 1-pixel wide wide contour without holes would improve the fingertip detection. Having a thicker contour can lead to errors during the encoding phase of the fingertip detection. While the method has means to offset these errors (fingertips track the closest contour point), it is less accurate compared to using the angle detection. It is possible that sacrificing compute performance would improve results. Performing the contour generation on the CPU would likely reduce the frame rate by half of what it currently is given the time necessary to transfer the image data between CPU and GPU is roughly the same as every step combined. However, it is possible that the improved contour would make up for this slower tracking by allowing more accurate results.

Finally this work should be compared to machine learning approaches. The method presented here has some pros and cons versus machine learning algorithms. The training data needed here is low; each background can be trained with less than 100 frames and no training is necessary for detecting gestures. Switching which gestures are detected by this algorithm would be easy as they are only reliant on the fingertip positions. No new training data of users performing new gestures would

be necessary. However, the background training used in this approach is not generalizable. Machine learning approaches typically train on a data set and can then handle unseen data where as this approach has to train on every new background. This approach also handles high resolution input frames in real time. As machine learning algorithms become more efficient they will likely be able to run in real-time on consumer hardware as well.

Appendix A

Appendix

A.1 Appendix A: Comparison of CPU and GPU

GPUs are primarily built, marketed, and optimized for graphics applications. Graphics pipelines, as well as graphics operations like tiling and multi-sample anti-aliasing, are built into hardware.

Immediate mode rendering (IMR) is a graphics pipeline which draws triangles in submission order; triangles are drawn in a first in first out queue regardless of their depth. If a developer specifies the order back-to-front (objects furthest from the camera drawn first), overdraw will occur. The entirety of background objects will be drawn even if parts will later be obstructed by foreground objects. If front-to-back order is used the GPU can perform an early z discard, pixels which are covered by foreground will not invoke overhead from texture look ups, shading, etc. IMR maps directly to the OpenGL pipeline (same steps like vertex processing shading, rasterize etc.), most of the operations are implemented in hardware. IMR makes up the majority of desktop GPU architectures. On desktop, GPUs are a separate device from CPUs. Data must travel from the CPU over some kind of bus to reach the GPU which is not always the case on mobile. Desktop GPUs also have a dedicated bus to retrieve models and textures from system memory. The pipeline approach hides memory access overhead by letting processing occur while data is loading.

Mobile GPUs do not have the space or power luxuries available on desktop. Mobile GPUs are built to conserve as much power and space as possible. While GPUs have their own memory, some is shared with the CPU. Mobile GPUs and CPUs reside on the same chip to save space as well. This means mobile GPUs do not have a dedicated

bus to main memory (which, on top of saving chip space, also saves power). The bus being shared with the CPU making off chip memory access very expensive as the CPU/GPU will need to wait for one another. Mobile GPU architecture is designed to mitigate these limitations and in most cases, tile based rendering (TBR) is used. TBR is a divide and conquer approach to drawing graphics. The screen is broken up into tiles to be rasterized separately. Splitting up the scene means each tile requires a small amount of memory which, can fit in GPU hardware buffers.

Mobile processors also use deep pipelines; many parts of execution (instruction decode, memory fetch etc.) are done simultaneously to increase throughput. Branches require the pipeline to be emptied as it is assumed next sequential instruction will be executed. Desktop processors use pipelining as well, but have more advanced branch prediction and mis-branch cost mitigation (like out of order execution). Desktops both minimize the number of missed branches and have a lower cost for missed predictions. Mobile CPUs are not as sophisticated and suffer large penalties for missed predictions so algorithms should minimize the use of conditional expressions where possible.

A lot of mobile architecture differences to desktop GPUs come down to which operations are supported in hardware. Floating point operations are very expensive on mobile as they are not implemented in hardware unlike desktop. This reduces bandwidth requirements by decreasing the number of texture reads from memory. Tegra 4 (a 2013 NVidia GPU) has dedicated core types [39]. Their GPU has fragment shader cores and vertex shader cores which are each optimized for different operations (in contrast to an ARM GPU from the same year which uses unified core types [40]). To complicate all of these differences, some mobile platforms (NVidia Tegra has an ARM CPU for example) include processors from multiple vendors. The newest mobile GPU architecture from NVIDIA (Tegra X1 [41]) mirrors its desktop counter part except for the memory layout. Both employ specific hardware for graphics operations like tessellation and vertex manipulation but there are no longer has separate vertex and fragment cores.

A.2 Appendix B: GPGPU Library Selection and Terminology

This section gives a general overview of the general purpose GPU tools available on Android and the rationale behind which library is used for this work.

A.2.1 OpenCL

Open Compute Library provides a framework for doing parallel computations on heterogeneous systems. OpenCL treats a CPU differently from a GPU or digital signal processor (DSP), while providing a uniform programming model across processor types. Data parallelism (workers perform same operation on different parts of the data) and task parallelism (different tasks on same data - may require some form of currency control like locks) are both supported. The maximum number of compute units that can be set, and other hardware restrictions, are defined in environment variables dependent on the device. Debugging on Android proved to be challenging; the interplay between Java, C/C++ and OpenCL are impossible to fully trace execution step by step.

A.2.2 Renderscript

Renderscript (RS) is an Android specific language for parallel applications. RS is derived from C but functions can be called directly through the Andriod SDK in Java. RS hides low-level hardware features and details; threads are assigned to computational devices automatically and it is not possible to trace where threads are assigned or to specify a number of threads. A RS kernel may run on GPU on one device but on CPU for another. The application developer has no control over where the code is run, nor do they have a way to determine where it is run without inferring from run times.

A.2.3 OpenGL ES

Open Graphics Library Embedded System was not built to support GPGPU applications. OpenGL utilizes a rendering pipeline specifically for drawing graphics; it operates on polygonal data to produce a frame buffer to be drawn on a device‘s screen Before OpenGL ES 2.0, the rendering pipeline was not flexible enough to support arbitrary, non-graphics code development. 2.0 introduced vertex and fragment shaders which can be used for general computations with some caveats. Shaders are simply programs which run on the GPU. Vertex shaders are specifically used to modify vertexes of polygonal data while fragment shaders operate on what colour is drawn at a particular pixel location. These will only accept certain data formats.

OpenGL only renders objects when a call to draw objects is made. General

computations can be done by ”rendering” objects off screen (which invokes these shaders that can have a purpose other than graphics). Odd view ports can be used to format the data as well. For example, setting the view port to 256x1 can be used for computing a grayscale histogram. OpenGL ES requires data to conform to some restrictions for example only certain kinds of texture layouts can be used for input data (for example 4 channel red, green, blue, and alpha textures).

OpenGL is still used for GPGPU despite these limitations because it is widely supported. OpenGL is cross platform and documentation is bountiful. In 3.0, compute shaders are added which allow for more flexibility as these programs run separate from set graphics pipeline unlike vertex and fragment shaders.

A.2.4 Choice of Library

Writing code for mobile GPU for non-graphics applications is fairly difficult. OpenCL is not supported much on Android, Renderscript task scheduling control is insufficient, and OpenGL ES pre version 3.0 requires data to have an odd format. Hardware vendor specific solutions are available like NVIDIA’s CUDA but these are not portable. For this work OpenGLES 3.1 is used (which requires use of a newer device) as it provides general use processing. While possible to do in other libraries, it also comes with the benefit of easily displaying results given.

A.2.5 GPGPU and OpenGL ES Terminology

Describing general purpose GPU terms is difficult as there as different vendors use different names for the same concept. A warp on a NVIDIA device is synonymous with a wavefront on an ARM device, CUDA threads/thread blocks are equitable to work items/work groups etc. This work utilized OpenGL ES (which has similar terminology to OpenCL) on a NVIDIA device. The described terminology will focus on the terms that library and device use.

GPUs have more cores compared to a CPU. For example, the device used in this work has a 4 core CPU but a 256 core GPU. GPU cores are not fully featured however and do not have hardware support each core to act independently. Groups of cores must all run the same instruction. If cores in a group diverge (for example cores evaluate an if statement or other branch differently), the instructions need to be repeated multiple times. The group will repeatedly run the instructions for one branch with the divergent cores disabled until all branches are executed. These cores

are a Single Instruction Multiple Data (SIMD) architecture; there is one instruction that all the cores execute but they operate on different data. CPU hardware also uses this architecture with vector operations (for example a 128 bit register can hold 4 32-bit floating point numbers allowing 4 operation to occur simultaneously). However, the potential width, that is the number of operations which can run at the same time, is higher on a GPU.

To encapsulate this hardware quality, OpenCL uses the terminology work item and work group for threads which the user spawns. Each work item executes the same instruction but works on different data. Consider an image processing task that is a function of pixel colour values. Each work item would operate on a single pixel, each would read a different location of an image but all would perform the same operations. The image would be broken into multiple work groups each consisting of a number of work items. During processing, these work groups are executed by hardware threads which run on groups of cores. In NVIDIA terminology, a warp consisting of 32 cores would execute the work groups. The size of the problem (in this case image size) determines the number of work items / work groups needed where as the number of number of warps is determined by hardware. For a per-pixel image processing task there there would be 1 work item per pixel.

Work groups also need to be independent. The order of work groups processed is not guaranteed without explicit synchronization. When a computation task is split into work groups, these are processed by the GPU as a queue. The amount of hardware resources determines how many work groups can be processed simultaneously. As work groups are completed, more are scheduled to run on the GPU. The size of the work group is dependent on the hardware; the work group size should be a multiple of the warp size so the work groups can be evenly divided across hardware resources [42].

In summary programs which run on the GPU use work items as a unit of work that should be as independent as possible from other work items. These are organized together into work groups whose size is dependent on the hardware. The work groups are essentially jobs which are enqueued to be run on the GPU warps. The software used in this work to generate the work groups is OpenGL ES 3.1 compute shaders.

Compute shaders are programs which run outside of OpenGL's graphics pipeline; they do not have to adhere to data restrictions or be run through fixed stages for rendering geometry unlike vertex and fragment shaders. OpenGL Shaders are written in the OpenGL Shader Language (GLSL) which is a C-like programing language which

has some built-in data types. The image data processed in this work was stored in image2d RGBA32f that is an array of pixels consisting of 32 bit floating points per red, green, blue and alpha channels. The other data type used are shader storage buffer objects (SSBOs) and uniforms. SSBOs can house arbitrary 32 bit data of various types (floating point, integer, unsigned integer etc.). Uniform data allow small amounts of data from the host CPU to be sent to the GPU. These were used to send the parameters discussed in this work to the GPU.

Compute shaders are compiled and linked from string data during run time. OpenGL provides API calls for allocating program resources, providing source code, as well as compiling and linking shaders. Once successfully linked, the programs are launched in OpenGL by setting the active program (glUseProgram) and then calling glDispatchCompute. Dispatch compute takes 3 arguments corresponding to the work group dimensions.

A notable downside of using compute shaders in OpenGL as opposed to OpenCL is there is no function which queries the device for a recommend work group size. On the hardware tested, the optimal size in terms of compute performance was found empirically to be 8 by 8 by 1. This evenly divides the input (1920x1080 video stream) into 32,400 work groups consisting of 64 work items. Any operation described here which operates on a per-pixel basis would have this many work groups. The given hardware has 2 streaming multiprocessors each which have 4 warps of 32 cores for a total of 256 cores. For this size of work group, each would be split to run on 2 warps (in total 64 cores) allowing for 4 work groups to be scheduled simultaneously.

A.3 Appendix C: Hardware Architecture

The device used for development and evaluation is a Google Pixel C tablet. This uses the NVIDIA Tegra X1 system on a chip (SoC) architecture. This includes a Maxwell GPU and a quad core ARM Cortex A57 CPU. Apart from these the SoC has external interfaces as well as built in processors which include hardware for processing video/audio feed from the devices camera as well as security offloads.

The Maxwell GPU contains 2 streaming multiprocessors (SMM) each of which have 4 warps of 32 cores for a total of 256 cores. Each SMM has specific hardware for graphic applications like vertex fetching and tessellation (graphic operations to retrieve vertex data and build primitive shapes). Each of the 4 warps has an instruction buffer and warp scheduler for distributing work to the cores in the warp. Each

core in the warps execute the same instruction.

The device does automatic lossless compression of 4x2 image regions if there is a repeating pattern. If a region is constant, 8:1 compression occurs to lessen the necessary memory transfer time. If regions of this size do not have repeating patterns, smaller regions (and therefor less compression) are found. Finally if the regions do not exactly repeat but have sufficiently close colour values, the deltas between neighbours are stored. In this work, there are large regions which have repeated colour values. After background subtraction, the image data sent to the GPU has every background pixel colour value being gray.

A.4 Appendix D: Parameter Summary

This appendix provides an overview of the parameters which are controllable in this work (these are the same parameters highlighted in Chapter 3). It includes speculation on what the parameters would be a function of if they were automatically configured (instead of controlled by UI input). For example the distance traversed along the hand contour during fingertip detection would be some function of the scale of the hand. Images with larger hands (closer to the camera) would have larger contours and therefore this distance would need to be bigger compared to smaller hands (farther from the camera).

A.5 Codebook Parameters

All of the codebook parameters depend on the background used. Changing these parameters affects the codebook model generated and thus dictate how well the background subtraction performs. The *I value* parameter is the basis of the codebook model and if it is changed, every other codebook parameter would also likely need to be changed for the background subtraction step to perform well. *I value* would unlikely be set problematically given the complexity inherit in changing this parameter. The number of code elements and the replacement scheme used affect compute costs in addition to changing background subtraction results (unlike the other parameters). These were chosen based on empirical tests; the output of the background subtraction was deemed to perform the same visually within a range of code elements (3-4) as well as between the replacement schemes (excluding no replacement). These parameters were instead set based on the computation time required. These parameters

would likely not need to change with different environments if the visual performance remained similar.

Evaluation of this work was done on multiple backgrounds using the same parameters so these parameters may not need to be adjusted for all backgrounds. More testing would need to be done to determine if these parameters should be automatically configured for different backgrounds. Note that, apart from *minMod* and *maxMod* which affect background subtraction after training, changing the parameters would require training to be done again as they affect codebook generation.

If these parameters were to be automatically configured an initialization step would likely be required. The user would hold up an open hand and if, for example, too much of the hand was removed during background subtraction *bounds* or *I_Inc* would be lowered. This would reduce the size of the code elements and therefore less of the image would be removed. The relationship between the background and how these parameters should be set is complicated however. Multiple parameters control the size of the code elements and there are trade offs between removing more background and removing parts of the hand during background subtraction. This work can handle noise isolated from the hand if the noisy regions are small (can be removed with morphology) but detecting this noise pattern may be difficult.

Table A.1: Parameters of the Codebook Step

Parameter Name	Description	Default Value
bounds	initial width of each code element. When a new *I value* comes in during training, the initial *I_Low* and *I_High* of the new code element are set to be *I* +/- *bounds*.	0.02
I_Inc	amount the range of a code element is extended during training when an update occurs. If the new pixel's *I value* lies within *bounds* distance to *I_Low*, then *I_Low* is decreased by this amount. Similarly, if the new pixel's *I value* lies within *bounds* distance to *I_High*, then *I_High* is increased.	0.01

minMod	during background substitution each code elements' *min* value is decreased by a small static amount	0.01
maxMod	like the minimum value, the maximum value is raised by a set same amount. These parameters widen the range of values the code elements will mark as background. This is done to try and prune as much background as possible at the cost of over pruning some foreground.	0.01
I value	the basis of the code elements. Other methods could be used to calculate this value.	$\sqrt{Y^2 + Cb^2 + Cr^2}$
Number of Code Elements	the number of code elements can also be configured. As stated previously higher values quickly degrade the compute performance at little benefit to segmentation.	3
Replacement	determines what happens when there are more unique *I values* than space to create code elements for all of them.	*least used*
Merging	Boolean value which indicates whether or not overlapping code elements should be merged	true

A.6 Hand Localization

These parameters change how the morphology sub-step performs, both in how the foreground changes as well as the compute cost. As the number of texture reads increases so does the compute time (larger structuring element require more reads as does an increased number of operations). Generally, for the codebook parameters used in this work, images had more false positives than false negatives; the entire hand was kept as foreground but some parts of the background were not removed. This noise pattern would change if the codebook parameters were changed and as

the goal of this step was to remove noise, the morphology parameters would need to be changed as well. Given the the codebook parameters remained static in testing for this work these parameters, the morphology parameters also were not changed. Additionally the shape of the structuring element might not need to be altered at all. The cross shape performed better than the other shapes in testing both in the compute time required and in the visual output. As with the codebook parameters, automatically adjusting these parameters would likely require a initialization step and they would then be based on noise in the resulting mask after background subtraction.

Table A.2: Parameters of the Codebook Step

Parameter Name	Description	Default Value
Structuring Element Shape	shape of the structuring element used. Three shapes were tested: box, cross and diamond.	Cross
Structuring Element Size	the size of the structuring, must be an odd size box	5x5
Morphology Operations	what combination of erodes or dilates done.	Erode, erode, dilate

A.7 Fingertip Detection

As stated in the introduction of this appendix, the *Number of Neighbours* traversed parameter if automatically set would be a function of the size of the hand in the input video. This parameter along with the
textitMinimum Distance Threshold both would be the simplest of any of the listed parameters to set automatically as depend solely on the scale of the hand in the input. The *Minimum Line Length* also would be set according to scale but would need to be some number less than the distance traversed on the contour (otherwise no candidate fingertips would ever be detected).

The *Minimum Angle Threshold*, *Remove Lines with Breaks* and *Minimum Line Length* parameters were introduced to reduce the effect of errors. Apart the *Minimum Line Length* parameter, these would unlikely need to be changed. Actual fingertips

on hand contours would not form angles that small and contour breaks indicate errors in the contour which can cause unpredictable angles.

Table A.3: Parameters for Fingertip Detection

Parameter Name	Description	Default Value
Number of Neighbours	how far along the contour the fingertip detection travels	100 neighbours
Minimum Angle Threshold	angle between $P_iP_{i\pm j}$ must be larger than this	10 deg
Maximum Angle Threshold	angle between $P_iP_{i\pm j}$ must be smaller than this	75 deg
Minimum Line Length	the lines $P_iP_{i\pm j}$ must be a certain length on top of forming a small angle	60 pixels
Remove Lines with Breaks	Boolean representing whether contour traversal which stopped due to a break should be removed	true
Minimum Distance Threshold	points on the contour must be a minimum distance away from the center of the palm to be considered fingertips	175 pixels

A.8 Fingertip Refinement

These parameters control how the K-Means algorithm operates. In particular, the *Reset Mean* would ideally be made automatic in future work. Currently, the user is required to change this parameter once their open hand is in frame. While this input is only necessary on the first frame or when reseting the algorithm, it should be automatically set when five fingers are detected in frame (5 clusters of candidate points roughly in the shape a hand appear). The algorithm also assumes that the means do not deviate from expected fingertip positions too much. Means far from fingers are assigned no points by design as this allows the algorithm to handle lowering of fingers by tracking the contour (means representing lowered fingers are assigned

no candidate points and are not moved based on raised fingers). However, if the means move far from the actual positions of the fingers due to a false detection the algorithm can get stuck. Detecting when these errors occur (for example means are no longer in a shape possible by a hand) would allow the algorithm to reset and make reacquiring the fingertips easier. Knowing when the algorithm deviated to this extent may require more computation time though.

The number of K-Means iterations is a parameter which will not need to automatically be set for different environments. The method was designed in a way to allow the K-Means algorithm to perform well with a low number of iterations. Unlike traditional K-Means, the algorithm used here uses a set number of iterations instead of running until convergence. This means no computation time needs to be spent detecting if the K-Means step has converged.

The number of labels necessary to move a mean as well as the maximum distance means can be moved to the closest contour points would both be set based on the scale of the hand. Larger hand contour would have more candidate points so the number of candidate points necessary to move the mean could be higher. Also the *Reposition Threshold* parameter was introduced to prevent means from switching between fingers when tracking the nearest contour point. Larger hand contour would have more space in between fingers so this threshold Would be based on scale.

Table A.4: Parameters of the Fingertip Refinement Step

Parameter Name	Description	Default Value
Reset Mean	Boolean representing if the mean positions should be reset	true until acquisition
Iterations	number of K-Means iterations performed	5
Reposition Threshold	limit on how far the correction step can move a mean	50 pixels
Number of Labels Threshold	number of candidate points necessary for a mean not to be considered lost	5

A.9 Summary

It is important to note the parameters which control the output of the presented algorithm were not changed during evaluation of this work indicating these parameters can handle varying conditions to a degree. However, test environments with large differences to those tested would require the parameters in this work to be adjusted. The parameter can be broken down into three broad categories: parameters which do not need to be changed, parameters which would vary with scale, and parameters that require detecting a change in the output.

The parameters which would not change in different environments are those that represent properties inherent to the method. These include the maximum angle threshold during fingertip detection or the number of iterations of the K-Means algorithm to perform. Detecting fingertips by finding small angles across a hand contour would always have a similar upper angle threshold regardless of scale or background. This algorithm was designed to reduce the number of necessary K-Means iterations so a higher or varying number should not be necessary.

Parameters which change with scale would be easy to calculate; all of these could likely be set automatically based on the number of foreground pixels relative to the input video resolution. Configuring these would not require much additional computation time given the number of foreground pixels is calculated as a part of determining the hand location.

The final group of parameters are those which would be complicated to calculate or compound together in non obvious ways. For example, consider the problem of too much of the hand is removed for a useful contour to be generated. This could be solved by altering any parameters which govern the size of the code elements during training, modifying the *minMod* and *maxMod* parameters which extend the code element ranges during live background subtraction, or by adding additional dilation steps to the morphology. These could be combined to varying degrees (use one parameter or change multiple to a lesser extent) and each come with trade offs. Additionally, some of these parameters may need additional computation, for example the *Reset Mean*. This would require assessing the location of the means each frame to determine if they moved outside of a possible hand configuration which, would add computation cost.

Bibliography

[1] OpenStreetMap contributors, *Planet dump retrieved from https://planet.osm.org*, https://www.openstreetmap.org, 2017.

[2] M. Szkudlarek and M. Pietruszka, "Fast gpu and cpu computing for head position estimation," in *Computer Science and Information Systems (FedCSIS), 2015 Federated Conference on*, Sep. 2015, pp. 231–240. DOI: 10.15439/2015F410.

[3] M. Konrad, "Parallel computing for digital signal processing on mobile device gpus," Master's thesis, HTW Berlin, Mar. 2014.

[4] N. Otsu, "A threshold selection method from gray-level histograms," *IEEE transactions on systems, man, and cybernetics*, vol. 9, no. 1, pp. 62–66, 1979.

[5] L. Xu, J. Li, and K. Wang, "Real-time and multi-view face tracking on mobile platform," in *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, May 2011, pp. 1485–1488. DOI: 10.1109/ICASSP.2011.5946774.

[6] A. Bulbul, Z. Cipiloglu, and T. Capin, "A face tracking algorithm for user interaction in mobile devices," in *CyberWorlds, 2009. CW '09. International Conference on*, Sep. 2009, pp. 385–390. DOI: 10.1109/CW.2009.9.

[7] G. Wang, B. Rister, and J. Cavallaro, "Workload analysis and efficient opencl-based implementation of sift algorithm on a smartphone," in *Global Conference on Signal and Information Processing (GlobalSIP), 2013 IEEE*, Dec. 2013, pp. 759–762. DOI: 10.1109/GlobalSIP.2013.6737002.

[8] B. Rister, G. Wang, M. Wu, and J. Cavallaro, "A fast and efficient sift detector using the mobile gpu," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, May 2013, pp. 2674–2678. DOI: 10.1109/ICASSP.2013.6638141.

[9] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *Int. J. Comput. Vision*, vol. 60, no. 2, pp. 91–110, Nov. 2004, ISSN: 0920-5691. DOI: 10.1023/B:VISI.0000029664.99615.94. [Online]. Available: http://dx.doi.org/10.1023/B:VISI.0000029664.99615.94.

[10] K.-T. Cheng, X. Yang, and Y.-C. Wang, "Performance optimization of vision apps on mobile application processor," in *Systems, Signals and Image Processing (IWSSIP), 2013 20th International Conference on*, Jul. 2013, pp. 187–191. DOI: 10.1109/IWSSIP.2013.6623485.

[11] H. Bay, T. Tuytelaars, and L. Van Gool, "Surf: Speeded up robust features," in *European conference on computer vision*, Springer, 2006, pp. 404–417.

[12] M. Hassan, M. Zhao, S.-h. Son, H.-s. Lee, H.-g. Kim, and B. Jang, "A low power and high performance face detection on mobile gpu," in *Energy Aware Computing Systems Applications (ICEAC), 2015 International Conference on*, Mar. 2015, pp. 1–4. DOI: 10.1109/ICEAC.2015.7352201.

[13] I. Hemdan, S. Karungaru, and K. Terada, "Facial features-based method for human tracking," in *Frontiers of Computer Vision (FCV), 2011 17th Korea-Japan Joint Workshop on*, Feb. 2011, pp. 1–4. DOI: 10.1109/FCV.2011.5739701.

[14] P. Vadakkepat, P. Lim, L. De Silva, L. Jing, and L. L. Ling, "Multimodal approach to human-face detection and tracking," *Industrial Electronics, IEEE Transactions on*, vol. 55, no. 3, pp. 1385–1393, Mar. 2008, ISSN: 0278-0046. DOI: 10.1109/TIE.2007.903993.

[15] H. Hasan and S. Kareem, "Human computer interaction for vision based hand gesture recognition: A survey," in *Advanced Computer Science Applications and Technologies (ACSAT), 2012 International Conference on*, Nov. 2012, pp. 55–60. DOI: 10.1109/ACSAT.2012.37.

[16] D. Mazumdar, M. K. Nayak, and A. K. Talukdar, "Adaptive hand segmentation and tracking for application in continuous hand gesture recognition," in *Recent Trends in Intelligent and Emerging Systems*, K. K. Sarma, M. P. Sarma, and M. Sarma, Eds. New Delhi: Springer India, 2015, pp. 115–124, ISBN: 978-81-322-2407-5. DOI: 10.1007/978-81-322-2407-5_9. [Online]. Available: http://dx.doi.org/10.1007/978-81-322-2407-5%5C_9.

[17] M. K. Ahuja and A. Singh, "Static vision based hand gesture recognition using principal component analysis," in *MOOCs, Innovation and Technology in Education (MITE), 2015 IEEE 3rd International Conference on*, Oct. 2015, pp. 402–406. DOI: 10.1109/MITE.2015.7375353.

[18] Y. Song, D. Demirdjian, and R. Davis, "Continuous body and hand gesture recognition for natural human-computer interaction," *ACM Transactions on Interactive Intelligent Systems (TiiS)*, vol. 2, no. 1, p. 5, 2012.

[19] G. Marin, F. Dominio, and P. Zanuttigh, "Hand gesture recognition with jointly calibrated leap motion and depth sensor," *Multimedia Tools and Applications*, vol. 75, no. 22, pp. 14 991–15 015, 2016.

[20] Z. Lai, Z. Yao, C. Wang, H. Liang, H. Chen, and W. Xia, "Fingertips detection and hand gesture recognition based on discrete curve evolution with a kinect sensor," in *Visual Communications and Image Processing (VCIP), 2016*, IEEE, 2016, pp. 1–4.

[21] S. Genç, M. Baştan, U. Güdükbay, V. Atalay, and Ö. Ulusoy, "Handvr: A hand-gesture-based interface to a video retrieval system," *Signal, Image and Video Processing*, vol. 9, no. 7, pp. 1717–1726, 2015.

[22] M. Mariappan, X. Guo, and B. Prabhakaran, "Picolife: A computer vision-based gesture recognition and 3d gaming system for android mobile devices," in *Multimedia (ISM), 2011 IEEE International Symposium on*, Dec. 2011, pp. 19–26. DOI: 10.1109/ISM.2011.13.

[23] S. S. Rautaray and A. Agrawal, "Real time hand gesture recognition system for dynamic applications," *International Journal of UbiComp*, vol. 3, no. 1, p. 21, 2012.

[24] Z. Pan, Y. Li, M. Zhang, C. Sun, K. Guo, X. Tang, and S. Z. Zhou, "A real-time multi-cue hand tracking algorithm based on computer vision," in *Virtual Reality Conference (VR), 2010 IEEE*, Ieee, 2010, pp. 219–222.

[25] A. Chaudhary, J. L. Raheja, and S. Raheja, "A vision based geometrical method to find fingers positions in real time hand gesture recognition," *Journal of Software*, vol. 7, no. 4, pp. 861–869, 2012.

[26] A. Bhandari, A. Chopra, and S. Rishi, "Gesture based control system," *IJAR*, vol. 2, no. 4, pp. 656–661, 2016.

[27] C.-J. Liao, S.-F. Su, and M.-C. Chen, "Vision-based hand gesture recognition system for a dynamic and complicated environment," in *Systems, Man, and Cybernetics (SMC), 2015 IEEE International Conference on*, Oct. 2015, pp. 2891–2895. DOI: 10.1109/SMC.2015.503.

[28] V. Bhame, R. Sreemathy, and H. Dhumal, "Vision based hand gesture recognition using eccentric approach for human computer interaction," in *Advances in Computing, Communications and Informatics (ICACCI, 2014 International Conference on*, Sep. 2014, pp. 949–953. DOI: 10.1109/ICACCI.2014.6968545.

[29] W. Ahmed, K. Chanda, and S. Mitra, "Vision based hand gesture recognition using dynamic time warping for indian sign language," in *2016 International Conference on Information Science (ICIS)*, Aug. 2016, pp. 120–125. DOI: 10.1109/INFOSCI.2016.7845312.

[30] P. Barros, N. T. Maciel-Junior, B. J. Fernandes, B. L. Bezerra, and S. M. Fernandes, "A dynamic gesture recognition and prediction system using the convexity approach," *Computer Vision and Image Understanding*, vol. 155, pp. 139–149, 2017, ISSN: 1077-3142. DOI: http://dx.doi.org/10.1016/j.cviu.2016.10.006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S107731421630159X.

[31] D. H. DOUGLAS and T. K. PEUCKER, "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature," *Cartographica: The International Journal for Geographic Information and Geovisualization*, vol. 10, no. 2, pp. 112–122, 1973. DOI: 10.3138/FM57-6770-U75U-7727. eprint: http://dx.doi.org/10.3138/FM57-6770-U75U-7727. [Online]. Available: http://dx.doi.org/10.3138/FM57-6770-U75U-7727.

[32] O. De, P. Deb, S. Mukherjee, S. Nandy, T. Chakraborty, and S. Saha, "Computer vision based framework for digit recognition by hand gesture analysis," in *2016 IEEE 7th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, Oct. 2016, pp. 1–5. DOI: 10.1109/IEMCON.2016.7746361.

[33] A. I. Maqueda, C. R. del-Blanco, F. Jaureguizar, and N. García, "Temporal pyramid matching of local binary subpatterns for hand-gesture recognition," *IEEE Signal Processing Letters*, vol. 23, no. 8, pp. 1037–1041, Aug. 2016, ISSN: 1070-9908. DOI: 10.1109/LSP.2016.2579664.

[34] K. Wang, B. Xiao, J. Xia, and D. Li, "A dynamic hand gesture recognition algorithm using codebook model and spatial moments," in *Intelligent Human-Machine Systems and Cybernetics (IHMSC), 2015 7th International Conference on*, vol. 1, Aug. 2015, pp. 130–133. DOI: 10.1109/IHMSC.2015.202.

[35] Khronos Group, *OpenGL ES*, https://www.khronos.org/opengles/, 2017.

[36] N. J. Leite, "An simd parallel algorithm for classifying binary image contours based on mathematical morphology," in *Image Processing, 1996. Proceedings., International Conference on*, IEEE, vol. 3, 1996, pp. 25–28.

[37] J.-S. Kwon, J.-W. Gi, and E.-K. Kang, "An enhanced thinning algorithm using parallel processing," in *Image Processing, 2001. Proceedings. 2001 International Conference on*, IEEE, vol. 3, 2001, pp. 752–755.

[38] T. Y. Zhang and C. Y. Suen, "A fast parallel algorithm for thinning digital patterns," *Commun. ACM*, vol. 27, no. 3, pp. 236–239, Mar. 1984, ISSN: 0001-0782. DOI: 10.1145/357994.358023. [Online]. Available: http://doi.acm.org/10.1145/357994.358023.

[39] "NVIDIA Tegra 4 Family GPU Architecture," NVIDIA, Tech. Rep., Feb. 2013. [Online]. Available: %7Bhttp://www.nvidia.ca/docs/IO//116757/Tegra_4_GPU_Whitepaper_FINALv2.pdf%7D.

[40] ARM. (2013). Mali-t720 high area efficiency gpu, [Online]. Available: https://developer.arm.com/products/graphics-and-multimedia/mali-gpus/mali-t720-gpu.

[41] "NVIDIA Tegra X1 NVIDIA's New Mobile Super Chip," NVIDIA, Tech. Rep., Jan. 2015. [Online]. Available: %7Bhttp://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf%7D.

[42] "OpenCL Programming Guide for the CUDA Architecture," NVIDIA, Tech. Rep., Aug. 2009. [Online]. Available: %7Bhttp://www.nvidia.com/content/cudazone/download/opencl/nvidia_opencl_programmingguide.pdf%7D.