MSc THESIS

Real-Time Gesture Recognition with a 2D camera

Srinivasan Yadunathan

Abstract

There has been a vast improvement in Human-Computer Interaction over the last decade. Yet there are only a very few systems with natural interfaces such as with speech and gestures. This thesis here addresses the topic of gesture recognition using a 2D camera and how they can be used as natural interfaces to control applications. The gesture recognition algorithm can identify six different gestures and was first developed in a PC and later moved to an embedded platform. A robust background subtraction technique is designed to obtain the hand segment. Two gesture recognition methods are implemented, their performances are measured and the angle-based recognition approach is chosen for its accuracy. The application is moved to an embedded platform i.MX515EVK based on ARM Cortex-A8 processor. To obtain a frame rate suitable for real-time applications, optimizations such as camera capture time reduction, algorithmic optimizations and utilizing SIMD unit of the Cortex-A8 processor known as NEON for data parallelism are performed. As experimentation, the optimized version of the algorithm is used to build a real-time application that recognizes gesture from images to control applications. The performance of the application is studied and a frame rate of 4 - 4.5 frames per second is achieved.
Real-Time Gesture Recognition with a 2D camera

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Srinivasan Yadunathan
born in Tirupur, India
Real-Time Gesture Recognition with a 2D camera

by Srinivasan Yadhunathan

Abstract

There has been a vast improvement in Human-Computer Interaction over the last decade. Yet there are only a very few systems with natural interfaces such as with speech and gestures. This thesis here addresses the topic of gesture recognition using a 2D camera and how they can be used as natural interfaces to control applications. The gesture recognition algorithm can identify six different gestures and was first developed in a PC and later moved to an embedded platform. A robust background subtraction technique is designed to obtain the hand segment. Two gesture recognition methods are implemented, their performances are measured and the angle-based recognition approach is chosen for its accuracy. The application is moved to an embedded platform i.MX515EVK based on ARM Cortex-A8 processor. To obtain a frame rate suitable for real-time applications, optimizations such as camera capture time reduction, algorithmic optimizations and utilizing SIMD unit of the Cortex-A8 processor known as NEON for data parallelism are performed. As experimentation, the optimized version of the algorithm is used to build a real-time application that recognizes gesture from images to control applications. The performance of the application is studied and a frame rate of 4 - 4.5 frames per second is achieved.

Laboratory: Computer Engineering
Codenumber: CE-MS-2011-13

Committee Members:

Advisor: dr. Georgi Kuzmanov, CE, TU Delft
Advisor: ir. Joost Mans, Philips Research, Eindhoven
Chairperson: dr. Ir. Koen Bertels, CE, TU Delft
Member: dr. Rene van Leuken, CAS, TUDelft
I would like to dedicate this thesis to my parents, my brother and all my best friends
# Contents

List of Figures viii

List of Tables ix

Acknowledgements xi

1 Introduction 1
   1.1 Problem Definition . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
   1.2 Thesis Contribution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
   1.3 Thesis Organization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2 Background Study 7
   2.1 Gesture recognition with external devices . . . . . . . . . . . . . . . . . 7
   2.2 Gesture recognition with Vision based techniques . . . . . . . . . . . . . 8
   2.3 Hand Segmentation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
   2.4 Gesture Recognition methods . . . . . . . . . . . . . . . . . . . . . . . . 11
   2.5 Chosen direction in this thesis . . . . . . . . . . . . . . . . . . . . . . . . 13
   2.6 Freescale i.MX515EVK Platform . . . . . . . . . . . . . . . . . . . . . . . 13
      2.6.1 Freescale’s i.MX515 Processor . . . . . . . . . . . . . . . . . . . . . 14
      2.6.2 ARM Cortex-A8 Processor . . . . . . . . . . . . . . . . . . . . . . . 15
   2.7 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

3 Gesture Recognition PC Implementation 17
   3.1 User Interaction with a retro reflective ring and an infra-red camera . . . 17
      3.1.1 Initialization and Set parameters stage . . . . . . . . . . . . . . . . . 18
      3.1.2 Process and Decision stage . . . . . . . . . . . . . . . . . . . . . . . 19
   3.2 Gesture recognition with a 2D camera . . . . . . . . . . . . . . . . . . . . 22
      3.2.1 Background subtraction Stage . . . . . . . . . . . . . . . . . . . . . 23
      3.2.2 Morphological Transformation stage . . . . . . . . . . . . . . . . . . 27
      3.2.3 Contours Detection and approximation stage . . . . . . . . . . . . . 28
      3.2.4 Gesture recognition stage . . . . . . . . . . . . . . . . . . . . . . . 29
   3.3 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

4 Gesture Recognition i.MX515 Implementation 35
   4.1 Porting Algorithm to i.MX515 platform . . . . . . . . . . . . . . . . . . . 35
      4.1.1 Building Linux OS on a SD card . . . . . . . . . . . . . . . . . . . . 35
      4.1.2 Building the gesture recognition algorithm for i.MX515 platform . . 37
      4.1.3 Algorithm Profiling . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
   4.2 Optimizations for the Algorithm . . . . . . . . . . . . . . . . . . . . . . . 38
      4.2.1 Optimization 1: Camera - Capture time reduction . . . . . . . . . . 38
List of Figures

1.1 Example: Using gestures to control medical images . . . . . . . . . . . . . 1
1.2 Opera browser and Google mail gestures . . . . . . . . . . . . . . . . . . 1
1.3 Example of a gesture control device-Cyberglove from Virtual Technologies 2
1.4 Variants of camera used for gesture recognition . . . . . . . . . . . . . . 3
2.1 Process flow for a gesture recognition system . . . . . . . . . . . . . . . . 7
2.2 Gesture recognition using data glove and multi-colored gloves . . . . . . . 8
2.3 3D capture - Stereo camera and structured light approach . . . . . . . . . 9
2.4 Background Subtraction technique - frame differencing . . . . . . . . . . . 10
2.5 Gesture recognition by machine learning approach . . . . . . . . . . . . . 11
2.6 Haar like features used for gesture recognition . . . . . . . . . . . . . . . 12
2.7 Gesture recognition using Convex hull technique . . . . . . . . . . . . . . 12
2.8 Block diagram of i.MX515 Processor . . . . . . . . . . . . . . . . . . . . . 14
2.9 Arm and Thumb ISA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.1 Flowchart for Retro-reflective ring based user-interaction system . . . . . 17
3.2 UI camera used for the setup with its IR - LED off and on . . . . . . . . . 18
3.3 Input image to the system and the ring’s bright spot in output image . . . 19
3.4 Mouse event simulation with retro-reflective ring - Flowchart . . . . . . . 20
3.5 Hand gestures recognized recognized by the system . . . . . . . . . . . . . 22
3.6 Hand Gesture recognition system - List of stages . . . . . . . . . . . . . . 22
3.7 General background subtraction methodology . . . . . . . . . . . . . . . . 23
3.8 Frame differencing and Skin color background subtraction approaches . . 24
3.9 RGB per channel background subtraction - Flowchart . . . . . . . . . . . . 25
3.10 RGB per channel background subtraction technique . . . . . . . . . . . . 26
3.11 Execution time comparison for several background subtraction approaches 27
3.12 An example for 3x3 median based noise filtering . . . . . . . . . . . . . . 27
3.13 Effects of Dilation and Erosion on an image . . . . . . . . . . . . . . . . . 28
3.14 Binary image of a hand segment and its contour image . . . . . . . . . . . 28
3.15 Contour image and corresponding approximated contour with fewer vertices 29
3.16 Gesture Recognition approach with peak detection model . . . . . . . . . 31
3.17 Gesture Recognition approach with angle calculation model . . . . . . . . 32
4.1 Linux OS file system distribution on a SD card . . . . . . . . . . . . . . . 35
4.2 Webcam capture time optimization graph . . . . . . . . . . . . . . . . . . 39
4.3 SISD and SIMD unit model . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
4.4 Two views of NEON’s thirty two 64-bit registers . . . . . . . . . . . . . . 41
4.5 NEON programming - Comparison of results . . . . . . . . . . . . . . . . 45
5.1 Flowchart for a real-time gesture control system . . . . . . . . . . . . . . . 47
5.2 Simulating keyboard events with gestures: Process flow . . . . . . . . . . 48
5.3 Flowchart for decision stage function that maps gestures to keyboard events 49
5.4 Energy consumption between algorithms with and without NEON . . . . . 53
## List of Tables

3.1 UI1220LE camera parameters for image capture .......................... 18
3.2 Mouse click simulation using retro-reflective ring ....................... 21
3.3 RGB per channel Subtraction addressing previous limitations ........ 26
3.4 Comparison of execution time for various background subtraction techniques 26
3.5 Comparison of peak-detection and angle-based gesture recognition approaches ................................................................. 33
4.1 Gesture algorithm execution time without optimizations .............. 38
4.2 Webcam capture time optimization results ................................. 39
4.3 Execution time after algorithmic optimizations .......................... 40
4.4 Programming NEON with intrinsics - Results ............................ 43
4.5 Programming NEON with assembly - Results ............................ 44
4.6 Execution time after NEON optimization ................................. 44
4.7 Gesture Algorithm execution time after optimizations ................ 46
5.1 Gestures to key symbol mapping table .................................... 50
5.2 Gesture Algorithm execution time after optimizations ................ 51
5.3 Comparison of execution times for various versions of application control algorithm ................................................................. 51
5.4 Comparison of application size when using ARM 32-bit instruction set ... 52
5.5 Comparison of application size when using Thumb 16-bit instruction set ... 52
5.6 Comparison of run time memory requirements for various versions of the algorithm ............................................................................. 52
5.7 Improvement in energy consumption when NEON SIMD unit is utilized for optimization ................................................................. 53
D.1 Using Arm and Thumb instructions - Program size ...................... 73
Acknowledgements

I express my utmost gratitude to Prof. Georgi Kuzmanov, TU Delft and ir.Joost Mans, Philips Research for guiding me throughout this thesis. Their constant encouragement and valuable suggestions helped me a lot throughout this thesis. A special mention goes to Joost Mans for the freedom I had in every aspect of this thesis.

I would like to thank my friend Vikram, who was always there whenever I needed help and for all the discussions we had and for his timely advices. I would like to thank my colleagues Harry Broers, Pieter-Jan Kuyten, Peter Tsang and Toon Bogers for their vital help and technical assistance whenever I approached them. I thank Levant, Raj Dhayal, Mustafa and all my colleagues, for the fun and joyous moments at office. I thank my friend Ram who always had time to listen to my experiences and for being supportive for the past two years.

Finally, I would like to thank all people who have helped and inspired me during my Masters study.

Srinivasan Yadunathan
Delft, The Netherlands
August 30, 2011
Introduction

Human Computer Interaction (HCI) is the study and design of systems that deals with how humans interact with the computers, to make the system more receptive and usable. The interaction may be on (a) how a system talks with the user, the output devices or (b) how the user talks with the system, the input devices. The main goal of HCI is to make the user achieve what he needs with minimum hurdle. There is a vast development in the HCI over the last decade but much of the noticeable improvements are in computer to user interface such as audio, video, graphics, operating systems etc. The way in which the user talks to the system has seen only a little growth. Currently there are only a very few systems that understand any form of inputs other than with mouse and keyboard. With the development in technology the future will bring lots of embedded devices that will need more natural interfaces such as with speech, gestures, head tracking etc. One such example to control medical images through gestures [1] is shown in fig 1.1.

Opera, on the year 2009 brought out a prank [2] that allows a user to interact with the browser using webcam and face gestures. Similar prank was brought out by Google in 2011 [3] where the user can interact with google mail with body gestures and spatial tracking algorithm fig 1.2. Although these are nothing more than pranks, it emphasizes the idea to have more natural interfaces such as gesture recognition in the system.

Figure 1.1: Example: Using gestures to control medical images

Opera browser and Google mail gestures
A gesture recognition system interprets human gestures into meaningful actions using algorithms in order to analyze the situation, to make decisions or solve some tasks. Gestures can originate from any portion of the body such as head, face, iris, hand and legs. There are several advantages in a gesture control system over a conventional one. Gestures are more natural form of interfaces, they are easy to use and can operate from a distance from the system without the need to have space for the communication device. Such a system can be used,

- as an input device for personal computer and gaming consoles
- as a control unit in consumer electronics
- to detect events such as in surveillance
- for scene control based on facial expressions
- interaction in the virtual environment

The first attempts to gesture recognition were mostly controller based and utilized an external controller such as a glove to acquire the gestures. As an example, the cybergloves by Virtual technologies [4] shown in fig 1.3, has several sensors attached to the glove that is capable of converting hand and finger motions into the three dimensional model using its software. The captured data is sent to the system with wires or even wirelessly. But due to its weight, complexity, cost of product and need for external medium to transfer data these methods are not preferred.

Figure 1.3: Example of a gesture control device-Cyberglove from Virtual Technologies

Now-a-days much of the interest is in capturing the hand gestures in its natural form without any external controller. This is possible with an external camera that detects hand gestures from the captured image frame. Several variants of cameras can be used for gesture recognition and a few are mentioned below,

**Time of Flight Camera** that uses the time of flight technique (camera emits light and measures its ‘time of flight’ after it reflects off the objects) to calculate the depth variant for each pixel in the image frame. Using the image frame and the depth information the three dimensional data is computed.

**Structured Light camera** that projects a known pattern of light onto the object and analyzes the deformation of the pattern to reconstruct the object in 3D. Xbox kinect is one such example.
**Stereo camera** that has 2 image lenses to capture the same image frame simultaneously. These cameras create the depth from the image frames from different lenses.

Methods with the above cameras can segment a hand easily from the background, using the depth map computed from the camera image.

**2D camera** produces a two-dimensional image frame without depth information. Although retrieving the foreground segment from the background is a challenging task with a 2D camera as it lacks depth information, it is preferred over the above variants for its lesser cost and wider usage. The choice of the camera depends on the application, accuracy and cost requirement.

Once the input frame is acquired there are several pre-process steps before it is possible to perform gesture recognition. First the foreground segment with the hand gesture is separated from the background. This might use background subtraction techniques in a 2D camera model or depth based selection in other 3D variants. Once the foreground segment is obtained it is then filtered to remove the noise. Several techniques (as listed below) are used to recognize the gestures from the foreground segment,

**Simple Template Matching** makes use of a template images for each gesture. The gesture is recognized from the foreground segment by comparing it with the templates and the match is obtained. This is the simplest of techniques but lacks robustness.

**Machine Learning** makes use of the image database which consists of hundreds of images with positive and negative samples of the same gesture. Based on the learning from these samples a feature set is developed. After training the system, when an image with gesture is fed to it, the the system can identify the gesture using the features. Haar-like features and the AdaBoost machine learning method that are used for face detection are good examples of this technique. This learning based method is computational intensive and consumes lot of memory.

**Contour Based Processing** makes use of the segment’s contour to recognize the gesture. The contour to be analyzed is obtained from the foreground segment and it is then processed by the implemented algorithm to decide if any gesture is identified and
CHAPTER 1. INTRODUCTION

if so what gesture it is. This method requires intelligent implementation to cover all the
gestures to be identified and comes as a compromise between the less robust template
matching and high memory Haar-feature matching techniques.

In the near future the gesture based control will make a key impact in the way the
user communicates with the system. The recent number of patents for the gesture based
ideas portrays the significance of this. Several notable patents among others are Multi-
Touch Gesture Dictionary by Apple Inc. [5] and hand gesture recognition input system

1.1 Problem Definition

The primary goal of this thesis is to get a fast and robust gesture recognition algorithm
using a 2D camera to run on Freescale i.MX515, the target embedded platform at a
frame rate suitable for real-time applications. Once the algorithm is implemented on the
final platform, the speed, memory usage and frame rate of the application are analyzed
to identify if it is suitable for embedded applications. Based on the detailed study on
related work, several problems are identified which are given below,

• Existing methods for gesture recognitions are usually carried out with depth aware
cameras such as stereo-cameras and time-of-flight cameras. These approaches
require special cameras which require much calibration and are not cost-effective

• Computer vision approaches that use regular cameras generally utilize external
objects such as multi colored gloves [10] and markers [12] for easier recognition

• Existing approaches for background subtraction to obtain hand segment from the
captured image were either carried out using plain background or skin colored based
approaches [21], that are less accurate as the skin color cannot be generalized

• Existing approaches to recognize gestures are mostly machine-learning based[25],
where the gestures are identified with a set of features obtained by analyzing
hundred’s of images. This approach is unsuitable for an embedded platform with
limited resource

1.2 Thesis Contribution

The previous section describes the various problems that exist in gesture recognition
field when achieving it with a regular camera. The goal of this thesis is to address those
issues by improving the design of the existing algorithm to provide the key contributions
as follows:

Algorithm for regular 2D camera

In order to achieve gesture recognition at lower cost, a regular 2D camera has to be
utilized in place of costly 3D cameras. As a first step, a system is designed with a
1.2. THESIS CONTRIBUTION

retro-reflective ring and an infra-red camera that can communicate to a system by simulating mouse movement and clicks. With this setup it was possible to simulate the mouse events and to control an application such as a power-point presentation. Existing algorithms are mostly suited for depth aware cameras, with which it is easier to select the hand region from background. But in this thesis the design was modified to use a regular camera, and in a way that the algorithm captures images and compares those with background reference image to obtain hand segment.

Background subtraction

A 2D camera lacks depth information and one of the main thesis contributions is developing a robust background subtraction technique for static backgrounds. The frame differencing method leads to ghost image formation while the skin colored method needs more time for processing and fails when the skin colored hand moves over skin colored objects. To address these issues a robust background approach was designed. The RGB per-channel subtraction is designed as a part of this thesis, splits the current image into three separate channels and difference them against their respective reference channels. The final foreground is obtained from the result of the three outputs. This approach gives better accuracy than frame differencing method and also operates in 11.66ms which is around 60% reduction in time compared to in skin colored subtraction.

Gesture recognition approach

In this thesis the gestures are recognized by processing the hand contour. This would result in less memory requirement and also this approach can be extended to recognize more gestures by modifying the algorithm unlike earlier method in which the system has to be trained again. In this thesis the hand contour is algorithmically processed to recognize the gestures. This approach is much suitable for low-memory embedded applications. Two different approaches are designed to recognize gestures: the first approach peak-detection classifies the gesture based on the number of peaks in the contour. This approach recognizes gesture in 14.61 ms, and although the approach is simpler it lacks robustness when the hand is tilted at some degrees and with distance. Hence to address this issue a second approach is designed. The second approach classifies gestures based on the angle of curvature with its neighboring points. This method is more robust than the previous approach and recognizes gestures in 15.78ms. The recognition system requires more robust algorithm to recognize gestures and hence the second approach is chosen over the first method.

Optimizations in embedded platform

The recognition algorithm that is built for the i.MX515 embedded platform resulted in a frame rate of less than 2 frames per second. The contribution in this aspect is to make the algorithm suitable for real-time applications, by performing optimizations such as camera capture time reduction, algorithmic optimizations and implementation optimization by utilizing SIMD unit of the Cortex-A8 processor known as NEON for data parallelism.
This resulted in bringing down the execution time of the algorithm by more than 50%.

1.3 Thesis Organization

The remainder of this thesis document is organized as follows:

**Chapter 2** starts with an overall idea on various hand gesture recognition techniques and the hardware used. A detailed study is done on various recognition methods with regard to parameters such as its accuracy, speed and robustness, cost of hardware and their limitations. Based on the detailed study and the problem definition the thesis direction is chosen such that it meets the research goals. Finally the architectural details and features of embedded platform i.MX515 along with its SIMD co-processor NEON are discussed.

**Chapter 3** covers the implementation details of the algorithm on a PC. Initially a brief overview on the design and implementation of an algorithm with an Infra-Red camera and retro-reflective material is discussed. Later we discuss on the improvement in algorithm design such that it uses only the bare hand gestures without any external objects attached. The complete design of the final algorithm and its implementation on a PC platform is discussed later along with the tools used for implementation.

**Chapter 4** covers the implementation details of the algorithm on the i.MX515 platform. Initially we discuss the details on setting up the Linux OS in a SD card and installation of various libraries to get the platform running. Later the algorithm is ported to the platform and its performance is discussed. The various speed up optimizations performed in order to meet the real-time constraints are discussed. A special emphasis is placed on the SIMD co-processor NEON, to show that it can be efficiently utilized for optimizing the algorithm. Finally the results obtained are discussed.

**Chapter 5** covers the details on verification of the gesture recognition algorithm to control a real world application. For algorithmic verification and experimental purpose two applications are developed, (1) a real-time application to control the PDF reader and presentation slideshow by emulating mouse events with gestures and (2) a real-time application to control a mp3 player by emulation keyboard events with gestures. An evaluation on the overall results obtained is discussed.

**Chapter 6** covers conclusions of the overall work, ideas for further improvements in the thesis along with the recommendations for future work.
Background Study

This chapter provides a detailed study on various approaches that are used for hand gesture recognition. Significant parameters such as approach of the algorithm, its speed and accuracy are taken into account. Finally in Section 2.5 based on the background study the direction for the thesis is set and in Section 2.6 the architecture of the hardware platform to be used is detailed.

In general most approaches to gesture recognition can be divided into two methods: one that use external device such as a glove and others that use a vision based method. The glove based technique is discussed in the later sections. The flowchart for a vision based gesture recognition system is given in fig 2.1. There are three major stages in recognizing gestures such as image acquisition, segmentation and gesture recognition. The image acquisition serves as the first step and can be achieved with many variants of camera as 2D camera, stereo camera, time of flight camera, IR based camera.

![Figure 2.1: Process flow for a gesture recognition system](image)

Once the image is acquired the foreground part is segmented from the background. The segmentation largely focuses on background subtraction techniques in a 2D environment to obtain the foreground segment. The background subtraction techniques can be simple and less computation intensive as frame differencing or can be more complex to obtain robust results. In a 3D space the segmentation can be easily realized with the help of the depth map computed from the camera image.

The final step is the recognition of gestures from the hand segment. The gesture in the foreground segment is recognized in many ways as in template matching technique, machine learning technique, convex hull based gesture recognition, contour based processing etc. Each of these techniques has their own advantage and disadvantages and we will discuss these approaches in the later section. Once the gesture is recognized it can be used to control an application or to perform a task or an action as required.

### 2.1 Gesture recognition with external devices

Before recognition methods based on computer-vision were popular, earlier attempts mostly utilized an external controller that is closely linked with the hand for gesture
recognition. The data glove and cyber glove [7] were the most prominent controllers in gesture recognition. These gloves and its variants used several sensors such as a bend and abduction sensors, accelerometer and gyroscopic sensors to detect hand motion. Approach discussed in [8] and [9] use several flex sensors and accelerometer sensors that are attached to the glove. The sensor values are in turn interpreted into hand gestures. Improved approach use multi colored gloves as shown in fig 2.2[10], that do not use sensors is discussed in [11]. Features such as the area of the colored regions, the distance and positioning of fingers from the wrist are collected. The features are then analyzed with pre-defined decision trees and the gestures are recognized. Other methods such as using markers at the finger-tip [12] are also tried to recognize gestures.

![Image of data glove and multi-colored glove]

**Figure 2.2: Gesture recognition using data glove and multi-colored gloves**

The above methods provide robust results yet they have various drawbacks which make them unpopular: mainly because these methods are costly since they use external devices. The use of sensors, wires and wireless modules to transmit signal make them uncomfortable and cumbersome. Also these methods need to be calibrated for each user before it can be successfully used.

### 2.2 Gesture recognition with Vision based techniques

With the drawbacks of the glove based methods for gesture recognition, approaches that utilized computer vision techniques are recently used. The main advantages of these methods are they do not use any external devices. Variants of cameras are used to capture the image of the hand and later user them for recognition. To obtain the hand segment easily from the background, methods that used depth information (3D) were preferred. Depth information for an image can be obtained using techniques such as time of flight, stereo camera, structured light coding etc. and are discussed below.

Time of Flight cameras were initially used for obtaining images with depth information [13], [14]. The whole area is illuminated by light source, say Infra-Red. The wave reflected by the objects lying around reaches the capturing sensor [15] which is a special type of sensor, that calculates distance value for each pixel from the received wave. The time of flight technique can be implemented by several ways to calculate the distance,
2.2. GESTURE RECOGNITION WITH VISION BASED TECHNIQUES

- RF-modulated light source: that modulates the illuminating beam with a RF carrier and analyzes the phase shift of the received wave. The distance the wave has travelled is proportional to the phase shift between reflected wave and reference wave.

- Range gated imagers: that have receivers that open and close at the same rate at which the illuminating beam is sent. In this case the distance information for each pixel is proportional to the amount of light the sensor acquires.

The methods discussed in [16], [17] use stereo cameras - two cameras that are separated by a known distance. The camera is calibrated, rectified and then used for capture. In the calibration step the radial and tangential lens distortions are removed to obtain an undistorted image. In the rectification step the cameras are positioned to obtain row aligned images. Two separate images of the same scene are captured and their features are matched to get a disparity map. The approach is shown in fig 2.3

![Stereo Camera and Structured Light Method](image)

Figure 2.3: 3D capture - Stereo camera and structured light approach

A disparity is the distance between x-coordinates of the corresponding points when the two images are superimposed. Since the arrangement of the camera is known, by triangulation a distance map can be obtained from the disparity map. The structured light coding technique uses a projector and a camera to obtain the depth map [18], [19]. The projector projects a known pattern of light onto the object and the camera analyzes the deformation of the pattern on the object. The amount of deformation in the pattern determines the distance of the object.

These cameras that provide depth information are costly and also need to be calibrated properly to obtain a good result. To obtain a depth map they need to process more pixels and are computational intensive. Yet these cameras are mostly preferred for gesture recognition as it is easy to obtain a hand segment for further processing based on the depth information.
2.3 Hand Segmentation

The main challenge for a gesture recognition system lies in segmenting a hand from the rest of the image for further processing. With a camera that is capable of producing a depth map, the segmentation turns down to selecting a group of points. The hand segment is obtained by selecting pixels that lie within certain threshold (which implies between certain distances). But the same method cannot be applied to 2D images as they lack depth details. So in order to obtain the hand segment the foreground has to be manually identified using the background knowledge. The background subtraction techniques that are generally used in gesture recognition in more static backgrounds are discussed below.

The approach discussed in [20] is the simplest of all methods that performs background subtraction based on frame differencing. The input frame in gray scale without hand is taken to be a reference. The successive frames are then compared against the static background reference frame and the foreground segment is obtained. The background is constantly updated to account for the variation in lighting conditions. This method though seems to have least competitions as it works with gray-scale images, it has more scattered noise and needs filtering. As it operates only on intensities ghost image formation as seen in fig 2.4 is its major drawback. Since the interest is in obtaining

![Reference Frame](image1.png) ![Input Frame](image2.png) ![Frame Differenced Image](image3.png)

Figure 2.4: Background Subtraction technique - frame differencing

hand segment, an improved version based on skin color is discussed in [21] and [22]. In this method the reference image is captured in RGB and converted to HSV mode [hue, saturation, intensity] as it is easy to process color in HSV mode. The hue range for skin color is known and the background image is segmented to have pixels in the hue range as the reference background mask. Similar procedure is performed for the input frame and is compared against the reference as in the frame differencing method. This method accounts for change only in skin colored elements which is an improvement over previous method. Yet the skin color cannot be generalized and it varies with illumination. Also this method becomes less effective when the hand moves over a skin colored object in the background. There are other approaches in finding a foreground segment by connected component method. It tends to be highly intensive in computation and not suitable for real-time requirements. Hence as a possible solution to this background subtraction problem a camera that provides depth information is used for easiness in segmentation.
2.4 Gesture Recognition methods

Once the foreground hand segment is obtained the next step is to recognize the gestures. Various approaches exist to analyze the segment and find out the gesture that is performed. Approaches discussed in [23] and [24] use template matching technique. It uses multiple templates for a particular gesture and stores a library of templates for all possible gestures. The hand segment is then matched against the templates to recognize the gestures. Improvement over this matching approach is based on machine learning technique [25] that uses AdaBoost (Adaptive Boosting) algorithm [26].

![Figure 2.5: Gesture recognition by machine learning approach](image)

The system is trained with a bunch of positive and negative samples. A positive sample is an image containing gestures that are to be recognized while the negative samples are images that do not contain gestures. During training a set of distinctive features are gathered as the gestures to be recognized often have some general properties. The machine learning process and gesture recognition flowchart is given in fig 2.5.

For example in face recognition the region of the eyes is darker than the region of the cheeks. Therefore a Haar feature is made with two adjacent rectangles that lie above the eye and the cheek region. Similar idea can be used for gesture recognition by having many features that can effectively classify an image containing gestures. Example of Haar like features is in fig 2.6.

Each feature collected by the system is a weak classifier, meaning it cannot solely identify the gesture but it pushes the image a step closer to the positive result. A cascade of weak classifiers results in a strong classifier. Once the system is trained it can then be used for recognition. The input image is passed to the system and a part of the image is scanned to match a particular feature. If the feature is present the classifier outputs 1 else it outputs 0. To search the entire image the similar search process is repeated moving the scan window over the entire image. If the number of outputs with 1 is more than a fixed value then it is said to have many features matching the gesture and as an outcome the gesture is recognized.
CHAPTER 2. BACKGROUND STUDY

Figure 2.6: Haar like features used for gesture recognition

A different approach to recognize gestures uses the convex hull points of the hand contour [27]. The convex hull is the smallest convex set that includes all the points in contour as in fig 2.7. The hand contour is computed from the segment and its convex hull is obtained from the contour points. This method uses hand-contour ratio for gesture recognition.

Figure 2.7: Gesture recognition using Convex hull technique

The hand-contour ratio is the ratio of area of the contour to the area of the convex hull. This ratio is pre-calculated for several gestures and the range of values is stored in a look up table. When a hand segment is given to the system it computes its hand-contour ratio and based on its value the corresponding gesture chosen from the lookup table. The method implemented in [27] can recognize 12 different gestures.

These are various approaches in recognizing the gesture after segmentation. The template matching technique seems the easiest of the approaches but it lacks robustness. The matching fails if the angle of the input image is tilted even by a small angle. The machine learning methods are very successful for face recognition but they have not achieved the same type of results in hand recognition due to a large variation in types of hands. Also the template matching and especially machine learning methods are
2.5 Chosen direction in this thesis

Several stages in gesture recognition and the various approaches that can be used at each stage were discussed in the previous sections. The main goal of this thesis is to achieve a real time gesture control for an embedded platform at a low cost. Based on the background study in the previous sections it is seen that the existing methods that can recognize gestures usually utilize depth aware cameras as image acquisition device. These devices are preferred because of the easiness of hand segmentation which drills down to selecting a group of points. On the other hand these devices need to be calibrated and are costly which means that they cannot be used for low cost applications. Hence a low cost 2D camera will be utilized for the thesis for image acquisition rather than a costly depth based cameras. The objective is to recognize gestures from the natural form of hand without utilizing any external objects and there have been only a few attempts that have tried to use regular cameras without external objects. Since a regular camera will be used, in order to obtain hand segment a background subtraction has to be performed. Comparing with the earlier works, the background subtraction were mostly done for plain background and in major cases done using skin-color approach, but the skin color cannot be generalized. On the other hand, the background subtraction techniques for moving background are too intensive in computation for an embedded platform. Hence a better background approach has to be designed to obtain hand segment. Recognition methods such as machine based learning which are used in most cases are to be excluded here as they require more memory which is not suitable for an embedded platform in general with limited resource. Methods that process the contour to recognize the gestures are to be considered as they will need less memory and achieve recognition with minimal processing. The design with all the computations should still meet the real time constraints. As a final step, the algorithm has to be verified and validated by building an application that will utilize the designed gesture recognition algorithm. In the following section the architecture details of the embedded platform i.MX515EVK on which the recognition algorithm is intented to run, is discussed.

2.6 Freescale i.MX515EVK Platform

The Embedded Platform - Freescale’s i.MX515 Evaluation Kit (i.MX515EVK) is the platform on which the final recognition algorithm is intended to run. This platform runs Linux OS, Ubuntu 10.04 distribution. The entire list of peripherals for the platform is given in Appendix E. Some of the important parameters of this platform are,

CPU: Freescale’s i.MX515[ARM Cortex-A8 based]
CHAPTER 2. BACKGROUND STUDY

Clock Speed: 800MHz
Memory: 4 X 128 MB DDR2

2.6.1 Freescale’s i.MX515 Processor

The i.MX515 is a Multimedia Processor which is one of Freescale’s latest additions to their growing multimedia-focused products, offering a high processing performance at a very low power consumption, which is due to the core of this processor, the ARM Cortex-A8. The architectural block diagram of the platform is shown in fig 2.8. The processor is suitable for applications such as,

- Smartbook and Netbooks
- Secure handheld devices: such as Smart Phones
- Gaming consoles

The core of the processor, Cortex-A8 and its features are discussed in the following section.

Figure 2.8: Block diagram of i.MX515 Processor
2.6. FREESCALE I.MX515EVK PLATFORM

2.6.2 ARM Cortex-A8 Processor

The ARM architecture is the most widely used 32-bit ISA. The ARM Core is built on a RISC architecture, which is aimed at delivering simple and yet powerful instructions that complete execution within a single clock cycle. The ARM processor has been specifically designed to consume less power which makes them suitable for low power applications. In 2005, more than 95% of all mobile handsets shipped contained at least one ARM Processors. Many of the top semiconductors manufacturing companies around the world produce products based on ARM processors which are used in various applications, such as Mobile Phones, Net Books, Smart books, Digital TVs etc.[28]. The ARM Cortex-A8 core is a 32-bit, dual-issue, in-order type processor, with dynamic branch predictor [29].

Some of the key features of this processor are,

- Clock Speed of 800MHz
- 32 KB Instruction and 32 KB Data Caches (L1)
- A unified 256 KB Data and Instruction Cache (L2)
- A Vector Floating Point Unit
- A SIMD unit called NEON
- Instruction Set Architecture (ISA) with ARM 32-bit and thumb 16-bit instruction support

The primary functional blocks of the platform which are widely utilized in this thesis are discussed below.

**ARM** - For Cortex-A8 processor, ARM instruction set provides the definitive and complete 32-bit instructions, using this instruction set gives best results in terms of performance.

**Thumb** - The Thumb instruction set is an extension to the 32-bit ARM architecture and it is used to obtain a high code density. It is a subset of the most commonly used 32-bit ARM instructions which have been compressed into 16-bit wide operation codes. These 16-bit instructions when decoded eventually enable the same functions as their 32-bit ARM instruction equivalents. This instruction set was primarily introduced to cater such situations where program memory is a constraint. Compared to full ARM up to 30 percent code size reduction is achieved, however what is achieved in low system memory use, is lost in performance.

**Thumb2** - The Thumb2 is a superset of Thumb instruction set. Several, new 16-bit instructions were introduced into Thumb2, but the specialty of Thumb2 lies in the new 32-bit instructions introduced, which were once again derived from ARM instructions. Although the performance is not completely at the level of pure ARM instructions, it
is definitely better than Thumb instructions. A comparison of code density and performance achieved by using all three instruction sets is presented in fig 2.9.

![Figure 2.9: Arm and Thumb ISA](image)

**VFP Unit** - These instructions are used to program the Vector Floating Point Unit (vfpv3). In simple words vfpv3 is a floating point hardware accelerator. The purpose of VFP is to speed up half, single and double precision floating point operations. The name Vector here is a misnomer, the VFP actually has no parallel architecture. It performs one operation on one set of inputs and returns one output.

**NEON Unit** - These instructions are used to program NEON - the SIMD unit of the Cortex-A8 processor. This is a true vector processor. Programs targeted to NEON can be written directly in Assembly or by making use of NEON Intrinsics, which are a C wrapper around assembly instructions. Some of the powerful NEON instructions are used extensively to program critical parts of the SURF algorithm implementation of this project.

These are the key blocks of that the i.MX515 platform that are most widely used in this thesis. The procedure to utilize these blocks to build the algorithm on this platform and the optimizations preformed using the NEON unit are covered in Chapter 4.

### 2.7 Conclusion

This chapter presented an outline to various stages in gesture recognition. Existing approaches that are used in each of the stages were discussed. The merits and demerits of the approaches are studied to know if they can be used in this thesis to achieve its goal. The approaches to be used in each of the stages are discussed and the thesis direction is set. Finally the architecture of the embedded platform is detailed in the last section. In the following chapter we will discuss the design approaches and implementation of the algorithm on a PC.
In the previous chapter based on the background study a roadmap to the thesis goal is set. And as a first step a gesture recognition algorithm was implemented in PC and later moved to the final platform. This chapter covers the design and implementation details of the algorithm. Section 3.1 covers the design of a user-interaction system with a retro reflective material and Infra-red camera. Section 3.2 covers the design and implementation details of a gesture recognition system with a 2D camera.

### 3.1 User Interaction with a retro reflective ring and an infra-red camera

As discussed in the previous chapters there are several ways to communicate to a system without a conventional mouse and keyboard. Before proceeding to the hand gesture recognition system, in order to get a feel of a user-interaction system and to get familiar with the development environment a preliminary step is performed. A system to control an application from a distance using a retro-reflector and an infra-red camera is designed. The architecture of the system is presented in fig 3.1. The blocks marked in gray are stages where design and algorithmic contributions were made and their implementation is discussed in later sections.

![Figure 3.1: Flowchart for Retro-reflective ring based user-interaction system](image)

Retro-reflection is the process of reflecting light back to the source with a minimum scattering of light. The retro-reflection idea is the key behind this system. The camera used is UI-1220LE(Monochrome) from IDS Imaging. For retro-reflection, the camera is manually altered to be sensitive to IR light and to have infrared LEDs to emit IR rays as...
seen in fig 3.2. The reflective surface is attached to the ring for convenience. The main objective of this step is to use the retro-reflective ring to communicate to the system to control some application. The design details are discussed below.

![UI camera - IR LED off and on](image)

Figure 3.2: UI camera used for the setup with its IR - LED off and on

### 3.1.1 Initialization and Set parameters stage

During the initialization stage the camera driver is started and it establishes connection to the camera. After successful initialization a camera handle is assigned which will be used by all subsequent functions.

In the set capture parameter stage the parameters for the capture such as pixel clock, exposure time, flash duration are to be set manually. The pixel clock determines the frequency at which the image data is read from the sensor. Trigger mode determines the type of trigger that will cause the sensor to capture the image. There are two main types of triggers. A hardware trigger: where the trigger signal is from electrical signal, and a software trigger: where the trigger event is determined by the software. The exposure time determines the duration for which the light enters the sensor and is usually kept minimum. The infrared flash (from the IR LED) can be set on / off and the flash delay determines the delay for driving the flash output and when the flash delay is set to 0 the duty cycle synchronizes with the exposure time and the flash remains on during the exposure time. Finally memory is allocated for storing the images from the sensor.

<table>
<thead>
<tr>
<th>No</th>
<th>Parameter</th>
<th>Value Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Pixel Clock Frequency</td>
<td>28 MHz</td>
</tr>
<tr>
<td>2</td>
<td>Trigger Mode</td>
<td>Software trigger</td>
</tr>
<tr>
<td>3</td>
<td>The exposure time</td>
<td>0.3 milliseconds</td>
</tr>
<tr>
<td>4</td>
<td>The infrared flash</td>
<td>On</td>
</tr>
<tr>
<td>5</td>
<td>Flash delay</td>
<td>0, same as exposure time</td>
</tr>
</tbody>
</table>

Table 3.1: UI1220LE camera parameters for image capture
3.1. USER INTERACTION WITH A RETRO REFLECTIVE RING AND AN INFRA-RED CAMERA

3.1.2 Process and Decision stage

**Design Contribution:** The camera parameters are set and it is now ready for acquisition. In the process and control stage the retro-reflective ring is tracked to communicate with the system. The ring movement is used to obtain a mouse movement and hence when the ring moves it simulates the mouse movement. The mouse clicks are simulated by closing the ring for certain period and then releasing it. The design of this stage is discussed below.

**Implementation:** The camera is initialized and flash is kept on. The infrared beam from the flash LED is reflected back to the sensor by the ring with minimum scattering. A fixed-level thresholding marks the image pixels high (1) if its value is greater than a threshold value (230 in this case) else it is marked low (0). This process converts a gray scale image into a binary image. Smoothing is performed on the binary image to reduce the noise level. After the thresholding and smoothing operation, the ring appears as a bright spot in the image as in fig 3.3(b).

![Figure 3.3: Input image to the system and the ring’s bright spot in output image](image)

The surface of the ring has a known shape (circle) and using Hough transform, a feature extraction technique [30] to extract lines and circles from the image, the ring can be tracked. The Hough transform is performed on the binary image to identify the circles. The ring’s surface is round and will be identified as a circle. We can limit retrieving other circles based on the fixed radius of the ring. Once the frame is processed we know if the ring is present or not based on the circle count which is 1 if ring is present else it is 0.

In the decision stage the presence of the ring and its location is used to simulate mouse events. The flowchart for the process is given in fig 3.4. In order to communicate to the system, mouse events such as movement and clicks are simulated by tracking the ring. The mouse movements along vertical and horizontal axes are paired with the ring movement. Hence whenever the ring is identified, the coordinates of the center of the ring are allocated to the mouse coordinates resulting in mouse movement. The camera’s resolution is 640 x 480 whereas the output screen’s resolution is 1280 x 1024. The camera captures mirror image of the real-world scene. So when the ring is moved to the right in real world, it is seen to move towards left in the camera frame. Hence to match the coordinates correctly the image x-position is subtracted from the screen x-coordinate (1280). Image coordinates are paired with the mouse coordinates based on the formula,
mouse\_x\_pos = (1280 - (\text{circle\_mid\_x} \times 1280 / 640))
mouse\_y\_pos = (\text{circle\_mid\_y} \times 1024 / 480)

Figure 3.4: Mouse event simulation with retro-reflective ring - Flowchart

To simulate mouse clicks, the ring is blocked for a certain period of time and then released. As explained in fig 3.4, when the ring is obstructed the timer 1 is incremented until it reaches the fixed counter value. If the ring is not released within this period then it waits for a small duration to check if there’s an chance for a double click action and repeats this process again to perform a double click else it performs a single click. In this way both the mouse movement and clicks can be simulated using the ring.

The pseudo code for the Mouse simulation algorithm is given below,

<table>
<thead>
<tr>
<th>Pseudo code: Mouse simulation with RR ring and IR camera</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. <strong>Input</strong> binary image with ring</td>
</tr>
<tr>
<td>2. Find the presence of ring with Hough transform</td>
</tr>
<tr>
<td>3. If circle found =1</td>
</tr>
</tbody>
</table>
3.1. USER INTERACTION WITH A RETRO REFLECTIVE RING AND AN INFRARED CAMERA

a. Update mouse x and y coordinate with circle mid-point
b. Get next image and goto step 2

4. End if

5. If circle found = 0
   a. Update timer 1 until timer1 = count
   b. If (timer 1 = count value) and ring count = 1 and goes to 0 again before end of timer 2
      i. Timer 1 = 0, Double click flag = 1
      ii. Update timer1 until timer1 = count
      iii. Execute double click action
c. End if
d. If (timer 1 = count value) and ring count = 0 until end of timer 2
   i. Execute single click action
e. End if

6. End if

7. Get new image and goto step 2

8. Result: Simulate mouse movement and mouse clicks with ring action

It is possible to communicate with the system now and the next step is to see if we can control an application. The algorithm is designed to operate on two modes: a normal and a presentation mode. In a normal mode the double click action results in an actual double click which is typically used for normal desktop operations, whereas the same action in a presentation mode results in a right click to control slides in a presentation. For the test purpose a Microsoft PowerPoint slideshow is controlled using the ring from a distance and its actions are summarized in the table table 3.2.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Ring Closed</th>
<th>Mouse Action</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>Once</td>
<td>Single click</td>
<td>Normal desktop application</td>
</tr>
<tr>
<td></td>
<td>Twice in succession</td>
<td>Double click</td>
<td>eg: single click to rename, double click to open folder</td>
</tr>
<tr>
<td>Presentation</td>
<td>Once</td>
<td>Single click</td>
<td>Slideshow control</td>
</tr>
<tr>
<td></td>
<td>Twice in succession</td>
<td>Right click</td>
<td>eg: single click for next slide, right click for previous slide</td>
</tr>
</tbody>
</table>

Table 3.2: Mouse click simulation using retro-reflective ring

Finally when the exit key is pressed the finish stage is reached where the driver is released and the allocated memory is freed for other applications.

To conclude it is possible to communicate to the system and to control an application (in this case a slide show) using the retro-reflective ring and the camera. Building this system helped me to understand user interaction with a computer vision based system and the need for such a system. The short coming of this system is that it used external device and special camera. Hence as a next step, the design is improvised in a way that it does employ any external device and a special camera, but can communicate to the system by recognizing hand gestures.
3.2 Gesture recognition with a 2D camera

In the previous section we achieved user interaction by communication with the system using a retro-reflective ring and an infra-red camera. However the main goal of this thesis is to communicate to the system without any external device or any special camera. As a result the above approach is to be modified in a way that it uses a cheap 2D camera without any external device. Therefore using bare hands to perform several gestures and to design the system to understand the gestures for user- interaction seems a possible solution.

The following section covers the design and implementation of a gesture recognition system. Possible gestures that can be recognized by the system are given in fig 3.5.

![Figure 3.5: Hand gestures recognized by the system](image)

The system recognizes hand gestures with computer vision based approach. It has a camera to acquire images and processes them to obtain hand contour. The contour is algorithmically processed to obtain the hand gestures and the approaches are discussed in the later sections. The recognition system consists of several stages as in fig 3.6. *Blocks marked in gray* show stages where my contributions are made either to improve a previous design or to bring out a new design. The overall system design and the implementation of each of the stages are discussed below.

![Figure 3.6: Hand Gesture recognition system - List of stages](image)
3.2. GESTURE RECOGNITION WITH A 2D CAMERA

The implementation begins with an image capture stage that uses a 2D camera to capture the hand gestures performed by the user. As a next step the system captures a background image, an image without hand and gestures which will be used as a reference image during the background subtraction stage.

3.2.1 Background subtraction Stage

In this stage a background subtraction is performed to obtain foreground hand segment. There was an important contribution in this stage to design and implement the RGB per channel based background subtraction. I have discussed background subtraction in general and the earlier methods that were employed for this thesis followed by the discussion about RGB per channel based subtraction approach.

As discussed in the section 2.2, the system employs a 2D camera for gesture recognition which lacks depth information and as a consequence a background subtraction has to be performed. General background subtraction approach is given in fig 3.7.

![Figure 3.7: General background subtraction methodology](image)

The gesture recognition system is designed to be used in a static environment where the movement in the camera field is occasional. The background subtraction techniques from section 2.3 are implemented and compared in order to see if they are able to provide a good foreground segment. The frame differencing on grayscale images was first implemented. It is simple and less computationally intensive which took around 4ms for computing the subtracted image. But it had a major drawback that it cannot distinguish between colors as it operates on grayscale images. As a result if the intensity of the image pixel matches the reference pixel and even if they are of different colors, the foreground pixel is marked zero. Also this technique accounts only intensity which results in ghost image formation as in fig 3.8(a).

The next approach skin colored based subtraction operates on colored images. Working with distinct colors is easier in HSV color mode (Hue, Saturation, and Intensity) than in RGB mode. With the current camera there is no option to capture the images in HSV mode and as a result the images are captured and then converted to HSV images. The hue range (5 to 20) represents skin color and a background mask (binary image) is created based on: if the pixel hue range falls within 5 to 20 it is marked 1 else it is marked 0. In a similar way the captured image is converted and a mask is obtained. The two masks are differenced to get the foreground segment. This approach accounts for the variation in skin colored elements only and hence its accuracy rate is better than the previous approach. But it is more computationally intensive and it takes more than 25ms to process the image for foreground. These two methods are discussed in fig 3.8.
3.2.1.1 Contribution in this stage

The contribution in this stage is to design a robust background subtraction algorithm with accuracy rate better than frame differencing and with lesser processing time than skin color subtraction to account for real-time. The details are discussed below.

From the above discussion it is evident that there are limitations with the previous approaches,

1. Frame differencing approach suffers when the intensities of different colored pixels are same as it operates only on the gray scale image fig 3.8(a)

2. Skin based subtraction suffers when hand moves over skin colored objects at the background fig 3.8(b)

As these limitations affect the contour that will be obtained from the foreground segment, it is important that a new background subtraction design solves this problem. The design has to be robust compared to the previous techniques and at the same time should be within real-time needs. Hence as a contribution we propose RGB per channel subtraction approach to address the problems of the previous techniques.
3.2. GESTURE RECOGNITION WITH A 2D CAMERA

In the **RGB per channel subtraction** design as in fig 3.9, the background image is captured and split up into individual channels as $R_{\text{ref}}$, $G_{\text{ref}}$ and $B_{\text{ref}}$ channel for further reference. The current image is captured each time and similarly split up into individual channels as $R_{\text{curr}}$, $G_{\text{curr}}$ and $B_{\text{curr}}$.

![Flowchart](image.png)

**Figure 3.9: RGB per channel background subtraction - Flowchart**

Each channel which is an 8 bit image is differenced with the respective reference channel using the frame differencing technique. For individual channels if the pixel value after differencing is greater than a threshold value then it is marked as foreground for that channel. Similar operations are performed on other two channels. After this process there will be three images with foreground information from each channel. Finally the foreground segment is obtained from all the three images. If the pixel is marked foreground in any of the channel then it is marked foreground in the final image.

The pseudo code for the Mouse simulation algorithm is given below,

**Pseudo code:** RGB Per Channel Subtraction

1. **Input:** Capture reference image
2. **Split reference** image into individual channels: $R_{\text{ref}}$, $G_{\text{ref}}$ and $B_{\text{ref}}$
3. Capture current image
4. **Split current** image into individual channels: $R_{\text{curr}}$, $G_{\text{curr}}$ and $B_{\text{curr}}$
5. Perform Frame differencing with $R_{\text{curr}}$ and $R_{\text{ref}}$
6. Apply **threshold function**, (threshold value 25)
   a. Pixel, with value $\geq 25$ are marked 1
   b. Pixel, with value $< 25$ are marked 0
7. Get binary image $R_{\text{foreground}}$ after thresholding
8. Repeat steps 5 till 7 to obtain $G_{\text{foreground}}$ and $B_{\text{foreground}}$
9. Combine $R_{\text{foreground}}$, $G_{\text{foreground}}$ and $B_{\text{foreground}}$ to get **foreground**
   a. If pixel in any of the 3 channels is marked 1 then foreground pixel = 1
   b. Else foreground pixel = 0
10. **Result:** Obtain foreground segment after background subtraction
This design addresses the problems with the existing approaches which is explained in the table 3.3,

<table>
<thead>
<tr>
<th>Approach</th>
<th>Failure Case</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame differencing</td>
<td>Grayscale image; The intensities of different colored pixels are considered same</td>
<td>Operates on colored images, hence same intensity of different colors are considered different</td>
</tr>
<tr>
<td>Frame differencing</td>
<td>Ghost image formation</td>
<td>No ghost image formation</td>
</tr>
<tr>
<td>Skin color subtraction</td>
<td>The skin colored hand moves over skin colored objects as intensities are seen equal</td>
<td>Can differentiate between intensities of skin colored hand and object</td>
</tr>
</tbody>
</table>

Table 3.3: RGB per channel Subtraction addressing previous limitations

The only way this method will suffer will be if the reference and image pixel are of same color and same intensity, but this is easily solved by reducing the exposure time so that the background appears dimmer than the foreground. The execution time and the accuracy rate for the three methods are compared in the table 3.4. The result of performing this background subtraction is given in fig 3.10.

![Figure 3.10: RGB per channel background subtraction technique](image)

Table 3.4: Comparison of execution time for various background subtraction techniques

<table>
<thead>
<tr>
<th>Approach</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame differencing</td>
<td>4.812 ms</td>
</tr>
<tr>
<td>Skin color subtraction</td>
<td>28.516 ms</td>
</tr>
<tr>
<td>RGB per channel</td>
<td><strong>11.663 ms</strong></td>
</tr>
</tbody>
</table>

As we see from the table 3.4, RGB per channel subtraction approach is more robust than skin color approach with greater accuracy rate (the accuracy rate is calculated based on the background used for previous images) and at the same time it executes at lesser time which is still within real-time requirements. Hence this approach is adapted and we move to the next stage morphological transformations.
3.2. GESTURE RECOGNITION WITH A 2D CAMERA

3.2.2 Morphological Transformation stage

The next stage is morphological transformation. The foreground segment obtained from the previous stage contains salt and pepper noise and has to be filtered. Noise is nothing but an abrupt intensity change in the pixel value and hence a 3x3 median based smoothing operation is performed to reduce the noise level. In this noise reduction technique every pixel is replaced with the median value in its 3x3 neighborhood to reduce noise level. The procedure is explained for a value 255 (which is noise) and it is replaced with value 60 as shown in fig 3.12.

Noise removal is followed by dilation and erosion. Dilation on a binary image gradually enlarges the boundaries of regions of foreground pixels. Hence the area of foreground pixels grows in size while holes within those regions become smaller. Similarly erosion on a binary image erodes away the boundaries of regions of foreground pixels and the areas of foreground pixels shrink in size while the holes within those areas become larger. The dilation and erosion effects are seen in in fig 3.13. A morphological closing operation which is a dilation followed by an erosion is performed on the noise reduced image. This operation results in a closed foreground segment by filling out the small holes that might have occurred in the noise removal stage.

![Execution time comparison for several background subtraction approaches](image)

**Figure 3.11:** Execution time comparison for several background subtraction approaches
3.2.3 Contours Detection and approximation stage

Once we have the foreground segment the next step is to analyze them for gestures. A contour is a sequential list of points that is used to extract the borders of objects in the image. In this implementation the gestures are recognized by algorithmically processing the contours. The inbuilt function of Opencv image library uses the suzuki85 algorithm \cite{31} to obtain the contours from the image. This function operates on a binary image and when an 8 bit gray scale image is passed to it all the non-zero pixels are treated as 1 and the rest pixels as 0.

![Figure 3.13: Effects of Dilation and Erosion on an image](image)

There are two types of contours: exterior contours that represent exterior boundaries of the white regions and interior contours that represent exterior boundaries of the black regions. We are interested in border of hand segment and hence only exterior contour is retrieved form the image. The function retrieves all exterior contours and stores them in an array (an array of sequence of points) and we can traverse from one contour to another in an array while processing. The binary image of the hand segment and the retrieved contour is shown in fig 3.14.

![Figure 3.14: Binary image of a hand segment and its contour image](image)

A single contour in the array that is represented by a maximum number of points will represent the hand segment. Also based on the contour area, contours representing objects other than the hand can be limited. The original contour will be represented by a huge number of points and although more number of points can accurately represent
3.2. GESTURE RECOGNITION WITH A 2D CAMERA

the contour the drawback is that it needs a lot of memory to store the points and more time to process the contour. Thus the system requires more time and memory which is against the real time needs.

As a result we can represent the hand segment contour with another contour with fewer vertexes and without any loss of accuracy. This process is called contour approximation. There are many methods to approximate the contour as Douglas-Peucker (DP) approximation, Rosenfeld-Johnson approach and Teh-Chin approach. The first method, Douglas-Peucker polygon approximation [32] is used to approximate the contour as it is readily available in Opencv as a function call. The function operates on original hand contour and returns an approximated contour with fewer vertices as in fig 3.15.

Now we have obtained the approximated contour from the original contour and we proceed to the next stages to recognize the gestures by processing the contour.

![Contour image and corresponding approximated contour with fewer vertices](image)

Figure 3.15: Contour image and corresponding approximated contour with fewer vertices

3.2.4 Gesture recognition stage

The main aspect of gesture recognition stage is to identify the gestures from the hand segment. Earlier approaches have used template matching and machine learning aspects as discussed in section 2.4.

In this section as a thesis contribution two gesture recognition approaches are designed to process the contour and recognize the gestures. The design details are discussed below and one of the approach is finalized based on accuracy and time consumed. The results are compared at the end.
3.2.4.1 Contributions

(i) Peak detection method: In the peak detection method the main idea is to design and implement an algorithm to recognize gestures from the contour, based on the number of peak points in the contour. The design and implementation details are discussed below.

**Design:** The approximated contour consists of number of points. This is the key idea of this approach is to examine classify each point as a peak point or non-peak point, based on the neighboring points. The current point and the neighboring points are projected on both x-axis and y-axis. If the current point x-coordinate is less than both the neighboring points coordinate in the x-axis then it is classified to be a peak point. Similar process is performed for y-axis.

Once this idea is implemented there were a few false positives and hence the design needed a small enhancement which is as follows. For a peak-point (based on x-axis), in addition to the previous idea, the difference in y-axis coordinates should be more than a fixed threshold. Similar procedure is performed for y-axis.

The pseudo code for the Peak detection approach is given below,

**Pseudo code:** Gesture - Peak detection approach

1. **Input:** approximated contour
2. Get coordinates for current point (px, py)
3. Get coordinates for neighboring points (ax, ay) and (bx, by)
4. Check for **peak point in x-axis**
   a. If px < min (ax, bx)
      i. If (min (ay, by) - py) > threshold then P = peak point
      ii. Else P = non-peak point
   b. End if
5. Check for **peak point in y-axis**
   a. If py < min (ay, by)
      i. If (min (ax, bx) - px) > threshold then P = peak point
      ii. Else P = non-peak point
   b. End if
6. Repeat step 2 till 5 for all the points in the approximated contour
7. Get the total number of peak points
8. **Result:** Recognize gesture based on the number of peak points in the contour

Let us consider the points 6 and 7 for our analysis. The neighboring points of 7 are 6 and 8 respectively. The x-coordinate of 7 is lesser than x-coordinate of 6 and 8 and hence it is classified to be a peak point. Whereas for the point 6 whose neighboring points are 5 and 7 respectively, the x-coordinate of 6 is between x-coordinates of 5 and 7. Therefore the point is non-peak point. This procedure is diagrammatically explained in fig 3.16.
With this design, six different gestures were successfully classified based on the number of peaks in the contour. This approach involves fewer computations which took around 9ms for execution but its accuracy rate becomes worse as the distance increases. Hence in order to design a robust algorithm we proceed to the next method.

(ii) Angle based Gesture recognition: In the angle calculation method a robust algorithm is designed to recognize gestures based on the interior angle of points in the contour. The design and implementation details are discussed below.

Design: Angle based detection recognizes gestures using interior angle between three consecutive points. Each point in the contour along with its neighbors form three consecutive points and are examined to see if their interior angle is less than a specific threshold angle value. If such is the case then it is classified to be a successful point, else as a non-successful point. Based on the number of successful points in the contour the gestures are recognized.

The pseudo code for the Angle calculation approach is given below,

**Pseudo code: Gesture - Angle calculation approach**

1. **Input:** approximated contour
2. Get coordinates for current point (px, py) and neighboring points (ax, ay) and (bx, by)
   a. Check if (ax-px) OR (bx-px) is positive, if both zero or negative - goto step 6
3. Calculate dx21, dx31, dy21, dy31, m12, m13
   a. dx21 = (ax-px) ; dx31 = (bx-px)
   b. dy21 = (ay-py) ; dy31 = (by-py)
   c. m12 = sqrt ((dx21*dx21)+(dy21*dy21))
   d. m13 = sqrt ((dx31*dx31)+(dy31*dy31))
e. \( \text{value} = \frac{(dx21 \times dx31) + (dy21 \times dy31)}{(m12 \times m13)} \)

4. Calculate interior angle for the value
   a. \( \text{angle} = \arccos(\text{value}) \)

5. Check for successful point in
   a. If angle < threshold angle
      i. then \( P = \) successful point
      ii. else \( P = \) non-successful point
   b. End if

6. Repeat step 2 till 5 for all the points in the approximated contour

7. Get the total number of successful points

8. **Result:** Recognize gesture based on the number of successful points in the contour

As an experimentation to get the best results the threshold angle is varied several times and the procedure is repeated. A threshold value of **48 degrees** for the design provided robust results. Let us analyze the algorithm with an example.

![Figure 3.17: Gesture Recognition approach with angle calculation model](image)

From the **figure 3.17**, let us consider the point 2 for our argument. The neighboring points of 2 are 1 and 3 resp. The interior angle between 1-2-3 is 29 which is less than the threshold value of 48 and hence it is classified as a successful point. While for the point 5 the interior angle between 4-5-6 is 124 which is greater than threshold value and it is non-successful point. In order to recognize gestures efficiently by reducing computations for each point, the angle is calculated for points only if the difference between its x-coordinate and that of neighboring points is positive. This enhancement reduces computations while it still provides required results.

The two approaches used for recognizing gestures are compared in the **table 3.3**. The first approach is simpler with fewer computations which takes about 14.6ms, but
it lacks robustness when the hand is tilted at some degrees and with distance. The second approach performs in 15.78ms which is slightly more than the previous method but the method is more robust. The parameters are compared and given in the table 3.5. Although the execution time and memory requirements are slightly greater for the angle based approach, it is chosen over the former approach for its accuracy level.

<table>
<thead>
<tr>
<th>#</th>
<th>Parameter</th>
<th>Peak Detection</th>
<th>Angle Calc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Execution time</td>
<td>14.61ms</td>
<td>15.78ms</td>
</tr>
<tr>
<td>2</td>
<td>Final output - memory requirement</td>
<td>6.08kb</td>
<td>6.29kb</td>
</tr>
<tr>
<td>3</td>
<td>Accuracy Distance</td>
<td>Less</td>
<td>More</td>
</tr>
</tbody>
</table>

Table 3.5: Comparison of peak-detection and angle-based gesture recognition approaches

This marks the final stage of the system and at the end of this stage the gesture that is performed by the user is recognized effectively.

3.3 Conclusion

This chapter covered the implementation details of a user-interaction system with retro-reflective ring and IR camera. Later the implementation of a gesture recognition system with a 2D camera that can identify six different gestures is discussed. The next step is to port the algorithm to an embedded platform and optimize it for real time applications and those are covered in the next chapters.
In the previous chapter the gesture recognition system was implemented on a PC with abundant resource. The next thesis goal is to port the algorithm to i.MX515 EVK platform and get it to run in real-time on the platform. This chapter mainly covers the implementation of the algorithm on the embedded platform. Section 4.1 covers details about setting up the platform and porting the algorithm to the platform. Section 4.2 covers the various optimizations performed to achieve a frame rate of 4 frames/second.

4.1 Porting Algorithm to i.MX515 Platform

In this section the details on porting the algorithm to the platform is discussed. As a first step in porting the algorithm, an operating system for the platform is built on a SD card, along with required libraries. The details to build an operating system on a SD card is given below.

4.1.1 Building Linux OS on a SD Card

The platform (i.MX515) boots OS from a SD card as discussed in Chapter 2. The Linux OS Ubuntu 10.04 is chosen to be built on the i.MX515 platform. Ubuntu 10.04 for i.MX515 does not have an image file that can install the entire OS automatically and hence the OS has to be built manually on the SD card. The boot loader, a kernel image and root file system as in fig 4.1, are parts of Linux OS to be built.

![Figure 4.1: Linux OS file system distribution on a SD card](image)

Generally, a Linux host is used to build the OS on a SD card. All build for the target system is performed using the host system running Ubuntu 11.04. The SD card along with the SD/MMC card reader is used to transfer the images and root file system. The procedure to build OS into the SD card is listed below.
1. **Identify SD card device node**: The Linux kernel running on the Linux host will assign a device node to the SD/MMC card reader. The kernel will decide the device node name. The device node name is used in all the further instructions and is found using the command,

```
cat /proc/partitions
```

which will be like /dev/sdb

2. **Copy boot loader and kernel image**: Boot loader is a small program in ROM whose job is to load and boot a kernel image as soon as the board is switched on. U-Boot and U-Image are the universal boot loader and kernel image respectively for ARM based embedded boards. They are flashed into SD card with the command,

```
sudo dd if=u-boot-bbg.bin of=/dev/sdb bs=512 sync
sudo dd if=uImage of=/dev/sdb bs=512 seek=2048 sync
```

3. **Create partition table**, file system and copy root file system: A partition table is a part of boot loader that helps to identify the active partition to find the operating system and to load it into memory. In our platform the partition table is to be flashed at address starting from sector #8192 (4MB), leaving space for the kernel, the boot loader and its configuration data. A partition table is created and a root file system is flashed using the commands below,

To create partition table,
```
sudo fdisk /dev/sdb
```

```
d [delete existing partitions ]
u [switch the unit to sectors from cylinders]
n [create a new partition]
p [create a primary partition]
I [first partition]
8192 [starting at offset 8192]
<enter> [create partition]
W [save and exit]
```

To create ext3 file system and to un-compress the root file system image use,
```
sudo mkfs.ext3
/dev/sdb1
```
```
sudo tar --numeric-owner -xvf <path>/</file_system.tar.gz>
sync
```

4. **U-Boot setup**: Finally the U-Boot is configured to boot the board from the SD card with DVI monitor support for display. The target board is connected to the Linux host through a serial console and the U-Boot is configured. The configuration is saved and the board is restarted for the changes to take effect.

At the end of this step the board boots Ubuntu 10.04 distribution from a SD card.
Now we have a Linux target system to which the algorithm is to be ported.
4.1. PORTING ALGORITHM TO I.MX515 PLATFORM

4.1.2 Building the gesture recognition algorithm for i.MX515 platform

At the end of previous chapter we had a gesture recognition algorithm that runs on a PC with Windows 7 operating system. The PC has abundant resources and processor with greater clock speed and hence there was no concern about the execution time. But now the algorithm has to be built for an embedded platform which has limited resources and after building the execution time plays a vital role. We will discuss how to build the existing gesture recognition algorithm for ARM Cortex-A8 processor. Also the algorithm is profiled to understand the execution time of various modules of the algorithm.

The i.MX515EVK runs a full-fledged operating system - Ubuntu 10.04 on a SD card. As a result the algorithm was built directly on the i.MX515EVK without any cross-compiler on a Linux host system. The development tools used for building the algorithm are,

- GNU Compiler Collection - gcc 4.3.3
- GNU Binutils - tools for manipulating binaries
  -as (GNU assembler)
  -ld (GNU Linker)
  -size (lists section and total size)
  -gprof (to profile the algorithm)
- GNU make
- Opencv 2.1.0 (library of imaging functions)

With the development tools mentioned, the gesture recognition algorithm was built for ARM Cortex-A8 processor. Compiling with the right set of compiler options plays a significant role in the outcome of the algorithm. The instruction to the GCC compiler is given below and the flags are explained,

```
gcc -mcpu=cortex-a8 -mfloat-abi=softfp -ffast-math -mfpu=vfpv3 -marm -O3 'pkg-config --cflags --libs opencv' -o gesture_algorithm gesture_algorithm.c
```

- `-mcpu`: indicates the name of the target ARM processor, **cortex-a8** in this case
- `-mfloat-abi`: indicates the floating-point ABI to be utilized, **softfp** in this case, generating output containing library calls for floating-point operations
- `-ffast-math`: use faster floating point instructions resulting in a minor loss in precision which is well within an acceptable level for this thesis result
- `-mfpu`: indicates floating point hardware to be used in the target board, **vfpv3** hardware In this case
- `-marm`: instruct compiler to generate output with ARM instruction


- `-O3`: instruct compiler to use highest level of optimizations. For list of flags that are enabled for this optimization kindly refer appendix A

- `pkg-config`: to link all the OpenCV dependencies using the file `opencv.pc`

With the above compiler flags the algorithm is built for the i.MX515EVK platform. The next step is to profile the algorithm and analyze the time spent in each segment.

### 4.1.3 Algorithm Profiling

The gesture recognition algorithm built for the platform will henceforth be referred as `GESTURE_ALGO` for clarity and easiness. The algorithm is profiled to understand the time spent in each of the modules and the results are discussed.

<table>
<thead>
<tr>
<th>#</th>
<th>Algorithm Module</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Reference Image Capture</td>
<td>163.76</td>
</tr>
<tr>
<td>2</td>
<td>Current Image Capture</td>
<td>139.89</td>
</tr>
<tr>
<td>3</td>
<td>Image Split</td>
<td>27.26</td>
</tr>
<tr>
<td>4</td>
<td>Background Subtraction</td>
<td>25.67</td>
</tr>
<tr>
<td>5</td>
<td>Morphological transformation</td>
<td>106.02</td>
</tr>
<tr>
<td>6</td>
<td>Contour processing</td>
<td>15.78</td>
</tr>
</tbody>
</table>

Table 4.1: Gesture algorithm execution time without optimizations

From the table we can see that that the total time spent by the un-optimized algorithm is around 480ms. As a result the frame rate obtained is around 2 frames/sec, which is acceptable but a low value for real time applications. Hence the most time consuming parts of GESTURE_ALGO: reference, current image capture and morphological transformation modules are to be optimized to obtain better frame rate. Optimizations such as camera capture time reduction and data-parallelism are employed to obtain better results and are discussed in the following sections.

### 4.2 Optimizations for the Algorithm

The most time consuming parts are identified from the table 4.1 and they need to be optimized to bring down the overall execution time. Several optimizations are performed by modifying the algorithm and by using the platform’s capabilities, in-order to achieve a frame rate around 4 frames/sec.

#### 4.2.1 Optimization 1: Camera - Capture time reduction

As seen from the table 4.1 the algorithm mainly depends on image capture. The camera used for this thesis is Logitech c250 in Ubuntu 10.04 environment. Measurement shows that the time required for capturing a frame is **139.89ms**. Hence by reducing the capture time the overall execution time of the algorithm can be brought down by several milliseconds.
4.2. OPTIMIZATIONS FOR THE ALGORITHM

Whenever a frame is captured, it is pushed to a temporary buffer which can store 4-8 frames and the data is copied from the buffer (with memcpy). This results in latency between frame capture time and processing time. As an optimization, the number of intermediate buffers is limited to 2 so that only the recent frames are picked for processing. Also the frame that is captured is pushed directly to the user without memory copy. This reduces the capture time for a frame to 98.92ms with an improvement of 29% as seen from fig 4.2. The results are given in table 4.2.

![Camera Capture Time Graph](image)

**Figure 4.2: Webcam capture time optimization graph**

<table>
<thead>
<tr>
<th>Before Optimization</th>
<th>Capture Time (ms)</th>
<th>Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>139.76</td>
<td>478.38</td>
</tr>
<tr>
<td>After Optimization</td>
<td>98.92</td>
<td>396.39</td>
</tr>
<tr>
<td>Improvement</td>
<td><strong>29.22%</strong></td>
<td><strong>17.14%</strong></td>
</tr>
</tbody>
</table>

**Table 4.2: Webcam capture time optimization results**

4.2.2 Optimization 2: Algorithmic Optimizations

In this optimization the algorithm is studied and several modifications were done on it to bring down the execution time. The combined execution time after optimization 1, for the modules 1, 2, 5 and 6 was **343.77ms**. The key idea here is to bring down the overall run time of the algorithm by bringing down the execution time of these modules. From experimentation it is seen that these optimizations although affect algorithm’s efficiency by a small margin, it can still produce the desired result.

The algorithmic modifications include,

(i) **Change in approach for reference image capture:** Earlier the algorithm captured image in both the reference and current image modules for background subtraction, causing higher execution time in worst-case scenario. The algorithm is modified to capture image every time at the beginning of the loop and assign it to the current image pointer. When the reference is to be captured the captured frame is assigned to the reference image and the process is continued. This optimization resulted in only one capture at the start and brought down the worst-case execution time.
(ii) Change in Morphological transformation module: In the morphological transformation module, several noise reduction techniques are performed on the foreground image to remove the noise and to obtain a closed contour. The algorithm is modified to use only a smoothing and morphological closing technique. This affects the output noise level in the frame, but it is within acceptable limit and does not affect the contour to be retrieved from the frame.

(iii) Change in contour processing: In the contour processing module the contour is analyzed to recognize the gestures. Processing the original contour will be costly in computations, as the contour uses more than 900 points to represent it. As a modification the contour is approximated and it is replaced with another contour with fewer vertices. Also the recognition technique is slightly modified to processing points if they match a certain criteria. Hence the algorithms will have to process only less number of points. These modifications were already explained in section 3.2.3 and 3.2.4.

Once the above modifications are done on the algorithm it is profiled again to check for the improvement in overall execution time. The execution time of the modules after modification was improved by 33% as in table 4.3.

<table>
<thead>
<tr>
<th></th>
<th>Module’s Execution time (ms)</th>
<th>Overall Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before Optimization</td>
<td>343.77</td>
<td>396.39</td>
</tr>
<tr>
<td>After Optimization</td>
<td>229.83</td>
<td>282.76</td>
</tr>
<tr>
<td>Improvement</td>
<td>33.14%</td>
<td>28.66%</td>
</tr>
</tbody>
</table>

Table 4.3: Execution time after algorithmic optimizations

4.2.3 Optimization 3: Data Parallelism (SIMD) with ARM NEON

Single Instruction Multiple Data (SIMD) allows a group of simple processors to all perform the same task at the same time on different data elements and helps in exploiting data parallelism. A digital image contains thousands of pixels that are all processed in the same way. Each time when an image is processed, major part of the execution time is spent in loops to apply the same operation over all the pixels. In a SIMD processor this operation can be performed by packing the data elements into a block and performing operations over the entire block.

In this thesis, the image resolution used for all processing is 640x480 pixels. The pixels are of 8-bits in length (data-type: char) and the same operation has to be performed on 307200 pixels over a loop. If using SIMD, four such pixel elements can be packed into a 32 bit block and a common operation is performed on the block, then the total number of operations will drop to 76800 as seen in fig 4.3. This is the main idea behind SIMD and data parallelism.

The platform i.MX515EVK consists of SIMD extension - NEON with thirty-two 64-bit registers. As shown in fig 4.4, NEON can access these registers in two ways as,
4.2. OPTIMIZATIONS FOR THE ALGORITHM

1. 32 x 64-bit registers [D0-D31], Double-word registers
2. 16 x 128-bit registers [Q0-Q15], Quad-word registers

The NEON instructions support signed/unsigned 8-bit, 16-bit, 32-bit, 64-bit integers and single precision floating point values [33].

4.2.4 Methods to program NEON unit

Three approaches are currently possible to utilize the NEON SIMD extension as follows,

4.2.4.1 Programming NEON with vectorizing compilers

The first method to program the NEON is to vectorize the data. Vectorization is the process of converting the program that performs one operation at a time (scalar) to perform multiple operations simultaneously (vector). By a process called automatic vectorization the GCC compiler can convert the scalar code to vector using compiler flags. The compile options used to achieve auto vectorization is given below,
From the above command, with the `-mfpu=neon` option we instruct the compiler to use the NEON SIMD unit unlike the vfpv3 unit that is used earlier. The flag `-ftree-vectorize` instructs the compiler to auto vectorize the code.

It is understood from [34] that GCC 4.4.1 was not matured enough to auto-vectorize the code. Similar tests were performed with a vectorizable code, to see if there is any improvement in GCC 4.4.3. But the compiler was still not able to auto-vectorize the code. The assembly instructions obtained without vectorizing and by enabling auto-vectorization flags are compared and it turns out that both the assembly files are similar. Hence it is evident that the compiler is still in the early days to generate vectorized code. The code used to test the compiler for auto-vectorization and the results are given in appendix B. And we move to the next approach to program NEON.

### 4.2.4.2 Programming NEON with C-Intrinsics

NEON intrinsics are C wrappers around NEON assembly instructions. An intrinsic function appears as a function call in C and is replaced by equivalent assembly instructions during compilation. Tasks such as type-checking and register allocation are managed automatically while compiling. Hence programming with intrinsics is the easiest way to program a NEON unit. To utilize the intrinsic functions the header `arm_neon.h` has to be included in the source file. The example below shows how NEON intrinsic is to be used.

**Code:** Intrinsic - `neon_split_8bit` splits color image into three individual channels

```c
void neon_split_8bit (uint8_t *dest_r, uint8_t *dest_g, uint8_t *dest_b, uint8_t *src, int n)
{
    int i;
    n/=8;
    for (i=0; i<n; i++)
    {
        uint8x8x3_t rgb = vld3_u8 (src);
        vst1_u8 (dest_r, rgb.val[0]);
        vst1_u8 (dest_g, rgb.val[1]);
        vst1_u8 (dest_b, rgb.val[2]);
        src += 8*3;
        dest_r += 8;
        dest_g += 8;
        dest_b += 8;
    }
}
```

where `uint8_t` data type stores integer of 8-bit. The `uint8x8x3_t` represents a
vector which stores 8 elements which are integers of 8-bit length with 3 channel data. The \texttt{vld3\_u8} instruction loads data into the vector and the \texttt{vst1\_u8} instruction stores data from the vector into the array. This way we can program the NEON unit using c intrinsics. The results obtained show positive improvements as seen from table 4.4.

<table>
<thead>
<tr>
<th>#</th>
<th>Function</th>
<th>Execution Time (ms)</th>
<th>Gain: wrt. (a) (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Normal(a)</td>
<td>Intr.64bit</td>
</tr>
<tr>
<td>1</td>
<td>Split</td>
<td>28516</td>
<td>9635</td>
</tr>
<tr>
<td>2</td>
<td>Convert</td>
<td>8231</td>
<td>7250</td>
</tr>
<tr>
<td>3</td>
<td>Abs Diff</td>
<td>4350</td>
<td>2780</td>
</tr>
<tr>
<td>4</td>
<td>Logical Or</td>
<td>2415</td>
<td>3030</td>
</tr>
</tbody>
</table>

Table 4.4: Programming NEON with intrinsics - Results

When using intrinsics for NEON, eight 8-bit integers are packed into a vector and operated. And the theoretical speed up should be around 8. But as it is seen from the table the speed up obtained was not half of the expected speed up. This is because the assembly code emitted by the compiler was not optimized for ARM boards. This is understood by comparing the hand-written assembly and generated assembly file. Kindly refer appendix C for details.

Hence the possible solution was to manually program NEON in assembly to obtain maximum efficiency, which is the next step.

4.2.4.3 Programming NEON with assembly

Assembly programming is a direct way of programming NEON and it optimize at the lowest level. The function to be optimized is first written using C intrinsics and the corresponding hand written assembly file is developed by replacing intrinsics with corresponding assembly codes from [33]. The GNU assembling tool can generates the object code for NEON from the hand-written assembly. For the RGB Split function detailed above, a hand-written assembly code is developed and is given below,

\textbf{Code:} Assembly - \texttt{neon\_split\_8bit} splits color image into three individual channels

\begin{verbatim}
neon_split_8bit:
    push {r4-r5,lr}
    MOV r4, #38400

.loop:
    # load 8 pixels:
    vld3.8 {d0, d1, d2}, [r3]!
    vst1.8 {d0}, [r0]!
    vst1.8 {d1}, [r1]!
    vst1.8 {d2}, [r2]!
\end{verbatim}
Here the \texttt{vld3.8} instruction (corresponding assembly instruction to \texttt{vld3_u8 intrinsic}) loads data into the vector and the \texttt{vst1.8} instruction (corresponding assembly instruction to \texttt{vst1_u8 intrinsic}) stores data from the vector into the array.

Hand-written assembly is tough to develop and is time consuming. Hence four functions that will have an impact on the final execution time and are relatively easy to be programmed are hand-written into assembly. The results for programming NEON with assembly (both 64-bit and 128-bit wherever applicable) is given below and discussed.

<table>
<thead>
<tr>
<th>#</th>
<th>Function</th>
<th>Execution Time (ms)</th>
<th>Gain: wrt. (a) (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Normal(a)</td>
<td>Assm.64bit</td>
</tr>
<tr>
<td>1</td>
<td>Split</td>
<td>28516</td>
<td>6991</td>
</tr>
<tr>
<td>2</td>
<td>Convert</td>
<td>8231</td>
<td>4440</td>
</tr>
<tr>
<td>3</td>
<td>Abs Diff</td>
<td>4350</td>
<td>2650</td>
</tr>
<tr>
<td>4</td>
<td>Logical Or</td>
<td>2415</td>
<td>2962</td>
</tr>
</tbody>
</table>

Table 4.5: Programming NEON with assembly - Results

We can see from the table 4.4 and table 4.5 that programming with Assembly instead of using intrinsics gives us better speedup. Also when we look at the last function, when using it with NEON intrinsic and assembly 64-bit registers the execution time is more for the SIMD function than the normal one. This is because the intrinsic and assembly functions spend significant time loading and off-loading the data to and from the registers. This contributes a lot to the execution time along with the actual work performed. Hence there is an increase in the execution time. But the loading time is less for 128-bit registers, as 16 integers are loaded at a time and hence we obtain a positive speedup when we program NEON with 128-bit registers.

<table>
<thead>
<tr>
<th></th>
<th>Module’s Execution time (ms)</th>
<th>Overall Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before Optimization</td>
<td>81.05</td>
<td>282.76</td>
</tr>
<tr>
<td>After Optimization</td>
<td>33.73</td>
<td>235.44</td>
</tr>
<tr>
<td>Improvement</td>
<td>58.38%</td>
<td>16.73%</td>
</tr>
</tbody>
</table>

Table 4.6: Execution time after NEON optimization

A significant improvement of 16.7% in the overall execution time is noticed when programming NEON with assembly. The results and the improvements are given in the table 4.6. Graphs comparing the execution time for individual modules by programming NEON with C-intrinsics and assembly approaches are in fig 4.5.
4.2. OPTIMIZATIONS FOR THE ALGORITHM

Thus the optimizations explained above are performed on various modules the execution time for each of the modules was brought down. The overall execution time after the optimizations from table 4.7 is found to be $235.44 \text{ms}$, thus giving a frame rate of $4.2 \text{ frames/sec}$, which is twice the previous result.

Hence we can see that the optimizations performed have brought down the execution time of the algorithm in i.MX515EVK significantly.
4.3 Conclusion

In this chapter we covered the porting of gesture recognition system to the i.MX515EVK platform. The details about setting up the platform and getting the OS into a SD card are covered. The frame rate was improved from 2 frames/sec to 4 frames/sec by performing several optimizations such as algorithmic optimizations and optimizations with the help of platform; especially utilizing NEON SIMD unit is detailed. In the next chapter, as experimentation an application interface is developed to be work with the gesture recognition system. The application will understand gestures and will perform tasks that are mapped with the identified gesture.
Previous chapter covered the implementation details and optimization performed on gesture recognition algorithm for the embedded platform i.MX515. In section 5.1, for experimentation, a real-time application that controls a music player with gestures is developed. This application is to verify the algorithm’s correctness and to test if it is possible to control the application using gestures. The results obtained and the performance of the application are presented in section 5.2 and are discussed.

5.1 Application Development

In the previous chapters, a gesture recognition algorithm is developed first in PC and then later built for the platform i.MX515 running Ubuntu OS. The algorithm was later optimized to bring down the execution time for real time applications. In this section an application to control Banshee music player is built. The application can read images, recognize gestures in them and with the gestures control the player. The flowchart for the process is described in fig 5.1.

![Flowchart for a real-time gesture control system](image)

There are two key stages in the application as seen from fig 5.1. They are,
1. Gesture recognition algorithm - to obtain gestures
2. Decision stage - to control Banshee with gestures

The application will control Banshee, an open-source media player for the Linux distributions. The gesture recognition algorithm to recognize gestures was already discussed in the previous chapter. We will now focus on the decision stage that will perform key press event with the gestures. In order to map the gestures to key press events, X11 libraries are used. The X11 system, its key functions and the implementation details are described in the following section.
5.1.1 X11 and Keyboard Events

X window system (X11) provides the framework for building GUI systems. It provides functions to control windows and to interact with them using keyboard, mouse, and touchscreen events. With this idea, a keyboard event can be simulated using X11 libraries. The \texttt{Xlib.h} header contains definition for structures and macros used in the X11 program, forms the integral part of the X11 system. The \texttt{display} is a structure defined in Xlib.h to represent the window on which the keyboard events will appear. The function \texttt{XOpenDisplay()} establishes a connection to the display on the machine and if no connection can be established \texttt{XOpenDisplay} returns NULL.

A Key code and a Key symbol play an important role in simulating keyboard events. A \texttt{KeySym} is an encoding of a symbol on the cap of a key. The library \texttt{keysymdef.h} contains pre-defined KeySyms, e.g Key symbol for English keyboards. A single key can correspond to different keysyms, for e.g, the key next to character ’m’ in English keyboard maps both to ’comma’ and to ’less than’ key symbols. A \texttt{KeyCode} is a value between 8 and 255 that represents a physical key. The mapping between keys and KeyCodes are fixed and cannot be changed. Generally in Linux OS whenever a key is pressed a keysym is generated and is converted to its respective keycode using the X11 function \texttt{XKeysymToKeycode()}. The OS finally performs a keyboard event using the key code.

![Figure 5.2: Simulating keyboard events with gestures: Process flow](image)

Similar idea is used to control the Banshee media player from a distance with gestures only and without any conventional input device. The flowchart to simulate keyboard events with the gestures is presented in fig 5.2. The gesture recognition algorithm running on i.MX515 captures images and recognizes gestures. Once the gestures are recognized they are then passed to a decision stage that maps the gestures with the corresponding key codes and simulate a key press event.

The flowchart for the function that maps gestures to events is presented in fig 5.3. As seen the input gesture for the current iteration is obtained from the recognition algorithm and compared with the gesture from previous iteration. If the same gesture is recognized for a period of time (say here: till counter reaches 10) then is considered valid input. If the input is valid, the decision stage is called with the gesture, else the function continues with next input. The decision stage takes in gesture and performs a key press event on Banshee player window, which will control the application. The series of steps from gesture to key press event is summarized in the following steps.
5.1. APPLICATION DEVELOPMENT

To simulate a keyboard event with gestures the following steps are carried out,

1. Map key symbol corresponding to gesture
2. Open display window - where the event is simulated (Banshee player)
3. Converts key symbol to key code
4. Simulate event - by calling XTestFakeKeyEvent (keycode) library call
5. Flush the output buffer - to perform event on display using XFlush (display) call
6. Close the display - and exit

The pseudo code for the key press event simulation function is given below,

**Code:** Simulate Keyboard Event with X11 - function

```c
#include <X11/Xlib.h>
#include <X11/keysym.h>

Display *display; // Banshee player window
key_event_function(key_symbol)
{
    display = XOpenDisplay(NULL); // open display
    keycode = XKeysymToKeycode(display, XK_1);
    XTestFakeKeyEvent(display, keycode, True, 0); // simulate key press event
    XTestFakeKeyEvent(display, keycode, False, 0);
}
```

Figure 5.3: Flowchart for decision stage function that maps gestures to keyboard events
The Banshee player can be controlled with several keyboard shortcuts. In this experiment, the main idea is to build an application that can read gestures using the recognition algorithm and the gestures are in turn converted to key events.

Six different gestures are identified by the gesture recognition algorithm. For each gesture that is recognized, a corresponding key symbol is mapped with it to perform key press event with X11. The gesture and key symbol mapping is given in Table 5.1. To compile the algorithm that uses X11 libraries the following command is used,

```
gcc -mcpu=cortex-a8 -mfloat-abi=softfp -ffast-math -mfpu=vfpv3 -marm -O3 -lX11 -lXtst 'pkg-config --cflags --libs opencv' -o control application gesture_algo.c
```

Most parameters in the above command were already explained in section 4.1 except X11 and Xtst. The use of these two parameters are explained below,

- `-lX11`: instruct compiler to utilize the X11 libraries
- `-lXtst`: instruct compiler to utilize the Xtst libraries

The number of successful points in gesture and the mapping of gestures with the key code and key event is given in the table below,

<table>
<thead>
<tr>
<th>Successful points in gesture</th>
<th>Mapped key symbol</th>
<th>Corresponding key code</th>
<th>Banshee player application</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>XK_Control_L</td>
<td>37</td>
<td>Exit Banshee</td>
</tr>
<tr>
<td></td>
<td>XK_W</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>XK_F</td>
<td>41</td>
<td>Full Screen</td>
</tr>
<tr>
<td>2</td>
<td>XK_N</td>
<td>57</td>
<td>Next Track</td>
</tr>
<tr>
<td>3</td>
<td>XK_B</td>
<td>56</td>
<td>Previous Track</td>
</tr>
<tr>
<td>4</td>
<td>XK_R</td>
<td>27</td>
<td>Restart Track</td>
</tr>
<tr>
<td>5</td>
<td>XK_space</td>
<td>65</td>
<td>Pause / Play</td>
</tr>
</tbody>
</table>

Table 5.1: Gestures to key symbol mapping table

The application to control the Banshee player is designed in this manner. In the following section we will study the various performance parameters of the algorithm.
5.2 Application Performance

The application to simulate key press events with the gestures is developed as discussed in the previous section. The next step is to measure the performance of the algorithm during execution. The main aspect considered is the execution time of each module and in turn the frame rate of the application to match a real-time application need. Other factors which are of primary interest in an embedded application such as the memory requirement and energy consumption of the application are also measured.

Execution Time:

<table>
<thead>
<tr>
<th>#</th>
<th>Algorithm Module</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Reference Image Capture</td>
<td>6.97</td>
</tr>
<tr>
<td>2</td>
<td>Current Image Capture</td>
<td>100.17</td>
</tr>
<tr>
<td>3</td>
<td>Image Split</td>
<td>7.21</td>
</tr>
<tr>
<td>4</td>
<td>Background Subtraction</td>
<td>19.55</td>
</tr>
<tr>
<td>5</td>
<td>Morphological transformation</td>
<td>92.25</td>
</tr>
<tr>
<td>6</td>
<td>Contour processing</td>
<td>9.29</td>
</tr>
<tr>
<td>7</td>
<td>Appl. Control</td>
<td>11.79</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td><strong>247.23</strong></td>
</tr>
</tbody>
</table>

Table 5.2: Gesture Algorithm execution time after optimizations

The execution time for each of the module after optimization is presented in table 5.2. The total time required by the application was about 247ms, resulting in a frame rate of 4 frames/sec. The overall time required for algorithm optimized with NEON is less compared to other versions of the algorithms as seen from table 5.3.

<table>
<thead>
<tr>
<th>#</th>
<th>Algorithm</th>
<th>Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Gesture Algorithm without optimization</td>
<td>461.27</td>
</tr>
<tr>
<td>2</td>
<td>Gesture Algorithm optimized without NEON</td>
<td>294.24</td>
</tr>
<tr>
<td>3</td>
<td>Gesture Algorithm optimized with NEON</td>
<td>247.23</td>
</tr>
</tbody>
</table>

Table 5.3: Comparison of execution times for various versions of application control algorithm

Application Size:

The cost of memory is as much important as the execution time for the algorithm and hence as a next step to evaluate performance the application size is measured. The i.MX515EVK supports both ARM 32-bit and Thumb 16-bit instruction set as discussed in section 2.4. To give an overview, the thumb instruction (16-bit) act as a shorthand for the corresponding ARM instruction (32-bit) and when compiling the code in the Thumb state, the processor simply expands the smaller shorthand instructions fetched from memory into their 32-bit equivalents. As a result the code-density of the algorithm increases and program size decreases as seen in appendix D. table 5.4 and table 5.5 presents the results obtained when using ARM 32-bit and Thumb 16-bit instruction set.
Application size and execution time when ARM 32-bit instruction set is used

<table>
<thead>
<tr>
<th>#</th>
<th>Algorithm</th>
<th>Size(bytes)</th>
<th>Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Algorithm optimized without NEON</td>
<td>6002</td>
<td>284.53</td>
</tr>
<tr>
<td>2</td>
<td>Algorithm optimized with NEON</td>
<td>5641</td>
<td>231.35</td>
</tr>
</tbody>
</table>

Table 5.4: Comparison of application size when using ARM 32-bit instruction set

Application size and execution time when Thumb-2 16-bit instruction set is used

<table>
<thead>
<tr>
<th>#</th>
<th>Algorithm</th>
<th>Size(bytes)</th>
<th>Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Algorithm optimized without NEON</td>
<td>5394</td>
<td>288.05</td>
</tr>
<tr>
<td>2</td>
<td>Algorithm optimized with NEON</td>
<td>5090</td>
<td>233.26</td>
</tr>
</tbody>
</table>

Table 5.5: Comparison of application size when using Thumb 16-bit instruction set

It is evident from the table that Thumb instruction set results in code that uses less memory, but the execution time is a little more than the standard 32-bit ARM instruction which is expected. But as a trade-off the Thumb instruction set can be used for its lesser code size.

**Run-Time Memory Requirements:**

Table 5.6 presents the runtime memory requirements for the application with various versions of the algorithm. The run time memory details are obtained using the 'top' command, which gives the memory usage of individual process in Ubuntu.

<table>
<thead>
<tr>
<th>#</th>
<th>Algorithm</th>
<th>Size(bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Algorithm without optimization</td>
<td>4.0% (17.84Mb)</td>
</tr>
<tr>
<td>2</td>
<td>Algorithm optimized without NEON</td>
<td>3.4% (15.16Mb)</td>
</tr>
<tr>
<td>3</td>
<td>Algorithm optimized with NEON</td>
<td>3.4% (15.16Mb)</td>
</tr>
</tbody>
</table>

Table 5.6: Comparison of run time memory requirements for various versions of the algorithm

As seen from the results, the memory required during runtime for both the optimized versions are the same, which is less when compared to the non-optimized version of the algorithm.

**Current Drawn and Energy Consumption:**

The power consumed by the embedded platform i.MX515EVK during its idle state is measured and compared against the power consumption when the algorithm is run. The observed data is presented in fig 5.4. From the data it is seen that the average current consumed by the i.MX515 platform during its idle state is **0.834mA**. The algorithm
that is optimized without NEON consumes an average current of 1.048mA whereas the algorithm with NEON consumes 1.052mA.

<table>
<thead>
<tr>
<th>#</th>
<th>Algorithm</th>
<th>Run Time (sec)</th>
<th>Current (mA)</th>
<th>Energy (Ws)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Algorithm optimized without NEON</td>
<td>29.97</td>
<td>1.048</td>
<td>32.07</td>
</tr>
<tr>
<td>2</td>
<td>Algorithm optimized with NEON</td>
<td>24.71</td>
<td>1.052</td>
<td>26.93</td>
</tr>
</tbody>
</table>

Energy Saving 16.02%

Table 5.7: Improvement in energy consumption when NEON SIMD unit is utilized for optimization

For energy consumption, the algorithm is made to capture 100 images and process them. The time required for the algorithm to process 100 frames with both the versions of the algorithm are 1.048mA and 1.052mA respectively. The algorithm without NEON consumed 0.214mA and the algorithm with NEON consumed 0.218mA additional current, which is 26% more compared to the idle state.

![Energy Consumption (Watt-Sec)](image)

Figure 5.4: Energy consumption between algorithms with and without NEON

From the table 5.7 we see that the energy consumption for algorithm without NEON is 32.07 Watt-sec where as to process the same data the algorithm with NEON consumed 26.93 Watt-sec energy, which is 16% less than the previous value. Hence we can see that to process the same data an algorithm with NEON requires lesser time and in-turn consumes lesser energy.

5.3 Conclusion

In this chapter, a real-time application to control a music player with gestures is discussed. Once the application is implemented, the vital parameters for an embedded application, such as its speed of execution, memory and power requirements for the application are studied using various algorithms and options. In the next chapter, an overall
conclusion for the thesis is presented and the performance of application is discussed with the results from this chapter.
Conclusion and future work

In this thesis a gesture recognition system is built for an embedded platform and an application to control a media player with gestures is developed. This chapter gives a summary and conclusions of the work carried out and also possible future work related to this thesis.

6.1 Summary and Conclusions

There is a vast development in the Human-Computer Interaction over the last decade but much of the noticeable improvements are in how a computer talks to a user. The growth is not the same in how a user talks to the system. Currently there are only a very few systems that can understand any form of inputs other than with mouse and keyboard. In future we will have devices with more natural interfaces such as with gestures.

In order to develop a robust gesture recognition algorithm, previous works were studied. Earlier approaches to gesture recognition mostly used costly depth aware 3D cameras with which a hand segment can be easily segmented. Also the gesture recognition approaches were mostly machine-learning based which required a lot of memory. The main challenge was to get the gesture recognition system to work with a 2D camera for an embedded platform with limited resources.

As a first step, a system to control an application from a distance using a retro-reflective ring and an infra-red camera is designed. This is done in order to get familiar with the development environment and to get a feel of how to communicate with a system from a distance. With this setup it was possible to simulate the mouse events such as mouse movements and clicks with the ring and to control an application such as a power-point presentation. This helped me to understand the need for a setup to communicate with the system from a distance without conventional devices and also to understand the development environment.

Further, the main challenge was to achieve the control without an external device and any special camera. As a result a gesture recognition system which can recognize six different hand gestures using an off the shelf 2D camera is developed. Since a 2D camera lacks depth information, one of the main thesis contributions is developing a robust background subtraction technique for a static background. The frame differencing approach takes 4.81ms and operates on grayscale images. The drawback with this method is that since it operates on grayscale values alone, the intensities of different colored pixels are considered same and it leads to ghost image formation. The better
solution is the skin color background subtraction that operates on colored images. This approach because of RGB to HSV conversions took 28.51ms which is much more than the earlier method. The ghost image formation is resolved with this approach but as it operates on binary masks with skin colored elements, this approach fails when the skin colored hand moves over skin colored objects. The RGB per-channel subtraction designed as a part of this thesis gives better accuracy than skin colored subtraction and at the same time consumes 11.66ms which is around 60% reduction in time. After the background subtraction is performed the hand segment is obtained using the contour based approach.

The next stage is to analyze the hand segment for gestures. Unlike the earlier approaches which used machine learning approaches, in this thesis the hand contour is algorithmically processed to recognize the gesture. This approach is much suitable for low-memory embedded applications. Two different approaches are carried out to recognize gestures; the first approach peak-detection classifies the gesture based on the number of peaks in the contour. This approach recognizes gesture in 9.82ms which is good for a real-time application. The approach is simple with fewer computations but it lacks robustness when the hand is tilted at some degrees and with distance, hence a robust approach is designed. The second approach classifies gestures based on the angle of curvature with its neighboring points. This method is more robust than the previous approach and involves more computations, which took around 12ms for recognizing gestures. Hence taking into account the computation time and robustness, the second approach is implemented in the recognition algorithm.

In the next step the algorithm is built for the i.MX515 embedded platform that runs a full-fledged OS Ubuntu 10.04. After the first build it is seen that the total time required by the algorithm was about 478.38ms, which will result in less than 2 frames per second after the final application is built. Hence in order to obtain better frame rate, the algorithm is profiled to understand the most time consuming parts and to optimize them. Three different optimizations were performed. The first optimization was to bring down the capture time of the USB camera. The number of intermediate buffers is limited to 2 and also the frame that is captured is pushed directly to the user without memory copy. This resulted in reduction of the capture time from 139.76ms to 98.92ms i.e. by more than 27%. The second optimization was done on the algorithm, to modify the way it captures reference and current image and also in approximating the contour. This optimization brought down the overall time for the modules by 33.14%. The third optimization performs data parallelism, using the platform’s SIMD unit called NEON. The NEON unit is first programmed using C intrinsic and later using assembly instructions. The assembly instructions provided better results and hence several modules of the algorithm are vectorized using assembly language to take advantage of the NEON unit. This optimization resulted in 41.62% reduction in the execution time of the modules. From the last two stages I was able to understand the approach to design the gesture algorithm and to port it for the embedded platform. Profiling the algorithm and performing optimizations to bring down the speed of execution so as to obtain a better frame rate, helped me to understand the need and methodology to
optimize an algorithm for real-time applications.

As a final step to verify the algorithm for correctness, an application that utilizes the gesture algorithm to control Banshee media player is designed and its performance is measured. The application captures image and finds the gesture with the recognition algorithm. The gesture that is analyzed is then mapped with several key symbols to simulate key events using X11 window system. Once this is done, when the gestures are recognized, they are converted to corresponding key events. The application was able to understand gestures and perform six different key events on the Banshee media player and in turn control it. Building this application helped me to verify the gesture algorithm that is built on the embedded platform and its working.

Once the application is built its performance is measured. Parameters such as algorithm size, its execution time, memory requirements and current drawn and energy requirements are analyzed. The overall execution time of the application that uses the optimized gesture algorithms is about 247.23ms which in turn results in a frame rate of more than 4 frames/second which is a good value for a real-time application. The algorithm when built with standard ARM 32-bit instruction set required 5.51KB while it required only 4.97KB when built with Thumb-2 16-bit instruction set, which saves around 10% in memory required to store the algorithm. The algorithm makes use of more number of image pointers and hence the run time memory requirement is 15.16MB. The current drawn by the application when running the algorithm is 1.052mA which is 26% more when compared to the idle state which is 0.834mA. To compare the energy consumption of the application is made to capture 100 images and then the energy consumption is measured. It is seen that the energy consumption by the algorithm with NEON is 32.07Watt-sec while the algorithm without NEON required only 26.93Watt-sec, which is 16% lesser than the former approach.

This way an application that utilize gesture recognition algorithm to control the system from a distance is designed.

6.2 Recommendations for Future Work

The recommendations for future work in the direction of this thesis are based on parameters below,

1. Extended use of NEON for speed

The SIMD unit of Cortex-A8 processor - NEON was utilized to achieve data parallelism. Compiler GCC 4.4.3 was not intelligent enough to auto-vectorize the algorithm for the NEON unit. So only four modules which were relatively easier to program with assembly for NEON is parallelized, and they showed 42% improvement in execution time. Hence more parts of the algorithm, especially the morphological transformations can be vectorized for NEON unit, which would in-turn improve the speed of the
application.

2. Future processors for speed

The ARM Cortex-A8 processor is the second of the four processors in the Cortex-A family. ARM has already launched its next two Cortex-A series processors Cortex-A9 and Cortex-A15 processors. Cortex-A9 is a 2 GHz and Cortex-A15 is a 2.5 GHz processor, both processors are scalable up to 4 cores with its individual floating point and NEON unit as shown in Figure 6.1. These processors can be used in-place of Cortex-A8 processors to perform computations in parallel, and to utilize multiple-cores which would bring down the execution time significantly.

![Figure 6.1: ARM Cortex-A9 and A15 processors architecture](image)

3. Low-Cost Embedded Platforms

The i.MX515 platform used in this thesis has support for external DVI display. The external display was required while building the algorithm for the embedded platform, to debug and to make changes to the algorithm and to observe its effects in external display. At the final step, the algorithm is more robust and its behavior is known. Hence it can be moved to low-cost embedded platforms which have fewer resources compared to the current platform to bring down cost.

4. Algorithmic Enhancements

The application currently recognizes six different gestures. As a final recommendation, the algorithm can be enhanced considering this design as a basic approach, to recognize more number of gestures and the algorithm would have wider scope of application.
Bibliography


Various levels of GCC optimizations

Entire list of GCC optimization flags and the levels at which they are enabled

<table>
<thead>
<tr>
<th>Optimization</th>
<th>-O1</th>
<th>-O2</th>
<th>-Os</th>
<th>-O3</th>
</tr>
</thead>
<tbody>
<tr>
<td>defer-pop</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>branch-jumps</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>branch-probabilities</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>copy-registers</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>guess-branch-probability</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>anti-frame-pointer</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>align-koops</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>align-jumps</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>align-labels</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>align-functions</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>optimize-sibling-calls</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>case-follow-jumps</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>case-skip-blocks</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>gcall</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>expensive-optimizations</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>strength-reduce</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>return-case-after-loop</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>return-loop-opt</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>caller-saves</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>force-c eax</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>pessphile2</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>regmove</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>strict-allocaing</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>delete-null-pointer-checks</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>reorder-blocks</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>schedule-intras</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>schedule-intra2</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>inline-functions</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>rename-registers</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

Figure A.1: Optimization with GCC

The figure given above gives the list of GCC optimizations -O1, -O2, -O3, -Os and the flags enabled at each level.
Auto-vectorization with GCC

Comparison between assembly files with and without vectorization flags to understand if GCC 4.4.3 is capable of auto-vectorization.

**Code:** Source-code used for vectorization

```c
#include <stdio.h>
void main()
{
    int a[128], b[128]=0;
    int i;
    for (i=0; i<128; i++)
    {
        a[i] = b[i] + 5;
    }
}
```

**File:** Assembly file obtained with vectorization flags disabled

```
gcc -mfloat-abi=softfp -mtune=cortex-a8 -ffast-math -mfpu=vfp3 -S vect_test.c
```

```assembly
main:
  @ args = 0, pretend = 0, frame = 1032
  @ frame_needed = 1, uses_anonymous_args = 0
  push {r7, lr}
  sub sp, sp, #1032
  add r7, sp, #0
  add r3, r7, #8
  sub r3, r3, #4
  mov r2, r3
  mov r3, #512
  mov r0, r2
  mov r1, #0
  mov r2, r3
  bl memset
  mov r3, #0
  str r3, [r7, #1028]
```

65
APPENDIX B. AUTO-VECTORIZATION WITH GCC

b .L2
.L3:
   ldr r2, [r7, #1028]
   ldr r1, [r7, #1028]
   add r3, r7, #8
   sub r3, r3, #4
   ldr r3, [r3, r1, lsl #2]
   add r1, r3, #5
   add r3, r7, #516
   str r1, [r3, r2, lsl #2]
   ldr r3, [r7, #1028]
   add r3, r3, #1
   str r3, [r7, #1028]
.L2:
   ldr r3, [r7, #1028]
   cmp r3, #127
   b .L3
   add r7, r7, #8
   add r7, r7, #1024
   mov sp, r7
   pop {r7, pc}

File: Assembly file obtained with vectorization flags enabled

gcc -mfloat-abi=softfp -mtune=cortex-a8 -ffast-math -ftree-vectorize -mfpu=neon
   -mvectorize-with-neon-quad -S vect_test.c

main:
   @ args = 0, pretend = 0, frame = 1032
   @ frame_needed = 1, uses_anonymous_args = 0
   push {r7, lr}
   sub sp, sp, #1032
   add r7, sp, #0
   add r3, r7, #8
   sub r3, r3, #4
   mov r2, r3
   mov r3, #512
   mov r0, r2
   mov r1, #0
   mov r2, r3
   bl memset
   mov r3, #0
   str r3, [r7, #1028]
   b .L2
Comparing the above assembly instructions, we can infer that GCC 4.4.3 is still in early days for auto-vectorization.
Comparison between GCC 4.4.3 and hand-optimized assembly file for NEON.

**Code:** Source-code used for NEON function

```c
#include "neon_test.h"

void neon_split_8bit (uint8_t *dest_r, uint8_t *dest_g, uint8_t *dest_b, uint8_t *src, int n)
{
    int i;
    n/=8;
    for (i=0; i<n; i++)
    {
        uint8x8x3_t rgb = vld3_u8 (src);
        vst1_u8 (dest_r, rgb.val[0]);
        vst1_u8 (dest_g, rgb.val[1]);
        vst1_u8 (dest_b, rgb.val[2]);
        src += 8*3;
        dest_r += 8;
        dest_g += 8;
        dest_b += 8;
    }
}
```

**File:** Assembly file obtained from GCC 4.4.3

```
neon_split_8bit:
    push {r4, r5, r6, r7, r8, r9, sl, fp}
    sub sp, sp, #72
    mov r5, r3
    ldr fp, [sp, #104]
    str r2, [sp, #20]
    add r2, fp, #7
    str r0, [sp, #12]
    bics fp, fp, fp, asr #32
    it cs
```
movcs fp, r2
str r1, [sp, #16]
lsr fp, fp, #3
cmp fp, #0
ble .L4
movs r4, #0
add ip, sp, #24
mov r6, r4
add r3, sp, #48
str fp, [sp, #0]
str r3, [sp, #4]

.L3:
    vld3.8 {d16-d18}, [r5]
    ldr fp, [sp, #12]
    vstmia ip, {d16-d18}
    ldr r3, [sp, #16]
    mov r7, ip
    add r8, fp, r4
    add sl, r3, r4
    ldr fp, [sp, #20]
    ldmi a r7!, {r0, r1, r2, r3}
    add r9, fp, r4
    ldr fp, [sp, #4]
    adds r6, r6, #1
    adds r5, r5, #24
    adds r4, r4, #8
    stmi a fp!, {r0, r1, r2, r3}
    fldd d16, [sp, #56]
    ldmi a r7, {r0, r1}
    fldd d17, [sp, #48]
    vst1.8 {d17}, [r8]
    vst1.8 {d16}, [sl]
    stmi a fp, {r0, r1}
    fldd d16, [sp, #64]
    vst1.8 {d16}, [r9]
    ldr r3, [sp, #0]
    cmp r3, r6
    bgt .L3

.L4:
    add sp, sp, #72
    pop {r4, r5, r6, r7, r8, r9, sl, fp}
bx lr
File: Hand Optimized assembly file

```assembly
neon_split_8bit:
push {r4-r5,lr}
MOV r4, #38400
.loop:
    #load 8 pixels:
    vld3.8 {d0, d1, d2}, [r3]!
    vst1.8 {d0}, [r0]!
    vst1.8 {d1}, [r1]!
    vst1.8 {d2}, [r2]!
    subs r4, r4, #1
    bne .loop
pop {r4-r5, pc}
```

From the assembly files it is evident that the file produced by GCC 4.4.3 is not optimized and it has to be manually optimized as in second file to obtain maximum achievable results.
Comparison between ARM 32-bit and Thumb 16-Bit program sizes.

**Table:** Arm and Thumb instructions - Program size

<table>
<thead>
<tr>
<th>Function</th>
<th>Program Size (bytes)</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>main.o</td>
<td>472</td>
<td>384</td>
</tr>
<tr>
<td>neon.o</td>
<td>208</td>
<td>152</td>
</tr>
<tr>
<td>executable file</td>
<td>2606</td>
<td>2462</td>
</tr>
</tbody>
</table>

Table D.1: Using Arm and Thumb instructions - Program size

From the table it can be inferred that using thumb 16-bit instruction set for a program will lead to smaller program size.
Overview

Freescale delivers cost-effective i.MX51 evaluation kit, allowing customers to develop, debug and demonstrate their next great product without compromising performance. As part of our new price, performance and personality series, the evaluation kit is designed to support all the features of the device in a small, single-board design to enable designers to complete a development platform at a low price point of less than an estimated $70 US. The i.MX51 EVK has two optional add-on modules: an LCD module and an expansion board which includes a camera, TV-out, keypad and UART. Based on a powerful ARM Cortex™-A7 core, the i.MX51 EVK delivers extreme performance and low power consumption, helping developers design products that meet today’s demands for energy efficiency.

A range of connectivity options makes the i.MX51 EVK suitable for developing many different types of user applications. The provided board support packages (BSP) for Linux® OS and Windows® Embedded CE enable rapid prototyping which helps to speed up the processor selection process and quickly deliver a demo into the hands of the project stakeholders. The i.MX51 EVK includes two SD cards: one pre-loaded with Linux and the other with Windows Embedded CE. Both options support a wide range of automotive, consumer, general embedded and industrial applications.

Key Benefits

- Explore multiple connectivity options with the i.MX51 applications processor: display, touch screen, USB, SDIO, Ethernet and others
- Investigate usage of the video and graphics through the hardware accelerated video processing unit, OpenGL® ES 2.0 and OpenVG™ 1.1 graphics processing units
- Develop with the MC13892 power management chip from Freescale that supports power sequencing of the i.MX51 device and output rails to supply power to external components such as memory and other system peripherals
- Use proven design examples and software drivers to reduce hassle associated with design-in of key connectivity and power management options
- Enable rapid prototyping of human-machine interfaces (HMI) via the on-board digital virtual interface (DVI) peripheral that allows the EVK to interface to a standard PC monitor
- Boot from SD, SPI or NAND flash

Performance

With the i.MX51 EVK, designers have access to key features needed for an end design offering hardware functionality and connectivity required for developing many applications, such as portable media players, mobile Internet devices, smartbooks, gaming consoles, e-books, media phones, digital photo frames, high-end appliances, video and navigation, security and surveillance, medical and factory automation. With production-ready software components, an optimized OS and a system-validated BSP, designers have the tools to test and maximize the performance of the applications they have developed.

Software and hardware engineers can also download this code to the target EVK to test and validate their software and to run and evaluate performance metrics. The ability to have all communications ports working (serial, USB) and to debug over JTAG and Ethernet is essential for product development. The EVK also provides boot select switches, which provide the user with the option to override the default boot setting of the CPU.

Personality

Freescale’s EVK for the i.MX51 applications processor allows designers to quickly prototype and demonstrate the results of their development efforts in a small, portable device the size of a 5 x 5 portlet, giving confidence to project decision makers that the product is much closer to production. Develop user-interactive software and display your product-specific graphical data on a high-quality, touch-screen-enabled 7” WVGA LCD available as an add-on module to the EVK. Connect additional input and output peripherals such as a camera, TV-out, keypad and UART with the expansion board add-on module. With the Freescale i.MX51 EVK, prototyping and development are simplified to improve time to market.

i.MX51 EVK Key Features

CPU
- i.MX51 applications processor
- 4 x 128 MB DDR2
- 4 MB SPI NOR
- PMIC Freescale MC13892
- NAND and ETM headers

Figure E.1: iMX51EVK peripherals one
**APPENDIX E. IMX515EVK LAYOUT**

**Peripherals**
- 7" WVGA touch screen LCD display (add-on module)
- Two USB connectors
- HDMI connector
- Two SD/MMC card slots
- USB host w/ USB 2.0 OTG
- Ethernet port
- Mini PCI Express®
- SATA/IDE connector
- SIM card connector
- Keyboard connector
- Mic input, stereo headphone output (for Li, V2P, headphone)
- Speaker connector
- USB camera connector
- RJ-45 Ethernet connector
- RGB output through DM-1 connector
- USB peripheral header
- Ambient light sensor footprint
- ATMAL expansion header
- Expansion board add-on module with camera, TV out, keypad and UART

**Debug**
- Debug serial port
- JTAG
- Reset, power switches
- Debug LED
- Power source
- Power control button
- Power measurement header

**Software Development Kit**
- Optimized and validated for both Linux and Windows Embedded CE operating systems
- Integrated and validated DSP for the IMX51 EVK feature set
- Highly optimized software that is coded by Freescale processor experts
- Complete application programming interfaces (API) and frameworks across software packages
- Evaluation and production software packages available through a streamlined, Web-based licensing and delivery system
- Freescale development tools, test streams, and documentation provided

**The IMX51/F2 Power Management and Reset Interface IC (PMR1 IC)**

The IMX51/F2 PMR1 IC is designed for use with the IMX51 applications processor requiring a highly integrated, bi-directional power management IC and communications device. Features include:
- Battery charging support
- Wall charging and USB charging
- 18-bit ADC for monitoring battery and input voltages
- Four adjustable output buck converters
- 12 adjustable output low dropout (LDO) with internal and external power devices
- Two, I2C, converters
- Serial backlight drivers
- Power control logic with processor interface and event detection
- Real-time clock and crystal oscillator circuitry
- Touch screen interface
- SPI/PC bus interface

---

Learn More: For current information about Freescale products and documentation, please visit www.freescale.com/imx51-evk

---

Figure E.2: IMX51 EVK peripherals two
Figure E.3: iMX515EVK Layout - Top
Figure E.4: iMX515EVK Layout - Bottom