Low vision assistance with mobile devices

Mark Stump

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation


This Thesis is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
Low Vision Assistance with Mobile Devices

by

Mark Alan Stump

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering

Supervised By

Dr. Andreas Savakis
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
January 2011

Approved By:

Dr. Andreas Savakis
Professor and Department Head, RIT Department of Computer Engineering
Primary Advisor

Dr. Sonia Lopez Alarcon
Assistant Professor, RIT Department of Computer Engineering

Dr. Andres Kwasinski
Assistant Professor, RIT Department of Computer Engineering
Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title: Low Vision Assistance with Mobile Devices

I, Mark Stump, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

_________________________________
Mark Stump

_________________________________
Date
Dedication

I dedicate this work to my parents, Walter and Susan, and my brother Jeffrey, who have always stood behind me throughout the whole of my college career. Additionally, I would like to dedicate this work to my friends Dan, John, and Mike, whose consistent support, friendship, and humor have helped me through my time in the Department of Computer Engineering at the Rochester Institute of Technology.
Acknowledgements

I would like to sincerely thank Dr. Andreas Savakis for his consistent guidance and mentoring during my thesis research, whose help gave me the ability to complete this work.

I also thank Dr. Andres Kwasinski and Dr. Sonia Lopez Alarcon for reviewing my work and offering valuable advice.

Finally I would like to acknowledge Gary Behm of the National Technical Institute of the Deaf who provided the initial direction of the research.
Abstract

Low vision affects many people, both young and old. Low vision conditions can range from near- and far-sightedness to conditions such as blind spots and tunnel vision. With the growing popularity of mobile devices such as smartphones, there is large opportunity for use of these multipurpose devices to provide low vision assistance. Furthermore, Google’s Android operating system provides a robust environment for applications in various fields, including low vision assistance.

The objective of this thesis research is to develop a system for low vision assistance that displays important information at the preferred location of the user’s visual field. To that end, a first release of a prototype blind spot/tunnel vision assistance system was created and demonstrated on an Android smartphone. Various algorithms for face detection and face tracking were implemented on the Android platform and their performance was assessed with regards to metrics such as throughput and battery usage. Specifically, Viola-Jones, Support Vector Machines, and a color-based method from Pai et al were used for face detection. Template matching, CAMShift, and Lucas-Kanade methods were used for face tracking. It was found that face detection and tracking could be successfully executed within acceptable bounds of time and battery usage, and in some cases performed faster than it would take a comparable cloud-based system for offloading algorithm usage to complete execution.
# Table of Contents

*Thesis Release Permission Form* ................................................................. ii  

*Dedication* ........................................................................................................ iii  

*Acknowledgements* ......................................................................................... iv  

*Abstract* .......................................................................................................... v  

*Table of Contents* ............................................................................................. vi  

*List of Figures* .................................................................................................. x  

*List of Tables* .................................................................................................... xii  

*Glossary* ............................................................................................................ xiii  

**Chapter 1**  

*Introduction* .................................................................................................... 2  

1.1  **Background** ............................................................................................. 3  

1.1.1  Low Vision: Scotomas and Tunnel Vision .............................................. 3  

1.1.2  Android .................................................................................................. 5  

1.1.3  Dalvik Virtual Machine .......................................................................... 7  

1.1.4  Android SDK ........................................................................................ 8  

1.1.5  Android NDK ......................................................................................... 9  

1.1.6  Java Native Interface ............................................................................. 10  

1.2  **Previous work on Mobile Devices** .......................................................... 10  

1.2.1  Mobile Device Potential for Users with Special Needs ......................... 10  

1.2.2  AndSURF ............................................................................................ 11  

1.2.3  Bartendroid .......................................................................................... 13  

1.3  **Previous work on Low Vision** ................................................................. 15
1.3.1. Visual Impairment and Disability ......................................................... 15
1.3.2 Augmented Reality and Visual Search .................................................. 15
1.3.3 ViSAR ................................................................................................. 16

1.4 Cloud Computing ..................................................................................... 17

Chapter 2 Supporting Work .................................................................................. 20

2.1 Face Detection ............................................................................................. 20
2.1.1 Viola-Jones .......................................................................................... 20
2.1.2 Pai et al .............................................................................................. 26
2.1.3 Support Vector Machines ........................................................................ 24

2.2 Object Tracking ........................................................................................... 29
2.2.1 Template Matching ................................................................................ 30
2.2.2 Lucas-Kanade ...................................................................................... 30
2.2.3 CAMShift ............................................................................................. 32

Chapter 3 Prototype Systems .............................................................................. 35

3.1 Prototype System Design ............................................................................ 35
3.1.1 Netbook System ................................................................................... 35
3.1.2 Android System .................................................................................. 36
3.1.3 Hybrid System ...................................................................................... 37

3.2 Prototype performance ................................................................................ 38
3.2.1 Android System .................................................................................. 38
3.2.2 Netbook System .................................................................................. 43
3.2.3 Hybrid System ...................................................................................... 45

3.3 Proposed Improvements ............................................................................... 46

Chapter 4 Algorithm Implementations on Mobile Devices .................................. 47
4.1 Face Detection 47

4.1.1 Viola-Jones 47

4.1.2 Support Vector Machine 48

4.1.3 Pai et al 48

4.2 Face Tracking 49

4.2.1 Template Matching 49

4.2.2 Lucas-Kanade 49

4.2.3 CAMShift 50

Chapter 5 Results 51

5.1 Algorithm Performance 51

5.1.1 Execution Time 54

5.1.2 Timing Limitations 61

5.1.3 Battery Usage 62

5.1.4 Memory Usage 65

5.2 Android System 66

5.2.1 Survey on the Android System 66

5.2.2 Final System Implementation 68

Chapter 6 Conclusion 72

Bibliography 75

Appendix A Sobel Edge Detection on Android 81

Overview 81

Theory 82

Implementation Notes 83

Results and Analysis 84
List of Figures

Figure 1 – Example of field of vision without (a) and with (b) scotoma......................... 4
Figure 2 - Example of field of vision without (a) and with (b) tunnel vision................. 5
Figure 3 - Android System Architecture....................................................................... 6
Figure 4 - Android Emulator Example [17].................................................................. 9
Figure 5 - Role of the JNI [20]................................................................................... 10
Figure 6 - Basic architecture of the Bartendroid System [2]......................................... 14
Figure 7 - Original Viola-Jones Weak Classifiers [1]...................................................... 20
Figure 8 – Extended Set of Weak Classifiers [30]........................................................ 21
Figure 9 - Integral Image Example [1].......................................................................... 21
Figure 10 - Top two weak classifiers found via boosting............................................. 23
Figure 11 - Cascade of Classifiers Example................................................................. 23
Figure 12 - SVM Two-Dimensional Example [7].......................................................... 24
Figure 13 - Pai et al Algorithm Flowchart [6]............................................................... 27
Figure 14 - CAMShift Tracking Flowchart [5].............................................................. 33
Figure 15 – CAMShift Image Sequence....................................................................... 33
Figure 16 - Initial System Prototype............................................................................. 35
Figure 17 – Hybrid System Operation Flowchart....................................................... 37
Figure 18 - Proposed Improved System Architecture............................................... 46
Figure 19 - System Difference Overview.................................................................... 52
Figure 20 - Android JPEG Compression Time............................................................ 53
List of Tables

Table 1 - YUV420 to RGB565 Conversion Timings (ms) ......................................................... 38
Table 2 - YUV420 to RGB565 Conversion Timings, Android 2.2 (ms) ................................. 39
Table 3 - Viola-Jones Face Detection Timings, Android (ms) ............................................... 40
Table 4 - Neven Face Detection Timings, Android (ms) ....................................................... 40
Table 5 - Original Prototype Resource Usage ....................................................................... 42
Table 6 - Modified Prototype (Android 2.2) Resource Usage ............................................. 42
Table 7 - Viola-Jones Face Detection Timings, 320 x 240, Netbook (s) ............................... 44
Table 8 – Hybrid System Resource Usage .......................................................................... 45
Glossary

Android
Operating system released by the Google Corporation in October of 2008. Designed for running on systems with limited resources, primarily smartphones.

Augmented Reality
Concept of overlaying virtual imagery over a physical environment in order to convey information not previously available.

Cloud Computing
Computing paradigm where parts of the system in use are not local to the machine in use, such as accessing a large database of images over an internet connection.

Dalvik
Java Virtual Machine used by the Android Operating System. Designed specifically for use on systems with limited resources (processing power, memory, battery, etc.).

Face Detection
Finding a human face within a digital image or video frame through computer vision algorithms.

JNI
Java Native Interface. Programming interface that allows libraries in other languages (Primarily C and C++) to be used in conjunction with Java development.
| **NDK** | Native Development Kit. Suite of development tools based on a native interface for a system. In the case of Android, NDK is allowing development of portions of applications in a more native language to the device, primarily C and C++. |
| **Object Tracking** | Process of locating a moving object (face, vehicle, etc.) throughout a sequence of video frames. |
| **Preferred Retinal Locus** | Favored area of the visual field for individuals with low vision conditions. |
| **Scotoma** | Blind spot. Area of degenerated vision in the visual field. |
| **SDK** | Software Development Kit. Suite of development tools for creation of applications on a specific platform. |
| **SURF** | Speeded-Up Robust Features. Feature Detector used for various computer vision tasks such as object recognition and 3D reconstruction. |
| **Tunnel Vision** | Loss of peripheral vision. |
| **Usher’s Syndrome** | A medical condition that affects both the hearing and the vision of a person. The vision affect presents as a centrally-located scotoma or as tunnel vision. |
Chapter 1 Introduction

The objective of this thesis research is to develop a system for low vision assistance that displays important information at the preferred location of the user’s visual field. The thesis consists of two parts. The first part of the thesis presents prototype systems based on the Viola-Jones [1] method of face detection with simple template matching. Additionally, an Android native face detection method is used for the initial Android-based prototypes. Mobile devices have numerous limitations when compared to computing devices such as laptop and desktop computers. Some of these issues come from the reduced computational power of the phone (slower processor, no native floating-point decimal support, etc) or memory issues (small amount of random access memory, slower memory, and small cache space) [2,3]. Due to these limitations, special care needs to be taken for development on these devices. Additionally, with these issues in mind, the prototype systems were examined in regards to processor and battery usage.

The second part of this thesis consists of an evaluation of various methods of face tracking and detection as implemented on a mobile device. Due to the various limitations of a mobile device, algorithms implemented on a desktop machine tend to perform in a much different manner versus their equivalents on a mobile device. A thorough evaluation of numerous algorithms was completed for both face tracking and face detection. This evaluation considered two criteria for each algorithm, primarily with regards to the constraining factors of a mobile device: throughput and battery usage. For face tracking, template matching, the Lucas-Kanade [4] and CAMShift [5] methods were evaluated. For face detection, in addition to Viola-Jones [1], a method from Pai et al [6] and Support Vector Machines [7] were evaluated for use on a mobile platform. After
these algorithms were theoretically evaluated for a desktop and netbook computer, they were implemented on the mobile device and then evaluated for their throughput and battery power. Finally, once these implementations were complete, a low vision assistance system was developed based on earlier system prototypes and the information taken from the algorithm evaluation.

The low vision assistance work of this thesis considered Usher’s syndrome, a condition that affects both the hearing and the vision of a person. The vision affect primarily appears as a centrally-located scotoma or as tunnel vision [8]. Consultation with the National Technical Institute for the Deaf (NTID) at the Rochester Institute of Technology proved very useful in the development of the interface of the system and provided useful input for the effectiveness of the proposed system.

1.1 **Background**

1.1.1 **Low Vision: Scotomas and Tunnel Vision**

Low vision is a part of daily life for many people and refers to a deficiency in eyesight that cannot be fully corrected. According to the United States Center for Disease Control, low vision conditions are much more common in older people than younger. From a 2002 report, approximately 2.4 million United States citizens ages 40 and over have some form of low vision. The report goes on to state that this figure is likely to double in the next 30 years [9]. Part of this figure comes from diseases that affect the eye, such as cataracts, diabetic retinopathy, glaucoma, macular degeneration, and Usher’s Syndrome [8,10].
Low vision conditions may include a scotoma, or blind spot, which is an area of degenerated vision surrounded by an area of unaffected vision. An example of a scotoma can be seen in Figure 1.

![Figure 1 – Example of field of vision without (a) and with (b) scotoma](image)

In addition to what is shown above, the size, shape, and location of a scotoma can vary greatly from case to case. Large or centrally-located scotomas can severely hamper the ability of someone to communicate with another person. Eccentric viewing (or eccentric fixation) is a condition that arises from an individual needing to use an alternate area of their vision for tasks such as reading, writing, or interacting with others. Over time, an area of vision begins to become preferred, much like the central vision of individuals without low vision. This new, favored field of vision is called a preferred retinal locus, or PRL [11].

In addition to scotomas, another common low vision condition is tunnel vision. Tunnel vision can be described as the opposite of a centrally-located scotoma; the peripheral vision of an individual is lost while the central vision remains. An example of tunnel vision is shown in Figure 2.
Tunnel vision has many causes such as retinitis pigmentosa, glaucoma, choroideremia, and Usher’s Syndrome [8,12]. With tunnel vision, tasks that may be easy for people with unaffected vision can become much more difficult, greatly affecting a user's day-to-day life [12]. However, the potential exists in numerous mobile devices available to aid people with low vision. Android-based mobile devices are one platform with that potential.

1.1.2 Android

Android is an open-source Linux-based operating system created by the Google Corporation for use in mobile devices, primarily for use in “smartphones”. The drive behind the Android operating system came from the Open Handset Alliance, or OHA. The OHA is a body of industry members, with member companies such as Motorola, Samsung, Intel, Qualcomm, and Google [13].

The Android Operating System is originally derived from the 2.6 version of the Linux kernel. With this, handset and device manufactures are able to tweak the operating
system to the needs of their applications and devices [13]. Amongst all the derivations, the basic Android architecture remains relatively consistent, as shown in Figure 3.

![Android System Architecture](image)

**Figure 3 - Android System Architecture**

Although the Android operating system is a relatively new development in the field of mobile devices, some work has been completed on research applications for the system. One relatively recent (2009) Master's thesis from Lund University in Sweden investigated the use of both on-device and cloud computing based approaches to complete operations such as Art Recognition, the ability to recognize various bottles of liqueur and suggest drink recipes (Bartendroid), and implemented a mobile-device optimized version of feature extraction with speeded-up robust features (SURF). As a testament to the speed at which developments occur in the field of mobile devices, the android device used for these implementations was an official Android development phone (a HTC Dream without operator locks) with the following specifications [2]:

- Android Operating System version 1.5 (Cupcake)
- Qualcomm MSM7201A 528 MHz Processor
• 256 MB ROM
• 192 MB RAM
• 3.2-inch TFT-LCD flat touch-sensitive screen with 320 x 480 (HVGA) resolution
• 3.2 megapixel color camera with auto focus

This thesis employed a Motorola Droid device that was donated to the department of Computer Engineering by the Google Corporation. The specifications for the Motorola Droid are shown below:

• Android Operating System version 2.2 (Éclair)
• Arm Cortex A8 550 MHz Processor
• 512 MB ROM
• 256 MB RAM
• 3.7-inch WVGA resolution widescreen
• 5 megapixel color camera with autofocus and dual LED flash

The technology related to the Android operating system is moving incredibly quickly, along with updates to the OS itself.

Finally, since Android is not limited to a single type of phone; numerous Android-based devices have been marketed, from devices with low cost and processing power to high-end phones that are being marketed as more media-centric devices.

1.1.3 Dalvik Virtual Machine

Java is a language that is generally compiled into a bytecode, which is then translated by a virtual machine (VM) to operate on a wider variety of computer hardware. However, since mobile platform architecture is quite different from desktop or laptop architecture, a different virtual machine was needed for the Android Platform. The Dalvik Virtual
Machine is designed to run on a relatively slow processor, use little RAM, and be used on an operating system that does not have any swap space, all while being powered by a battery [14]. These traits make it ideal for running on a mobile device such as an Android smartphone.

While most Java virtual machines are stack-based, the Dalvik VM is a register-based architecture. Additionally, Android applications are transformed into a ‘Dalvik Executable (.dex)’ for execution instead of the standard Java Archive (.jar) or Java class files. The Virtual Machine relies on the underlying operating system (Android’s Linux Kernel) for threading and the lower-level memory management, while at the same time providing an easy-to-develop interface for those familiar with the standard versions of Java available on a wider range of systems [2,15].

### 1.1.4 Android SDK

To aid in the development of applications, the Google Corporation has released a Software Development Kit specifically made for the Android platform. The majority of user-developed software for Android is written in version 6 of the Java programming language. For Android development, it is encouraged to use the Eclipse development tool (http://www.eclipse.org/), since it provides for easy-to-use interfaces and tools as well as very good integration with the Android SDK [16].

The SDK provides many useful tools for the Android developer. Among these are tools for emulation (called Android Virtual Devices) that provide testbeds for application functionality on a variety of platforms [16]. An example of an Android emulator is shown in Figure 4.
Additionally, there are tools for viewing debug information from the device (the Dalvik Debug Monitor Service) as well as some command-line tools, such as the Android Debug Bridge [18]. Overall, the tools provided by Google aid the developer greatly with an application structure that is simple to understand and robust enough to be used in various ways.

1.1.5 Android NDK

The Android Native Development Kit is a tool that is used in conjunction with the normal SDK. The NDK primarily provides the tools and build files to generate libraries from C and C++ source code, as well as a way of embedding these libraries into the APK, the package where Android applications are compiled. These compiled libraries are optimized for the processors in use, which at the time of this writing are all ARM based processors. For the future, support for x86-based architectures is being investigated [19].
1.1.6 Java Native Interface

For communication between the SDK and the NDK, the Java Native Interface (JNI) is used. JNI acts as the communications medium between native code, generally written in C or C++, and the primary application code, written in Java. At a high level, the JNI generally operates as is shown in Figure 5 [20].

![Figure 5 - Role of the JNI](image)

As Figure 5 shows, the JNI works in conjunction with the Java Virtual Machine to provide two-way communication with native libraries and applications [20]. Although the JNI does allow for applications to be written in native code, the Android Operating System and Dalvik Virtual machine only supports libraries being written in native code; applications and user interfaces still must be written in Java with calls to native functions when applicable [19].

1.2 Previous work on Mobile Devices

With the prevalence of mobile devices in the world today, there is a large amount of background work. Three areas of previous work with mobile devices were examined due to their insight into the development of this thesis research.

1.2.1 Mobile Device Potential for Users with Special Needs

As mobile devices gain in processing power and capabilities, the potential for applications to aid those with special needs greatly increases. According to [21], mobile
devices can provide numerous applications to individuals with various needs. Among these functions, mobile devices can act as aids to carry out functions, such as remotely interact with PCs, chair lifts, opening doors, and more. Additionally, mobile devices can be used as an aid in communication, such as using them for text-to-speech functionality. Mobile devices can even be used as assistants with applications such as alerting users of dangers, acting as guides [21], and in the case of this research, helping individuals with low vision.

Mobile devices have also been researched for helping individuals with motor impairments. An application where this was found useful is for people with Muscular Dystrophy, who lose their gross motor control (moving wrists, arms) while maintaining their fine motor control (fingers). A project from Carnegie Melon University was created using a Palm Pilot Personal Digital Assistant to access the internet, e-mail, and even play computer games [22]. The integration of mobile devices for assisting those with disabilities or the elderly has wide-ranging implications. Personal communication is one field, where smartphones can allow those with severe motor issues (both gross and fine) to communicate when they may not be able to use a standard telephone. Another application aspect would be with security, in order to give those with disabilities a sense of safety in case of illness, accidents, or other incidents. Finally, mobile devices can be integrated to allow people with impairments, whether they are from age or disability, to have a sense of autonomy, which springs from the previous subject areas mentioned [23].

1.2.2 AndSURF

A Master’s Thesis from Lund University in Sweden focused on mobile computer vision on the Android platform, where the researchers implemented numerous computer vision
algorithms. Notably, a version of the Speeded Up Robust Feature, or SURF, feature extraction algorithm was implemented in various forms on an early Android device [2].

Three versions of the SURF algorithm were implemented on the Android platform, and their performance was compared. The first method, called AndSURF (Android SURF) was written as an optimized version of a previously-created Java-optimized version of SURF called “JSURF,” that was developed for the same thesis as an optimized Java port of the open-source “OpenSurf”, version 1.2.1. Numerous Java-based optimizations were implemented in JSURF, and further Android-based optimizations were considered to “overcome the algorithmic flaws and choices of the Android platform” [2].

Additionally, a native-code version of SURF, called “Native AndSURF,” was implemented using the Java Native Interface and C programming language. The resulting implementation was found to be marginally faster than the original PC implementation and notably faster on the Android implementation [2].

The performance increase with the use of the JNI can become significant for many different applications. The results found in [2] are reinforced by results presented for Sobel edge detection, presented in Appendix A. It can be seen that the implementation completed with the Android native libraries is generally over fifty times faster than its java-implemented Android counterpart. Therefore, it stands to reason that for mathematically complex portions of code, there is a tangible benefit to creating native libraries with the Android NDK and JNI.
1.2.3 Bartendroid

Another application that came from the thesis out of Lund University was a system that would identify multiple types of liquor and then suggest drink combinations based upon a picture taken [2]. This application is mentioned because it uses a much different paradigm than the previous SURF implementation. Since the drink identification process requires much more feature extraction, it is unsuitable for implementation on a mobile device due to the execution time required. Additionally, it would be unsuitable to have a large database of drink recipes and a liquor recognition database on the phone itself, in both terms of search time and amount of data present in the database [2].

With those considerations, a cloud computing based system was created with the aid of the Amazon Elastic Compute Cloud, or EC2. The general setup of the system is shown in Figure 6, with the local mobile device portions of the system is shown in the left of Figure 6, and the cloud based portion shown in the right of Figure 6 [2].
This system is discussed primarily to illustrate the potential of applications with mobile devices. A cloud-based approach to solving problems with high computation or large database requirements (or both) is quite relevant and useful for systems with limited resources, such as mobile devices. While the current state of wireless data networks, both wireless fidelity (Wi-Fi) and cellular data (EDGE, GRPS, 3G, and soon 4G), the systems are generally not fast enough for real-time applications. In the case of Bartendroid, the system took approximately ten seconds from the taking of the picture to receiving of results across a Wi-Fi network [2]. However, as next-generation cellular systems are created and data throughput of wireless radios increases, communication time will decrease. Additionally, as the processing power of the devices increases, more computationally intensive tasks can be completed locally on the mobile device.
1.3 Previous work on Low Vision

Low vision can refer to many different types of conditions leading to a decrease in an individual’s sight. Three pieces of work have been chosen for further review, due to their general nature [24] and their application to mobile devices [12, 25].

1.3.1. Visual Impairment and Disability

Studies have found that general visual impairment has a strong association with functional dependence, especially with those 65 years and older. Generally, vision impairment appears to be directly associated with losing of self-sufficiency. A survey completed on 222 subjects age 65-90 found that 49% of the individuals surveyed had some sort of ocular disorder (Cataracts, Macular Degeneration, Glaucoma, Trauma, Amblyopia, Diabetic Retinopathy, and “other”) [24].

From further questioning, 29% of subjects reported difficulty with basic activities, such as grooming and bathing, and 19% of subjects then reported some difficulty with “instrumental activities” such as shopping or managing money. Finally, 32% of subjects reported difficulty with some mobility tasks [24]. Overall, the results from this study show that there is a notable link between those with visual impairment and their ability to interact with the world around them.

1.3.2 Augmented Reality and Visual Search

Numerous works has been completed with individuals with low vision conditions. One of these works used an augmented reality, or AR, system to aid individuals with tunnel vision. The system was designed to use visual or auditory cues to help and individuals find a target in scene.
The system uses a combination of buzzers and a target contour to direct a user with tunnel vision to find the designated target in the system. The target contour is an edge-detected version of the target to find which increases in size as a user’s visual search approaches the actual target, until the contour is directly overlaid on the target. Both of these pieces of the system make up the augmented reality system, since these are implemented on top of the normal environment in order to convey information to the user [12].

For testing the system, three cases were considered: visual cues (target contours) only, audio cues only, and finally with no cues. Users with the device and one of the cues enabled exhibited a significantly reduced search time (28% to 74%) compared with users without the cues [12].

The system used a head mounted display, or HMD, to convey information to the user. The display was monocular, designed to only use one eye for the display of data. Additionally, a camera was used to capture data from the environment. These pieces were connected to a purpose-built edge detection processor, which took input from the camera and displayed the processed data for the given target on the HMD [12].

Overall, the lesson taken from this system is that a small amount of information can aid people significantly if the information is presented in a useful manner. Additionally, the system shows that using a head-mounted display can be a suitable way to aid individuals with low vision conditions.

1.3.3 ViSAR

Scherlen and Gautier have completed work with patients with centrally-located scotomas [26] and have taken further into a system a concept called Visual Signal Adaptive
Restitution, or ViSAR. Their system is built around the idea of taking information that would normally be obscured by a scotoma and moving it around the scotoma for the individual to read. The system makes use of a head-mounted display and an eye-tracking system to detect where a user’s eyes are looking and therefore, what information their scotoma is obscuring. Then the system moves the information around the scotoma for viewing.

An application for this system, in addition to text, would be face recognition and relocation. This is similar to the goals of the system for this thesis, however the ViSAR system completes the goal in a relatively different way. While the system in this case relocates the face around a scotoma, the system proposed for this thesis redraws the face in a new area in the user’s preferred retinal locus [25].

The ViSAR system is conceptually similar to the system developed in this thesis, but the ViSAR system has not been developed further than its original concept. Therefore, the system created for this thesis does parallel some other work, although the implementations and some design choices are very different.

1.4 Cloud Computing

Cloud computing is a very popular recent topic in the field of computer vision as well as in many other fields with computationally-heavy tasks. As defined in [27], cloud computing is “both the applications delivered as services over the Internet and the hardware and systems software in the data centers that provide those services.” Both Amazon and Google corporations offer cloud computing services named the Amazon Elastic Compute Cloud, or EC2, and Google’s AppEngine [27]. At a non-corporation level, clouds can be seen as a relatively powerful computer, or perhaps a computing
cluster, often referred to as a “micro-cloud.” For mobile phones and other devices, wireless communication can be completed with a cloud over data networks such as third-generation (3G) cellular networks [28].

The ‘Looktel’ computer-aided visual assistance program is an example of a system created for low vision assistance that integrates a cloud-based approach for object recognition. In the case here, a mobile device is used as a relatively ‘dumb’ piece of the system, used for taking images, displaying results, and providing a user interface for the system. The processing of the system is completed on a remote system (a micro-cloud), which receives images from the phone and returns the processed results. This approach is conceptually simple and makes use of both the strong points of a mobile device (touchscreen, camera, cellular radios) and the strong points of a cloud-based system (processing power, system resources, etc). This system can get results of 4-14 frames per second over a 3G network with a round-trip delay time of approximately 100-300ms [28].

Another system using both a mobile phone and cloud-system is described in [29]. Instead of the mobile phone acting as a mostly dumb interface for this system, the phone captures an image and then completes feature extraction and feature compression on the capture image. This system was designed to identify compact disc, DVD, and Book covers based on features extracted CHoG (Compressed Histogram of Gradients) descriptors. In lieu of sending an entire image wirelessly over a data network, the extracted and compressed features were sent, which greatly decreased the required transmission time [29]. Overall, the two systems mentioned here represent the majority of approaches in use for cloud-based systems with computer vision algorithms and mobile devices; to keep the processing on a phone as little as possible and off-load the
heavy computational tasks to a remote server, and receive and display results on the mobile device.

With the prevalence and availability of cloud-based systems, the algorithms used for face detection and tracking are evaluated with a theoretical cloud-based approach in order to compare and contrast the performance of a mobile-device only and cloud-based system for the six algorithms tested.
Chapter 2  Supporting Work

2.1  Face Detection

For the purpose of this research, three different face detection methods were implemented on the Android platform. The Viola-Jones method is a commonly used as a benchmark in the face recognition community, as it is very accurate and its performance serves as a baseline for the other algorithms. The second method is Support Vector Machines, and was chosen to determine how this commonly used classifier along with a sliding-window approach would perform on the Android platform. The last method, by Pai et al, was chosen to evaluate how a color-based detection algorithm would perform on the system.

2.1.1 Viola-Jones

The Viola-Jones method of face detection was created by Paul Viola and Michael Jones and was originally in 2001 [1]. This method uses weak classifiers based on Haar features, such as the ones shown in Figure 7. To use the weak classifiers, each feature is moved across the given image, and the mean area of the white region is subtracted from the mean area of the black. More current methods additionally use ‘extended’ weak classifiers, such as shown in Figure 8 [30].

![Figure 7 - Original Viola-Jones Weak Classifiers](1)
Originally, a training set is created with tens of thousands of faces, with each face having a resolution of 24 x 24 pixels. All of the weak classifiers are then run across each training image, with all possible scales and locations for the classifiers used. The weak classifiers are called ‘Haar-like’ due to the fact that they resemble Haar basis functions, originally used by Papageorgiou et al [31]. In lieu of running the weak classifiers on a basic image, each image is transformed into an ‘integral’ image, originally proposed in [1]. This integral image is created by setting each pixel value in an image to the sum of the pixels above and to the left of the image, including the original pixel value. This allows feature extraction to be completed much more quickly than with the base images, since any Haar classifier can be computed from simple lookup table entries in the integral image. The example given by Viola-Jones is shown in Figure 9 [1].
As an example of the use of an integral image, the area B can be calculated from the area of rectangles A and B [1]. In Figure 9, to calculate a Haar classifier (white area on area A, black area on area B) the equations below would be used, where the notation ‘ii’ refers to ‘integral image’.

\[
\text{Haar Classifier} = \text{Area White} - \text{Area Black} \tag{1}
\]

\[
\text{Haar Classifier} = ii(1) - [ii(2) - ii(1)] \tag{2}
\]

\[
\text{Haar Classifier} = 2ii(2) - ii(2) \tag{3}
\]

Extrapolated from the above, any classifier in the image can be calculated from a relatively small amount of linear combinations of the integral image values.

With the sheer number of weak classifiers possible for training (180,000), which is far more than the number of pixels in a 24 x 24 image (576), a different technique must be implemented to reduce the computational cost of the training. Instead of using all of the features available, a learning classifier is built to select between faces and non-faces. When used together, the cascaded classifiers become a ‘strong’ classifier [1]. This is completed through a process called “Boosting.” There are a variety of boosting methods available, but the one most commonly used is called AdaBoost [1,32]. When training, an iterative process updates each of the many (180,000) weights through multiple rounds, and during each round selects which feature gives the best distinction between faces and non-faces. Viola-Jones only used the top 6,061 features for its implementation [1,32]. The top two features are shown in Figure 10.
Finally, once the features to use are found through the boosting algorithm, a cascade of progressively more complex features is used to determine if a face is present or not. Statistically, it is unlikely for any given window in any given frame to contain a face. As each window in each frame is searched for a face we would like to quickly determine the window as being a non-face area and move on to the next location. As such, a few simple features are used to determine if the location contains a face or non-face. If the classifier even thinks there might be a face, the window is then run against a second stage of more complex classifiers. A simplified example of the cascade process is shown in Figure 11.

This process continues until a face is deemed found from reaching the end of the cascade, which is 38 stages in the original Viola-Jones process, or one of the classifier stages fails, at which point the window moves to the next position in the image. The original Viola-Jones algorithm, with 10 cascades with 20 features per classifier, was
found to be nearly as accurate as a single 200 feature classifier with an almost 10x speedup [1].

2.1.3 Support Vector Machines

The second face detection approach that was investigated involved the use of Support Vector Machines (SVMs). Support Vector Machines were originally created by Cortes and Vapnik in 1995 [7] and are used for classification of data primarily into two groups (e.g. faces and non-faces). With this, an SVM needs to be trained before its use, with examples of both classes (faces/non-faces) [33]. These training sets can be of any size, but generally more samples of each lead to better results.

Figure 12 - SVM Two-Dimensional Example [7]

For SVM to work correctly there must be a clear distinction in the training data between the two classes. A simplistic example of this is shown in Figure 12. In the above example, the two sets of data are separable. The support vectors are represented
by the three grey squares, so this dataset has three support vectors. These vectors allow for the largest margin of separation between the two classes of data. The training of the support vector machine is what creates the margins and the hyperplane, which determines the way by which new data is classified. When a new data point is tested with the trained SVM, its location with regards to the hyperplane determines the data classification [7].

Support vector machine (SVM) classifier is a machine learning algorithm that determines the hyperplane that separates the training data with the maximum possible margin [33]. Formally, let the input training data be represented by a set of m labeled examples $x_i \in \mathbb{R}^d$, $i = \{1 \ldots m\}$ and their associated labels $y_i \in \{+1,-1\}$. The optimal hyperplane is obtained by maximizing a convex quadratic programming problem given by the following equation:

$$W(a) = \max \sum_{i=1}^{m} a_i - \frac{1}{2} \sum_{i,j=1}^{m} a_i a_j y_i y_j \langle x_i, x_j \rangle$$  \hspace{1cm} (4)

subject to $\sum_i a_i y_i = 0$ and $0 \leq a_i \leq C$ for $i = \{1 \ldots m\}$ where $a_i$ are the Lagrange multipliers of the optimization. The training samples associated with the non-zero Lagrange multiplies are called support vectors. A test vector $\tilde{x}$ is classified according to the decision function given by:

$$D(\tilde{x}) = w \varphi(\tilde{x}) + b = \sum_{i=1}^{m} a_i y_i \langle x_i, \tilde{x} \rangle + b$$  \hspace{1cm} (5)

In cases where the data cannot be linearly separated, the kernel trick is employed to transform the input data into a higher dimensional space where the transformed data is linearly separable. In this case the inner products in (4) and (5) are substituted by inner products in a higher dimensional space $\langle \Phi(x_i), \Phi(x_j) \rangle$. Unfortunately, evaluating all the
inner products in the $\Phi$ space requires great computational effort. A solution to the problem is given by Mercer’s theorem, where the inner products are given by a kernel function such that $K(x_i, x_j) = \langle \Phi(x_i), \Phi(x_j) \rangle$. The modified optimization function is then given by:

$$W(a) = \max \sum_{i=1}^{m} a_i - \frac{1}{2} \sum_{i,j=1}^{m} a_i a_j y_i y_j K(x_i, x_j)$$

(6)

And the decision function is given by:

$$D(\vec{x}) = w\varphi(\vec{x}) + b = \sum_{i=1}^{m} a_i y_i K(x_i, \vec{x}) + b$$

(7)

There are many different implementations of kernels for SVM, and four of the basic ones are shown below in Equations (8) through (11). Please note, in the equations below, $\gamma$, $r$, and $d$ are all kernel parameters that can be tuned for the data in use.

**Linear:** $K(x_i, x_h) = x_i^T x_j$

(8)

**Polynomial:** $K(x_i, x_h) = (\gamma x_i^T x_j + r)^d, \gamma > 0$

(9)

**Radial Basis Function (RBF):** $K(x_i, x_h) = \exp(-\gamma \|x_i - x_j\|^2), \gamma > 0$

(10)

**Sigmoid:** $K(x_i, x_h) = \tanh(\gamma x_i^T x_j + r)$

(11)

The methods range in computational complexity and may vary in accuracy with certain datasets, making the choice of kernel dependent on the platform as well as on the training data.

### 2.1.2 Pai et al

The last face detection algorithm investigated for the purposes of this research was a color-based algorithm introduced in 2006. This algorithm uses the color of human skin
as well as the geometry of the human face to detect faces very quickly in a scene [6]. A flowchart of the algorithm is shown below in Figure 13.

![Figure 13 - Pai et al Algorithm Flowchart [6]](image)

The first stage of the process, lighting compensation, is accomplished by computing the average luminosity of the image. This is completed with Equations (12) and (13), where \( Y \) refers to the luminance plane of the image while \( R, G, \) and \( B \) refer to the red, green, and blue color planes of an image respectively. The subscripts \( i \) and \( j \) refer to the two-dimensional coordinates of the pixels in each plane [6].

\[
(12)
\]

\[
(13)
\]

Depending on the value of \( Y_{\text{avg}} \), a compensated image is created by the following parameters [6]:

27
Finally, with the compensated color planes, the chrominance image can be found through the use of Equation (17). Blue is left with its original value since it does not factor much into chrominance. The chrominance image is used because it is a color space that represents human skin in a consistent fashion [6].

\[
\begin{align*}
R'_{ij} &= (R_{ij})^\tau \\
G'_{ij} &= (G_{ij})^\tau \\
\tau &= \begin{cases} 
1.4, & Y_{\text{avg}} < 64 \\
0.6, & Y_{\text{avg}} > 192 \\
1, & \text{Otherwise}
\end{cases}
\end{align*}
\]

(14) (15) (16)

With the chrominance image found, the limits for skin can be set, with any point in the image between pixel values of 10 and 45 representing human skin. Next, since a simple binary image is prone to noise in this situation, a low-pass filter of size 5 x 5 is run through the image in order to quickly remove the high-frequency noise present. Finally, with the blobs now shaped, bounding boxes can be found for the face-colored blobs found in the image [6].

With the boxes found now of potential faces, numerous tests are run to find the valid faces present in the image. The quickest test run compares the width and height of the face candidates. Generally, a human face has a height-width ratio of 0.8 to 1.5, so if a face candidate falls outside of this range, it can be determined with relative certainty that the current candidate is not a real human face [6].

The last two tests run on face candidates are for the detection of known human face geometry: the mouth and the eyes. For mouth detection, the \( \theta \) for each pixel in the
candidate box is calculated, as shown in Equation (18), and then it is determined whether or not to be a mouth pixel by Equation (19) [6].

\[
\theta = \cos^{-1}\left( \frac{0.5(2 * R' - G' - B)}{\sqrt{(R' - G')^2 + (R' - B)(G' - B)}} \right)
\]  \hspace{1cm} (18)

\[
M_{pq} = \begin{cases} 
0, & \theta < 90 \\
1, & \text{Otherwise}
\end{cases}
\]  \hspace{1cm} (19)

In Equation (19), a 0 denotes a mouth pixel while a 1 denotes a non-mouth pixel. From this, a histogram of the mouth pixels is found, and the location of the highest number of mouth pixels found in the horizontal dimension is noted.

Once the mouth is found, the area above the mouth can be searched for the eyes of the potential face. The eyes are determined by a region of darker chrominance above the mouth region; a chrominance value of between 65 and 80 denotes an eye pixel for this algorithm. The test in this case is simpler than prior; a threshold value of eye pixels is defined through experimentation. If the eye pixels found are within the defined threshold, the candidate is confirmed to be a face, and the process continues to the next candidate [6].

\section*{2.2 Object Tracking}

As with face detection, three different algorithms were chosen for use with the purpose of object tracking. The first method, Template Matching, was chosen due to its relatively simple operation and implementation. The second, Lucas-Kanade, was chosen because of its application of using feature points to track throughout frames, while CAMShift was chosen because it is color-based. Overall, the three algorithms were chosen because of their different approaches, which make it worthwhile to examine how each type performed on a mobile device.
2.2.1 Template Matching

The implementation of the template matching algorithm begins by finding an initial template to use. This template is taken from the face detection process, i.e. a face is detected with one of the previous face detection methods, and the face region serves as the first template in the image. When a new frame is captured from the camera, the surrounding area is searched for a match to the template through correlation of the template with the search region. A normalized square difference is run with the template and image, as shown below in Equations (20) through (22), where the image under test, called I, and the template (the face), called T.

\[
x' = 0 \ldots width - 1
\]
\[
y' = 0 \ldots height - 1
\]
\[
R(x,y) = \frac{\sum_{x',y'}(T(x',y') - I(x + x', y + t'))^2}{\sqrt{\sum_{x',y'}T(x',y')^2 \cdot \sum_{x',y'}I(x + x', y + y')^2}}
\]

This method returns an intensity image R, where the global minimum represents the best match to the given template. The object being tracked, therefore, is located at the global minimum. This location information is then used to determine the new face returned to the user. Finally, this new tracked face is set as the current template, and is used in the subsequent frame of the video sequence. This process continues until detection is run again. Overall, this is a relatively simple method and is chosen as a baseline for gauging the performance of the other tracking methods investigated.

2.2.2 Lucas-Kanade

The Lucas-Kanade method of tracking uses a concept called ‘Optical Flow’ to track objects throughout a scene. Given two images, the motion between them can be found by
examining image pixel intensities using the difference between corresponding points, such as corners. The technique used most often for optical flow tracking is called ‘sparse’, which refers to using a subset of points in an image for tracking, while dense optical flow tracking refers to tracking every pixel in a scene, which can be computationally intensive [4]. Additionally, in order find pairs of features, a robust feature detector is used, such as the Harris Corner Detector [34].

With optical flow, a motion vector can be found from the difference of a location of a point in two images. The Lucas-Kanade optical flow takes this idea and adds the assumption that a location patch of pixels has spatial coherence. That means that pixels around a feature pixel tend to travel in the same direction, since it is very likely that the pixels belong to the same object [4].

The Lucas-Kanade algorithm is a differential method for the determination of optical flow. This flow can be written as shown in Equation (23).

\[ I(x, y, t) = I(x + \delta x, y + \delta y, t + \delta t) \]  
\[ (23) \]

In the above, \( I(x, y, t) \) is a representation of a pixel value at a specific \( x \) and \( y \) coordinate at a specific time \( t \). Additionally, Lucas-Kanade make another assumption where the pixel values around a chosen feature point will also remain constant. This leads to Equations (24) and (25) below.

\[ I(x + i, y + j, t) = I(x + i + \delta x, y + j + \delta y, t + \delta t) \]  
\[ (24) \]
\[ i, j \in (-w..w, -w..w) \]  
\[ (25) \]

In the above, the parameter \( w \) refers to the window size used for the pixel neighborhood around a given tracking feature. Additionally, Luca-Kanade strengthened their technique by increasing the weight given to pixels close to the central pixel. This
compensates for occurrences when parts of the neighborhood of pixels used do not all
belong to one object [4].

Overall, the Lucas-Kanade method of tracking provides a relatively robust
method of tracking moving objects in a scene. While its individual parts are more
computationally intensive than the other two methods in use, it is a relatively adjustable
tracking method; the amount of computations completed can be tuned to an optimal point
on a mobile platform. Due to this, it makes a suitable algorithm for test on a mobile
platform.

2.2.3 CAMShift
CAMShift, or continuously adaptive mean-shift, is an algorithm for tracking an object
through a scene based on color. The base of the algorithm, ‘mean shift’, works on basic
probability distributions based on the colors contained in an image [35]. The probability
distributions of color change over time (from frame to frame in a video sequence), and
the algorithm must compensate for these changes, hence the ‘continuously adaptive’
addition to mean-shift. Overall, CAMShift tracks the x and y coordinates of a given
tracked object, and also tracks the color probability distribution of the object. This area
can be seen as proportional to depth in the image, represented by the Z axis. The general
flow of the algorithm is shown in Figure 14 [5].
For explanation purposes, the images shown in Figure 15 will be used to explain the flow of the algorithm.

Figure 15 – CAMShift Image Sequence

In 20.a, the initial search window size and location is chosen. This initial region will be given by the face-detection stage of the system. Next, the calculation region of
the image is set at the middle of the initial window, but slightly larger than the original search window (20.b). From this new calculation region, a color histogram look-up of the calculation region is taken from the HSV transformed image. From this, a color probability distribution of the calculation region can be found [5].

Finally, an iterative process is run to successfully track the object. The center of mass contained within the search window is found, and the search window is then centered at this center of mass (20.c). The area in the search window is calculated, and if the mass converges, the x, y, z, and roll of the tracked object is reported. Finally, the x and y coordinates reported are used to set a new search window as shown in Equation (26) (20.d), and the process is then repeated for the next image submitted to the tracker [5].

\[
\text{New Search Window} = 2 \times \text{area}^{1/2} 
\]

(26)

Overall, CAMShift provides a very fast tracking solution that uses few resources on a system. These criteria make it ideal for test and usage on a mobile platform, especially for the application considered with the low vision assistance system.

With the algorithms chosen for use, prototype systems could be developed to act as proofs of concept for the algorithm implementations, and to determine if the algorithms chosen are viable candidates for use on a mobile device.
Chapter 3  Prototype Systems

3.1 Prototype System Design

3.1.1 Netbook System

The initial prototype for the low vision assistance system was completed on a netbook computer. This prototype was created with the aid of OpenCV, and was created to simulate a scotoma in various areas of a view and redraw a single detected face in a user-defined region. A screenshot of this system is shown below in Figure 16.

![Initial System Prototype](image)

Figure 16 - Initial System Prototype

In the scene above, three boxes are used during system testing. The outermost box denotes the region of the image that is being searched for faces. This prototype uses the classical Viola-Jones method of Face Detection [1]. The middle box denotes the region of the image that is being searched for a template match. This is only created after an initial face has been found with Viola-Jones, and is smaller than the detection region for two reasons: a smaller region will decrease processing time for a template match and a subject being viewed is unlikely to move beyond this box in consecutive frames.
Finally, the central box denotes what the algorithm detects as a face in the image. This face region is redrawn in the defined preferred retinal loci of the scene, denoted in the top left of Figure 16.

### 3.1.2 Android System

A second prototype was created on the Android operating system. While the netbook prototype used OpenCV for the majority of the program (video feed, graphical user interface, underlying algorithms), the Android prototype only used OpenCV for the face detection; the remainder of the interface was written using the Android Java interface.

This original prototype only implements face detection. Face tracking was implemented with the final revision of the system completed for this thesis. The prototype created in Android also has the capability to simulate a scotoma, and the location of the preferred retina loci can be manually changed by the user.

A similar prototype was written in Android using only Java and native Android libraries. The Android application programming interface, or API, contains a native face detection interface, called 'FaceDetector'. This interface uses what is referred to as the Neven method of face detection, created by Hartmut Neven, who is currently with the Google Corporation. Little public documentation exists on the operation of this algorithm, and the Neven method was not used in further work; it was only used as an initial test of the functionality of a system. As with the previous Android-only prototype, this system only implements face detection, with tracking implemented in the next iteration of the system.
3.1.3 Hybrid System

A third and final prototype was created as a hybrid system between the two previous prototypes. This system was created to make use of the strengths of both the laptop computer and the Android phone. A flowchart of the operation of this hybrid system is shown in Figure 17. A laptop computer is used for image capture as well as the processing for both face detection and tracking. The laptop also serves as the image processor, changing the image as the parameters of the system dictate. The Android device is used to display the processed image from a phone and serves as a controller for the system. From user input, the phone sends commands to the laptop to change parameters, such as the location where a detected face should be redrawn and the size or zoom of the redrawn face. Communication between the two systems is completed over the Bluetooth wireless protocol. With this implementation, the phone controls and its display can be separated from the laptop system. While this system does allow for the better parts of each system to be used, it also adds a complicating and potentially limiting factor with the Bluetooth wireless protocol requirement.

![Figure 17 – Hybrid System Operation Flowchart](image-url)
3.2 Prototype performance

3.2.1 Android System

For basic initial timing comparisons based on execution speed, the two Android-based prototypes were compared with two different image sizes. Due to past limitations in the Android Operating System version 2.1, images from a video feed (called the video preview) are supplied only in a YUV420 image format. While this format is suitable for compressing images very well, it is unsuitable for use with Viola-Jones or the Neven face detection algorithm. Because of this, the image must first be converted into an RGB565 format for use with the face detection algorithms. This conversion is unfortunately quite time-consuming and adds additional time to the execution of the system. The timing found for this conversion is shown for two different image sizes below in Table 1. Each image size test was run thirteen times to obtain an average of the data.

<table>
<thead>
<tr>
<th>Image Resolution</th>
<th>Sample</th>
<th>320 x 240</th>
<th>848x480</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>149</td>
<td>805</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>151</td>
<td>803</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>149</td>
<td>802</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>150</td>
<td>796</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>152</td>
<td>803</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>149</td>
<td>804</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>149</td>
<td>801</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>149</td>
<td>798</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>151</td>
<td>802</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>152</td>
<td>807</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>148</td>
<td>803</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>149</td>
<td>809</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>150</td>
<td>802</td>
</tr>
<tr>
<td></td>
<td>Average=</td>
<td>149.85</td>
<td>Average=</td>
</tr>
<tr>
<td></td>
<td></td>
<td>802.69</td>
<td></td>
</tr>
</tbody>
</table>

Table 1 - YUV420 to RGB565 Conversion Timings (ms)
The two image sizes were chosen from the supported resolutions of the Android camera on the phone in use. 848x480 is the default size for the camera preview which was used, and 320x240 is a resolution that is much smaller than the default but still provides a suitable viewing of a scene. As expected, the smaller resolution leads to a much shorter conversion time.

With advancements in the Android OS, this conversion time has become much lower than prior conversions. In the current release of the Android operating system (version 2.2), the capability to retrieve a live video sequence from the camera as JPEG images is implemented, and so is the ability to quickly and natively transform the YUV type image into a RGB type image. This version of the Android OS has already been released to many mobile devices, the Motorola Droid used for this research being one of these. Timings from this new conversion method are shown below in Table 2.

<table>
<thead>
<tr>
<th>Image Resolution</th>
<th>Sample</th>
<th>320 x 240</th>
<th>848 x 480</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>19</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>18</td>
<td>85</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>82</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>17</td>
<td>83</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>17</td>
<td>83</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>19</td>
<td>82</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>17</td>
<td>83</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>17</td>
<td>84</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>17</td>
<td>82</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>19</td>
<td>82</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>18</td>
<td>84</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>17</td>
<td>82</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>17</td>
<td>83</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Average=</th>
<th>Average=</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.69</td>
<td>83.15</td>
</tr>
</tbody>
</table>

*Table 2 - YUV420 to RGB565 Conversion Timings, Android 2.2 (ms)*
Finally, both image sizes were tested with the two face detection algorithms in use. The tests were completed thirteen times for each method, and the results are shown below in Table 3 for Viola-Jones and Table 4 for the Neven face detection algorithms.

<table>
<thead>
<tr>
<th>Image Resolution</th>
<th>Sample</th>
<th>320x240</th>
<th>848x480</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>278</td>
<td>1628</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>276</td>
<td>1622</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>292</td>
<td>1527</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>294</td>
<td>1519</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>296</td>
<td>1481</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>293</td>
<td>1484</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>293</td>
<td>1487</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>296</td>
<td>1488</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>296</td>
<td>1544</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>293</td>
<td>1550</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>296</td>
<td>1552</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>293</td>
<td>1551</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>297</td>
<td>1538</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>291.77</strong></td>
<td><strong>1536.23</strong></td>
<td></td>
</tr>
</tbody>
</table>

*Table 3 - Viola-Jones Face Detection Timings, Android (ms)*

<table>
<thead>
<tr>
<th>Image Resolution</th>
<th>Sample</th>
<th>320x240</th>
<th>848x480</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>271</td>
<td>1349</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>262</td>
<td>1311</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>268</td>
<td>1308</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>266</td>
<td>1301</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>266</td>
<td>1309</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>264</td>
<td>1306</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>271</td>
<td>1302</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>211</td>
<td>1303</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>269</td>
<td>1297</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>279</td>
<td>1299</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>259</td>
<td>1300</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>263</td>
<td>1299</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>261</td>
<td>1301</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>262.31</strong></td>
<td><strong>1306.54</strong></td>
<td></td>
</tr>
</tbody>
</table>

*Table 4 - Neven Face Detection Timings, Android (ms)*
As the above tests show, the timings for the Neven face detection algorithms are slightly faster than those for the Viola-Jones algorithm. This could be seen as a function of the operation of Viola-Jones in the context of the Android OS, requiring translation to an OpenCV image type and then communication with the Android Native Library system, while the Neven implementation uses faster calls to the Android system implemented by Google engineers. This approach also hides a large amount of the complexity of the system, allowing implementation of systems to be primarily concerned with higher-level issues, instead of low-level implementation issues. However, as the average difference between the two methods is quite small (< 100 ms for 320 x 240 images), this system implementation with Viola-Jones will serve as a useful benchmark for future implementations. These results resulted in the decision to use only an image size of 320 x 240 pixels for future measurements, since this would be the size used in the real system.

For resource usage, three metrics were examined: Processor usage, heap size, and the percentage of the heap used. The ‘FroYo’ version of the Android operating system was released during the time of these tests, so usage tests were run for both the original prototype and the prototype modified to work with the new version of the operating system. Table 5 below shows the resource usage from the original prototype system.
### Table 5 - Original Prototype Resource Usage

<table>
<thead>
<tr>
<th>Test</th>
<th>Processor Usage (%)</th>
<th>Memory(% of Heap)</th>
<th>Heap Size: (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>70</td>
<td>72.49</td>
<td>3.57</td>
</tr>
<tr>
<td>2</td>
<td>66</td>
<td>74.48</td>
<td>3.883</td>
</tr>
<tr>
<td>3</td>
<td>61</td>
<td>66.94</td>
<td>3.57</td>
</tr>
<tr>
<td>4</td>
<td>48</td>
<td>66.94</td>
<td>3.57</td>
</tr>
<tr>
<td>5</td>
<td>62</td>
<td>68.5</td>
<td>3.883</td>
</tr>
<tr>
<td>6</td>
<td>59</td>
<td>66.94</td>
<td>3.57</td>
</tr>
<tr>
<td>7</td>
<td>55</td>
<td>66.94</td>
<td>3.57</td>
</tr>
<tr>
<td>8</td>
<td>78</td>
<td>68.35</td>
<td>3.883</td>
</tr>
<tr>
<td>9</td>
<td>53</td>
<td>68.15</td>
<td>3.883</td>
</tr>
<tr>
<td>10</td>
<td>70</td>
<td>79.43</td>
<td>3.883</td>
</tr>
<tr>
<td>11</td>
<td>53</td>
<td>67.6</td>
<td>3.883</td>
</tr>
<tr>
<td>12</td>
<td>77</td>
<td>67.58</td>
<td>3.57</td>
</tr>
<tr>
<td>13</td>
<td>76</td>
<td>70.19</td>
<td>3.57</td>
</tr>
<tr>
<td>14</td>
<td>67</td>
<td>70.68</td>
<td>3.57</td>
</tr>
<tr>
<td>Average =</td>
<td>63.93</td>
<td>69.66</td>
<td>3.70</td>
</tr>
<tr>
<td>Std. Dev. =</td>
<td>9.65</td>
<td>3.63</td>
<td>0.16</td>
</tr>
</tbody>
</table>

Table 6 shows the resource usage from the system using the new version of the Android operating system.

### Table 6 - Modified Prototype (Android 2.2) Resource Usage

<table>
<thead>
<tr>
<th>Test</th>
<th>Processor Usage (%)</th>
<th>Memory(% of Heap)</th>
<th>Heap Size: (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>55</td>
<td>81.08</td>
<td>6.195</td>
</tr>
<tr>
<td>2</td>
<td>66</td>
<td>55.98</td>
<td>7.758</td>
</tr>
<tr>
<td>3</td>
<td>56</td>
<td>55.98</td>
<td>6.508</td>
</tr>
<tr>
<td>4</td>
<td>49</td>
<td>81.04</td>
<td>7.758</td>
</tr>
<tr>
<td>5</td>
<td>58</td>
<td>82.83</td>
<td>7.758</td>
</tr>
<tr>
<td>6</td>
<td>67</td>
<td>71.17</td>
<td>6.508</td>
</tr>
<tr>
<td>7</td>
<td>47</td>
<td>71.12</td>
<td>7.758</td>
</tr>
<tr>
<td>8</td>
<td>60</td>
<td>69.74</td>
<td>7.758</td>
</tr>
<tr>
<td>9</td>
<td>57</td>
<td>71.16</td>
<td>7.758</td>
</tr>
<tr>
<td>10</td>
<td>54</td>
<td>70.54</td>
<td>7.758</td>
</tr>
<tr>
<td>11</td>
<td>73</td>
<td>49.1</td>
<td>6.508</td>
</tr>
<tr>
<td>12</td>
<td>65</td>
<td>69.3</td>
<td>6.508</td>
</tr>
<tr>
<td>13</td>
<td>62</td>
<td>70.53</td>
<td>7.758</td>
</tr>
<tr>
<td>14</td>
<td>70</td>
<td>70.46</td>
<td>7.758</td>
</tr>
<tr>
<td>Average =</td>
<td>59.93</td>
<td>69.29</td>
<td>7.29</td>
</tr>
<tr>
<td>Std. Dev. =</td>
<td>7.66</td>
<td>9.76</td>
<td>0.66</td>
</tr>
</tbody>
</table>

Table 6 - Modified Prototype (Android 2.2) Resource Usage
As the above results show, the processor use is generally lower with the Android 2.2 version (4 percent lower, on average) while the size of the heap increased by 3.5 megabytes. This can be seen as the conversion method for raw images changing what is the primary resource in use for the system; the previous method being more processor intensive, while the new conversion method is more memory intensive. Since the development phone has ample memory for this application (as well as with speed advantage presented), the conversion method used in Android 2.2 was used for further development.

### 3.2.2 Netbook System

When compared to a mobile phone, a netbook has a very large amount of resources available. The netbook used for initial development and testing was an Asus Eee PC 1005HA, which has the specifications below:

- Ubuntu Linux 10.04 (Lucid Lynx)
- Intel Atom N280 Processor (1.66 GHz)
- No ROM of note
- 1 GB RAM
- 10.1” LED Backlight WSVGA Screen (1024x600)
- 1.3 megapixel color camera

Since the measurements of resource usage would be comparable to those on the Android-based system due to the very different system architectures, only the timing of the algorithms was recorded, as is shown in Table 7.
Table 7 - Viola-Jones Face Detection Timings, 320 x 240, Netbook (s)

<table>
<thead>
<tr>
<th>Test #</th>
<th>Viola-Jones</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.05</td>
</tr>
<tr>
<td>2</td>
<td>0.05</td>
</tr>
<tr>
<td>3</td>
<td>0.06</td>
</tr>
<tr>
<td>4</td>
<td>0.05</td>
</tr>
<tr>
<td>5</td>
<td>0.05</td>
</tr>
<tr>
<td>6</td>
<td>0.05</td>
</tr>
<tr>
<td>7</td>
<td>0.04</td>
</tr>
<tr>
<td>8</td>
<td>0.05</td>
</tr>
<tr>
<td>9</td>
<td>0.05</td>
</tr>
<tr>
<td>10</td>
<td>0.05</td>
</tr>
<tr>
<td>11</td>
<td>0.05</td>
</tr>
<tr>
<td>12</td>
<td>0.05</td>
</tr>
<tr>
<td>13</td>
<td>0.04</td>
</tr>
<tr>
<td>14</td>
<td>0.04</td>
</tr>
<tr>
<td>15</td>
<td>0.06</td>
</tr>
<tr>
<td>Average</td>
<td>0.05</td>
</tr>
</tbody>
</table>

As expected, the Viola-Jones implementation on the Netbook is markedly faster than its implementation on the Android mobile device. However, due to its faster speed and relative low amount of resources when compared to a normal desktop PC, it will serve as a decent test bed and initial development platform for the algorithms.

The netbook could serve as a test bed that would represent the processing capabilities of the next generation of smartphones. Systems with similar specifications as the netbook and form factors similar to a mobile phone, such as the mobile internet devices based on the Intel Atom architecture, are already in development and are nearing the release phase. Therefore, the netbook represents the performance possible on smartphones in the near future.
3.2.3 Hybrid System

As shown in Figure 17, the hybrid system uses both the Android and Netbook systems in tandem. The resource usage on the Android device is markedly different than previous incarnations of the system, as is shown in Table 8.

<table>
<thead>
<tr>
<th>Test</th>
<th>Processor Usage (%)</th>
<th>Memory(% of Heap)</th>
<th>Heap Size: (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>46</td>
<td>76.54</td>
<td>3.008</td>
</tr>
<tr>
<td>2</td>
<td>51</td>
<td>76.53</td>
<td>3.07</td>
</tr>
<tr>
<td>3</td>
<td>50</td>
<td>75</td>
<td>3.07</td>
</tr>
<tr>
<td>4</td>
<td>51</td>
<td>75.01</td>
<td>3.07</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
<td>75.03</td>
<td>3.07</td>
</tr>
<tr>
<td>6</td>
<td>46</td>
<td>75.07</td>
<td>3.008</td>
</tr>
<tr>
<td>7</td>
<td>47</td>
<td>75.07</td>
<td>3.008</td>
</tr>
<tr>
<td>8</td>
<td>48</td>
<td>75.08</td>
<td>3.008</td>
</tr>
<tr>
<td>9</td>
<td>44</td>
<td>75.08</td>
<td>3.07</td>
</tr>
<tr>
<td>10</td>
<td>45</td>
<td>75.08</td>
<td>3.07</td>
</tr>
<tr>
<td>11</td>
<td>45</td>
<td>75.05</td>
<td>3.07</td>
</tr>
<tr>
<td>12</td>
<td>49</td>
<td>75.08</td>
<td>3.008</td>
</tr>
<tr>
<td>13</td>
<td>48</td>
<td>75.11</td>
<td>3.008</td>
</tr>
<tr>
<td>14</td>
<td>49</td>
<td>75.09</td>
<td>3.007</td>
</tr>
<tr>
<td>Average =</td>
<td>47.79</td>
<td>75.27</td>
<td>3.04</td>
</tr>
<tr>
<td>Std. Dev. =</td>
<td>2.33</td>
<td>0.54</td>
<td>0.03</td>
</tr>
</tbody>
</table>

*Table 8 – Hybrid System Resource Usage*

As can be seen, both the processor usage and heap size are smaller than the performance of both versions of the Android-only system, exhibiting a difference of 12.14 % and 0.64 MB respectively when compared to the Android-only system. The increased complexity of the hybrid system makes the system less desirable as a low vision assistance system. However, if these complexity issues are addressed, there is great potential for applications for future development with it.
3.3 **Proposed Improvements**

The proposed improvements for the system focus on changing the algorithms in use. For the purposes of improvements and measurements, the focus will be on the Android systems only in order to measure the performance possible on the mobile platform. Additionally, the new applications will be written to make use of the POSIX-based threading available on the Android platform through Java.

Finally, the use of the Java Native Interface will be examined very closely in order to minimize calls to it. Every call to the JNI involves a context switch [20], and these context switches can add a sizable amount of time to the execution of a program. Therefore, the flow of the programs will be examined in order to minimize usage of the JNI and C++ libraries present in the system. The proposed general system architecture is shown in Figure 18.

![Proposed Improved System Architecture](image_url)
Chapter 4  Algorithm Implementations on Mobile Devices

Implementing the chosen algorithms required some special allotments to be viable on a mobile device. Additionally, parameters used for the algorithms, e.g. SVM, are also given in this section. All of the algorithms were implemented with the aid of OpenCV as well as some native C/C++ functions. Each algorithm was run at seven different resolutions: 176x144, 320x240, 352x288, 640x480, 720x480, 848x480, and 720x576. These resolutions correspond to the available video capture resolutions on the Motorola Droid.

4.1  Face Detection

4.1.1  Viola-Jones

For the Viola-Jones implementation, a boosted Haar cascade was used. While there are many cascades available for the implementation of Viola-Jones, trained for different things such as eyes, fully body, nose, etc., a commonly-used cascade was used (“frontal face alt 2”). This cascade was used after experimentation to determine what cascade gave the best results overall in testing the systems on real subjects. Additionally, in order to speed up the detection, the minimum window for searching the image was 40 x 40 pixels in size, and used a Canny Edge Detector to identify regions of the image that could be rejected due to an abundance (or shortage) of edges. A scale factor of 1.2 was used between scans of the image, i.e. increasing the window size by 20% after each scan. Finally, the Haar cascade used on the machine was stored on a flash memory chip (Micro
SD) available on the device in order to use as little amount of on-board memory as possible on the mobile device.

### 4.1.2 Support Vector Machine

As with Viola-Jones, the support vector machine was implemented with the aid of OpenCV. The SVM was based on a linear kernel, because its computational complexity is lower than that for the other kernels available (such as RBF or sigmoid).

The SVM was trained using the face library published by the Massachusetts Institute of Technology Center for Biological and Computational Learning. This library contains a total of 6,977 images, 2,429 faces and 4,548 non-faces. [36]. The training was completed with a cost parameter of 0.05. The training generated a total of 407 support vectors for the dataset given.

For the actual detection, a sliding window and image pyramid approach was used. An increment of 2 pixels was used for scanning an image, and a 3-level image pyramid was used for processing, with scaling the image to a quarter of its original size, then to half of its original size, and then scanning the image at its original size. As with Viola-Jones, the trained library was stored on a removable flash memory card in order to conserve the on-board memory of the device.

### 4.1.3 Pai et al

Of the methods used, the Pai et al algorithm was the one that changed the most from its theoretical form proposed in [6]. After initial implementations and tests were completed, it was clear that the face candidates found were not detailed enough to locate the mouth and eyes of a subject. Instead of face and eye detection, a simple calculation
of the area of a candidate region was applied and compared to average facial area values obtained through numerous tests of various subjects at various distances. This implementation assumes that a detected face would have area within certain bounds when facilitating communication between individuals.

### 4.2 Face Tracking

#### 4.2.1 Template Matching

Template matching is completed in the manner described in Section 2.2.1. However, to keep processing time to a minimum, only the area surrounding the location of the original detected face is searched for a match to the template. This area is of size 75 x 75 pixels, and is centered at the point where the original face is detected.

#### 4.2.2 Lucas-Kanade

For the Lucas-Kanade tracking, a corner detector using minimum eigenvalues is used to find the features to track [37]. With this, only ten features are detected between the previous and current image. This is to keep computations to a minimum while maintaining the ability to track a moving face throughout a scene.

To compensate for the moving of an object throughout a scene, the differences between the features of the previous and current frames are summed and averaged in both the x and y direction. This resulting value is then used to maintain tracking and keep the original face in the tracking frame. Averaging was introduced to prevent the tracking from exhibiting erratic behavior, which would be present when a very aggressive correction value was used.
4.2.3 CAMShift

The CAMShift tracking method was implemented in an almost identical fashion to its theoretical implementation described in Section 2.2.3. Short of small changes completed to conserve memory usage on the mobile device, there is no notable implementation change present for the implementation of CAMShift. The initial face detected is used as the starting image, and then the algorithm is run as explained.

With the algorithms implemented across the Android, Netbook, and Desktop systems, a battery of tests were run to compare and contrast their performance across systems with regards to execution time and battery life.
Chapter 5 Results

5.1 Algorithm Performance

The results presented in this chapter are taken from a battery of tests run with respect to two aspects of a mobile device: execution time and battery usage. Graphs of the results are presented below for simplicity and comparison of methods. In addition to results taken from the Android device, results were also taken from a netbook and a desktop machine. Additionally, the systems are compared to a cloud-based solution for both execution time and battery usage. The netbook results can serve as an approximation to the performance of future smartphones. The desktop results are included, both as a standalone unit and as a hypothetical cloud-based system where time for image compression and for transmission over a 3G network is added.

For timing comparisons, the Android and Netbook systems were examined with respect only to processing time, since that is all is required to execute the algorithms on the system. For the cloud system, however, time needed to be added for compression and transmission to simulate a proper cloud-based system. A diagram showing this difference is shown in Figure 19.
For the cloud-based system, two values of time were added to the desktop system results. These values are JPEG compression time at a quality factor of 75 and upload time over a 3G network. For the upload times, a study completed by Tan, Lam, and Lau was examined [38] and upload times for three different carriers were averaged to find a representative upload rate for a 3G network. To calculate the upload time, the average sizes of the compressed images tested was used in addition to the upload rate of the 3G network. Finally, since it is assumed that this system will return a very small amount of data representing coordinates from the tracker or detector, the download time is ignored. The results across each resolution for compression and transfer time are shown in Figure 20 and Figure 21.
Figure 20 - Android JPEG Compression Time

Figure 21 - Image Transmission Time across 3G Network
5.1.1 Execution Time

Timing tests were run on an Android device for each of the algorithms implemented. Timing tests were also completed on a netbook for comparison purposes. Detection results for the desktop/server, netbook, and Android systems are shown in Figure 22.

![Figure 22 - Face Detection Timings](image)

As can be seen in the above, the color-based method, Pai et al, is the fastest, followed by Viola-Jones, followed by SVM. This holds true across all three systems. Comparisons of each algorithm by themselves across all three systems, as well as with values from a ‘cloud’ based system are shown below.
As can be seen in Figure 23, the expected values are found, with the desktop system being the fastest, followed by the netbook, followed by the Android system. Additionally, performance of the algorithm is relatively similar at lower resolutions, but time difference between systems becomes notably greater as the resolution of the processed image increases. When comparing the performance of the algorithms, the times all increase in a very linear manner across the different image resolutions. The interesting result occurs between the netbook system and the cloud system. From the data given, it can be seen that the netbook system is quicker than a cloud-based system across all image resolutions. The trend shows that this will hold true across even higher image resolutions. What can be taken this is that from examination, regardless of how
fast the processing of the image is completed on the cloud-based system, the transfer time will overcome the cloud speed advantage when compared to a near-future smartphone approximation. This trend continues with the other two algorithms, Pai et al as shown in Figure 24.

Figure 24 - Pai et al Comparison

Pai et al, as a color-based method, executes much faster than Viola-Jones across all systems. This further strengthens the point shown in Viola-Jones, that the transfer time required for the images across a network overcomes the processing advantage present on a cloud-based system. Finally, a comparison of SVM is shown in Figure 25.
Figure 25 - SVM Comparison

The SVM method, when compared to Pai et al, demonstrates that if the execution time of an algorithm is relatively large compared to the transmission time across a network, there is a great benefit in offloading the algorithm to a cloud-based system. While these points are relatively logical, and interesting result is the comparison of results across all three systems. While each system performs a fair amount differently than the others, the trends show that each algorithm appears to perform in a near-identical manner across resolutions. In each of the three trend lines, the timing performance appears quite similar, almost as if the trend lines are only offset a certain amount. The difference in overall trend is minimal across the systems, even though they vary greatly in resources and processing power.
For object tracking, the same approach was taken, and results across the three systems are shown in Figure 26.

Figure 26 - Object Tracking Timings

The above shows that template matching is by far the most time-consuming method used. Interesting, for smaller resolutions, both CAMShift and Lucas-Kanade provide very similar performance at smaller resolutions, and only begin to slowly diverge at larger resolutions. Additionally, each algorithm’s performance, while markedly different on each platform, still appears to have the same overall trend as noted with face detection methods. For further examination as before, a comparison of the template matching implementations is shown in Figure 27.
The most notable information taken from this graph is that there appears to be little difference between the cloud-based implementation and the netbook system. In this case, as far as execution time is concerned, there is little benefit as far as timing performance is concerned. Additionally, this shows the similarity in trends of the system across the systems tested. A comparison of CAMShift timing performance is shown below in Figure 28.
The performance difference shown for CAMShift is similar to the results previously shown with Pai et al. The algorithm execution time is small enough that all cases, even the slowest Android-based time, actually beat the time for a cloud-based solution. This can be seen as a function of CAMShift being a very fast and algorithmically simple algorithm to implement, and therefore is able to be performed relatively quickly across all platforms. And, as before, the trends appear quite similar across systems. Finally, a comparison for Lucas-Kanade tracking is shown in Figure 29.
The results here diverge a bit from previous comparisons. The netbook and desktop systems exhibit relatively similar performance, and the Android and cloud-based systems also exhibit similar performance. This can be seen as a product of both the algorithm taking a relatively short time to execute as well as the compression time on the Android system forcing the cloud-based system to exhibit similar timings to the Android-only system. For this algorithm, there is little benefit to using a cloud-based solution; the difference between the Android and cloud-based timings is minimal, and the difference between the netbook/near-future smartphone analog is quite notable.

5.1.2 Timing Limitations

From examination of the systems in use, it was found that there was a clear limitation in the measurements of algorithm processing time on the netbook and desktop/cloud
systems. The “time.h” standard library in C was used, and on the two systems in use this only gave timing precision in the tens of milliseconds. While this would not be a problem for the longer-executing algorithms, some algorithms (such as Pai et al and CAMShift) execute fast enough that this lack of precision can result in some inexact timing results for the netbook and desktop/cloud systems. This issue is not present on the Android platform, since the timing construct provided (getElapsedCpuTime()) gives sub-millisecond precision for results.

5.1.3 Battery Usage

The other metric used for comparing the face detection and tracking algorithms involved measuring the battery usage of each algorithm over numerous runs. This was done because a single run of an algorithm would not make a notable enough change to be detected in the energy remaining in a battery. Therefore, the initial and final battery levels (in millivolts) were measured after a set number of runs of an algorithm across the testing resolutions. The results from these the tests run for face detection on both the netbook and Android systems are shown in Figure 30. For reference, the maximum voltage of the netbook battery is approximately 11.5 volts, while the maximum voltage of the Android battery is approximately 3.7 volts. To note, the measurements of the battery level for the netbook system are slightly less precise than those of the Android system, due to the differing precision of the measurement hardware in each system. Additionally, measurements were not run on the battery consumption of the 3G radio of the phone because it was inoperable at the time of testing.
The above trends show that SVM takes the most energy per algorithm execution by a large margin. The three face detection algorithms require a similarly trending amount of energy per execution. This is logically consistent with the previous timing results; the most time consuming method takes the most energy to run, while the quickest method takes the least amount of energy. Additionally, and not as obviously, the screens of the mobile devices were consistently on during these tests, so all of these other active pieces of the mobile are active during the algorithm execution, using battery power. It would be very difficult and not a particularly suitable real-world test to isolate the algorithm by itself for test. However, these factors are unchanging across implementations and testing runs, therefore affecting each algorithm identically and leaving the trends unaffected. Therefore, it can be seen that as an image size increases, the battery usage increases a near linear manner.
Battery usage for the tracking algorithms is shown below. As before, each algorithm was run numerous times, and the before and after voltages on the battery measured. The results are shown below in Figure 31.

![Figure 31 - Object Tracking Battery Usage Comparison (Netbook)](image)

Each system has clear separation between the three algorithms in use. Additionally, each algorithm’s battery usage correlates with its execution time, where CAMShift takes the least time overall to execute and therefore uses the least energy, and template matching takes the longest to execute and consumes the most energy. Additionally, the battery power used for each algorithm appears to increase linearly as the area of the image to be processed increases.

For comparison with a cloud-based system, the algorithms in use make a barely-noticeable impact on the battery from examination of the tests. However, as the images increase in resolution, the battery power needed to process the images on the phone increases to a point where it is beneficial to offload them to a cloud. Overall, the most
battery-intensive pieces of a phone’s hardware are the cellular radios while the CPU energy usage is markedly smaller [39].

5.1.4 Memory Usage

Memory usage was not measured for this research; execution time and battery life were chosen as the focus. Additionally, measuring memory usage properly on a mobile device is not a trivial manner, since to keep resource usage to a minimum the system uses various types of memory pages for execution.

As an example, in the Android platform, each application has a proportional set size, as well as private dirty pages of memory as well as shared dirty pages to be used among applications. Additionally, there are private and dirty pages reserved just for native processing for compiled C/C++ libraries on the system. Therefore, with all of the sharing being completed in the Android, it is difficult to retrieve a consistent and accurate reading of memory usage for a single process.

On the algorithm side, the six algorithms implemented all use a varying degree of memory. For face detection, Viola-Jones and SVM both use a large amount of memory with their processing as well as to store the Haar cascade for Viola-Jones or to store the trained data for SVM. Pai et al, on the other hand, uses very little memory since it consists only of very fast mathematical operations.

For face tracking, a similar story is told for memory usage. Lucas-Kanade and Template Matching both use a very large amount of memory for their processing. Lucas-Kanade processes with an image-pyramid based method, which is memory intensive, and Template Matching must store results for an entire image while it is being processed.
CAMShift, however, uses very little memory since like Pai et al, it executes mostly with fast mathematical operations that take a relatively small amount of memory to complete.

5.2 Android System

5.2.1 Survey on the Android System

During the development of the Android system the National Technical Institute for the Deaf (NTID) held an event called “From OUR Perspective: A community Approach Regarding Students with Vision Loss.” This event served as a forum in which to receive feedback on the initial systems to obtain some real feedback from the system implementations.

Three questions in an anonymous survey were posed to the audience after a presentation describing the system. Although the sample size of individuals was relatively small (12 subjects), the results can be seen as a first step towards obtaining feedback on the Low Vision assistance system. The first question asked was “Do you think this system would be useful for low vision assistance,” and the response to this is shown in Figure 32.

![Figure 32 - Response to Survey Question 1: Do you think this system would be useful for low vision assistance?](image)

Figure 32 - Response to Survey Question 1: Do you think this system would be useful for low vision assistance?
The majority of those surveyed agreed with the statement of the system’s usefulness. This served as a confirmation that the development of the system has potential. The second question asked was “What aspect of the system did you find most interesting or exciting,” and the results are shown in Figure 33.

![Bar Chart](chart.png)

*Figure 33 - Response to Survey Question 2: What aspect of the system did you find most interesting or exciting?*

There was notable interest in the overall concept of the system. The two ‘other’ responses mention finding the new technology aspect of the system interesting as well as finding the system as a whole (“everything”) to be interesting. With the given responses, it can be seen that the system as a whole is the most interesting aspect; not any single portion of the system really stands out against the rest. The third and final question posed to the attendees of the presentation was “How would you rate the system interface,” and the results are shown in Figure 34.
The response to the user interface of the system was consistent from excellent to neutral. This can be seen as both a function of the system being in prototype form as well as a function of the limited time for the presentation given.

Overall, the response taken from the NTID event was that the system would be of great aid, but there was still much future work to be completed. Additionally, six of the twelve subjects surveyed indicated that they would be interested in volunteering for future work on the system, such as testing newer implementations and providing feedback on newer systems.

### 5.2.2 Final System Implementation

For the final implementation of the system, all of the algorithms for face detection and tracking were implemented into one application on the Android platform. The interface of the system is split into two parts: the algorithm selection screen and the actual low vision assistance system. The system is created to allow a user to select which combination of face detection and tracking algorithm will be used for the current run of the system. This was created to examine how different algorithms perform together and
would not be suitable for a release of the system to individuals with Low Vision. Screenshots of the user interface are shown in Figure 35 and Figure 36.

![Figure 35 - Final System Main Menu](image)

![Figure 36 - Final System Face Detection Method Selector](image)

The system provides an easy-to-use interface for selecting the combination of algorithms to use, as well as an intuitive way to move the program onto the next stage of operation: the actual low vision assistance system.

An example image of the operation of the low vision assistance system is shown in Figure 37. A picture was taken externally from the device due to the Android
Development Tools not providing functionality to take screenshots with a camera feed present.

The system defaults to redrawing a face in the top-left corner of the feed at its original size. However, this can be changed with input from the user. Touching the screen at any point will change the coordinates of the redrawn image, and pressing up or down on the D-Pad of the phone will increase or decrease the size of the image respectively. An example of this move and resize is shown in Figure 38.
Overall, the system is designed to modularly use different algorithms for face detection and tracking. The Java portion of the system sends a captured image to the given C++ library, and the library will return the detected or tracked face to the Java interface. This is identical to the flowchart presented previously in Figure 18.
Chapter 6 Conclusion

In this thesis, an initial testing release of a low-vision assistance system was created and tested. Additionally, six algorithms, three for face detection and three for object tracking, were implemented and their performance was compared across numerous metrics.

With the algorithm results, it was shown that the majority of methods could be completed in an acceptable time for a low vision assistance system, without making use of a cloud-based solution for the implementations. Viola-Jones and Pai et al for face detection and Lucas-Kanade and CAMShift all were shown to execute fast enough so that the impact on the user would be minimal. Additionally, the multi-threaded aspect of the programs in use guaranteed an always-running live feed for the user.

The two algorithms that did not execute in a suitable on-device time across all cases were the Support Vector Machine face detection method and template matching object tracking method. These algorithms would need to be offloaded to a remote server to be effective. Additionally, this is evidence that a basic sliding-window approach is ill-suited for use on smartphones, and the large amount of computations required for the convolution of template matching require too much time on a mobile device. While these results will improve as technology improves, the algorithms will still perform poorly when compared to the other methods used in the system.

The feedback received for the system was quite promising, showing that there is a real interest for a system such as this. While much development and testing still needs to be completed in order to create a system that could be used in practice for low vision
assistance, the work completed here will serve as an excellent starting point for further development.

Future work has the opportunity to be taken in a variety of directions. The primary directions would be associated with the algorithm portion of the work or the low vision portion of the work. For algorithms, more methods of face detection and tracking could be benchmarked on the system, in order to find methods that perform better than the ones used for this research, or algorithms more suited to a mobile device. Additionally, the future performance of algorithms can be studied. While speeds of processors on mobile devices are continually increasing, it would be of use to see how performance would be enhanced by future additions to mobile architectures, such as more support for hardware acceleration of floating point calculations and optimizing implementations for future multi-core processors.

Additionally, use with external displays should be an area to be examined for future development. Integration with an HMD should be investigated as it becomes possible. As a bonus, HMDs are generally powered by their own source, meaning that the screen could be turned off of a mobile device. This would save a large amount of battery on the device, allowing for longer uptimes for the system.

Finally, on the application side of research, more work should be completed with receiving input on the system itself. While the received response to the system was quite positive, it would be of great use to involve individuals with low vision to test the system in an iterative fashion; testing new features and strengthening the current features of the device. This would ensure that the system would be useful for its intended purpose, as well as provide useful direction on future development.
What has been shown in this thesis is that a low vision system can be implemented on a common, low-cost, general-usage device. Additionally, it has been shown that it is possible to run various face detection and object tracking methods on a mobile device. Overall, this work and its future extensions have great potential to aid those with low vision and potentially other conditions. The performance capabilities of mobile platform will continue to increase and support various computer vision tasks in an increasing number of applications.
Bibliography


Conference on Computer Vision and Pattern Recognition Workshop, pp.73-80, 2010.


Appendix A  Sobel Edge Detection on Android

Overview

Mobile operating systems have gained much prevalence in the world of today. What used to be a simple apparatus for making telephone calls and receiving text messages has evolved into a much more advanced device, capable of browsing the Internet, giving directions to locations, and much more through a multitude of Apps. The Android operating system has become one of the prime tools in this evolution.

Developed by Google, the Android operating system is used primarily in 'smart' phones, but also is beginning to be used in other platforms, such as small laptops and even electronic readers, like the Barnes and Noble 'Nook.' With the prevalence of the system now, numerous companies and individuals have begun developing applications for the Android platform. This has been aided by Google releasing a Software Development Kit (SDK) free of charge to any interested developer.

Android applications are generally written in the Java programming language. This is a great aid for general development, but also does not provide an optimal platform for processing-intensive applications like those required for computer vision. For purposes like this, Google released what it calls the Native Development Kit (NDK). This is an environment that allows resource or calculation intensive portions of an application to be written in the C or C++ programming languages. The Java Native Interface, or JNI, is used for communication between the two difference code sets.

To find out what performance differences between the SDK and the NDK libraries, an edge detection application was written, one that would complete the
operation in Java, and one that would complete the operation in C++. Overall, the conclusions found were quite decisive; the operations completed in C++ were orders of magnitude quicker than the ones completed in Java.

**Theory**

Many potential applications were investigated for the purposes of time evaluation. A simple edge-detection algorithm was chosen due to its relative simplicity, which would allow for quick development as well as relatively similar measurements across implementations.

Edge-detection based on the Sobel operator, which involves convolving two separate kernels with the image under examination; one for vertical edges and the other for horizontal edges. These two operators are shown in Equations (1) and (2).

\[
G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \ast \text{Image} \quad (1)
\]

\[
G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \ast \text{Image} \quad (2)
\]

After the two results (Gx and Gy) are calculated, the Euclidean distance between corresponding points of the resulting images can be calculated, and this gives the gradient of the points, showing the edges in an image. This operation is shown in Equation (4). A suitable approximation can be found by using the magnitude of two pixel locations, as shown in Equations (3) and (4) [1].
The resulting gradients of $G$ provide the edge-detected image, which would be displayed to the user at the end of computation [1].

**Implementation Notes**

The algorithm was written in Java, and then an OpenCV C++ version was used as well. A copy of the Java algorithm was attempted to be written in C++, but this proved difficult with trying to stay as similar to the Java implementation as possible. Due to some constructs used in the Java implementation, which are native in android but without equivalents in the C++ NDK, attempts to replicate the Java implementation were unsuccessful. Therefore, two implementations were used and compared, the Java implementation and a native library implementation using the OpenCV graphic processing library.

After the implementations were complete, tests were run on three different devices. First, each implementation was run (and debugged) on a created Android emulator, local on the machine in use. Then, each implementation was tested on two Android phones, the HTC Magic and Motorola Droid, e running Android versions 1.6 and 2.1 respectively.

The OpenCV version of the algorithm was only run on the two actual devices. Due to an error in the emulation devices provided by Google, the native code version of the code was unable to be run on an emulator; however it was able to be run on the actual Android devices.
Four images were used for testing, a small logo of size 48 x 48 pixels, a medium-sized image with a size for 640 x 480 pixels, a larger image with a size of 1136 x 852 pixels, and finally an image taken from the camera of the phone with a resolution set to 320 x 240 pixels. The camera pictures were only tested on the actual devices, since the emulator does not have a real camera for taking images.

**Results and Analysis**

The program was written to provide an easy-to-use and informative display for testing each implementation. The welcome screen, directing the user to each interface, is shown in Figure 1.

![Application Welcome Screen](image)

*Figure 1 – Application Welcome Screen*

Selecting any of the buttons will execute the specified method on an image, which was defined per-compilation. Results for the small image, 48 x 48 pixels, are shown in Figures 2, 3, and 4 for the Java implementation.
Figure 2 – Java, Small Image, Emulated Device

Figure 3 - Java, Small Image, HTC Magic
In the interest of space and to avoid unnecessary redundancy, only a single result from each run will be displayed for the remaining views.

As can be seen above, processing times noticeably decrease moving from the emulator, to the HTC Magic, to the Motorola Droid. This can be seen as the emulator having the slowest processor and amount of resources allotted by its creation, the HTC magic having an intermediate amount of processing power and resources, and the Motorola Droid being the most powerful phone available to test.

The next test run, with the larger 640 x 480 image, yielded similar results, as did the test with the large image (1136 x 852). These results are shown in Figures 5 and 6. Additionally, a sample result from the camera test is shown in Figure 7. Finally, the complete aggregate results from the static image tests are shown in Table 1.
Figure 5 - Sample Mid-Sized Image (Emulator)

Figure 6 - Sample Large-Sized Image (Motorola Droid)
As can be seen in the three base images, the times for all implementations are quite shorter for the Native/OpenCV implementation. Interestingly, the times for the Native/OpenCV implementation for the two hardware implementations are very similar for the non-camera images.

An unexpected result involves the times recorded for the camera image. According to the results, the HTC magic is quicker with processing an image taken by the camera. On paper, the Motorola Droid is a superior device, with a faster processor and a more efficient implementation of the Android operating system, so this result is

---

**Table 1 – Edge Detection Timings**

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Emulator</th>
<th>HTC Magic</th>
<th>Motorola Droid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small Image</td>
<td>Java 1839</td>
<td>N/A</td>
<td>1037 9</td>
</tr>
<tr>
<td>Medium Image</td>
<td>N/A 275051</td>
<td>158689 1237</td>
<td>77313 1221</td>
</tr>
<tr>
<td>Large Image</td>
<td>N/A 608070</td>
<td>436238 3608</td>
<td>250587 3759</td>
</tr>
<tr>
<td>Camera Image</td>
<td>N/A</td>
<td>N/A 59178 774</td>
<td>320034 4954</td>
</tr>
</tbody>
</table>
unexpected. A possibly reasoning for this is that the camera in the Motorola Droid is capable of taking pictures up to 5 megapixels, while HTC magic camera can take pictures up to 3.2 megapixels. While each implementation was designed to take a picture of resolution 320 x 240 pixels, there appears to be some disconnect with how the camera captures images and then represents them within the system. This is a key portion of the system to be investigated in the future, so as to fully understand the functionality of the Android camera system.

**Conclusion**

Overall, the implementations were created without much issue. It was discovered that for a proper comparison and to properly port code to and from the native interface, much planning is necessary so that interfaces that are used in the Android Java are supported or can be implemented easily in the native code. From the results, it appears that for computer vision applications the OpenCV library is an excellent choice, even on Android, although future testing would be wise to ensure that algorithms are being completed as efficiently as possible.

**References**