FLEXIBLE COMPUTING ARCHITECTURE FOR REAL
TIME SKIN DETECTION

THESIS

Matthew P. Hornung, 2$^{nd}$ Lieutenant, USAF

AFIT/GCE/ENG/10-02

**DEPARTMENT OF THE AIR FORCE**
**AIR UNIVERSITY**

# *AIR FORCE INSTITUTE OF TECHNOLOGY*

**Wright-Patterson Air Force Base, Ohio**

AFIT/GCE/ENG/10-02

FLEXIBLE COMPUTING ARCHITECTURE FOR REAL TIME SKIN DETECTION

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Engineering

Matthew P. Hornung, BS

2$^{nd}$ lieutenant, USAF

March 2010

FLEXIBLE COMPUTING ARCHITECTURE FOR REAL TIME SKIN DETECTION

Matthew P. Hornung, BS
2$^{nd}$ Lieutenant, USAF

Approved:

_____/signed_____      _____
Michael Mendenhall, Maj, USAF (Chairman)      Date

_____/signed_____      _____
Yong Kim, Dr. (Member)      Date

_____/signed_____      _____
Jeffery Hemmes, Maj, USAF (Member)      Date

AFIT/GCE/ENG/10-02

*Abstract*

In both the Air Force and Search and Rescue Communities, there is a current need to detect and characterize persons. Existing methods use red-green-blue (RGB) imagery, but produce high false alarm rates. New technology in multi-spectral skin detection is better than the existing RGB methods, but lacks a control and processing architecture to make them efficient for real time problems. We hypothesize that taking a minimalistic approach to the software design, we can perform image preprocessing, feature computation, and skin detection in real time.

A number of applications require accurate detection and characterization of persons, human measurement and signature intelligence (H-MASINT), and SAR in particular. H-MASINT requires it for the detection of persons in images so other processing can be performed. It is useful in the SAR community as a method of finding persons partly obscured, in remote regions, and either living or deceased.

We have developed a modular computing architecture to perform the acquisition and processing in real time, as well as separate programs to perform processing and analysis of images post-acquisition. The architecture is flexible, as one can easily add additional functionality to meet growing demands. All programs were organized using a basic Model-View-Controller design, designed using Universal Modeling Language principles, and coded using a bottom-up approach.

Based on the results presented in this thesis, image acquisition, processing, skin detection, viewing, and saving can be performed in real time, at nearly 10 fps. Not only does this support the SAR community, the Air Force now has a new capability to help address its H-MASINT mission.

*Acknowledgements*

**Table of Contents**

*List of Figures*

## List of Tables

FLEXIBLE COMPUTING ARCHITECTURE

FOR

REAL TIME SKIN DETECTION

*I. Introduction*

*1.1 Background*

Recent research has produced an engineering model of human skin [1]. From evaluating the output of this model, the author was able to develop an efficient skin detection methodology using two distinct frequencies in the near infrared. Additional studies showed that a majority of the surrounding environment could be suppressed using information contained in the visible wavelengths.

Figure 1 shows an example of the capabilities of this detection strategy. This detection strategy works regardless of the color of the skin, or if the subject is alive or dead (depending on the condition of the body). Since it does not rely on shape based recognition, there is no requirement for the entire person to be exposed, just a single pixel of skin. The same body of work also presented the capability to estimate the melanin content of skin detected pixels, providing a mechanism to characterize the detected skin. An example of this is shown in Fig. 2. Due to recent work in [2], the Air Force now has a capable multispectral imaging system specialized for the task of skin detection. What is currently missing is an architecture that controls the cameras and processes the images from the multispectral imaging system in [2]. This thesis addresses that current deficiency.

**Figure 1.** Example of skin detection using hyperspectral imagery from the HyperSpecTIR3 (HST3) hyperspectral camera [1].
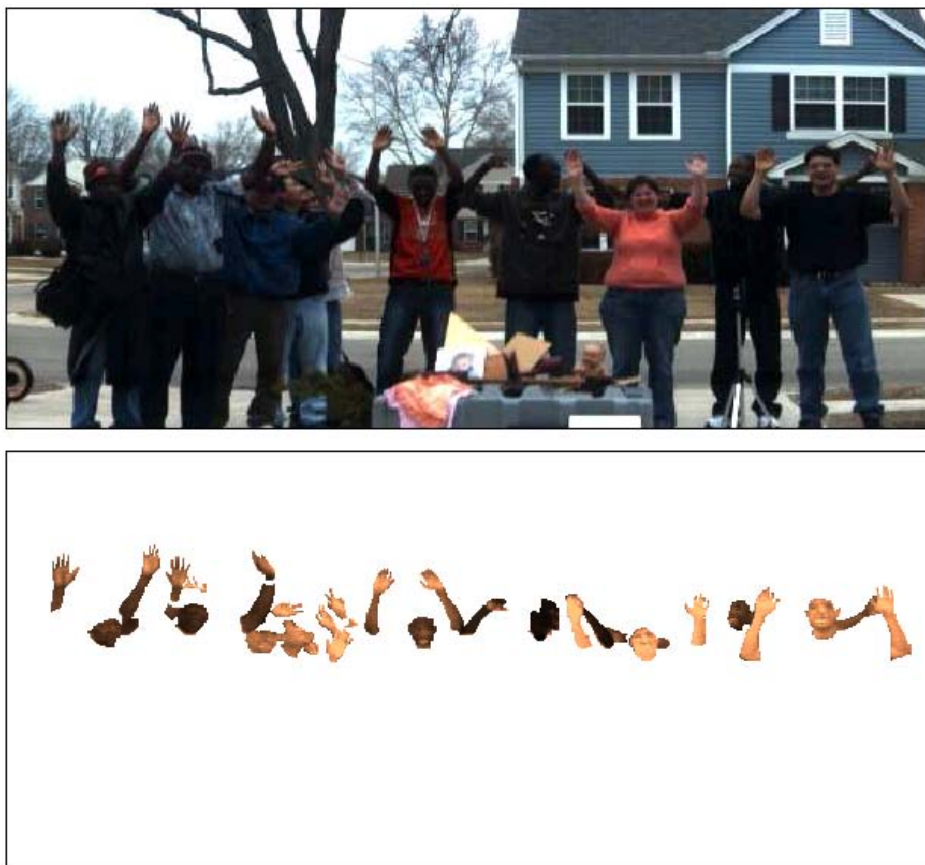


**Figure 2**. Example of melanin estimation on skin detections [1].

*1.2  Research Scope*

The goal of this thesis is to create a compact software architecture to control diverse camera hardware interfaces and to provide a fast and flexible computing platform that allows the creation, incorporation, and testing of new detection methodologies and algorithms, that is compatible with the multispectral system developed in [2].  The hypothesis is that, by taking a minimalistic approach to the design, we can perform image preprocessing, feature computation, and skin detection in real time.  Furthermore, this system should provide a platform capable of operating in real-life scenarios.

Our eventual goal is to create a mobile system for use in a variety of environments.  The primary use will be to have a small aircraft mounted system for use in search and rescue.  This environment requires a system to operate fast enough to obtain complete image ground cover while at the moving at high speeds, and also work with changing aspect and distances, as well as size and weight restrictions.

The hyperspectral cameras used in [1] are large, expensive, and are not nearly fast enough to provide real-time imagery, or even complete ground coverage for a fast, low flying aircraft.  The system developed in [2] uses near infrared cameras with filters to obtain the two necessary frequencies for skin detection.  False alarm suppression is handled with a color camera, while melanin estimation is planned for future work, and uses a broadband monochromatic camera.

## 1.3  Methodology

The software is built using a bottom up approach.  UML modeling is utilized to aid in development.  Testing is accomplished through field testing.  Runtime analysis of the algorithms and cameras is performed on the end-to-end system (hardware and software).

## 1.4  Resources

The software with both the National Instruments cards, (PCI interface) and the Imperx (Express54 interface) Camera Link frame grabbers provided drivers and software for both C++ and Matlab.  Due to the flexibility Matlab offers to the research group, it was chosen as the language of choice for image acquisition.  However, since higher level programming is difficult and runs slow in Matlab, Java was chosen to implement the architecture developed in this thesis (Java reference materials we use include [3,4,5]).  The optical system used in this thesis was developed in [2], and the algorithms in [1, 6].

## 1.5  Thesis Organization

Chapter 1 covers the introduction and basic goals of the thesis.  It defines the boundaries and focus of the thesis.

Chapter 2 discusses the background necessary to understand the methodology.  It begins by discussing the basics of the imagery and human skin model.  Next the algorithms used for skin detection in this thesis are presented.  Finally the basics of UML modeling are covered.

The methodology is covered in Chapter 3.  The overall program layout is discussed, and the design process for each piece of software is discussed.  First, the requirements and functionality are laid out, and then the sequence diagrams of the method calls required to make

each function happen.  Next, class diagrams are developed.  Finally, this chapter describes the video file and parameter file format as well as the parameter definitions.

Chapter 4 covers the testing and analysis of the software.  First, all known errors and flaws are discussed.  Next, a demonstration of the system is provided.  An effort is taken to show the output of each algorithm in order to show that it has been correctly implemented.  The results are discussed and analyzed.  Finally this chapter analyzes the performance of the cameras and the algorithms, both from a mathematical perspective, and time-based comparison.

Chapter 5 provides a summary of the present work.  It highlights the important aspects of the methodology and results, and draws the necessary conclusions.  Finally, the chapter discusses the possibility of future work.

Finally, Appendix A is the required software and the steps that must be taken to set up and use the software, and Appendix B is the User's Manual for the software.

## II.  Background

This chapter discusses the basics of the imagery used for this research.  We then discuss the basic human skin model, and the algorithms used to process the images.  This chapter also gives the basic UML modeling techniques used to develop the software.

### 2.1  Hyperspectral Imaging

Standard color cameras take images at three different wavelengths of light, corresponding to red, green, and blue.  The bandwidths of these three wavelengths are generally on the order of 100 nm wide.  Hyperspectral cameras take an image, but often at several hundred different wavelengths, ranging from the visible through the near infrared, typically 400-2500 nm.  The bandwidth at each of these wavelengths is generally on the order of 10 nm wide.  This additional information allows analysts to examine the reflectance properties of the imaged materials.

A standard hyperspectral image can be thought of as a cube, where the X-Y plane is the two dimensional image, and the Z plane is the wavelength.  Alternatively, examining a single pixel through the height of the cube gives us the magnitude of the reflection across the spectrum for that material.  Figure 3 shows the construction of a hyperspectral cube in comparison to a multispectral image [7].

Hyperspectral imagery contains a lot of information about the material being imaged.  Different materials respond differently at various wavelengths, and a analyzing a single pixel throughout a hyperspectral cube provides us with important information regarding the material content in the image at that pixel.  These curves can be used to discriminate between multiple classes of materials in images.
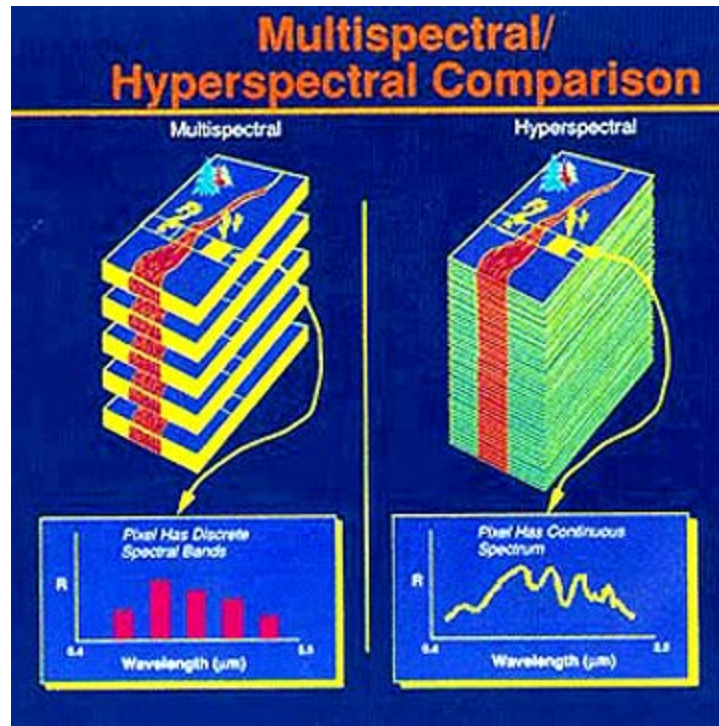
**Figure 3.** Comparison of a hyperspectral cube to a standard multispectral image [7].

*2.2 Human Skin Reflectance*

Previous research [1, 2, 6, 8] developed a multispectral approach to skin detection, and demonstrated using hyperspectral imagery. We use broadband monochrome cameras that take images in the near infrared portion of the electromagnetic spectrum. We place a filter in front of each camera, so the image is only of a very particular bandwidth, about 10? nm wide. By using two near infrared cameras with filters, a standard color camera, and a monochrome camera in the visible portion of the spectrum, we can create a composite image with six distinct wavelengths [2]. For the purpose of this research, these wavelengths are 1580 and 1080 nm in the near infrared, 750 nm for just beyond the visible, and 660, 540, and 475 nm in visible.

A model of human skin reflection was developed in [1]. Figure 4 shows an example skin reflectance spectrum generated by that model. The lines show light skin (2.4% melanin) and

dark skin (24% melanin). Darker skin has reduced reflectance at the shorter wavelengths due to melanin. The basic shape remains the same beyond 1000 nm, due primarily to water absorption.
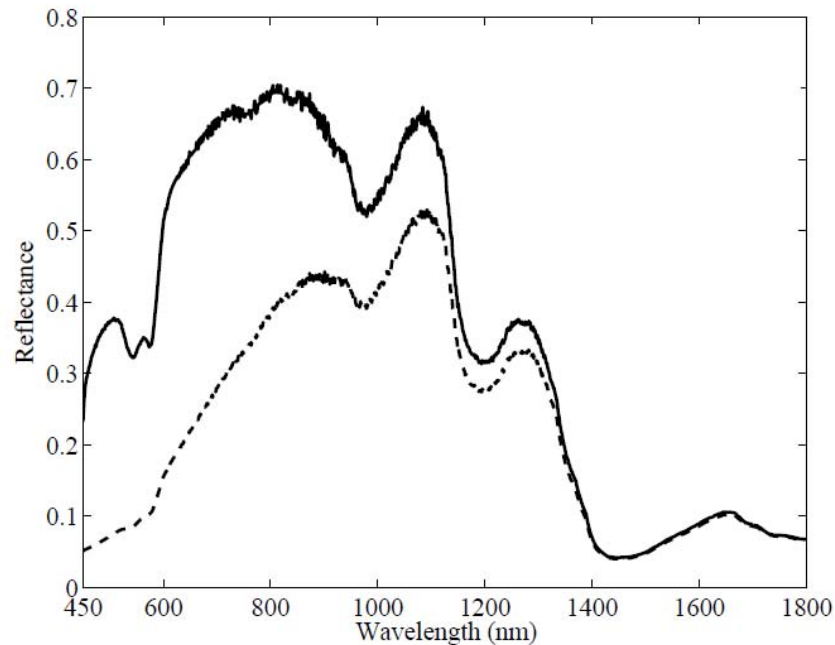


**Figure 4.** Human Skin Model showing the effect of melanin on reflectance [1]. The solid line is a plot of skin reflectance with 2.4% melanin, and the dashed line of 24% melanin.

For the purpose of this research, the important features are the high reflectance at approximately 1080 nm, and the low reflectance around 1580 nm. The significantly lower reflection at the longer wavelengths is due to water absorption. However, other water-heavy materials also tend to share this same drop in reflectance, which can cause problems as discussed later. Figure 5 shows the difference of skin reflectance between 1580 nm and 1080 nm. We use 1580 nm instead of 1400 nm because there is a large amount of atmospheric absorption around 1400 nm, and therefore very little natural illumination at that wavelength [1].
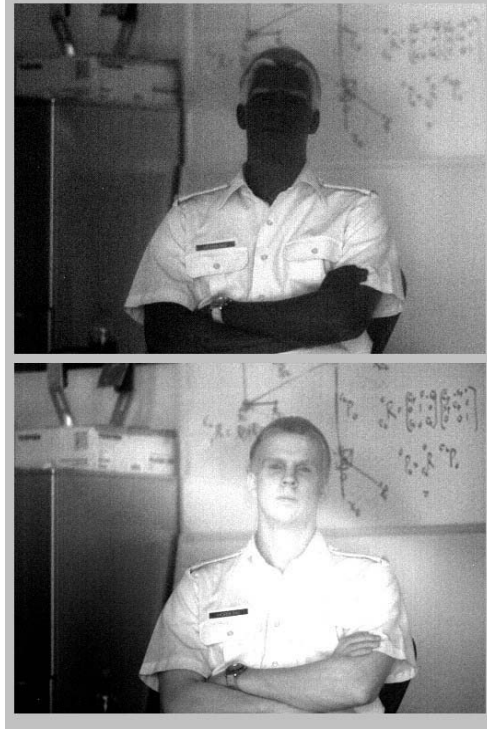
**Figure 5.** Filtered NIR images showing the difference in skin reflectance between 1580 nm (top) and 1080 nm (bottom).

Another important feature is the difference between the reflectance in the red (660 nm) and green (540 nm) portion of the spectrum. Skin is significantly more red than it is green, whereas most common skin confuses are typically either more green than red (i.e. vegetation) or approximately equal (i.e. snow) in reflectance.

Figure 6 shows the difference between the skin spectra and typical skin "confusers". Skin colored plastics and cardboard have very similar reflectance in the visible spectrum, which cause color based approaches to fail. However there are clear differences in the near infrared wavelengths that allow our methods to discriminate between the materials.
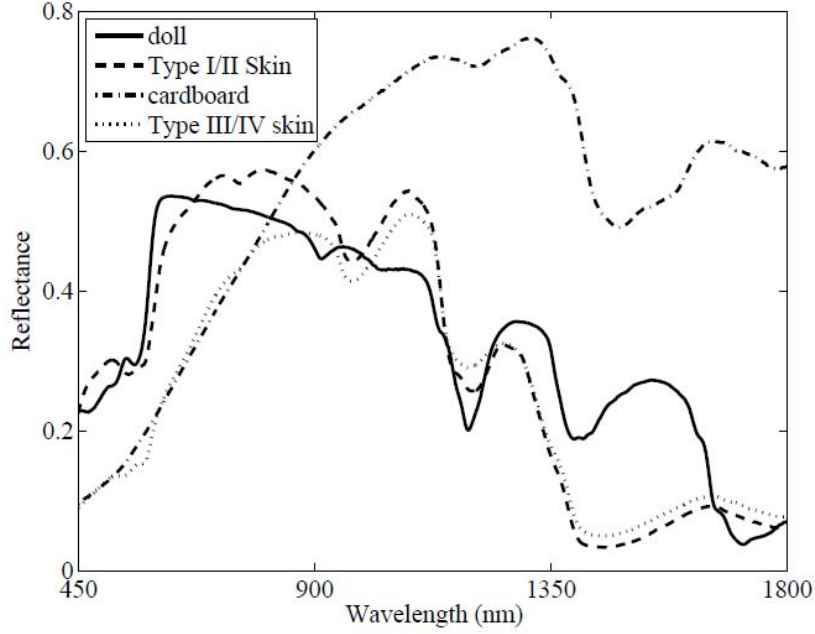
**Figure 6.** Spectra of Type I/II (light) and Type III/IV (dark) skin (dashed and dotted lines respectively) and spectra of a plastic doll and brown cardboard (solid and dashed-dotted respectively) [1].

## 2.3  Features for Skin Detection and False Alarm Suppression

For the remainder of this thesis, $\hat{\rho}_{\text{wavelength}}$ is the estimated reflectance in a given pixel, calculated from the measured reflectance of light and dark panels, using the empirical line method (ELM) [6].

### 2.3.1  Normalized Difference Skin Index

The Normalized Difference Skin Index (NDSI), developed in [1], describes a method for skin detection by examining the difference between 1080 nm and 1580 nm.  The NDSI value $\gamma$ is found by

$$\gamma = \frac{\hat{\rho}_{1080} - \hat{\rho}_{1580}}{\hat{\rho}_{1080} + \hat{\rho}_{1580}} \tag{1}$$

where $\hat{\rho}_{1080}$ and $\hat{\rho}_{1580}$ are the estimated reflectances of the 1080 and 1580 nm wavelengths, respectively.

*2.3.2 Normalized Difference Vegetation Index*

Reasonable performing skin detection algorithms cannot be accomplished using NDSI alone. Even though it may be possible to identify nearly all pixels in an image that contain skin, it will also falsely identify several other materials as skin. Common confusers for skin detection using only NDSI include some types of vegetation, snow, mud, or a raw steak. Therefore, additional features must be used to remove these common false positives from the detections.

The equation to detect vegetation, thus enabling it to be ruled out from skin detections, is shown in Eqn. 2, where α is the value of the Normalized Difference Vegetation Index (NDVI).

$$\alpha = \frac{\hat{\rho}_{750} - \hat{\rho}_{660}}{\hat{\rho}_{750} + \hat{\rho}_{660}}$$

(2)

In Eqn. 2, $\hat{\rho}_{660}$ and $\hat{\rho}_{750}$ are the estimated reflectances of the 660 and 750 nm wavelengths, respectively. As can be seen from Eqn. 2, this feature depends on the addition of the 750nm camera. It is currently incorporated into the architecture but disabled.

*2.3.3 Normalized Difference Green-Red Index*

Another feature useful for ruling out false positives is the Normalized Difference Green-Red Index (NDGRI). The assumption is that skin is more red than green, while most other skin confusers are not [1]. This feature is especially useful in ruling out vegetation and snow. The NDGRI is defined as:

$$\beta = \frac{\hat{\rho}_{540} - \hat{\rho}_{660}}{\hat{\rho}_{540} + \hat{\rho}_{660}}$$

(3)

where $\hat{\rho}_{540}$ and $\hat{\rho}_{660}$ are the estimated reflectances at 540 and 660 nm, respectively.

## 2.4 Algorithms for Skin Detection

Currently, three algorithms for skin detection are specified [1,6] in our system. The Basic Skin Detector only uses the NDSI values, while the rules based detectors use a rectangular bound on either (NDSI, NDVI) or (NDSI, NDGRI) pairs. The last detection algorithm is the likelihood ratio test, which uses a continuous function as the decision boundary (basically a bounding polygon).



(a) (NDSI/NDVI) Pair          (b) (NDSI/NDGRI) pair
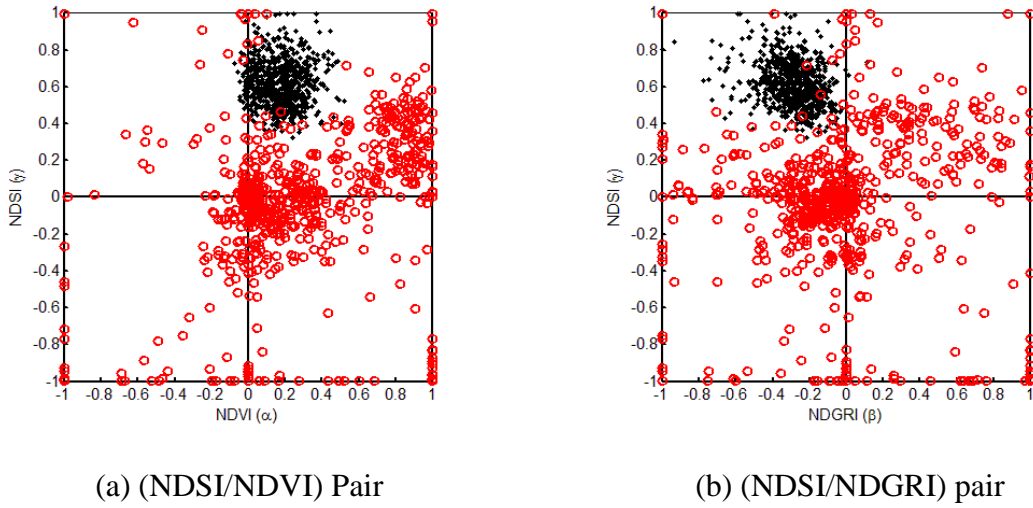
**Figure 7.** (a) Joint distribution of NDVI and NDSI values using spectral measurements and skin model-generated data. (b) Joint distribution of NDGRI and NDSI values using spectral measurements and skin model-generated data. Spectral measurements of a random sampling of materials are shown as red, and skin model-generated data are shown as black [6].

### 2.4.1 Basic Skin Detector

The most basic detector, as discussed above, uses only the NDSI values. This also means the detector can be run using only two cameras. However, because it will generate a large number of false positives it is not particularly useful. The NDSI values that are probably skin are $0.657 \le \gamma \le 0.768$ [8].

## 2.4.2 Rules Based Skin Detectors

Indentifying a particular pixel as skin should not be based on the NDSI alone, as it results in too many false positives. Two rules based algorithms are shown in Eqn. 4 and Eqn. 5. Equation 4 shows the rules for determining if a pixel is skin based on NDVI and NDSI, where the constants $a_1$ and $a_2$ are the bounds for NDVI. The default values are 0.004 and 0.503, but may change depending on the exact environment [8]. The constants $c_1$ and $c_2$ are the bounds for NDSI, with defaults at 0.657 and 0.768 [8]. If the result S for any particular pixel is 1, then it should be considered skin, otherwise it should not. Because this equation depends on the NDVI feature, which depends on the 750 nm camera, it is fully incorporated into the architecture but disabled.

$$S = \begin{cases} 1 & \text{if } a_1 \leq \alpha \leq a_2 \text{ and } c_1 \leq \gamma \leq c_2 \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

Equation 5 shows the skin detection rules based on NDGRI and NDSI. The default values for $b_1$ and $b_2$ are -0.541 and -0.062 [8].

$$S = \begin{cases} 1 & \text{if } b_1 \leq \beta \leq b_2 \text{ and } c_1 \leq \gamma \leq c_2 \\ 0 & \text{otherwise} \end{cases} \quad (5)$$

These detection algorithms have the advantage of relying on the values of each feature independently, so the entire algorithm consists two feature computation and four comparisons.

## 2.4.3 Likelihood Ratio Test Algorithm

The likelihood ratio test is a hypothesis testing methodology where the two hypothesis are the pixel is skin ($H_1$) or the pixel is not skin ($H_0$). The likelihood ratio is the ratio between the estimates of the two likelihoods $\hat{f}_1(\theta)$ and $\hat{f}_0(\theta)$, where $\hat{f}_1(\theta)$ is the distribution of the data when it is skin and $\hat{f}_0(\theta)$ is the distribution of the data when it is not skin. The likelihood ratio is defined as:

$$\Lambda_\Theta(\theta) = \frac{\hat{f}_1(\theta)}{\hat{f}_0(\theta)} \overset{\overset{H_1}{>}}{\underset{\underset{H_0}{<}}{}} \eta \qquad (6)$$

where $\eta$ is the threshold.

The likelihood ratio as designed in [6] assumes each distribution is a Gaussian mixture model. The Gaussian mixture models describe the joint distribution of NDSI and NDGRI for skin present and skin absent. If the resulting hypothesis is greater than $\eta$, then it is most likely skin, otherwise it is not.

There are 43 parameters that must be specified per [6]. The first 42 are grouped in sets of six sets of [$P_i$ $a_i$ $b_i$ $d_i$ $\mu s_i$ $\mu g_i$], where $i$ varies from 1 to 7. $P_i$ is the prior probability, $a_i$, $b_i$, and $d_i$ are the elements of the covariance matrix ($\Sigma = \begin{bmatrix} a & b \\ b & d \end{bmatrix}$), and $\mu s_i$ and $\mu g_i$ are the means of the NDSI and NDGRI samples assigned to the distribution. The 43$^{rd}$ parameter is $\eta$, which is used in the final comparison.

The likelihood ratio is expressed in Eqn. 7, where S is the NDSI value, and G is the NDGRI value.

Equation 7 must be run seven times, once for each value of $i$, resulting in $F_1$ through $F_7$, where F is defined as (less the subscripts for clarity)

$$F = \frac{ad - b^2}{2\pi} * \exp\left[-0.5(d(S - \mu_s)^2 - 2b(S - \mu_s)(G - \mu_g) + a(G - \mu_g)^2\right] \qquad (7)$$

After all seven F values are calculated, the likelihood ratio is computed with

$$R = \frac{P_1 F_1 + P_2 F_2 + P_3 F_3}{P_4 F_4 + P_5 F_5 + P_6 F_6 + P_7 F_7} \qquad (8)$$

where $P_1+P_2+P_3=1$ and $P_4+P_5+P_6+P_7=1$. If $R > \eta$, then the pixel is most likely skin, otherwise it is not skin.

The default parameters used by this system are shown in Table 1. The threshold $\eta$ is a user specified parameter and is used to "tweak" the system. However $\eta = 0.0226$ is considered in [6] a reasonable default value for the current system configuration.

**Table 1.** Default parameters for the Likelihood Ratio Test [6]

| Variable Name | i=1 | i=2 | i=3 | i=4 | i=5 | i=6 | i=7 |
|---|---|---|---|---|---|---|---|
| P | 0.0501 | 0.4286 | 0.5213 | 0.2966 | 0.2034 | 0.3733 | 0.1267 |
| a | 0.0331 | 0.0164 | 0.0076 | 0.0808 | 0.0868 | 0.1408 | 0.0059 |
| b | 0.0016 | 0.0002 | -0.0006 | -0.0111 | -0.0145 | -0.0026 | -0.0006 |
| d | 0.0161 | 0.0153 | 0.0078 | 0.0202 | 0.0174 | 0.1981 | 0.0138 |
| μs | 0.7318 | 0.7008 | 0.5548 | 0.3446 | 0.0875 | 0.2332 | 0.8983 |
| μg | -0.5921 | -0.3185 | -0.2306 | 0.4085 | -0.1872 | -0.0441 | 0.0063 |

## 2.5 Near Infrared Melanosome Index

Another important aspect of our system is the ability to estimate melanin content of the detected skin. While this is not directly part of identifying skin, it is useful in determining if a detection is possibly the subject desired. The Near-Infrared Melanosome Index (NIMI) makes use of 750 nm and 1080 nm images, and is only applied to pixels that have already been identified as skin. As with the NDVI, this feature is currently disabled until the addition of a 750 nm camera.

The first step in determining the melanin content of skin is to calculate the ratio between the two wavelengths as shown in Eqn. 9.

$$ N = \frac{\rho_{750}}{\rho_{1080}} \tag{9} $$

The second step is to estimate the actual percentage of melanosomes present. In humans the range of values is 0% to 43% [1]. The melanosome estimate for typical ("standard") person

can be found by Eqn. 10, and for a median person Eqn. 11. Figure 8 shows the corresponding

lines overlaid on the range of NIMI values for each melanin value. The solid line corresponds to

the standard person, and the dashed line to the median person.

$$\widehat{M}_{MD} = -1.07N^5 + 4.93N^4 - 9.13N^3 + 8.80N^2 - 4.90N + 1.40 \qquad (10)$$

$$\widehat{M}_{SP} = -1.78N^5 + 7.37N^4 - 12.35N^3 + 10.84N^2 - 5.50N + 1.45 \qquad (11)$$
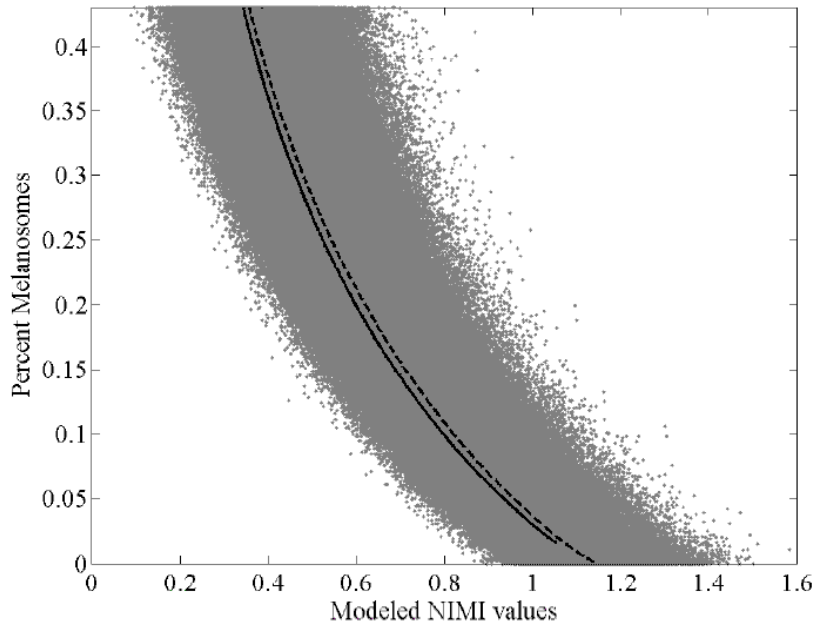


**Figure 8.** The gray dots represent actual melanosome percentage vs. the calculated NIMI value. The dashed line represents the median NIMI value, and the solid line is the average NIMI value. The regression shown in Eqn. 10 corresponds to the median value (dashed line), and Eqn. 11 to the standard person (solid line) [8].

*2.6 Empirical Line Method*

There are many factors that can affect the image that a camera produces. It can vary

depending on lighting, shadows, objects in the image, or any number of other factors. These can

affect different portions of the spectrum differently. The most notable issue is atmospheric

absorption. Empirical Line Method (ELM) is frequently used in the remote sensing communities

to remove linear atmospheric effects [9].

We use a pair of calibration panels for atmospheric correction, one light and one dark. Since we know the exact reflectance at each panel at each wavelength, and we can measure the panel values in the images, we can apply a gain and offset to adjust the received images to an estimate of the actual reflectance.

To perform ELM, we first calculate two ratios, $\hat{a}$ and $\hat{b}$, as shown in Eqn. 12 and Eqn. 13. The variables $\rho_{white}$ and $\rho_{gray}$ are the actual reflectance values of the panels used, the defaults of which are shown in Table 2, which shows the dependence on wavelengths. The variables $L_{white}$ and $L_{gray}$ are the measured values of the two panels from the camera.

**Table 2.** Default Values for the ELM calculation [2].

| Wavelength | $\rho_{gray}$ | $\rho_{white}$ |
|---|---|---|
| Visible | 0.99 | 0.075 |
| 750 nm | 0.99 | 0.09 |
| 1080 nm | 0.989 | 0.108 |
| 1580 nm | 0.987 | 0.131 |

$$\hat{a} = \frac{\rho_{gray} - \rho_{white}}{L_{gray} - L_{white}} \tag{12}$$

$$\hat{b} = \frac{L_{white}\rho_{gray} - L_{gray}\rho_{white}}{\rho_{gray} - \rho_{white}} \tag{13}$$

For implementation purposes, the values of $\hat{a}$ and $\hat{b}$ only need to be calculated once. After they have been calculated, they are stored and used on all subsequent images. Equation 14 shows the operation that must be run on every pixel of every image, where L is the measured value of the pixel and $\hat{\rho}$ is the estimated value of the reflectance of that pixel.

$$\hat{\rho} = (L - \hat{b})\hat{a} \tag{14}$$

*2.7  UML Modeling and Software Development*

The first step in developing the computing architecture is to create a number of use cases. After writing out a number of brief use cases, a general program flow and functionality is established, and then basic sequence diagrams are created. The sequence diagrams shown in Chapter 3 are created using the MaintainJ plug-in for Eclipse [10] and are generated automatically during run time.

The next step is to create class diagrams to determine exactly where classes should be placed and what access they should be given. As with the sequence diagrams, only the final versions are shown. The class diagrams shown are created using eUML2 [11] to directly reverse engineering the code. The final steps in the development process are testing and analysis. The following discussions on UML design are derived from [12].

*2.7.1  Use Cases*

Use cases are text based descriptions of how users interact with the computer system. The goal of use cases is to provide the developer with an idea of what functions the system needs to perform and all required interactions with the user in order to complete its goal. They are generally the first step in the development process because this is how the developer determines what the system actually is.

There are several levels of use cases. The most basic are brief use cases, which are generally single paragraph summaries covering the main success scenario. Next are casual use cases, which are several paragraphs covering various scenarios. The final level is fully dressed use cases, which are formal and can be several pages long. Formal use cases cover every step and variation in detail, and contain all prerequisites and result guarantees. Fully dressed use cases are generally written out in an outline form instead of paragraph form.

For the purpose of our system, only brief use cases are developed. Since this is as much a scientific as it is end user piece of software, anything off the main success path is usually considered unacceptable and will cause the program to end. Furthermore the exact dialog between user and system is not as important as the ability of the system to perform some function. For both these reasons, there is little purpose in progressing beyond brief use cases. Additionally, the use cases have not been included in this document as they do not significantly add to the readers understanding of the architecture post-development.

## 2.7.2  Sequence Diagrams

After the system functions have been determined, the developer must determine how the system will perform its required functions. Sequence diagrams are useful for this, because they allow the developer to determine what classes and objects are needed, the methods required in those classes, and the data required to be passed to which methods. Post development, the sequence diagram help the reader understand exactly what is being done in the program for each function.

## 2.7.3  Class Diagrams

The class diagrams are created from the sequence diagrams. The developer takes the classes and methods deemed necessary, and organizes them into a useful structure. There is very little new information over the sequence diagrams, as this is basically a direct derivation. However, class diagrams are a much easier format to understand and program the basic structure from.

*III. Methodology*

The goal of this chapter is to present a design that can be used to create a skin detection system. It describes the design in detail so future developers can modify and expand this design. This chapter discusses the methods and processes used to design the software architecture. The first topic is the top level design that decides how the programs are divided. Then, the design of each program is discussed.

*3.1 Top Level Description*

The capability of the system is contained in three distinct programs. This is due to a logical separation in the requirements for any particular application. While in the field, a user needs to be able to perform real time processing and view the raw data and results. The user also needs to be able to save multiple data streams of his or her choosing. Because the field deployed system needs to be as compact and efficient as possible, a minimalistic program is needed, causing the separation of these functions from any others. This software will be referred to as the Acquisition Program for the remainder of this thesis.

Two other capabilities are desired, one to allow the user to process saved data from one or more previously saved streams and save the result. The other allows the user to play back a previously saved file. While it would be entirely possible to place both of these functions in the same piece of software, it is the opinion of the author that the additional complexity is detrimental to the portability to less capable systems and to the speed of the software. These pieces of software will be referred to as the Processing Program and the Playback Program. Figure 9 shows the interactions and ordering of these programs.
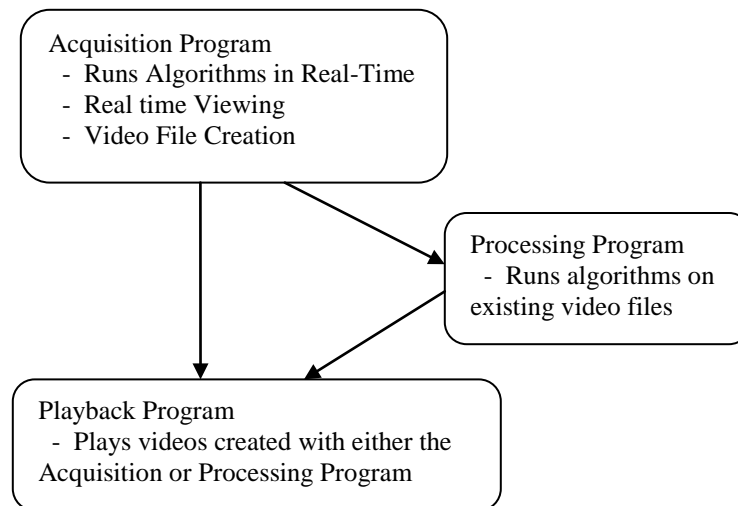
**Figure 9.** Interactions and basic purposes of the three programs in the architecture

*3.1.1 Video File Format*

All three programs must share the same video format in order for the system to work. Table 3 shows the format of the different fields in the file and their sizes. The size of the data field in an image will change depending on the height and width of the image, as specified in the header. The data size shown is for a 640x512 image (1,310,720 pixels). Each pixel of the data is stored as a 32-bit integer, regardless of the bit depth, due to limitations of Java. If the coordinates for a panel are not set, they are set to '-2'.

Figure 10 shows an example sequence in a video file. The first item is always the file header. This is the only place in the file this should occur. Following the file header is a sequence of interspersed images and panel coordinates (locations of the grey and white panels as required for the Empirical Line Method algorithm). When the program is reading through the file, it will start by reading a long integer (8 bytes) from the file. If it is equal to '-1' then the item is a set of panel coordinates, otherwise it is an image.

**Table 3.** Data types and their fields and associated sizes defined in our video file format.

| Data Type | Field Descriptor | Size (Bytes) |
|---|---|---|
| File Header | Bit Depth | 4 |
| | Height | 4 |
| | Width | 4 |
| | Dark Panel X | 4 |
| | Dark Panel Y | 4 |
| | Light Panel X | 4 |
| | Light Panel Y | 4 |
| Image | Frame Number | 8 |
| | Time Stamp | 8 |
| | Data | 1310720 (640 x 480) |
| Panel Coordinates | '-1' | 8 |
| | Dark Panel X | 4 |
| | Dark Panel Y | 4 |
| | Light Panel X | 4 |
| | Light Panel Y | 4 |

| File Header | Image | Image | … | Image | Panel Coordinates | Image | Image | … |
|---|---|---|---|---|---|---|---|---|

**Figure 10.** Example video file layout showing how the data types are interspersed. The beginning of the file is always a header, followed by images with a periodic set of panel coordinates mixed in.

This methodology of interspersing item types can be easily extended to including other additional information into the file. Once such possible upgrade might be to include information to highlight a skin detection region. Table 4 shows what such an extension might look like. All three programs would have to be updated to deal with the new item type.

**Table 4.** Example of another possible data type that could be added to extend the functionality of the video file format.

| Data Type | Field Descriptor |
|---|---|
| Boundary Definition | '-2' |
| | Center X |
| | Center Y |
| | Radius |

*3.1.2 Parameter File Format*

The parameter file is shared by both the Acquisition Program and the Processing Program. Since the Playback Program does not run algorithms, there is no need for the parameter file in that program.

The Parameter file is called "ParamFile.txt" and must be placed in the same folder that the video files are placed in. It is arranged as a comma separated list, where one variable is placed on each line. All variables, except those for Empirical Line Method (ELM), have defaults in the program, but those defaults can be overwritten by listing them in the parameter file. There is no particular order of variables required by the file. If a variable is entered twice the second entry will take precedence.

Table 5 provides a list of available parameters and the algorithm they are associated with. The parameters for the rules-based algorithms are shared by the basic detector, and the two rules-based detectors. The $a_1$ and $a_2$ parameters are the thresholds applied to Normalized Difference Vegetation Index (NDVI), $b_1$ and $b_2$ for Normalized Difference Green-Red Index (NDGRI), and $c_1$ and $c_2$ for Normalized Difference Skin Index (NDSI).

The $F_i$ parameters are not defined in the program, and therefore must be defined in the parameter file if the likelihood ratio algorithm is to be run. The $F_i$ variable name must be followed by six numbers, all comma separated. All other variables should be followed by only

one parameter.  Comments can be included in the file by placing a "//" as the variable name, followed by the comment.  Figure 11 shows an example parameter file.

**Table 5.**  Loadable parameters associated with the algorithms and their default values.

| Algorithm | Parameter | Default Values |
|---|---|---|
| Rules Based | a1 | 0.004 |
| | a2 | 0.503 |
| | b1 | -0.541 |
| | b2 | -0.062 |
| | c1 | 0.657 |
| | c2 | 0.768 |
| NIMI | p1 | -1.78 |
| | p2 | 7.37 |
| | p3 | -12.35 |
| | p4 | 10.84 |
| | p5 | -5.5 |
| | p6 | 1.45 |
| Likelihood Ratio | eta | 0 |
| | F1 | Undefined |
| | F2 | Undefined |
| | F3 | Undefined |
| | F4 | Undefined |
| | F5 | Undefined |
| | F6 | Undefined |
| | F7 | Undefined |
| ELM | rw1580 | 0.987 |
| | rw1080 | 0.989 |
| | rw750 | 0.99 |
| | rw660 | 0.99 |
| | rw540 | 0.99 |
| | rw475 | 0.99 |
| | rd1580 | 0.131 |
| | rd1080 | 0.108 |
| | rd750 | 0.09 |
| | rd660 | 0.077 |
| | rd540 | 0.075 |
| | rd475 | 0.073 |
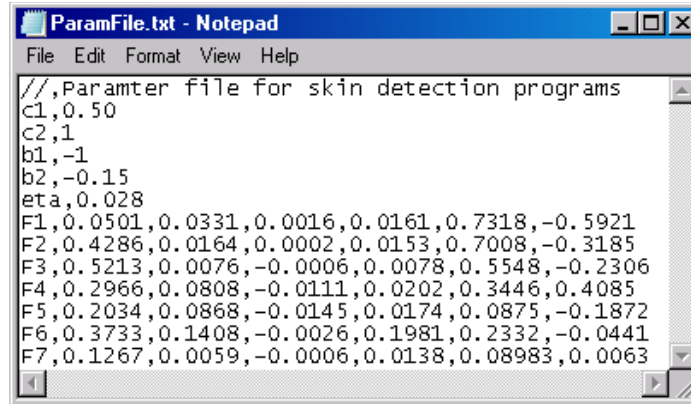
**Figure 11.** Example parameter file. The $F_i$ lines are required for the likelihood ratio because there are no defaults in the software. The *//* parameter is used for comments.

*3.2 Acquisition Software Design*

The goal of this system is to enable the user to view and save images from the camera, as well as view and save the output of the higher level algorithms. All operations are on real time images and must be performed in real time.

*3.2.1 Program Overview*

The software will follow a basic Model-View-Controller pattern [12]. This pattern allows the Graphical User Interface (GUI) of the program to be separate from the basic logic of the system. It also allows changes to be made in the underlying hardware without changing the logic of the software. This program consists of two primary threads. The GUI controls all asynchronous operations, and the main program loop within the main() method controls all synchronous operations. Everything that occurs based on a user input is controlled by the GUI, and all the continuous functions of the program, such as receiving the next image from each camera and updating the outputs, is controlled by the main loop.

Another physical requirement of the system is that each camera is controlled by a separate class. Therefore, the software has three input classes: one for each Near Infrared (NIR) camera and one for the color camera. A fourth input class for the 750 nm camera is

incorporated into the system, but is disabled. Based on the functionality of the program determined by the use cases, there are two output classes: an output window, and a video file creator. There are a large number of distinct algorithms that can be performed. Furthermore, there is another class that loads the parameters file. These classes are all wrapped in the DataHandlers package.

The controller and the DataHandlers classes (input, output, and algorithms) are given access to the model. The model consists of a large number of images, where each image represents the most recent image from a particular camera or algorithm output. It also contains the parameters used by all the algorithms. When the program is run, there are two instances of the Model created. One contains the raw images, and the other contains preprocessed images (those that have been processed using the ELM algorithm). Furthermore, there is a Picture class that defines exactly what data is contained in an image. Throughout the architecture, wherever images are used, they are instances of the Picture class.

The controller is responsible for translating the basic operations called from the GUI and main program loop into the lower level logic required to synchronize the rest of the program.

The CameraAcquisition class is the compiled Matlab code. Each of its methods work directly with the cameras. There are methods for initializing a camera, triggering all the cameras at once, getting the most recent frame from a camera, and closing all the cameras. This class is compiled using the Matlab Builder JA, and is wrapped in a separate .jar file.

*3.2.2  Sequence Diagrams*

When the program is started, it must initialize each camera and start the GUI. The sequence diagram in Fig. 12 shows the method calls required. As can be seen, the main() method in AcqMain first initializes the controller, and then the GUI. This is because the GUI

needs to have knowledge of the controller, so it is necessary to start the controller first. On initialization, the controller loads the parameters file, then starts each camera. The final two function calls from the main() method, run() and updateFrameRate(), are the two calls repeatedly made from the main program loop. The sequence diagrams of Controller.run() are discussed in more detail later in this section.

When no algorithms are running, the sequence of calls made from the Controller.run() method are shown in Fig. 13. It triggers the cameras and gets the most recent frame from each. These frames are directly stored in the model by each camera class, the controller is not responsible for managing the pictures directly.

When the user first opens a window, the sequence shown in Fig. 14 occurs. The GUI must retain a link to the open window in order to close it when the user chooses. The controller, however, is responsible for making all other method calls to the output windows. The identifier determines what should be displayed in the window. If the user opens a video file creator instead of a window, the function call is OpenImageFile() instead of OpenImageView(), but otherwise identical.

After a window is open, the sequence of calls from the Controller.run() method changes to that shown in Fig. 15. The window opened shows only the raw data from one of the cameras, with no algorithms applied. The only difference from the sequence without the window open is the Update() call. If multiple windows are open, then this method call is made to each.
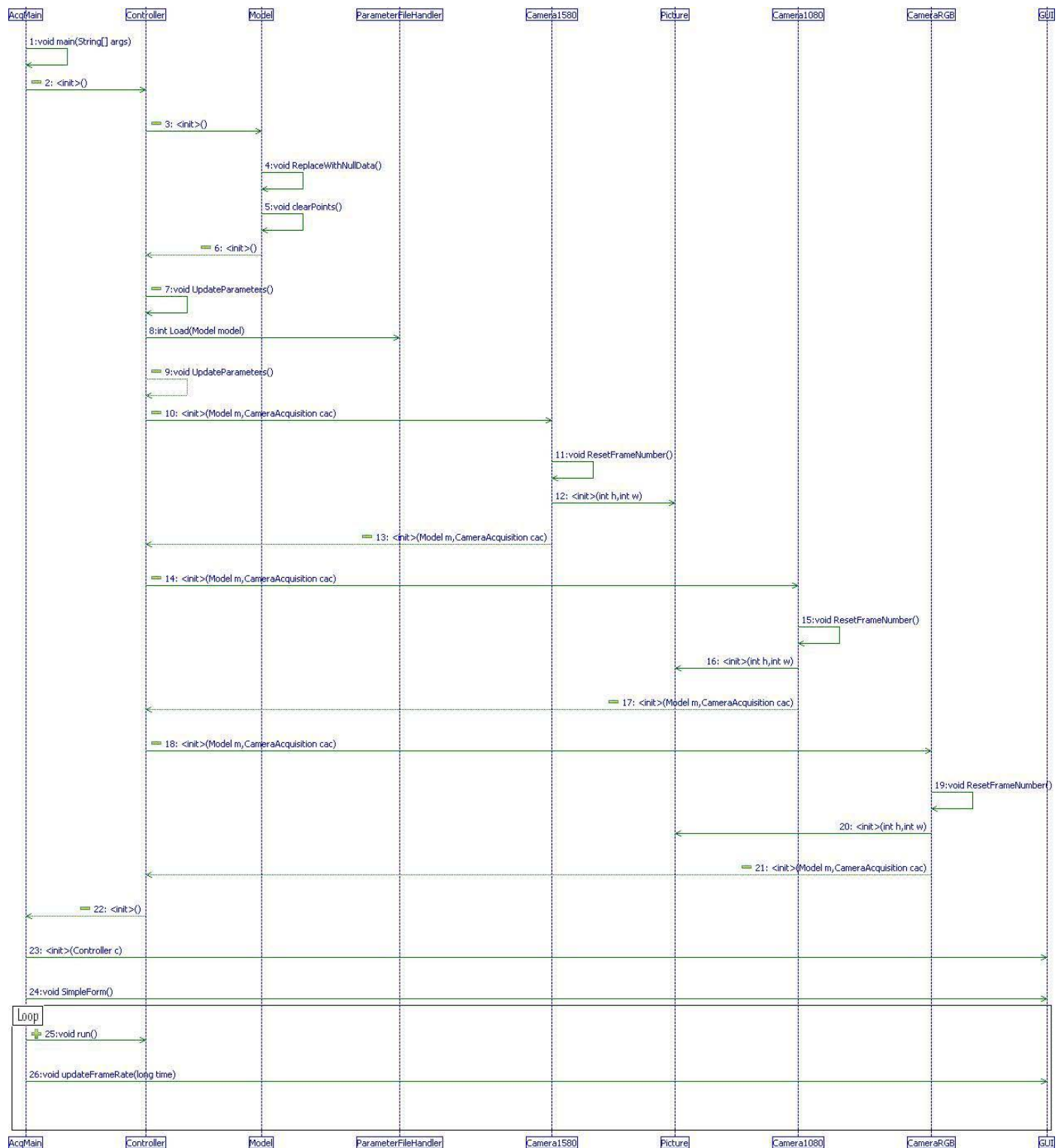
**Figure 12.** Startup sequence of the Acquisition Program. The Controller, GUI, Models, and Cameras are started, and then the program is placed into a continuous loop updating the model and outputs with the most recent data.
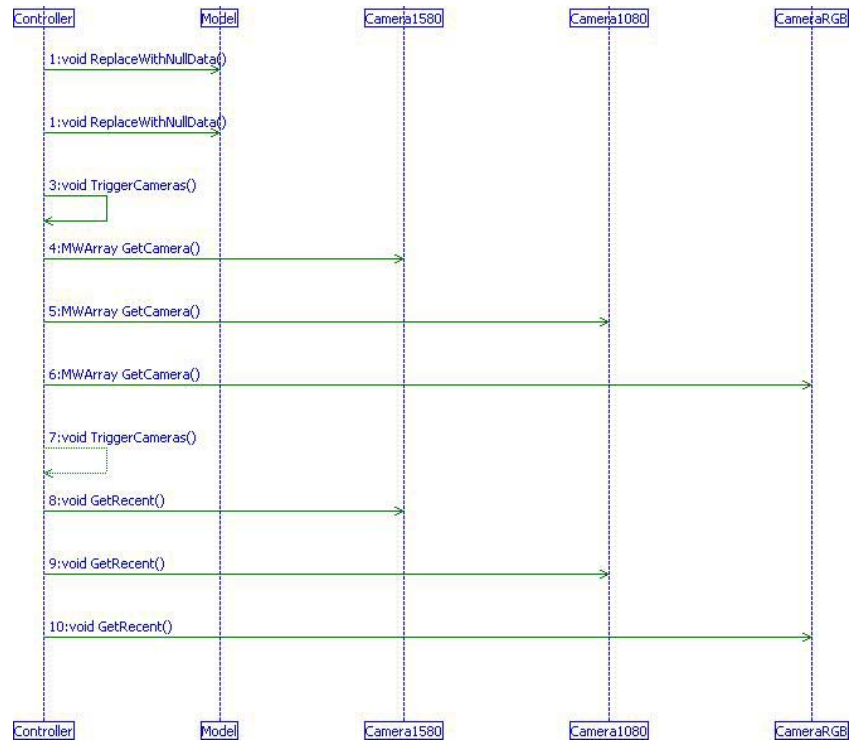
**Figure 13.** Controller.Run() sequence with no open outputs.
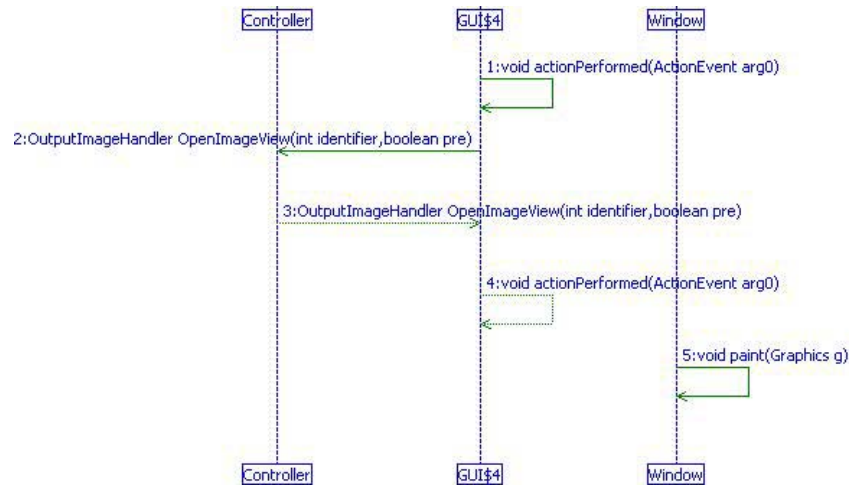


**Figure 14.** Sequence of calls from GUI to open a Window.
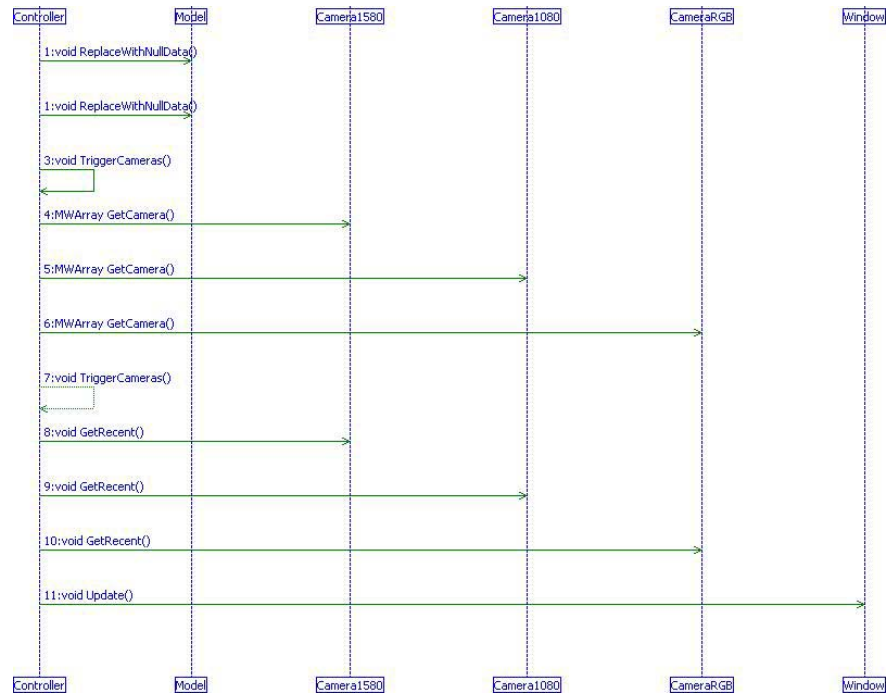
**Figure 15.** Sequence of Controller.Run() following the opening of a window to view raw images from a camera. Notice the addition of the final call, Window.Update(), that is not present in Fig. 13.

The sequence diagram in Fig. 16 shows the sequence of calls following an update call with an open window that is showing images with the rules-based skin detection algorithm, using NDGRI, NDSI, and ELM. This combination is chosen because it requires running multiple algorithms, showing the dependency structure.

Another important feature of this sequence is the call to the ELM algorithm (AlgorithmPre). This function call populates the second instance of the Model class, and is only run if an open output requires it. Because windows have no knowledge of which model they are actually using, AlgorithmPre is run directly from the controller and given access to both instances of Model.

The final feature of interest is the algorithm dependencies. Any open output has knowledge of which algorithm it relies on, and which image it uses in the model. If the image is

not updated in the model, the window calls the associated algorithm (in this case, AlgorithmRulesNDGRI). This algorithm only has knowledge of the image it directly relies on, and their associated algorithms. This approach ensures that no algorithm or output window has to have knowledge of more than its direct dependencies, which further avoids running the same algorithm multiple times if multiple windows are open.



**Figure 16.** Sequence diagram of Controller.Run() when an output Window showing images processed by the rules based, NDSI, NDGRI, and ELM algorithms.

Figure 17 shows the sequence to close an open window or file. This sequence is identical regardless if the output is a video file creator or a window. Not shown is that the GUI immediately removes it from its list of open outputs, but does not close the window thread itself until 500 ms later in order to prevent a race condition between the Controller and GUI over the OutputImageHandler from occurring. Without this delay, the Controller may attempt to access a thread that no longer exists in memory.
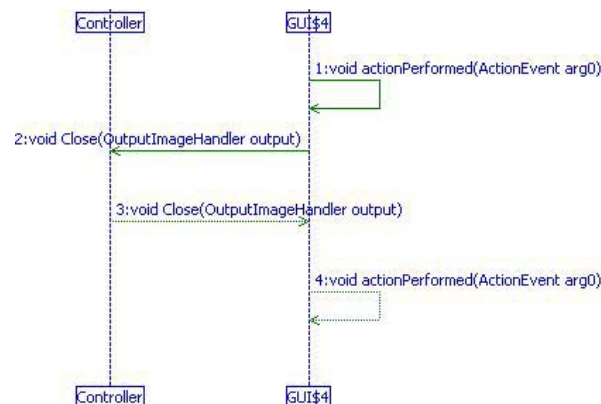


**Figure 17.** Sequence diagram showing the function calls that occur when a user selects to close an open window or file.

Another key piece of functionality for this program is the ability to set points in the scene containing the light and dark panels, so that the ratios required for ELM can be generated. Due to the requirement that the Processing Program be able to run all algorithms on the raw data but with different parameters, the Acquisition Program cannot save the generated ratios because those rely on parameters. Instead, it must save the exact coordinates at the moment the user sets them.

Coordinates for either panel can be set in any open window viewing the raw data. Right clicking sets the point containing the light panel, and left clicking sets the dark panel. After the points are set, the location of the light panel is shown as a light blue dot, and the dark panel as a blue dot. These dots will appear in every open window showing uncorrected images.

In every open video file, whenever a calibration point is changed, the VideoFileCreator.append() method will write a '-1' instead of the frame number in the video file, and then write both coordinates. The program will then write the subsequent picture from the beginning as normal.

Figure 18 shows two sequences related to establishing and removing calibration points. The first occurs when a point is clicked on the image. The second is when the "clear points" button on the main GUI is clicked.
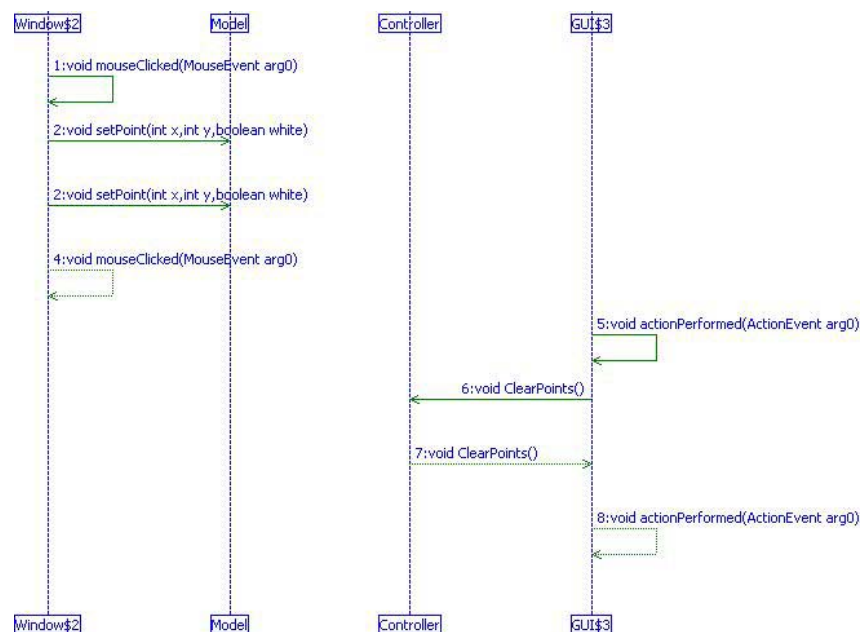


**Figure 18.** Sequence diagram showing the user setting points in the scene representing a panel (left), and clearing the points from the main GUI (right).

The final sequence diagram for the Acquisition Program, Fig. 19, shows the method calls that must be made when exiting the program. These are required to close the camera interfaces and properly close out all files and visual windows. If the cameras are not shut down correctly by the controller, then they may not be able to start back up correctly.
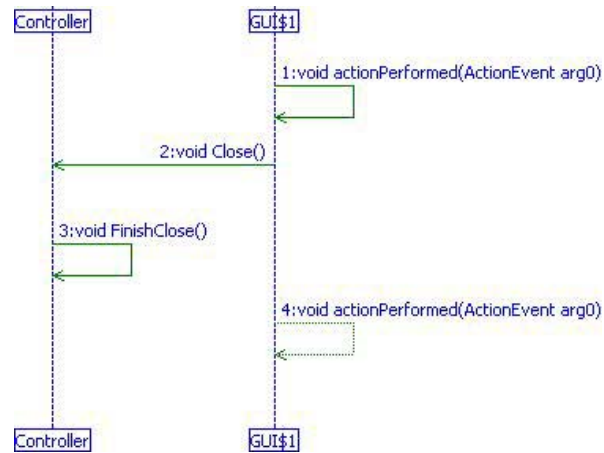
**Figure 19.** Sequence diagram for shutting down the system closing.

*3.2.3  Class Diagrams*

Figure 20 shows the final class diagram for the Acquisition Program.  The layering in the program is visible here.  The GUI package is at the topmost level, with the Controller directly between it and the remaining classes.  The DataHandlers package is below the Controller, but still above the Model and CameraAcquisition packages, which make up the bottommost layer. The CameraAcquisition package is the compiled Matlab code.

**Figure 20.** Class diagram of Acquisition Program. The application follows the Model-View-Controller pattern.

*3.2.4 General Discussion*

While the structure of the program is fairly modular and flexible, we did not take it as far as we could. For example, if a later programmer chooses to add another algorithm they must at least touch part of almost every class. While most of the edits would be straight forward, such as adding another item to a list, it still requires detailed knowledge of the system. However, if we chose to make everything completely modular, it would have significantly increased the size and complexity of the program, slowing down its operation. Therefore we have tried to strike a balance between expandability and simplicity.

*3.3  Processing Software Design*

The goal of this program is to allow the user to generate video files using different algorithms or parameters than those used in real time.  For example, in order to obtain the fastest frame rate of acquisition possible and still be able to generate all outputs later, the user must save the raw data streams and avoid running algorithms in real time.  However, they can later use this program to generate the required files.  It also allows the user to tweak the parameters, running algorithms on the same pieces of data multiple times.

The underlying design of this piece of software is very similar to the Acquisition Program.  However, there are three key differences.  The first is that the acquisition is from files instead of cameras and sensors.  The second is that there are no options to display streams, only to save them.  This is primarily because the input from the files and the processing will be done as fast as possible, not in relation to the rate that the images were taken.  The third primary difference from the Acquisition Program is that only one algorithm can be run at a time, and the user is responsible for selecting the correct files as required by that algorithm.

*3.3.1  Program Overview*

The video files generated by both the Acquisition and Processing Programs are saved as a sequence of pictures.  There is a header at the beginning of the file, with the dimensions of the image and the bit depth, as well as the initial X-Y coordinates of both panels as required for ELM.  The rest of the file is a collection of pictures.  Each picture consists of an array with the actual image, a frame number, and a timestamp of when the picture was taken.  If the frame number read is '-1', the four following integers are the new X-Y coordinates of the panels, and then the standard sequence resumes.

Since the video files from different inputs will almost invariably have been started at different times, they must be synchronized in order to get any useful results. The program first reads the picture from both files, and then steps through the file that starts first until the timestamp matches that of the other file. If the input files do not overlap, then the program does not produce an output file and warns the user.

*3.3.2 Sequence Diagrams*

This program is completely sequential, following a single path from start to finish. There is an option to exit when selecting the algorithm, however the program must otherwise continue along the single path.

The sequence diagram shown in Fig. 21 illustrates the method calls that take place when the program is first opened. The user must first supply the program with the output filename, before the main GUI opens. The actionPerformed() on FileHandler$2 is the user clicking the button to start the file save. The controller has one associated file handler, which is the output file. All input file handlers are controller by the algorithm. This is one primary difference from the Acquisition Program, where input, algorithms, and outputs are all associated directly with the controller.
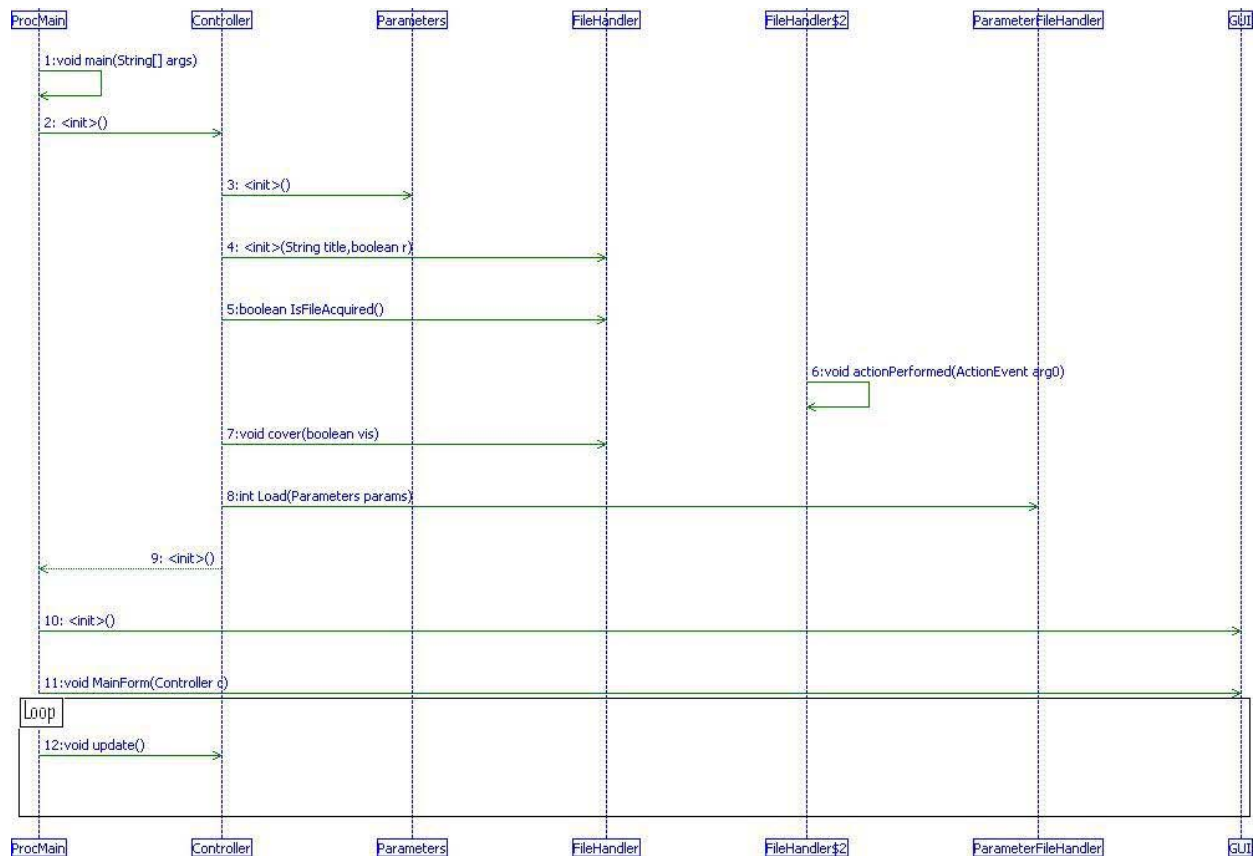
**Figure 21.** Startup sequence of the Processing Program. All work for the remainder of the program is performed by logic inside Controller.Update().

The sequence diagrams for the remainder of the program are not particularly helpful in understanding its operation. The program from here on out is a continuous loop. First it checks to see if an algorithm has been selected, then uncovers the file selection windows in turn, until all input files have been selected. Once the last file is selected, the program calls the synchronize() method, and then the Algorithm.Run() method on the selected algorithm. Once the end of either input file is detected, the program closes.

### 3.3.3 Class Diagrams

One of primary differences from the Acquisition Program is the replacement of the Model class with a Parameters class. It is basically identical to the model class except it does not

contain any pictures. Another difference is the FileHandler class replaces the Camera, Window, and VideoFileCreator classes of the Acquisition Program. There is also no CameraAcquisition class as there are no cameras. The class structure of the Processing Program is shown in Fig. 22.
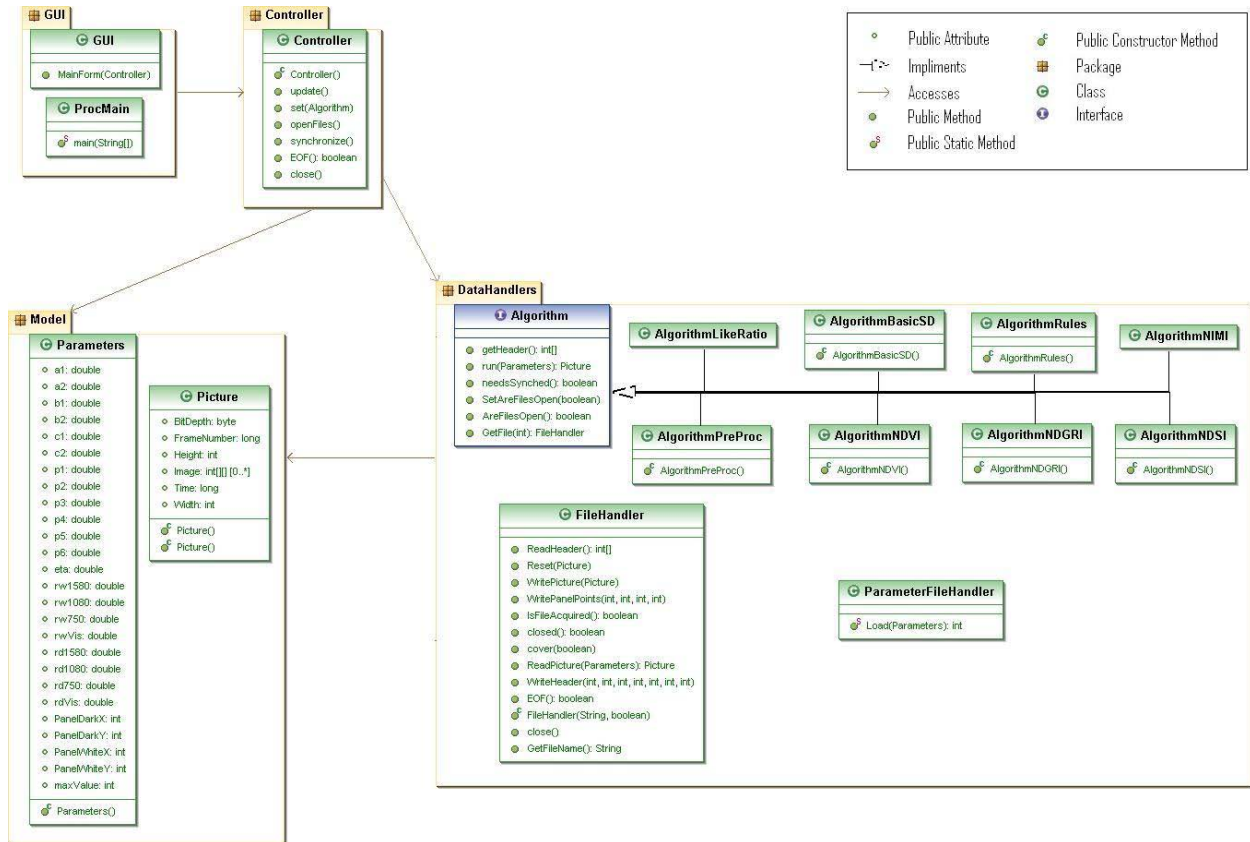


**Figure 22.** Class diagrams for the Processing Program. The Processing Program has a simplified version of the Acquisition Program class structure.

### 3.3.4 General Discussion

The Processing Program is much more straightforward, and less elegant than the Acquisition Program. It also requires the user to have knowledge of how each processing algorithm works and what information is needed, as there are no dependencies built into this aspect of the system. For example, to perform likelihood ratio based skin detection, the user must first know to run the NDSI and NDGRI algorithms. Because each video file does not store

what sort of image is saved in it, the user must keep track of what they store in each file (e.g., by using filename conventions).

*3.4  Playback Software Design*

The Playback Program is the simplest of the three programs.  Its only function is to open and display a video file, with basic functionality for video playback.  These functions include the ability to play, pause, and move around in the video.  It must also ensure that the playback speed corresponds with the speed it was recorded.  The source code borrows heavily from the FileHandler class of the Processing Program and the Window class of the Acquisition Program.

*3.4.1  Program Overview*

There are two options for moving in the video.  The first is by frame number, which moves the current view to that frame number, and the second is time, which searches the video for the closest timestamp (since the exact matching timestamp might not exist).  The frame number and time of the current image is displayed to the user with each frame.  In order for the video to display at the correct speed, the program looks at the difference in time between the current frame and next frame ($\Delta t$).  It then displays the next frame $\Delta t$ after the current frame.

*3.4.2  Sequence Diagrams*

Like the Processing Program, this is a one use utility.  A user cannot open the utility, and then start multiple video files from one instance.  The first sequence diagram for the Playback Program, Fig. 23, shows the procedure for opening a video file.
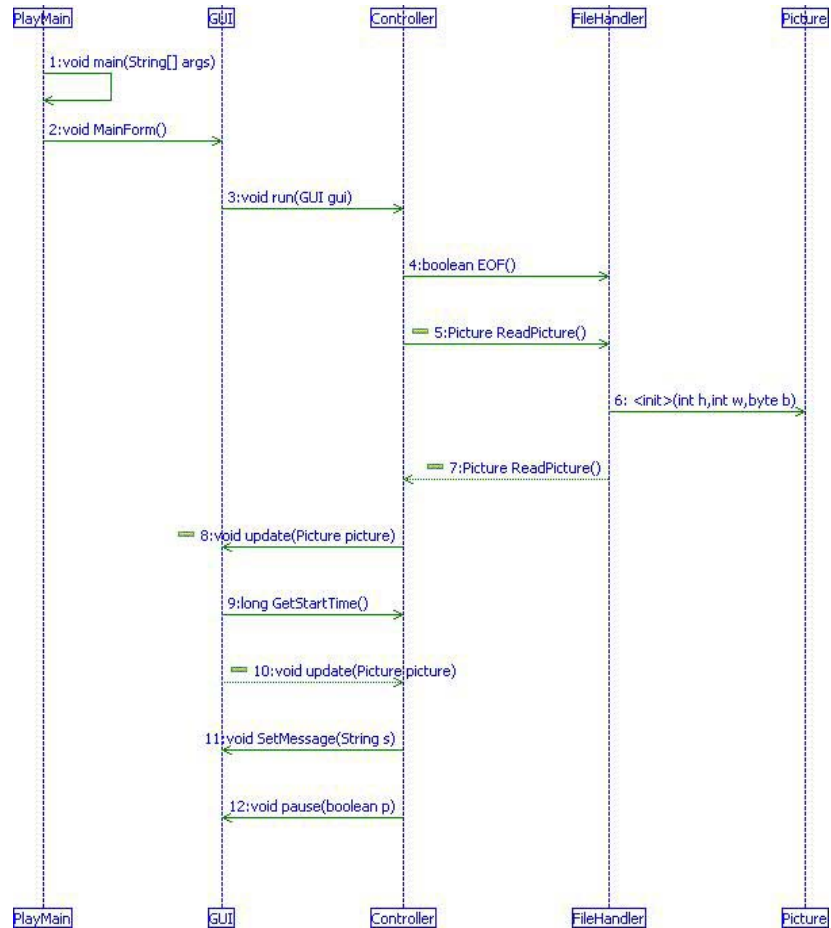
**Figure 23.** Sequence of opening the Acquisition Program. Notice there is a single input file opened, and no separate outputs.

After the user selects "play" to play the selected video file, the main program loop in the GUI makes a method call to Controller.Play() every cycle. This loop is only broken when the user selects some other function, or the end of the file is reached. Figure 24 shows the sequence of a calls made before the pause button is pressed, and after. When the GUI is updated, it no longer reads the most recent picture from the file, it just uses the current one over and over.
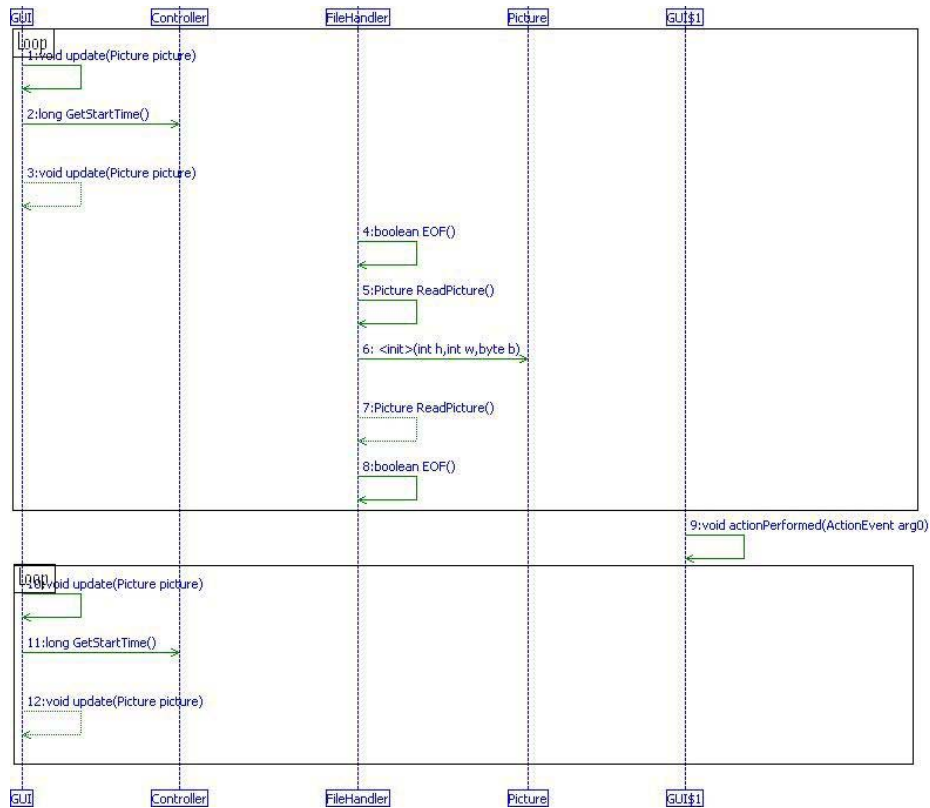
**Figure 24.** Sequence before and after the "Pause" button is pressed. Notice that no new pictures are read in the update loop following the pause action.

The most complicated functions of the program are the ability to move within a file. Due to the linear nature of the file layout, the searches precede based off a simple comparison of equality with the search parameter provided. Whenever either type of search is started, the file is closed and reopened to reset it to the beginning. If the selected frame number is before the beginning of the file, the first image in the file is displayed, along with a message informing the user of such and the number of the first frame from the file. The method continues to loop until either a match or the end of the file is found. The search by frame number sequence is show in Fig. 25. Searching for a particular time is virtually identical to searching for a frame number; the only logical difference is the value being compared.
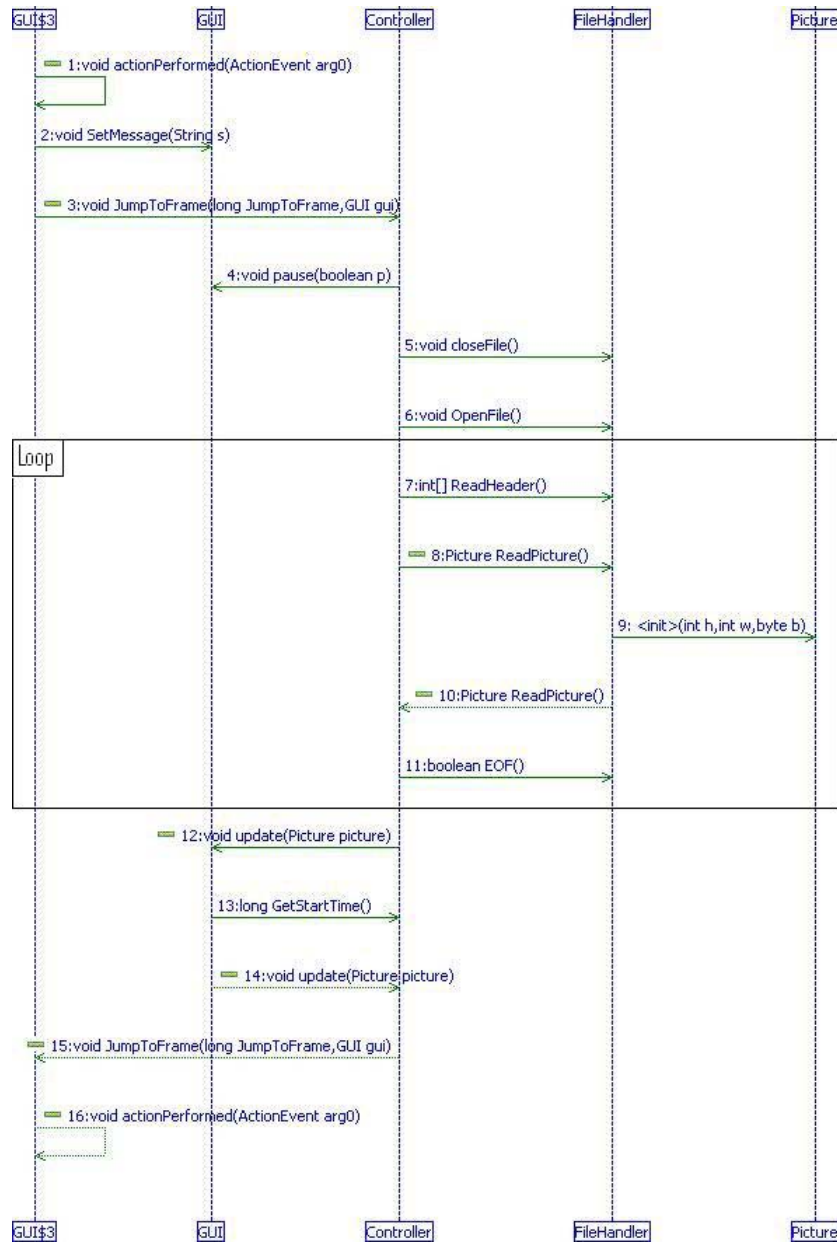
**Figure 25.** Sequence of calls following a "Jump" command. The Playback Program will close, then reopen the file and search for a matching frame.

Figure 26 shows the sequence of method calls when the exit button is pressed. Since there are no hardware inputs or outputs, the only requirements are to shutdown the input file and close the GUI.

3-24
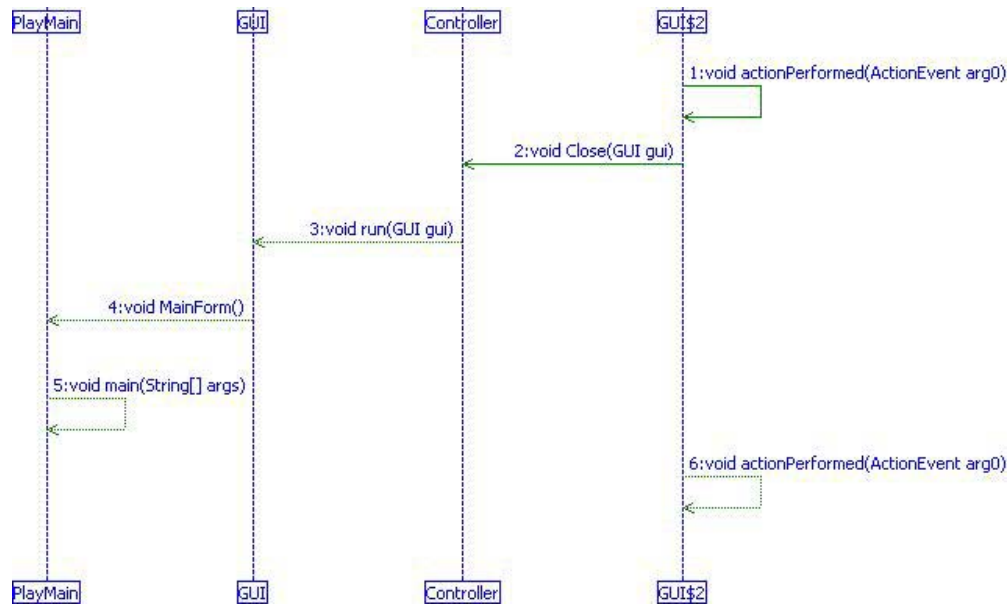
**Figure 26.** Sequence following the exit command.

### 3.4.3 Class Diagrams

Figure 27 shows the Playback Program class diagram. The package layout is identical to the Processing Program and similar to the Acquisition Program. However, the DataHandlers class is significantly smaller as there is only one input class, and no output classes. Furthermore, since no algorithms are executed, there is no Model or Parameters class in the Model. The final important feature is the dependency loop between the Controller and the GUI. This is because the GUI is not only the asynchronous input of the program, it is also the synchronous output.
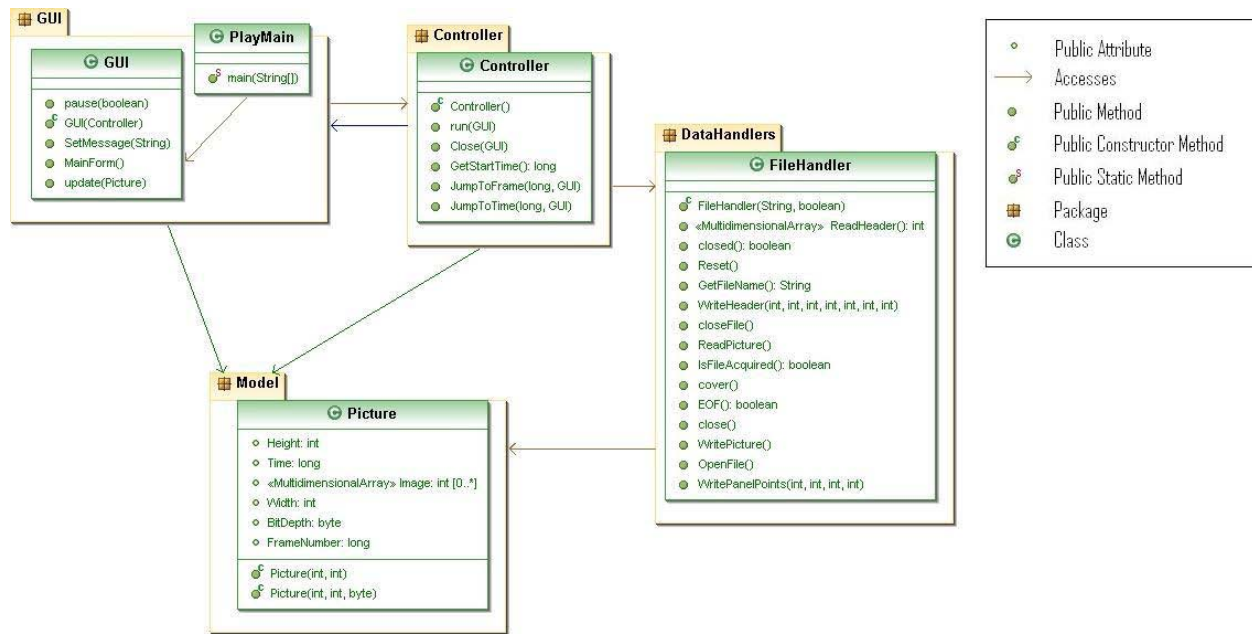
**Figure 27.** Playback Program class diagram. Notice the significantly reduced DataHandlers and Model packages, and the circular dependency between the Controller and GUI.

### 3.4.4 General Discussion

The two biggest issues are the circular dependency between the Controller and GUI, and the extremely slow search time when moving around in a file (approximately 40 fps). While it would be difficult to solve the circular dependency without completely redesigning the program, the slow search time has been identified as an item for future work. While changing the design of jumping to a particular time would be problematic as the time between frames is not constant, jumping to a frame number could be changed to skip over the required number of bytes in the file instead of reading through (although keeping track of reference panel frames could be an issue).

Another problem that has been identified is the slow playback when multiple instances of the program are running simultaneously. This is because the hard drives simply cannot keep up with the demand. Assuming the hard drive switches between video files exactly once per frame,

which is a reasonable assumption due to the applications waiting period between frames while playing, the time to read a frame on the current system is 16.83 ms. This means that only five players can be running at a time to maintain the current frame rate of 10 fps. If time advance searches are being performed, the hard drive attempts to switch between files more, causing the frame rate to decrease. A fix for this would be to upgrade to significantly faster hard drive setup, such as 10,000 rpm drives in RAID, or a solid state drive setup. The potential read time per frame could be reduced to approximately 4.2 ms per frame with a solid state drive on a SATA II interface, even faster if used in striped RAID.

*3.5 Summary*

The construction of the software is fairly straightforward, except for the multithreaded aspects of the code. It provides a reasonably modular and expandable system while maintaining a smaller program size. The Java coding also allows one to place the system on any computer that can access the camera interfaces through Matlab, and does not require the code to be recompiled or significantly modified.

## IV.  Testing and Analysis

### 4.1  Testing and Flaws

The software was designed using a bottom up approach.  Testing is accomplished for all possible program branches occurring at each level.  While no single test plan has been developed and implemented, the structure has been tested in data collections and any errors found have been fixed or recorded.

### 4.1.1  Known Errors in Software

The following sections describe, in detail, known errors with the system as implemented. These following errors are limited to the Acquisition Program, and include periodic output file corruption, errors when opening or closing outputs, and an improper refresh of the viewing window.

### 4.1.1.1  File Corruption in the Acquisition Program

The most serious error is a sequence of seemingly random numbers that are sometimes written to a video file by the acquisition program when a file is first created.  This occurs in one out of every approximately ten files, and we have been unable to correlate it to any particular event and thus have been unable to find the cause of the error.  The only correlation found is that it tends to occur more frequently as the number of open windows is increased.  In some cases, the extra integers can be removed and the rest of the file copied into a new file.  However, most of the time, the files appear to be irrecoverable.  The simplest way to check for a corrupted file is to open it in the playback program, and if instead of playing a video, the screen says "waiting for image", then the file is most likely corrupted.

*4.1.1.2  Concurrent Modification of Open Outputs in the Acquisition Program*

If the Acquisition Program is running slowly, it is possible for a concurrent modification exception to occur. This error occurs because the main loop keeps a separate list of open windows, copied once per frame from the GUI's master list. If the GUI closes a window while the controller is in the middle of updating it, the program will hang. A delay of 500 ms to open or close a window is added to prevent this from occurring. However, if the frame rate drops below 2 fps, this problem can still occur.

The solution to the problem is to better synchronize the threads, not just provide a set delay. We tried to use a set of semaphores for thread synchronization, however, they took too long and system performance decreased to a point that is was unable to meet performance requirements.

*4.1.1.3  Improper Refresh in Window Display of Acquisition Program*

The window output of the acquisition program does not correctly refresh the blank space at the bottom of the window, in the area around the label and checkbox, whenever other windows are dragged overtop of it. We are unsure why Java is not handling this part of the GUI properly, but it is not a performance hindering flaw so we have not investigated the error any further. An example of this issue is shown in Fig. 28. Notice the bottom of the window, there is a streaked image from another window being dragged across, as seen inside the red box in Fig 28.

**Figure 28.** Window showing the improper refresh error.

*4.1.2 Known Shortcomings in Software*

*4.1.2.1 Reference Panel Selection*

Currently, when setting the location of a white or dark panel in the acquisition program, only a single pixel is selected for each. This should be extended to include the selection of a rectangular region of pixels. The program should be modified so the user drags a rectangle across the panel, and the program uses the average of all points in the rectangle. Using the mean will result in a cleaner estimate of the true panel properties as the noise is averaged out.

Another shortcoming is the inability to change or set new panel points in the processing program. The parameters used to calculate the ratios can be changed, but not the locations themselves. Therefore, if a user saves the raw images but does not select the panels, or selects the wrong location, there is no way to correct the mistake later.

*4.1.2.3 Difficulty Closing Out of the Processing and Playback Programs*

Difficulty closing the Processing and Playback Programs is not so much an error as a minor design flaw. In the processing and playback programs, whenever a file selection window is open, there is no other GUI visible, and the selection window has no option to close. In the

acquisition program, windows can be closed through the main GUI, but in the other programs, the user must continue inserting all necessary filenames. There should be a cancel button added that cancels everything the program was doing and exit cleanly.

### 4.1.2.4 Slow Jump Times in Playback Program

Because the playback program must read sequentially though a file to find the matching frame, searching for a point near the end of the file can be extremely time consuming, since even moderate length video files are very large. There is basically no way to avoid this when jumping to a time because the time between one frame and the next is not consistent and therefore cannot be pre-calculated. However, when jumping to a frame number, the current frame number is known, as is the size of each image. The program should be changed to skip over the required number of bytes in the file rather than reading them, tremendously increasing the speed of jumping to a frame of interest.

### 4.1.2.5 Inefficient Parameter Modification

Currently, the only method of editing parameters is to open the associated text file, which is a comma separated list of variable names and values. The user can edit the file, save it, then click the "Update Parameters" button on the main GUI of the acquisition program. This file is also used by the post processor, so any changes are reflected there as well.

It would be helpful in the future to have a dedicated GUI window to edit parameters, where changes are instantly reflected in the outputs. This approach has the distinct advantage would decrease the direct knowledge of the system and algorithms that the user would need to know. Furthermore, it would be faster to make parameter changes for the algorithms.

*4.1.2.6 RGB Camera Issues*

One major limitation to the system is the color camera. First, this camera is extremely slow. It takes approximately 87 ms to acquire each frame, as discussed in Section 4.3.1. The longer time compared with the NIR cameras is not unexpected, as the RGB camera acquires eight times the amount of data as each of the NIR cameras (24 bit per pixel versus 12 bits per pixel, and four times the spatial resolution). Lowering the resolution of the camera could speed this up significantly, but this function only exists in the code provided by the manufacturer and not in the Matlab camera interface.

The other problem with this particular camera is that it will only work on a 32-bit computer. All other parts of this system will function on the newer 64-bit computers. This is, in part, responsible for the reduced algorithm performance and reduced acquisition frame rates. System performance could be significantly improved if this camera was upgraded.

*4.2  System Demonstration*

The system is demonstrated using the parameters specified below in Table 6 and Table 7. Table 6 shows the $F_i$ parameters, which each have six arguments, used by the Likelihood Ratio Test, while Table 7 shows all the single argument variables used by all the algorithms.

**Table 6.** Likelihood Ratio Test parameters used in demonstration of the programs.

| Variable Name | i=1 | i=2 | i=3 | i=4 | i=5 | i=6 | i=7 |
|---|---|---|---|---|---|---|---|
| P | 0.0501 | 0.4286 | 0.5213 | 0.2966 | 0.2034 | 0.3733 | 0.1267 |
| a | 0.0331 | 0.0164 | 0.0076 | 0.0808 | 0.0868 | 0.1408 | 0.0059 |
| b | 0.0016 | 0.0002 | -0.0006 | -0.0111 | -0.0145 | -0.0026 | -0.0006 |
| d | 0.0161 | 0.0153 | 0.0078 | 0.0202 | 0.0174 | 0.1981 | 0.0138 |
| µs | 0.7318 | 0.7008 | 0.5548 | 0.3446 | 0.0875 | 0.2332 | 0.8983 |
| µg | -0.5921 | -0.3185 | -0.2306 | 0.4085 | -0.1872 | -0.0441 | 0.0063 |

**Table 7.**  Basic parameters used in the demonstration of the programs.

| Algorithm | Parameter | Default Values |
|---|---|---|
| Rules Based | b1 | -1 |
| | b2 | -0.15 |
| | c1 | 0.5 |
| | c2 | 1 |
| Likelihood Ratio | eta | 0.028 |
| ELM | rw1580 | 0.987 |
| | rw1080 | 0.989 |
| | rw750 | 0.99 |
| | rwRed | 0.99 |
| | rwGrn | 0.99 |
| | rwBlu | 0.99 |
| | rd1580 | 0.131 |
| | rd1080 | 0.108 |
| | rd750 | 0.09 |
| | rdRed | 0.077 |
| | rdGrn | 0.075 |
| | rdBlu | 0.073 |

*4.2.1  Acquisition Program*

Figure 29 contains a screenshot of the main GUI for the program.  The disabled rows of check boxes rely on the 750 nm camera.  Opening and closing windows and video files is accomplished with the check boxes on the GUI.  The left two columns are for viewing and saving raw images from the cameras.  The right two columns are for viewing and saving images that have been processed through the ELM algorithm.  The "RGB" row does not have a save option because it requires three images at once, which is not possible using the current video file format.  However, the three colors can each be saved separately, allowing one to have access to the full color data for post processing for future algorithm exploration in other applications (i.e. Matlab).

**Figure 29.** Acquisition Program main GUI. The disabled rows rely on the 750 nm camera not currently present in the system.

One of the most important requirements of the program is to be able to correctly implement the algorithms. Figures 30, 31, and 32 show the outputs of the Basic Skin Detection, Rules Based, and Likelihood Ratio Algorithms respectively. The associated color image is shown in Fig 33. The background of the scene contains a large amount of snow. Snow is a known skin confuser and provides a reasonable demonstration of the detection portion of the system. The goal of this set of images is to qualitatively show the algorithms are working as designed. All three of the algorithms are being run on images that have been through the ELM

algorithm.  Both the grey and white panels used for calibration are visible on either side of the subject of Fig. 33.

The result of the basic skin detection algorithm on the scene is shown in Fig. 30, where one clearly sees that the Basic Skin Detection algorithm is working.  All exposed skin is clearly between the thresholds defined as $c_1$ and $c_2$ in Table 7.  There are some highlighted shadows and edges visible, but most important is the large amount of snow behind the subject that is declared as skin.  However, this is expected from this algorithm, showing that it is functioning correctly.

Figure 31 shows the result from the rules based algorithm and the parameters shown in Table 7.  The most prominent difference between the basic skin detection and the rules based detection is that the latter does not declare the snow as skin, resulting in the black background.  Also notable is the reduced highlighting in shadows and edges, although the tripods and feet still have some minor edge effects.  Another point is that shadowed skin is not fully highlighted.

Figure 32 shows the Likelihood Ratio algorithm.  The algorithm is clearly working, the skin is highlighted more than the surrounding environment.  We were unable to achieve better results by simply adjusting the value of η.  While there is reduced highlighting of shadows and edges over the basic detector, there are still many false detections from the snow.  Furthermore, the algorithm does not detect skin as well as the rules based detector, particularly on the face.

**Figure 30.** Output of the basic skin detection (NDSI only) algorithm and ELM. Notice the large number of snow pixels that are declared as skin.
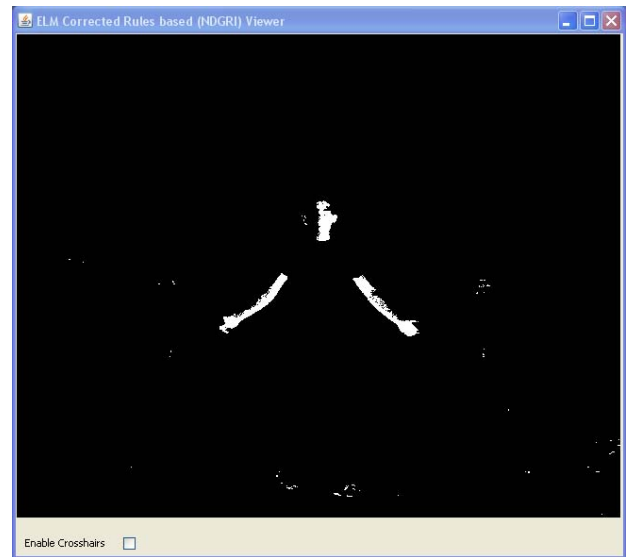


**Figure 31.** Output of the rules based (NDGRI and NDSI) Detection Algorithm and ELM. Notice that the snow is not declared as skin in this image.



**Figure 32.** Output of the likelihood ratio test algorithm and ELM. Notice that snow is still declared as skin, shadowed regions of skin are not identified, and the corners of the image are also declared as skin.
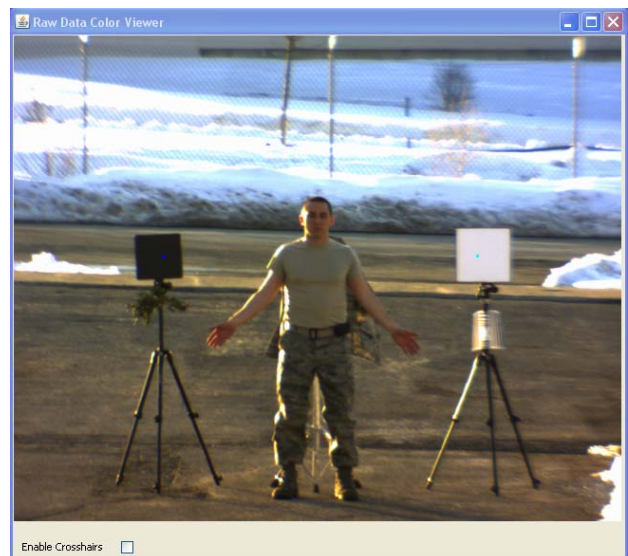


**Figure 33.** Color image of the test scene. The panels used for ELM are visible beside the subject.

Figures 34 through 37 show the operation of the ELM algorithm. Figure 34 shows the raw color image from the RGB camera. Looking closely reveals a light blue dot on the white panel, and a dark blue dot on the dark panel (these are also visible in Fig. 33). The dots represent the coordinates from which the ELM ratio computations in Eqn. 12 and Eqn. 13 are computed. Figure 35 shows the ELM corrected color image. Also notice that the dots do not appear in the corrected images (the points also cannot be set in the corrected image windows).

Another feature of importance is power thresholding based off the 1080 nm camera. Whenever a pixel is below a threshold in the 1080 nm image, the same pixel in all the other images are set to black, to help avoid false detections due to noise. The 1080 nm camera was chosen because skin should be lighter than the surroundings at this wavelength, so dark regions should not be skin, and thus can be safely eliminated from consideration. This is responsible for the black regions in the upper corners or Fig. 35.

Figures 36 and 37 show images from the same scene as in Fig. 34 but the images are from the 1080 nm camera. Notice that while the ELM correction is not extremely visible on the color images, where it is just a slight color shift, the difference between corrected and uncorrected images from the 1080 nm NIR camera is significant.

The final demonstration of the Acquisition Program is the ability to save video streams. The truest demonstration of this is to be able to use them correctly in the Processing and Playback Programs, as is shown in following sections. Figure 38 and 39 show the dialog for video file creation. The first box, Fig. 38, appears when the associated checkbox is checked, but the file only begins to save after the "Save" button is clicked. Once the save is started, the second box shown in Fig. 39 remains visible until the save is ended by un-checking the box in the GUI.
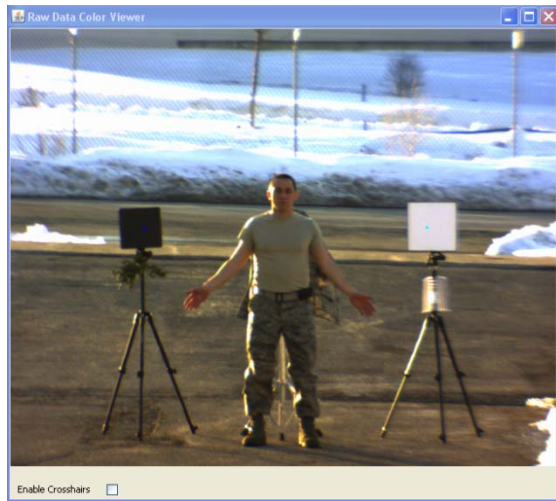
**Figure 34.** Raw color image.
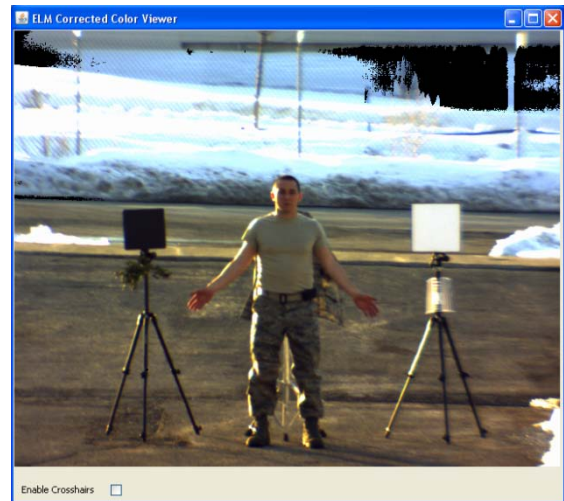


**Figure 35.** ELM corrected color image.
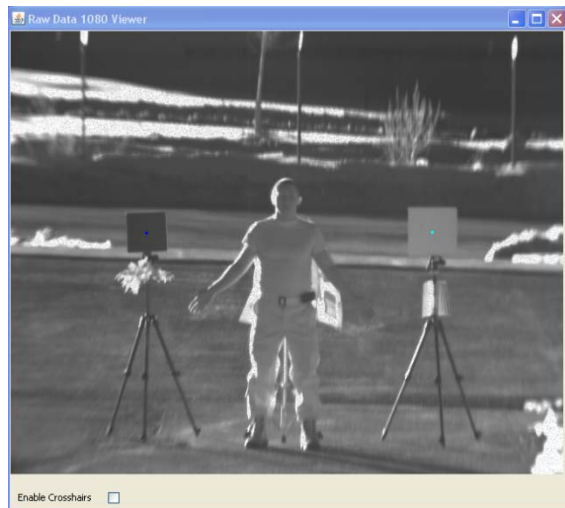


**Figure 36.** Raw 1080 nm image.



**Figure 37.** ELM correction 1080 nm image.



**Figure 38.** File selection window.



**Figure 39.** Window visible during file creation.

*4.2.2  Processing Program*

The Processing Program is slightly more difficult to demonstrate the correctness for when compared to the acquisition program, as there is no visible output.  All outputs must be checked in the playback program after they are generated, and compared against results saved from the acquisition program.

Figures 40, 41, and 42 are screenshots showing the program flow.  The input file window shown in Fig. 42 may be shown multiple times depending on the algorithm.  The required frequency or image type for each input is shown in the title bar of the window.
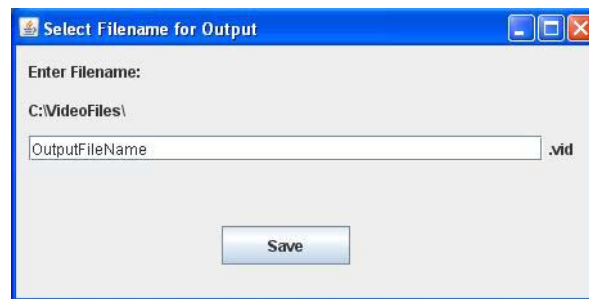


**Figure 40.**  File selection window for output file.



**Figure 41.**  Algorithm selection window.



**Figure 42.**  File Selection Window for input file.  The file type is specified in the title bar.

*4.2.3  Playback Program*

   While this is the simplest of the programs, it is still vitally important.  It is also currently the only method of confirming the correctness of the video creation functionality of the acquisition program, and the correctness of the post processor.

   To help show the functionality of this program, and of the post processor, we have chosen to show both the direct NDSI and NDGRI outputs in the player window.  These files can only be generated by the post processor, because the acquisition program can only save and display either direct data or the output of a higher level algorithm.

   Figure 43 shows the first window presented to the user, allowing them to select the video file.  Figures 44 and 45 show the player window itself.  The scene in Fig.44 and Fig. 45 is the same as that used in Figs. 30-37.

   Figure 44 shows the NDSI results from 1580 nm and 1080 nm video files.  Both inputs were previously run through ELM.  Both skin and the snow are clearly visible as brighter white than all of the surroundings, as expected.

   Figure 45 shows the NDGRI results from 660 nm and 540 nm videos, run through ELM first.  Skin appears darker than the surroundings, representing negative values.  The snow in the scene is still bright.  One of the issues with NDGRI is immediately visible from the figure. NDGRI tends to have a much lower contrast than NDSI, making it harder to discriminate features than with the NDSI.
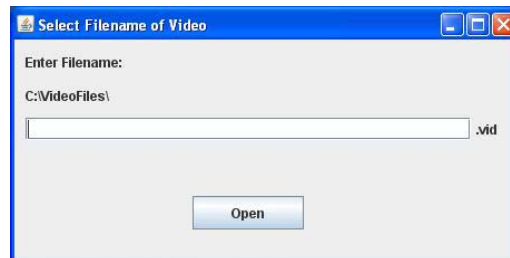


**Figure 43.**  File selection window of Playback Program.
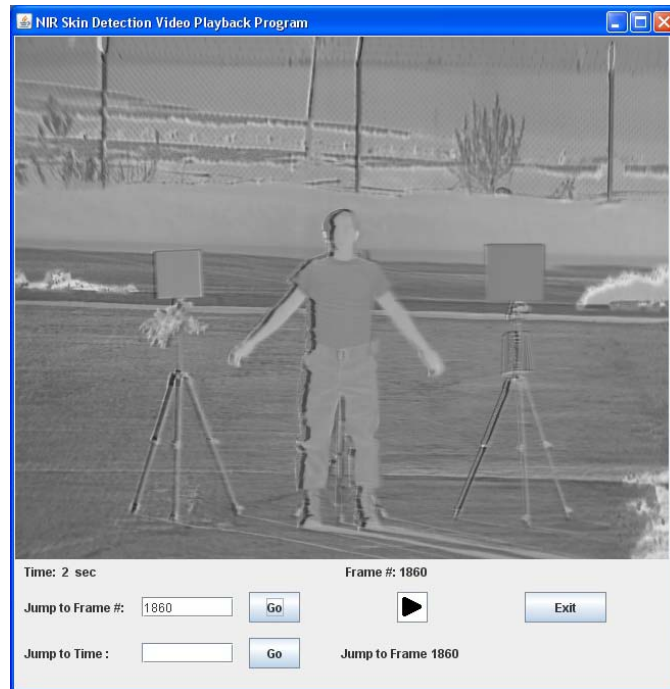
**Figure 44.** Main GUI of Playback Program showing an image of NDSI values. Both skin and the snow in the background have high values.
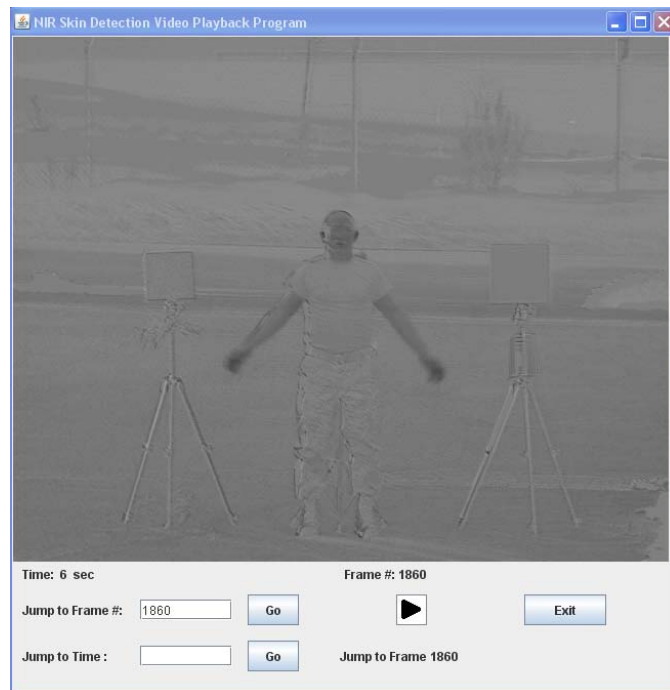


**Figure 45.** Main GUI of Playback Program showing an image of NDGRI values. Skin has a lower value than most of the surrounding environment, while the snow has a higher value.

*4.3  Performance Analysis*

*4.3.1  Camera Performance*

In order to analyze the performance of the cameras, we recorded the time to acquire a frame from a camera during standard runtime, with no algorithms being run.  We averaged the times over 1000 acquisitions, as shown in Table 8.  Why the difference between the acquisition times of the 1080 nm and the 1580 nm is not known for certain, it is possible that the because all three cameras are triggered immediately before the 1580 nm cameras images is read it may have to wait some additional time for the image to be ready, which is not the case for the 1080 nm camera.  It is, however, clear that the color camera is significantly slower than the NIR cameras.

**Table 8.**  Acquisition time per image for each camera.

| Camera | Acquisition Time (ms) |
|--------|-----------------------|
| 1580 nm | 27.8 |
| 1080 nm | 18.0 |
| RGB | 87.8 |

*4.3.2  Algorithm Performance*

In order to compare the performance of each algorithm, we have compared the number of mathematical operations in each.  The mathematical analysis are based on the "as written" math, and do not take into account any optimizations or additional operations added at compile or runtime.

As a secondary comparison of efficiency, we analyze the length of time it takes to perform each algorithm, averaged over 1000 runs.  While these times are not universal and will change depending on the machine the program is run on, and the amount of load on the system at that particular time, they do provide a relative comparison of one algorithm to another.

As the program is written, there are also steps outside of the algorithm itself, such as copying images in memory, calculating new times and frame numbers, etc. The ELM in particular must perform a deep copy of all the raw images, which is very time consuming but does not appear in the mathematical analysis. The mathematical operations only take into account the operations that are done on each and every pixel.

The times shown in Column 2 of Table 9 are the times required only to run that particular algorithm, and not the sub-algorithms prior to it. The estimated total times shown in Column 3 take into account all prerequisite algorithms, including the optional ELM. Neither of these times takes into account the acquisition time of the cameras, or the output times, or any other program function. They are purely the times required to run that particular algorithm on one image.

**Table 9.** Mathematical Complexity of the algorithms, and the time to complete each algorithm, as run on one particular machine. The * indicates that the first value is as implemented, and the second value is if the 750 nm camera is added to the system.

| Algorithm | Time (ms) | Total Time (ms) | Addition | Multiply | Divide | Compare | Power | Prerequisites |
|---|---|---|---|---|---|---|---|---|
| NDSI | 6.44 | 39.27 | 2 | | 1 | | | {ELM} |
| NDGRI | 8.60 | 41.43 | 2 | | 1 | | | {ELM} |
| NDVI | N/A | N/A | 2 | | 1 | | | {ELM} |
| NIMI | N/A | N/A | 5 | 8 | 1 | | 4 | {ELM} |
| BasicSD | 7.09 | 46.36 | | | | 2 | | NDSI |
| RulesBased | 4.24 | 52.11 | | | | 4 | | NDSI, NDGRI/NDVI |
| LikeRatio | 53.39 | 101.26 | 68 | 84 | 1 | 1 | 1 | NDSI, NDGRI |
| ELM | 32.83 | 32.83 | 5/6* | 5/6* | | 10/12* | | |

*4.3.2.1 NDSI*

The NDSI algorithm takes 2 adds and 1 division per pixel. The total time to perform the algorithm is 6.44 ms.

*4.3.2.2 NDGRI*

The NDGRI algorithm takes 2 adds and 1 division per pixel. The total time to perform the algorithm is 8.60 ms. This time is somewhat odd because the algorithm is completely identical to NDSI except for the images from the model that are used. However, we ran the experiment several times with similar results, so it is not a fluke, but we have no explanation for the extra 2 ms.

*4.3.2.3 NDVI*

Because there is no 750nm camera, we cannot run the NDVI algorithm to determine the time it takes. However, it is virtually identical to the NDSI and NDGRI mathematically, so it would take about the same length of time. It also takes 2 adds and 1 division per pixel.

*4.3.2.4 NIMI*

As with the NDVI, the NIMI algorithm cannot be run until the 750 nm camera is connected to the system. Mathematically, this algorithm required 8 multiplications, 1 division, 5 additions, and 4 calls to Math.pow(), which raises a number to a power. Without visibility into this function, we do not know the runtime of this operation.

*4.3.2.5 Basic Skin Detection*

Basic skin detection requires that NDSI algorithm also be run, and then takes an additional 2 comparisons per pixel. The additional time required to run is 7.09 ms. This makes the total start to finish time for the algorithm approximately 13.53 ms.

*4.3.2.6 Rules Based Algorithm*

There are two virtually identical algorithms, the Rules Based Detector with NDGRI, and Rules Based Detector with NDVI. Because the 750 nm camera is not operational, the NDVI version is disabled, and this is not analyzed here.

This rules based algorithm requires that NDSI and either NDGRI or NDVI both be run, and then takes an additional 4 comparisons per pixel. The additional time required to run is 4.24 ms, as shown in Table 4. This makes the total start to finish time for the algorithm approximately 19.28 ms.

Another point that is unclear is why this algorithm is almost 3 ms faster than the Basic Skin Detection algorithm. The two algorithms are nearly identical except that the Rules Based algorithm must make more comparisons, using two images as input instead of one. If anything it should be slower, but multiple tests still show it as faster.

*4.3.2.7 Likelihood Ratio Test*

The Likelihood Ratio test algorithm requires that NDSI and NDGRI both be run, and then takes an additional 70 additions, 84 Multiplies, 3 divisions, 1 exponential, and 1 comparison per pixel. The additional time required to run is 53.39 ms, as shown in Table 4. This makes the total start to finish time for the algorithm approximately 68.43 ms. If it is combined with ELM, the runtime is over 100 ms, resulting in a dramatic depreciation of system performance.

*4.3.2.8 Empirical Line Method*

As has been discussed elsewhere in the thesis, the ELM algorithm is somewhat different than the other algorithms. When it is run, it generates an entire second model, not just a new image. This algorithm requires 1 addition, 1 multiplication, and 2 comparisons for each wavelength. The current system has five input frequencies, so the total cost is 5 additions, 5 multiplications, and 10 comparisons. If the 750 nm camera is added, the cost will be 6 additions, 6 multiplications, and 12 comparisons. The total computational cost is 32.83 ms, with an estimated 39.4 with the 750 nm camera. This algorithm does not depend on any other algorithms, but will be run if the user selects to run any other algorithms on its output.

*4.3.3 Algorithm Scalability*

All of the algorithms implemented run on O(n) complexity, where n is the number of pixels. Therefore, doubling each dimension of the camera resolution will cause a four-fold increase is runtime. Figure 46 shows the estimated runtime for various common image dimensions, with the Rules Based Detector and the Likelihood Ratio detector, both using ELM. These are estimated times, not results from actual experiments.

If we assume that the acquisition and output times are linear; meaning the operation of the entire system will scale proportional to the number of pixels in the images, we can get a rough estimate of the effect larger images have on system performance. For example, the current system runs at approximately 4 fps with the rules-based algorithm, and 2 fps with the likelihood ratio test algorithm. If the camera resolution is upgraded to 2048x2048, without upgrading the computer hardware being used, the frame rates will drop to approximately 0.29 fps and 0.14 fps, respectively, which is far below desired performance.
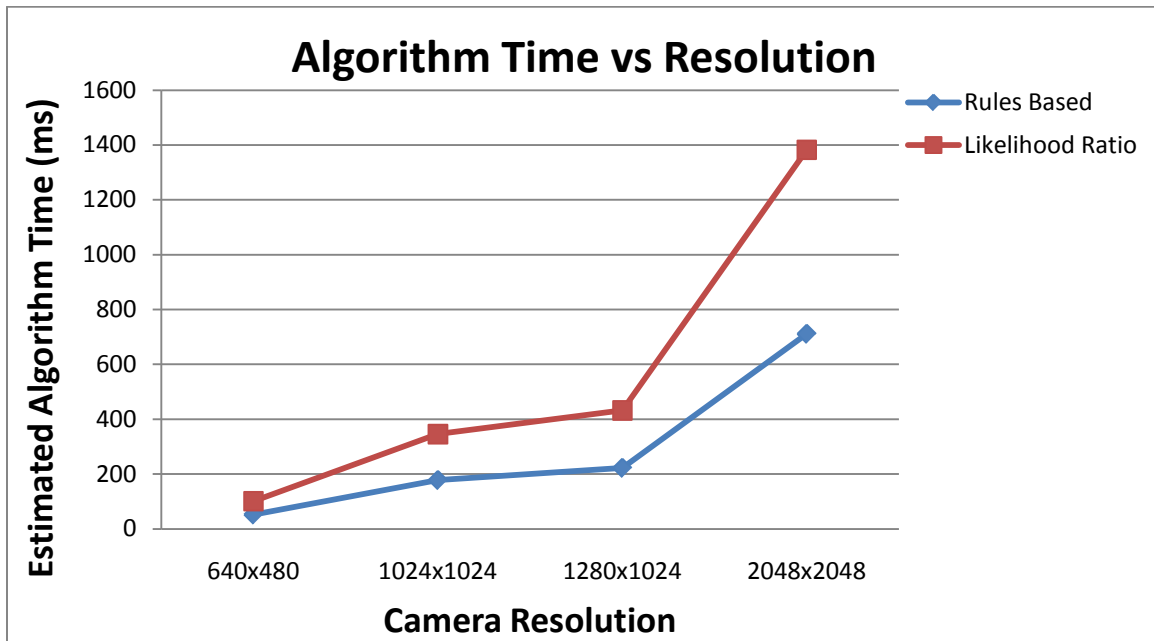


**Figure 46.** Estimated algorithm run times for various common camera dimensions. Both algorithms are running ELM on all data first.

*V. Discussion*

*5.1 Design and Methodology*

The system is designed with three distinct programs. The first is a versatile Acquisition Program that can acquires images and view, save, and run algorithms on them in real time. The next program, the Processing Program, can run algorithms and generate new videos based off of saved video streams from the Acquisition Program. The final program is a Playback utility for the video files.

All programs were generated using a bottom up approach, and using UML modeling practices. Basic use cases were developed for each program, then sequence diagrams, class diagrams, and finally the programs themselves. The bottom-up approach was used as it was necessary to determine exactly what was required to access the cameras and the shape of the data they returned, and build the rest of the system around that information. All three programs are modeled after the Model-View-Controller pattern. However, the programs are all multithreaded so the pattern is not strictly adhered to.

The acquisition program was built first, with the other two programs adhering to the standards and limitations it created. Testing was performed through usage. By using the program in data collections, whenever a bug was found, it was dealt with. There are still a few known minor errors, listed in Section 4.1.

*5.2 Results and Performance*

The algorithm outputs are functioning as expected, and as shown in the figures in Section 4.2. Algorithm run times and complexity are shown in Section 4.3, and are reasonably close to expectations. Empirical Line Method (ELM) and the Likelihood Ratio Test are computationally

intensive and both take a long time, whereas all the feature calculations and Rules Based algorithms are much faster.

On the current machine, the frame rate is approximately 10 frames per second (fps) when under light load, with a reasonable load where only a few simpler algorithms are run, the frame rate drops to about 5 fps. If all algorithms and multiple windows and video file saves are in operation, the system can run as slow as 0.5 fps.

## 5.3 Future Work

Needed changes and fixes were noted in Section 4.1. However, there are several major expansions that we have recognized the need for.

### 5.3.1 Exporting Images to Matlab

A utility should be developed to export individual images in a format that can be read in Matlab. Having the image available in Matlab would allow for numerical analysis and manipulation of the actual values of the image, not just as a visible image.

### 5.3.2 Addition of the 750 nm camera

Most of the required coding and algorithms are already present in the program but disabled. Adding support for the 750 nm camera would allow for the use of NDVI, NIMI, and the rules based detector with NDVI.

### 5.3.3 Target Different Language Environment

Java's advantage lies in its flexibility. Programs can be run on multiple different operating systems and hardware platforms. However, the advantage is lost with our camera system because some of the drivers are windows specific. Therefore, if the system as designed were developed in C++ or another similar language, it could run faster, potentially up to 30 fps.

### 5.3.4  Implement Imperx Laptop Cards

The current National Instruments camera link cards use a PCI interface, and are therefore limited to desktop machines.  The Imperx camera link cards are an ExpressCard/54 interface to run on a laptop.  If a slightly different version of the Matlab code is used it would be possible to access these cards.  Opening a selection window when the acquisition program is run, or auto-detecting which cards are present would be a substantial flexibility boost to the system.

### 5.3.5  Use of Different RGB Camera

As discussed in Section 4.1.2.6, the RGB camera that was used has some issues.  Most prominent are the slow frame rate, and lack of 64-bit compatibility.  We recommend finding a better camera for this application.

### 5.3.6  Additional Threads and Optimization of Synchronous Program Elements

There are several ways to optimize the efficiency of the Acquisition Program.  For example, using multiple threads to pipeline the acquisition, processing, and output of images, or only running NDGRI on pixels identified by the basic NDSI based detector.  Such a pipeline might look like that in Fig. 47, where each box represents a different thread.
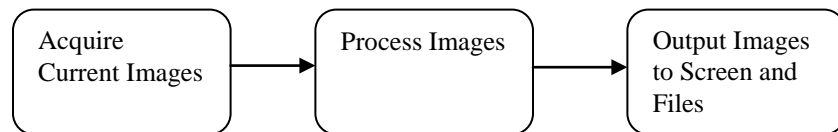


**Figure 47.**  Example of a possible pipeline implementation to increase speed.

### 5.4  Conclusion

The architecture designed here allows a common user to quickly and easily acquire data, view raw and processed data, and save detections in real time.  For the search and rescue community, this provides a unique mechanism to perform skin detection algorithms in real time.

It also allows users to easily save, post process, and view video files. The program correctly implements all common algorithms required for skin detection, for use both in real time and post processing. Furthermore it provides a platform suitable for future expansion in support of a critical Air Force research area: human measurement and signature intelligence.

*Appendix A.   Software*

*A.1  Software Installation and Setup*

*A.1.1  Required Software*

>Windows XP 32-bit

>Matlab® 7.9.0.529 (R2009b) 32 bit

>Matlab® Compiler™ 4.11

>Matlab® Builder™ JA 2.0.4

>Matlab Component Runtime 7.11

>Matlab Image Processing Toolkit

>Matlab Image Acquisition Toolkit

>Eclipse IDE for Java Developers (win32 version)

>Java JDK6 (1.6.0_17)

>Java SE Runtime Environment (1.6.0_17)

>SUI Image Analysis Software and .ICD files

>NI-Imaq

>Thor Labs Software for DCx USB Cameras (CD3.32)

*A.1.2  Software Setup:*

- Standard installation for Matlab, Eclipse, and MS Visual Studio.

- Install the Matlab Component Runtime (MCR).

  The MCR is installed by running

  >*C:\Program Files\MATLAB\R2009b\toolbox\compiler\deploy\win32\*

  >*MCRInstaller.exe*

   and following the default installation instructions.

- Set up Windows and Matlab environment variables

  Open the environment variables box in Windows and add the following to the PATH variable:

  *C:\Program Files\Java \jdk1.6.0_17; C:\Program Files\Java \jdk1.6.0_17\bin*

  Edit (or add) JAVA_HOME to:

  *C:\Program Files\Java \jdk1.6.0_17*

  Open Matlab and enter the following at the command prompt:

  *setenv('JAVA_HOME',' C:\Program Files\Java \jdk1.6.0_17')*


- Set the C++ Compiler that the Matlab Compiler will use

  In either the Matlab or DOS Command prompt, type:

  *mbuild –setup*

  Allow mbuild to locate installed compilers, and select lcc when presented with options. Finally verify the choices and the program will exit.

- Copy .ICD files for the NIR cameras to location required by NI-Imaq.

  This location can be found by doing a windows search for *.icd, and you will find one directory full of these files.

  Make a second copy of this file, and use one for each camera so that settings can be saved independently.

*A.2  Explanation of Software Operation*

Functions are programmed into Matlab, with one function per m-file. The Matlab Builder JA allows a user to create any number of classes, and place any number of function m-files into each. It then wraps all the classes and functions into a JAR file.

In order to create the JAR file, the Matlab Builder JA makes use of the Matlab Compiler (mcc.exe) and the java compiler (javac.exe). The Matlab Compiler makes use of a C++ compiler, which is the reason for setting its location in mbuild.exe.

When setting up the Java project in Eclipse, both the compiler generated JAR and another file, JavaBuilder.JAR, must be imported. JavaBuilder.JAR contains the necessary classes for the type conversion between standard Java classes and the types required for the Matlab function calls. The user generated JAR actually contains the compiled functions.

At runtime the JAR will link the Java classes that are Matlab function calls to an associated .dll contained in the same directory as the JAR. This .dll will link to another .dll in the Matlab Runtime Compiler to actually run the Matlab functions.

*A.3  Instruction for Eclipse Project Setup*

To setup the Java project in eclipse, add the JAR file that was generated by the Matlab Builder JA. Also add *Javabuilder.Jar* found in *C:\Program Files (x86)\MATLAB Compiler Runtime\v711\toolbox\*. Both should be added as external JARs. Once they are added to the project, add this line to the top of any java files that use Matlab: "*import com.mathworks.toolbox.javabuilder.\*;*". Finally import the package of the compiled functions as "*import* {package name}.*\*;*".

*Appendix B.  Users Manual*

*B.1  Acquisition Program*

- Camera Configuration

    Depending on the light conditions, the operational settings of the NIR cameras have to be setup.  Open the NI-Imaq software, and select the first channel camera.  Click the "Grab" button, then change the Operational Setting and the Digital Gain parameters until the image is reasonable.  Click the "Save" button.  Repeat for the second channel, then close the program.

- Startup

    In Eclipse, open AcqMain.Java in the GUI package of the AcquisitionProgram project, and select run.  System messages are displayed at the bottom of the screen.  The GUI will appear when everything is running.

- Closing

    To close the program, click the "Exit" button on the bottom of the main GUI.  This will close all open outputs in turn, and then close the program.  The "X" in the upper right corner will not close the program.

- Viewing Windows

    The first and third columns of checkboxes on the main GUI are for the window viewers. The first column is to view the raw data, and the second to view the ELM processed data. Selecting the checkbox opens the window, and unselecting the checkbox closes it.  The windows cannot be closed using the "X" in the upper right corner.

    Crosshairs can be overlaid onto the image by selecting the checkbox under the image. These are not stored in the image in any way, only displayed in the viewer.  They can be used for lining up the cameras onto a target.

- Video File Creation

The second and fourth columns of checkboxes on the main GUI are for the video file creation windows. The second column is for the raw data, and the fourth to save the ELM processed data. Selecting the checkbox opens the window, and unselecting the checkbox closes it. The windows cannot be closed using the "X" in the upper right corner.

The first window to appear is a filename selection window. Entering a pre-existing filename will cause a message to be displayed. Entering a valid filename will cause this window to change to a smaller window simply listing the filename, and will start the video file creation. The file does not begin acquiring until after a filename is selected. Closing the window through the main GUI ends the video file creation.

All video files are saved to "C:/VideoFiles/", and automatically are assigned a ".vid" extension.

- Setting Panel Coordinates

When the program starts there are no panel coordinates set, and the ELM will do nothing. Panel coordinates can be set in any open raw image window (first column of checkboxes). They cannot be set in the ELM processed windows.

The points are set by clicking on the panels in the image. A left click will set the grey panel, and a right click will set the white panel. The raw image displays will all show a light blue dot at the coordinates of the white panel, and a dark blue dot over the grey panel. As with the crosshairs, these do not overwrite any data in the image, they are only overlaid in the display. The ELM processed images do not show these dots. There is no requirement of the order they must be selected in.

Clearing the panel coordinates is done by clicking the "Clear Points" button in the main GUI.

Whatever panel coordinates are set when a file save is started are saved at the beginning of the file. Any subsequent panel coordinate changes are reflected when they occur.

- Parameter File Modification

The parameter file is saved as "C:/VideoFiles/ParamFile.txt". Open the file in a text editor, make the necessary changes, then save the file. The parameters in the system are updated only on startup or when the "Update Parameters" button is clicked, not when the file is saved.

Available parameters and formatting is discussed in Section 3.1.2.


*B.2 Processing Program*

- Startup

In Eclipse, open ProcMain.Java in the GUI package of the ProcessingProgram project, and select run. The first window presented is an output file name, which must be entered in order to continue.

- Closing

The program closes automatically when finished.

- Algorithm Selection

After an output filename is selected, a small GUI containing a list of available algorithms is presented. Pull down the list and select the desired algorithm, then click run. Note: the algorithms have no built in dependency structure, so be sure you have created all videos the algorithm will require prior to attempting to run the algorithm.

- Input File Selection and Algorithm Execution

After the algorithm is selected, input file selection windows will appear. The required wavelength is listed in the window title. Selecting a video containing data other than that required will still produce an output. System messages and progress are displayed in the Eclipse console box.

- Concerns for ELM

Due to the nature of the Processing Program, ELM can be difficult. The initial panel coordinates are saved in the file header, and are applied based off the first image. If the panels are moved or have too much noise at that exact pixel, it may be impossible to perform useful ELM.

The recommendation is to start all desired file saves, then click on the panels again so all data after that will be known to be useful. A better alternative is to save the ELM processed streams instead of the raw, although that will result in slightly reduced frame rates.

- Synchronization

In algorithms requiring more than one file, the images are synchronized by time stamp. Any frames not overlapping the other file are discarded in the output, and if no overlapping frames are found, then no output is produced.

- Parameter File

The parameter file is shared with the Acquisition Program, and is loaded at startup only. The location of the file is "C:/VideoFiles/ParamFile.txt". See Section 3.1.2 for details on available parameters and formatting.

*B.3 Playback Program*

- Startup

    In Eclipse, open PlayMain.Java in the GUI package of the PlaybackProgram project, and select run. The first window presented is an input file name, which must be entered in order to continue. Following the file selection, the main GUI immediately appears, and the video begins to play.

- Closing

    To close the program, click the "Exit" button on the bottom of the main GUI. The "X" in the upper right corner will not close the program.

- Pause/Play

    Pausing and Playing is done by clicking the button below the image with the play or pause symbol on it, which alternates depending on the state. Current frame number and time in seconds since the beginning of the file are displayed directly below the image.

- Moving in the Video

    Two options are available: moving to a time and moving to a frame number. They are clearly marked in the lower left corner of the GUI. Enter a number into the appropriate box, and click the button next to it. The video changes to the appropriate image, and the video is automatically paused.

    One thing to keep in mind is the move function is slower the farther into a file the desired frame is, and can take a good length of time. Also the frame number is the number of frames since the Acquisition Program was started until that image was taken, while the time displayed is in seconds from the beginning of the file. So, if attempting to view the same image in multiple

video files, searching by frame number will pull up the same scene while searching by time will

not.

*Bibliography*

1. Nunez, Abel. *A Physical Model of Human Skin and Its Application for Search and Rescue*. Wright-Patterson AFB, OH: USAF/AFIT, 2009. Print.

2. Peskosky, Keith. *Design of a monocular multispectral skin detection, melanin estimation, and false alarm suppression system.* Master's thesis, Air Force Institute of Technology, March 2010.

3. "Java 2 Platform SE v1.4.2." *Developer Resources for Java Technology*. Sun Microsystems Inc, n.d. Web. 11 Feb. 2010. <http://java.sun.com/j2se/1.4.2/docs/api/>.

4. Grand, Mark. *Java Language Reference (Nutshell Handbook)*. 1 ed. Sebastopol, CA: O'Reilly, 1997. Print.

5. Geary, David M.. *Graphic Java 1.1: Mastering the AWT*. 2nd ed. Mountain View, CA: Sun Microsystems Press, 1997. Print.

6. Brooks, Adam. *Improved multispectral skin detection and its application to search space reduction for histograms of oriented gradient–based dismount detection.* Master's thesis, Air Force Institute of Technology, March 2010.

7. Short, Nicholas M., Sr. " Remote Sensing Tutorial Page 13-9." *The Remote Sensing Tutorial*. N.p., n.d. Web. 11 Feb. 2010. <http://rst.gsfc.nasa.gov/Sect13/Sect13_9.html>.

8. Nunez, Abel, Adam Brooks, Michael Mendenhall, and Richard Martin. "Human Skin Detection and Color Estimation in Hyperspectral Imagery." *IEEE Transactions on Geoscience and Remote Sensing* (2009). Submitted.

9. Eismann, *HyperSpectral Remote Sensing.* 2006

10. "MaintainJ." MaintainJ Inc. Web. 2009. <http://www.maintainj.com/>.

11. "Soyatec - Open Solution Company: XAML for Java, UML for Eclipse and BPMN designer." *eUML2*. Soyatec, n.d. Web. 11 Feb. 2010. <http://www.soyatec.com/euml2/>.

12. Larman, Craig. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. 3 ed. Upper Saddle River, NJ: Prentice Hall PTR, 2004. Print.

| REPORT DOCUMENTATION PAGE | | *Form Approved* OMB No. 074-0188 |
|---|---|---|

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of the collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to an penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
**PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

| 1. REPORT DATE *(DD-MM-YYYY)* 26-03-2010 | 2. REPORT TYPE Master's Thesis | 3. DATES COVERED *(From – To)* Sep 2008 – Mar 2010 |
|---|---|---|

| 4. TITLE AND SUBTITLE Flexible Computing Architecture for Real Time Skin Detection | 5a. CONTRACT NUMBER |
|---|---|
| | 5b. GRANT NUMBER |
| | 5c. PROGRAM ELEMENT NUMBER |
| 6. AUTHOR(S) Matthew Paul Hornung, Second Lieutenant, USAF | 5d. PROJECT NUMBER 09ENG287 |
| | 5e. TASK NUMBER |
| | 5f. WORK UNIT NUMBER |

| 7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way WPAFB OH 45433-7765 | 8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCE/ENG/10-02 |
|---|---|

| 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Dr. Brian Tsou 711 Human Performance Wing Information Operations and Special Programs Branch 2210 8th Street, Bldg. 164, WPAFB, OH 45433 937-255-8896; brian.tsou@wpafb.af.mil | 10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/RHXB |
|---|---|
| | 11. SPONSOR/MONITOR'S REPORT NUMBER(S) |

**12. DISTRIBUTION/AVAILABILITY STATEMENT**

Approval for public release; distribution is unlimited

**13. SUPPLEMENTARY NOTES**

**14. ABSTRACT**

In both the Air Force and Search and Rescue Communities, there is a current need to detect and characterize persons. Existing methods use red-green-blue (RGB) imagery, but produce high false alarm rates. New technology in multi-spectral skin detection is better than the existing RGB methods, but lacks a control and processing architecture to make them efficient for real time problems. A number of applications require accurate detection and characterization of persons, human measurement and signature intelligence (H-MASINT), and SAR in particular. H-MASINT requires it for the detection of persons in images so other processing can be performed. It is useful in the SAR community as a method of finding persons partly obscured, in remote regions, and either living or deceased. We have developed a modular computing architecture to perform the acquisition and processing in real times, as well as separate programs to perform processing and analysis of images post-acquisition. The architecture is flexible, as one can easily add additional functionality to meet growing demands. All programs were organized using a basic Model-View-Controller design, designed using Universal Modeling Language principles, and coded using a bottom-up approach. Based on the results we have presented in this thesis, image acquisition, processing, skin detection, viewing, and saving can be performed in real time, at nearly 10 fps. Not only does this support the SAR community, the Air Force now has a new capability to help address its H-MASINT mission.

**15. SUBJECT TERMS**
Skin Detection, Melanin Estimation, False-Alarm Suppression, Multi-Spectral Imaging, Search and Rescue

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON Maj Michael J. Mendenhall |
|---|---|---|---|---|---|
| REPORT U | ABSTRACT U | c. THIS PAGE U | UU | 93 | 19b. TELEPHONE NUMBER *(Include area code)* (937)-255-3636, ext 4614  Michael.Mendenhall@afit.edu |

Standard Form 298 (Rev: 8-98)
Prescribed by ANSI Std. Z39-18