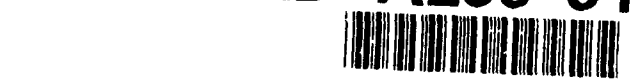


AFIT/GE/ENG/92S-06

AUTONOMOUS FACE SEGMENTATION

THESIS

Kevin Patrick Gay
Captain, USAF

AFIT/GE/ENG/92S-06

Approved for public release; distribution unlimited

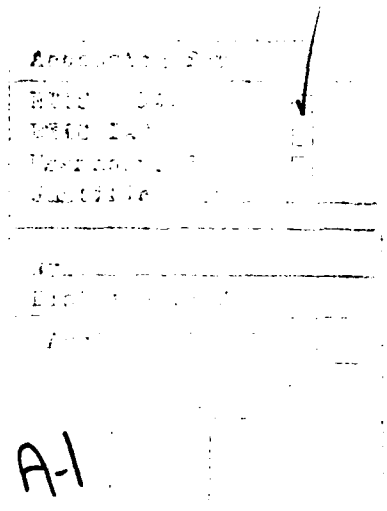
AFIT/GE/ENG/92S-06

AUTONOMOUS FACE SEGMENTATION

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Electrical Engineering

Kevin Patrick Gay, B.S.E.E.
Captain, USAF



September, 1992

Acknowledgments

My thanks to Dr. Steve Rogers (Captain America) and Dr. Matt Kabrisky for all the time, knowledge, and gentle persuasion they gave me during this research. Their humor and free-flowing style of teaching made my time here a joy instead of a drudgery. An additional thank you to Major Rogers for going the extra mile to help me with my Air Force and post-Air Force career. The help was truly appreciated.

I also want to acknowledge Mr Dan Zambon and Captain Dennis Ruck for their technical expertise with computer systems and their patience with students. Dan was always quick to help on the Suns, saving me countless hours, and he was always ready for a little friendly verbal jousting as well. Dennis assisted me with innumerable "C" and Unix problems, and he could always manage a smile, even when I repeatedly brought down the Next computer system.

And I would be ungrateful indeed if I did not thank the other two "face guys", Ken Runyon and Dennis Krepp. Their friendship was a major reason I survived my time here, and it is nice to know I can take it with me.

The last group I want to acknowledge are those that matter the most to me on a personal level. My wife, who has carried the family through the difficult times here. I love her, and she is truly better than I deserve. My children, who have given up time we would normally spend together. Their sacrifice is more my sacrifice, because the joy they bring me is sweeter than anything else on this earth. My father, who passed away only a few months ago, and yet still strengthens and uplifts me. And my sweet mother, who is love personified. These are but a few of the many to whom I am indebted, but I will end here by publicly acknowledging God and expressing my gratitude to Him. There may be some who doubt that God exists, but that does not alter the fact that He does exist, and He blesses us daily.

Kevin Patrick Gay

Table of Contents

List of Figures

Figure

Page

Abstract

The purpose of this study was to implement an autonomous face segmentor as the front end to a face recognition system on a Sun SPARCStation2. Face recognition performance criteria, specifically, the capabilities to isolate and resize faces in an image to a consistent scale, were analyzed to determine current practical limitations. Face images were acquired using a S-VHS camcorder. Segmentation was accomplished using motion detection and pre-defined rules. Tests were run to determine the suitable of the autonomous segmentor as the front-end to a face recognition system. The segmentation system developed consistently located faces and rescaled those faces to a normalized scale for subsequent recognition.

AUTONOMOUS FACE SEGMENTATION

I. Problem Description

1.1 Introduction

Autonomous face recognition systems have many potential applications, such as scanning airports for criminals and verifying users at bank teller machines. Current systems, however, are primarily research tools, ill-equipped for practical applications. This research focuses on some of the major obstacles to practical face recognition by constructing an autonomous face segmentor which will operate as the front end to a complete, autonomous face recognition system residing on a Sun SPARCStation2. Complementary theses by Krepp and Runyon will focus on the remaining portions of this complete face recognition system (14)(22). The face recognition system is intended for use in a realistic office environment.

This problem description begins with a background review of the most relevant face recognition research, followed by the problem statement. Next, the research objectives, assumptions, scope, and standards are stated. The approach is then presented, and the chapter concludes with an overview of the remaining chapters.

1.2 Background.

1.2.1 **Suarez and Goble.** Suarez and Goble researched face recognition at the Air Force Institute of Technology using the Karhunen Loève and Discrete Cosine transforms, respectively (24)(7). Their recognition accuracies were respectable; however, face segmentation was done manually, and recognition software resided in pieces on several machines.

A face recognition system being developed by Runyon is based on the techniques developed by Suarez's research. This face segmentation research will be the front end for Runyon's recognition system.

1.2.2 **Turk and Pentland.** Turk and Pentland, from the Massachusetts Institute of Technology, constructed a completely autonomous face recognition system (25). Like Suarez, they used Karhunen Loève transforms for the recognition process. Their system used three processors: the first two dedicated to segmentation, and the third, recognition. Their segmentation technique uses motion detection, motion analysis, and the error in reconstructed images. Turk and Pentland did not address the limitations of their system in a practical environment although the amount of processing power required is an obvious drawback.

The concepts of using motion to find faces is directly applicable to this research, but the processing power will be significantly different.

1.2.3 **Lambert's AFRM.** The AFRM (Autonomous Face Recognition Machine) was developed at the Air Force Institute of Technology before Lambert's research, but it was his enhancements which make the execution time and segmentation results respectable. Lambert's segmentation technique used motion detection, brightness normalization, and "face" brightness patterns. The system as a whole did not meet expectations, and it was clearly not useful in a practical environment.

The motion detection and brightness normalization are the concepts most applicable to current research. His recognition technique, which is based on cortical thought theory, takes more time and is less accurate than current techniques.

1.3 Problem Statement

Current face recognition systems are not yet practical for commercial use. The purpose of this study is to investigate limitations of autonomous face segmentation

which is part of a complete, autonomous face recognition system operating in a realistic environment.

1.4 Research Objectives

The objective of this research is to uncover, and overcome, the major obstacles to autonomous face segmentation for a single workstation autonomous face recognition system.

1.5 Assumptions

The face segmentation assumes:

1. A fixed-position imaging device.
2. Fixed focal length.
3. Automatic intensity compensation.
4. Continuous head movement.
5. No movement directly behind the system user which is shoulder height or higher.
6. Consistent office lighting.

1.6 Scope

The scope of this thesis is to investigate the limitations of an autonomous face segmentation system operating on a Sun SPARCstation2.

1.7 Standards

The performance criterion for this face segmentation is a suitable face for face recognition. The key criterion for the face recognition system is classification accuracy, as well as user interaction and/or constraints. The key constraints for face

segmentation with regard to classification accuracy in this face recognition system are :

1. Minimal background in each image, and
2. Consistent scale across all images.

The key user constraints to minimize for face segmentation:

1. Tailored backgrounds, and
2. User actions.

1.8 Approach/Methodology

A Super VHS camcorder connected to a Sun SPARCStation2 with a VideoPix frame grabber card provides the input imaging capability of the system. A software environment developed and executing on the Sun SPARCstation2 performs the face segmentation and recognition. This software combines commercial software with new software written in "C."

1.9 Overview

Chapter Two presents a review of current literature related to face recognition systems. Chapter Three provides a detailed description of the methodology used in this thesis, and Chapter Four provides test results. Chapter Five presents conclusions based on the test results and makes recommendations for future study.

II. Literature Review

2.1 Introduction

When people look at photographs of friends, or run into a co-worker at the supermarket, or talk with family members at the dinner table, they probably never stop to consider how they recognize these people. Recognizing people is so commonplace to humans they usually don't think about how remarkable this process is. Faces are one of the most common discriminators humans use to recognize people, and humans are very good at recognizing faces (experiments have shown that monkeys, sheep, and even pigeons are also very good at recognizing human faces) (19)(12)(21:35). Yet, as seemingly effortless as this process is for humans to perform, no one has been able to build a machine that performs anywhere near as well as a pigeon, let alone a human. In spite of this fact, there are some practical applications which would benefit from a high confidence face recognition system. Scanning for criminals in airports or verifying account owners at a bank teller machine are just two of many possible examples. One of the requirements for this thesis is to build an autonomous near-real time face recognition system. Given the performance disparity between humans and machines, it seems prudent to review the latest research regarding biological face recognition, as well as the latest face recognition technology. The results of this literature investigation are provided in the following section. The final section of this chapter will briefly summarize the current status of face recognition technology as found in the literature search, and state what can be concluded regarding research objectives.

2.2 Face Recognition Research

One of the most interesting facts about investigating how the brain "does" face recognition is that no one *knows* how the brain "does" any process, face recognition included. There are, however, many hypotheses; some of which enjoy greater accep-

tance than others. The latest trends in face recognition research indicate that there are at least two major subprocesses in recognizing faces. First, taking in a visual scene and realizing there is a face in that scene at all. This detection of a face is not recognition of a specific person's face, it is simply realizing that a face exists in the scene (for example, as someone looks around, he realizes there is a face looking at him from the bushes). This process is referred to as *segmentation.* The second major subprocess is recognizing that a face "belongs to" a specific person (returning to the example, upon further study, he realizes it is that weird kid down the street that is looking at him from the bushes). This second process is usually referred to as face recognition by itself, but it will be referred to herein as *individual face recognition* to distinguish it from the overall process.

2.2.1 **Face Segmentation** When humans look at a photograph of someone, they can immediately determine if there is a face in that picture, but how do they know this? Some research suggests humans have cells which respond uniquely to faces, regardless of whose face it is or how it is represented (e.g., photograph, line drawing, or some other representation) (20)(19)(9). *Face sensitive* cells suggest that face segmentation is a *biological process.* Support for this hypothesis can be found in the research conducted on prosopagnosia patients. These people exhibit no deficit in the ability to find and discriminate faces, but they cannot identify the person to whom the face belongs (4)(3)(18).

Other research suggests that face sensitive cells are excited by particular combinations of cells (i.e., patterns) on the visual cortex. These cells on the visual cortex respond as if a *localized frequency analysis* has been applied to the visual field (10). These experiments have spawned research with *wavelet* segmentation; wavelets perform a localized frequency analysis. Wavelet segmentation results with tanks and such are encouraging (23), but very little face segmentation with wavelets has been attempted (26).

While the biological research on face segmentation progresses, other researchers have designed and built systems to segment the faces in an image. The success rate of these systems has been modest, but may be sufficient for some applications. These systems find faces by using expected brightness patterns to find areas in the image which may be the eyes (1)(15)(16) or face outline (27)(8)(16). To reduce the computational load, most systems use pre-defined rules, such as head size (25)(8) or motion (25) to reduce the possibility of falsely identifying a non-face object as a face. To speed up the segmentation process, some face recognition systems use motion to limit the search area (25)(15). The underlying assumption of this technique is that people almost never keep their heads perfectly still. Experimental results have shown this to be a reasonable assumption. Using motion is reasonable from a biological standpoint, as well, since research has shown that (a) some animals use motion to detect objects and (b) we have cells which respond solely to motion (11). Even with motion and pre-defined rules, segmentation is usually the most time consuming and computationally intensive portion of any autonomous face recognition system.

2.2.2 **Individual Face Recognition** Once the face is found in the image, the face recognition process must be completed by identifying the individual to whom the face belongs (individual face recognition). This is true regardless of whether we are working with a biological or man-made process. The two most common approaches to individual face recognition are *feature-based* and *holistic*. Biological experiments have been cited to support feature-based and holistic recognition (2), although neither approach can claim superiority based on the experimental results. Recent studies suggest humans may do both feature-based and holistic recognition in parallel (5:6).

The feature-based recognition, as might be guessed from the name, uses measurements based on the features of a face (e.g., eyes, nose, and mouth) to characterize a person's face. Which features are the right ones to use, or which features does the brain use? We don't know. Researchers have tried a variety of measurements with

modest results. Feature-based techniques were used in the early seventies when face recognition was first attempted by computers, and feature-based research still continues today. One of the most significant limitations of the feature-based approach is the autonomous segmentation of the features. One recently developed autonomous feature-based face recognition system required forty minutes on a VAX 8530 just to find the face and get it into a format the system could use to extract the feature measurements (16).

The holistic approach uses the "entire face", not just selected features, as an input to the individual recognition process. The holistic research has been based primarily on data compression techniques which have the advantage of representing the entire image with just a few coefficients (6)(13)(7). In fact, face recognition systems have been built which use one of three following data compression techniques:

- The Karhunen Loève Transform (25)(13)(24)
- The Discrete Cosine Transform (7)
- A Neural Network with Unsupervised Feature Extraction (6)

There is no clear advantage to using any one of these three techniques based on recognition success rates; recognition success rates were essentially equivalent for systems using any one of these techniques. An additional, and perhaps more significant, advantage of holistic recognition is that rigorous segmentation may not be required; the location of the face is needed, but not individual features. Face recognition research using the holistic approach has been around for only a few years, but results have been encouraging.

2.3 Summary and Conclusions

Current autonomous segmentation techniques have had only limited success in locating faces in an image. To improve their success rate, most systems make a number of assumptions about the objects that will appear in the visual field. These

assumptions severely limit the usefulness of these systems in practical applications. A secondary issue is the computational burden and execution time. Although motion and pre-defined rules may reduce the execution and the computational burden, they intrinsicly make further assumptions about the system's environment and, therefore, further limit their portability to new environments. Face segmentation with wavelets may benefit the face segmentation field, but that research is still open.

The individual face recognition portion of the face recognition process can be done by current equipment with reasonable accuracies using either the feature-based or holistic approach. A major drawback to feature-based recognition is the rigorous segmentation feature-based systems require; as discussed above, current segmentation techniques are limited. Holistic recognition is promising, partly because it does not appear to require rigorous segmentation to achieve reasonable results.

As mentioned earlier, one of the requirements for this thesis is to build a portion of an autonomous near-real time face recognition system, and sponsor requirements dictate this system will be integrated into a Sun SPARCstation2. Given the results of the literature search, a motion-aided holistic face recognition system seems to be the most reasonable solution given the technology available today. The assumptions about motion are acceptable for the intended environment of this system, and if better segmentation techniques become available they can be easily integrated into this type of system.

III. Methodology

3.1 General

The main objective of this thesis was to implement segmentation techniques on a Sun workstation to determine the capabilities of these techniques to segment a face from an input scene which is suitable for use as an input to a face recognition system. The approach of this research was to use an algorithm to analyze the movement between two successive images to find possible face regions. This approach is based on the fact that humans do not keep their heads perfectly still; a certain amount of motion is always present (17). A side benefit to this approach is that if this technique cannot segment suitable faces by itself, it can be used by other techniques to find areas of interest.

3.2 Motion Analysis

3.2.1 **Capturing Images** The first step was to develop the capability to capture images and bring them into the Sun for analysis. The tools used were the Sun VideoPix and a S-VHS camcorder. The reasons for choosing the VideoPix were as follows:

1. The tool capability seemed adequate for our application. The specification stated it could capture up to four grey-scale images per second; color images at up to one per second.
2. The tool can be controlled via "C" routines which come with the tool.
3. The hardware and software were available and already integrated into a Sun workstation.

The S-VHS camcorder was chosen because it provided automatic brightness control, and could supply color or grey-scale, live or video-taped images. Appendix A contains information on using VideoPix.

The images provided by the VideoPix tool are 720 by 480 non-square pixel *YUV* images, where *Y* is luminance and *UV* are chrominance[1]. The conversion to NTSC format square pixel data produces a 640 by 480 image. The decision was made to use 8-bit grey scale images because these types of images had been used successfully for both motion and recognition in the past (25)(15).

3.2.2 **Finding Motion Regions** After developing the capability to capture images on the Sun workstation, the next step was to find the regions of movement. The technique for finding moving regions, and one that had been used successfully in previously research (25)(15), was frame-to-frame subtraction. The reasoning behind this approach is, given that the camera and focal region remain fixed, the pixel values that change between two images are considered regions of movement.

One obvious consideration is the time lag that occurs between capturing the first and second images. If this time lag is too large, the resolution of the motion regions deteriorates. This time lag was measured using the Unix **date** command. Additionally, the motion region created when a subject greatly exaggerates his motion was determined. Although the acquisition rate for the VideoPix tool is fairly slow, it is fast enough that a subject would have to intentionally exaggerate his motion to cause any significant deterioration of the motion image, and in this research a cooperative user was assumed.

Noise was filtered out with a median filter; a median filter assigns pixel values based on the median value of the pixels surrounding a pixel as well as the pixel value itself.

As can be seen in Figure 1, the "motion regions" that result from this approach are not usually solid regions, but rather loose outlines of moving objects, with gaps in the outline, and random patterns of pixels internal to the object "outline" which

[1]More detail on the *YUV* data is given in Appendix A

depend on both the pattern of the moving object and the motion internal to that object (i.e., blinking eyes or moving mouth).

Figure 1. Typical Motion Image

To form a defined region, the "motion" pixel closest to the top was found for each column of the image and all pixels below that pixel were considered part of the motion region. An example of this modified image can be seen in Figure 2.

The decision to find the pixel closest to the top was based on:

1. Many times the object "outlines" were so poor that no standard algorithm could be used to simply "close the gaps."
2. Given the target environment for this system (office environment), it was assumed that if a person was in the scene there would not be motion above that person's head. It was recognized that a second person standing directly behind a person sitting in a chair, would obscure the sitting person; however, that was felt to be an acceptable limitation.
3. The top portion of the motion outlines is the most consistent outline edge.

Figure 2. Motion Outline I

To close small gaps that still exist in the modified motion image, the columns of this image were then grouped, and all columns in a particular group were assigned the "highest" motion pixel value for any column within that group. An example of this image is shown in Figure 3.

As is evident from Figure 3, random noise or large gaps in the motion outline occasionally caused "spikes" to occur in the motion region. Therefore, single column-group spikes were eliminated by looking for large changes in motion pixel location from one column to the next followed by another large change in motion pixel location in the opposite direction. An example of the resulting "cleaned up" motion image is provided in Figure 4.

Alternately, motion regions formed by subtracting two color (rgb) images were tested to determine if there was any advantages to using color images instead of grey scale to determine the motion regions.

3.2.3 **Segmentation by Region Analysis** Once motion regions were defined, these regions were then analyzed to find features which were common to all

Figure 3. Motion Outline II

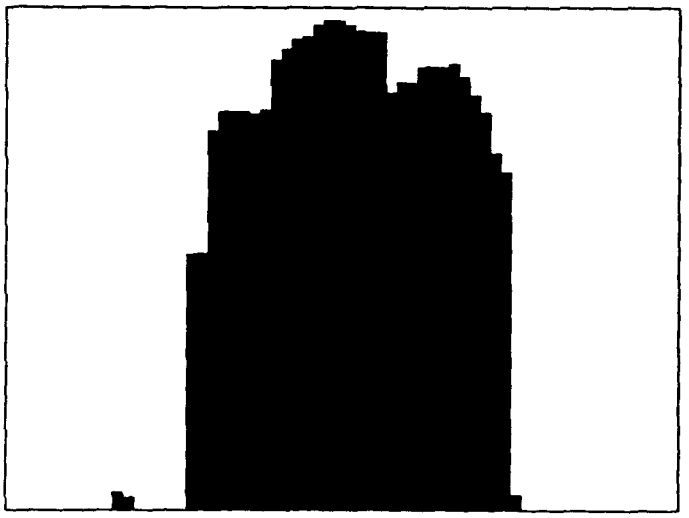


Figure 4. Final Motion Outline

face regions. A secondary analysis was to find features which were unique only to face regions, but this was only secondary because it was felt that later processing could eliminate non-faces as long as the segmentation was providing suitable face regions when they were part of the input scene.

Because (a) motion regions were created by "blanketing" objects from the top, and (b) human heads are tall oblong shapes on top of a wider shape (shoulders and body), the difference between the height of the shoulders and the top of the head is manifested as a dramatic change, or large slope, in the motion region. The analysis, therefore, simply looks for a large positive slope, a fairly stable region, and then a large negative slope. The entire image is searched for regions which fit this profile over several slope thresholds. Multiple slope thresholds are an attempt to overcome the variations in subjects and the "quality" of the motion outline.

Once the portions of the motion region had been found which fit the "head" profile, these regions were "cut" out of the image and rescaled to a standard size. It was hoped that since the region segmented out of the original image was dependent on the width of the moving head, that rescaling the segmented region to a standard size would provide sufficient scale invariance for an automatic recognition system. The scale invariance was tested by segmenting the face of a subject at three different camera "zoom" settings. These camera settings significantly altered the scale of the subject's head in the image input. The segmented faces were then visually inspected to determine how well the segmentation algorithm "normalized" the scale given the three scale variations in the input images. Faces segmented during the previous segmentation tests were also visually inspected to further determine the scale consistency of faces segmented by this technique. The actual "sufficiency" of this scale normalization for a recognition system may be tested in a complementary thesis by Ken Runyon (22).

3.3 Binarized Face Patterns

To provide an alternate method of finding the faces in an image, it seemed reasonable to search for specific features of faces in the image. One technique used previously at AFIT which appeared to be fairly successful was to "lambertize" an image and then search for features such as eyes, nose, and mouth (15). To overcome feature obscuring due to brightness variations across a scene, Lambert found the brightness variations within some window of the visual scene. The variations remained fairly consistent even when the overall brightness of the image change radically. Thus, by finding these local variations, it was hoped that facial features would appear consistently in a "lambertized" image.

Local variations in an image were found using Lambert's technique and several different window sizes. The input image was then reduced and then local variations were found in the reduced image as well. Algorithms for finding faces, based on these images, may be developed in future research.

3.4 Summary

The face segmentation approach taken in this research can be summarized as:

1. Capture two images from a fixed camera in rapid succession.
2. Perform frame-to-frame subtraction to determine movement.
3. Eliminate noise from the motion image.
4. Group pixels to form a more defined motion region.
5. Analyze the motion region to find face regions.
6. Cut these face regions out of the input image.
7. Resize these face regions to create a standard size vector (these vectors become the input to a face recognition system).

The details of this algorithm and the tests conducted to investigate this algorithm have been described in this chapter. The results of these tests are documented in the chapter that follows.

IV. Results

4.1 General

4.2 Capturing Images

The VideoPix hardware and software performed as expected. An example of two 8-bit grey scale images captured in a software loop are provided in Figure 5. The time lag between two successive captures was measured at 0.5 seconds. This

Figure 5. 8-Bit Grey Scale Images

time lag was determined by executing the Unix **date** command, capturing twenty images in a software loop, executing **date** again, and then dividing the difference by twenty. This difference was taken ten different times and each time the same measurement was found—footnoteIt was believed that capturing twenty images and dividing would produce a better estimate of the time between two successive grabs because the **date** command gives time to the nearest second, and the measurement was expected to be in fractions of seconds..

4.3 Finding Motion Regions

As mentioned in the previous section, the frame by frame subtraction using 8-bit grey scale images did not provide the consistent motion expected. The funda-

mental assumption was that biological subjects "wobble", and thus a clearly defined motion outline was expected even when subjects were trying to remain as still as possible (17). Figure 6 clearly shows that this was not the case. Multiple attempts at

Figure 6. Bad Motion Image

forming this motion image revealed that if the subject simply sat as if he or she were working at the computer terminal, the motion regions formed were too inconsistent to use for segmentation. This was particularly unfortunate in light of the fact that this was the projected user scenario for the final face recognition system.

On the opposite end of the spectrum, a motion image was formed with the subject greatly exaggerating his motion. As can be seen in Figure 7, severe motion will also produce a potentially unusable motion image. This type of motion is essen-

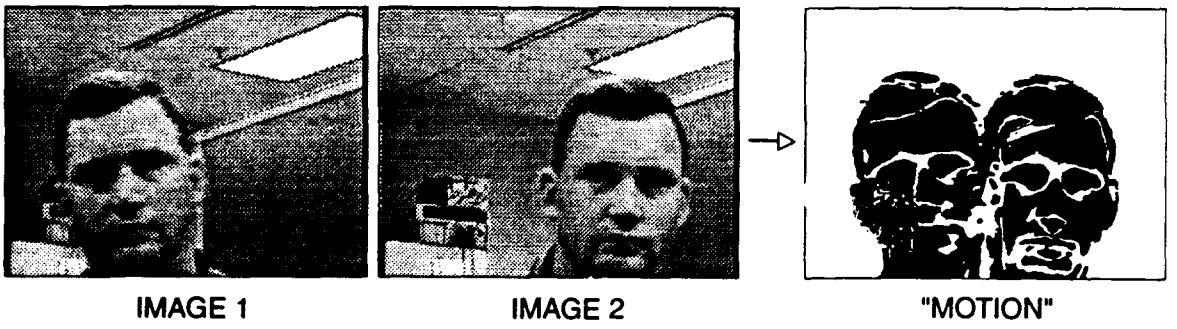


Figure 7. Severe Motion Image

tially ignored in this research, since a cooperative user was assumed. However, this test revealed limitation of the system caused by the relatively slow acquisition rate of the VideoPix tool.

The noise in the motion image was able to be filtered out very effectively by a median filter. Figure 8 shows this noise elimination by providing an example of a noisy motion image and the resulting images after passing through the median filter. Typically, a difference threshold of ten was set for "motion" pixels from the

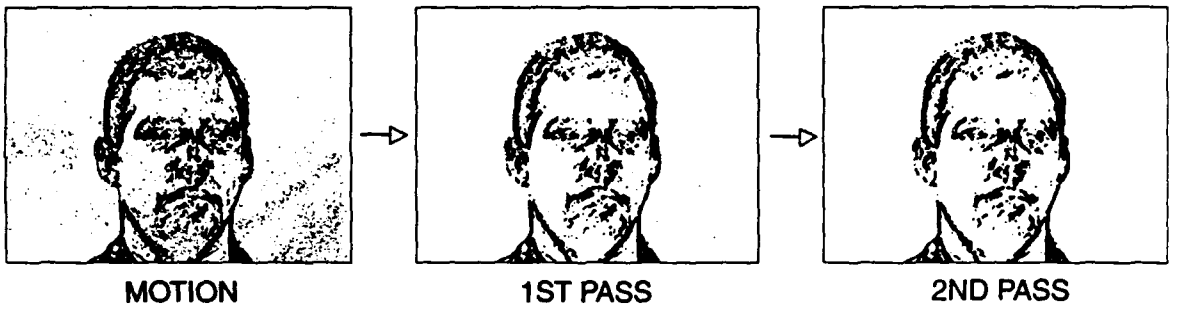


Figure 8. Median Filtering an Image

frame to frame subtraction, and the image was passed twice through a median filter. Through experimentation, this difference threshold and two passes through the filter were found to remove the noise in the image consistently.

To overcome this motion "inconsistency", the decision was made to threshold the number of pixels "turned-on" in the motion image. The drawbacks of this approach are:

1. The higher the threshold the more dramatic the motion must be on the part of the user and usually the longer that user must wait for segmentation.
2. The more dramatic the motion the less precise the segmentation and, therefore, the less consistent the scale in the segmented images.
3. The threshold does not guarantee a good motion outline, it simply indicates a greater likelihood of a good outline given stable conditions.

Experimentally trading-off the quality of the motion image with the user movement requirements and wait time, a motion threshold of 3000 pixels was found to be the best motion threshold. Of course this threshold will vary with the number of pixels in an image and the size of the moving objects in the image as well.

As mentioned in the previous chapter, moving objects found using frame-to-frame subtraction are not usually solid, but somewhat abstract outlines and random internal patterns. The process to form a defined motion outline is shown visually in Figure 9 and briefly summarized as:

1. The noise in the motion image was removed by median filtering.
2. The "motion" pixel closest to the top was found for each column of the image and all pixels below that pixel were considered part of the motion region.
3. The columns of this motion outline were then grouped to close these gaps.
4. Finally, single column "spikes" were eliminated by looking for large slopes in one direction followed by a large slope in the opposing direction.

Forming motion regions by subtracting two color images was attempted to determine if there were any improvement over using grey scale images. However, the motion region created from two color images did not improve the consistency of the motion outline at all, and the noise in the color image was not filtered out effectively. Figure 10 is an example of the motion image created using color images.

An alternate method of finding the outline of moving objects could be developed using an edge enhancement algorithm. The edges of the objects in the input image are found using an edge enhancement algorithm, and the motion analysis could be used to determine the edges of the moving objects. Finding the exact edges of the head would allow easy elimination of all background and also produce a rigid scale standard for resized head regions. An example of an edge enhanced input image is provided in Figure 11.

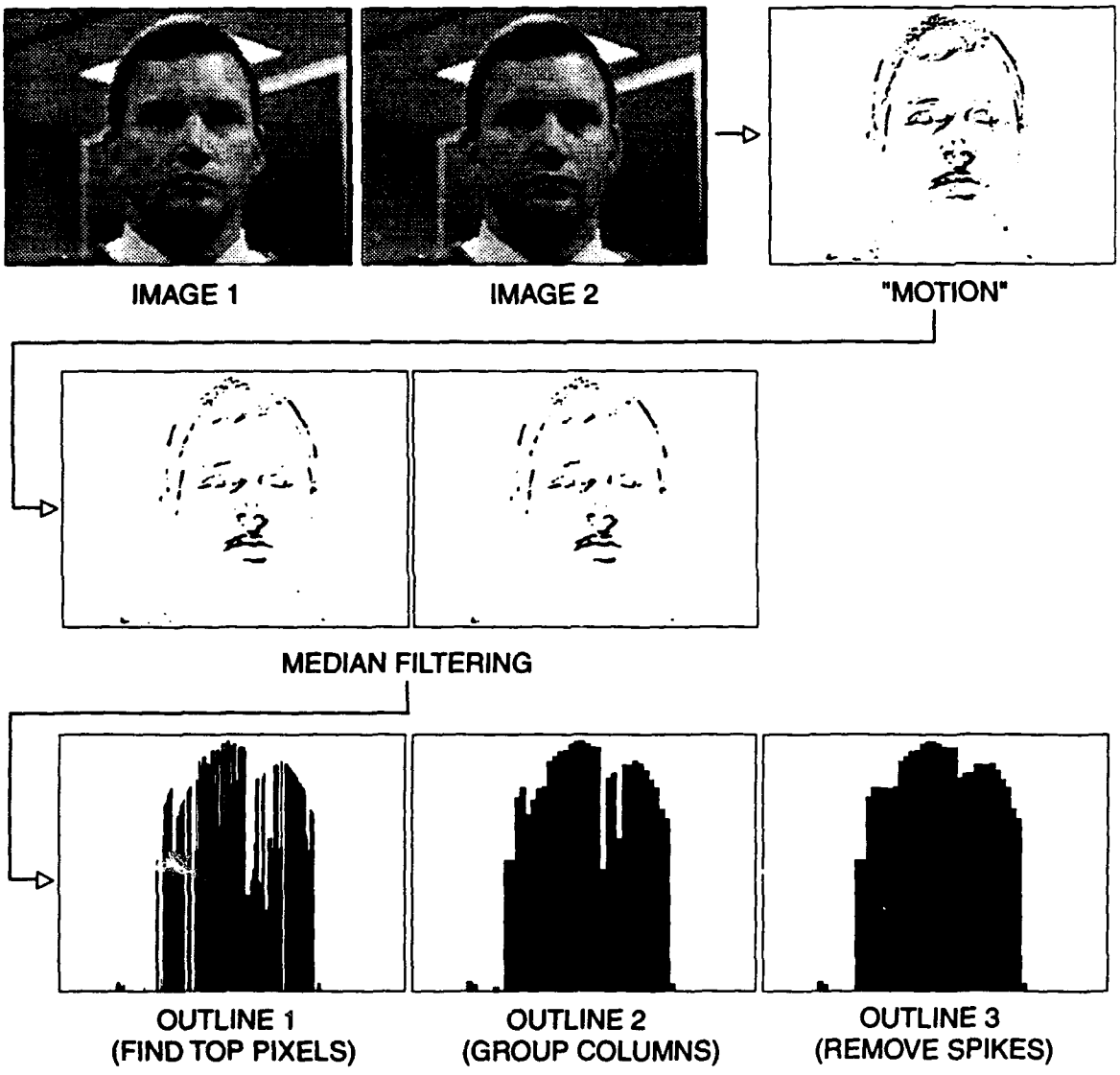


Figure 9. Forming a Motion Region Image

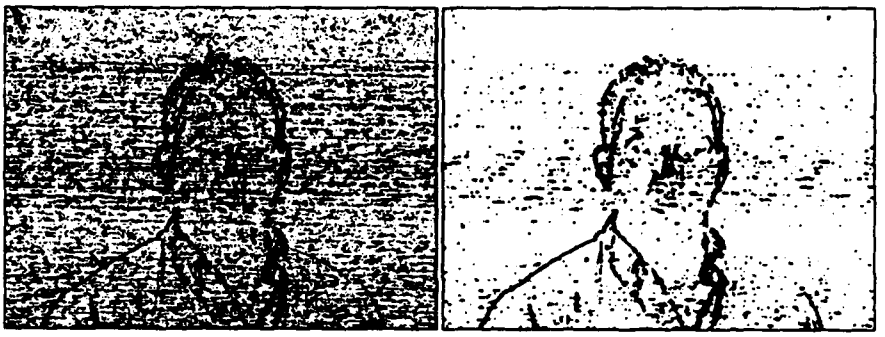


Figure 10. Motion Image From RGB Images

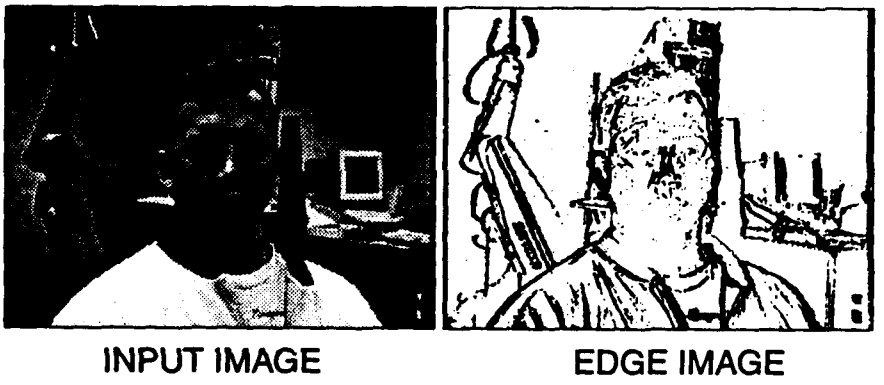


Figure 11. Sample Edge Image

4.4 Segmentation Algorithm Test

Possible face regions are segmented according to the algorithm described in Chapter three. This segmentation algorithm searches the motion image for large positive slope-level region-large negative slope regions. The image is searched using several slope thresholds to account for poor motion images and variations between subjects. The top of a motion outline of a face was rarely flat with a smooth curve at the sides of the head. Many times there were small positive and negative slopes along the top of the outline and very often small plateaus at the side of the head. By ignoring small slopes that did not exceed some moderate threshold, these small deviations could easily be overcome. Some hairstyles however caused a more moderate slope at the sides of the head, forcing lower thresholds to detect the sides of the head. By searching the motion outline three times using three different slope thresholds, it was experimentally found that all these irregularities could be overcome.

A visual representation of the entire segmentation process is provided in Figure 12.

The first part of the segmentation test was performed by videotaping several subjects and allowing the segmentation routine to segment all regions that might be valid faces. The segmentation test was run three times on the videotaped subjects with three different thresholds on the number of motion pixels. Figure 13 shows a sample 8-bit grey scale input image, the motion image associated with that image, the segmented regions in that image, and the standard size image (32 x 32) for each segmented region. A complete set of these segmentation test images are provided in appendix C. For scale comparison, all of the segmented regions for the highest motion threshold test are provided in Figure 14.

The segmentation test results showed (a) each "successful" capture consistently had at least one "properly" segmented face, and (b) "properly" segmented faces were not one, but a few, consistent scales. A "successful" capture is one in which the number of pixels changed in the motion image exceeded a pre-defined threshold.

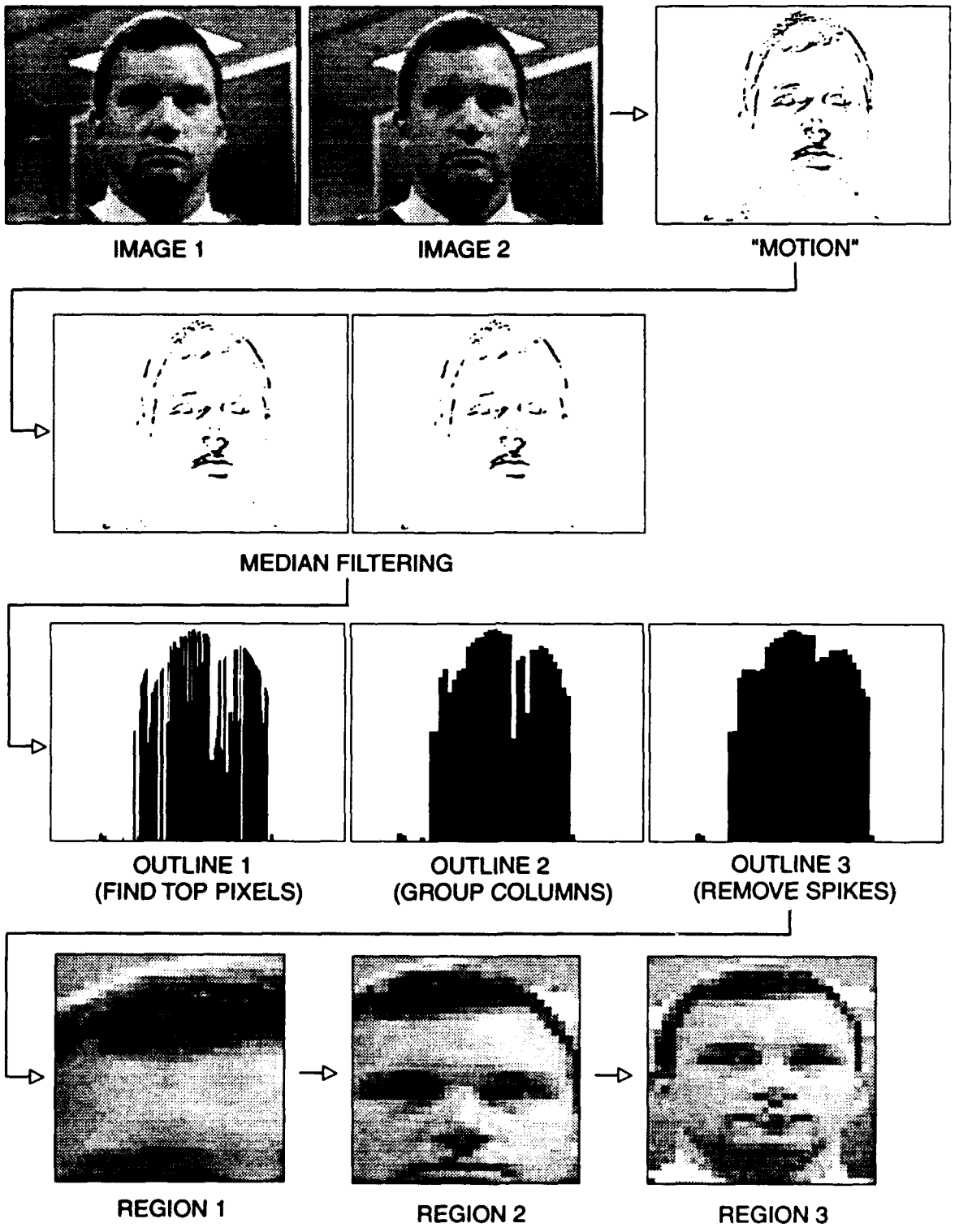


Figure 12. The Segmentation Process

Figure 13. Sample Segmentation Test Images

"Properly" segmented means the regions segmented from the original image appear suitable for input into a face recognition system. The lowest motion threshold setting (1500) had a few badly segmented face regions due to the poor motion regions on the front end of the segmentation algorithm. The higher threshold settings (3000 and 4500) consistently segmented suitable faces for recognition provided that one existed in the input scene.

The next part of this test investigated the scale "normalization" capability of this algorithm. The test was conducted by placing a single subject in the system field of view at three very different "zoom" settings on the camera. The three "zoom" settings made the subject's face three very different sizes in the input image. Three faces were autonomously segmented at each of the three "zoom" settings, resulting in a total of nine faces segmented for that subject. The results of this test, shown in Figure 15, and the results of the previous segmentation tests, indicate the segmentation and rescaling is somewhat tolerant of changes in the input scale, although more prototypes of each subject may need to be taken to account for the slight variations in scale that did occur.

Face orientation may alter scale depending on the facial view segmented, but each view will be a consistent scale. Thus, if a recognition system should attempt to handle multiple orientations, prototypes taken from different viewpoints using this algorithm should still be suitable.

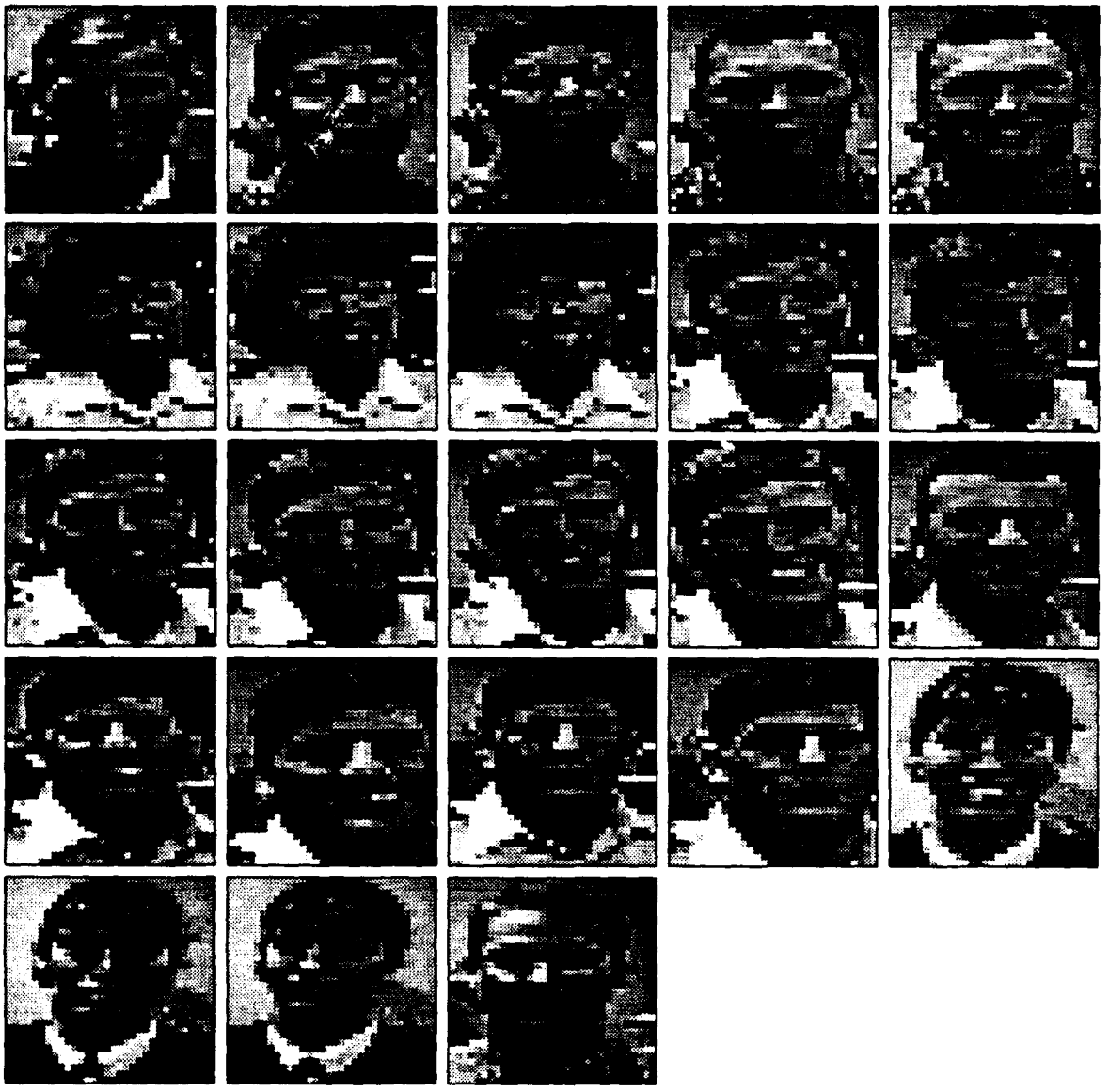


Figure 14. Sample Set of Segmented Regions

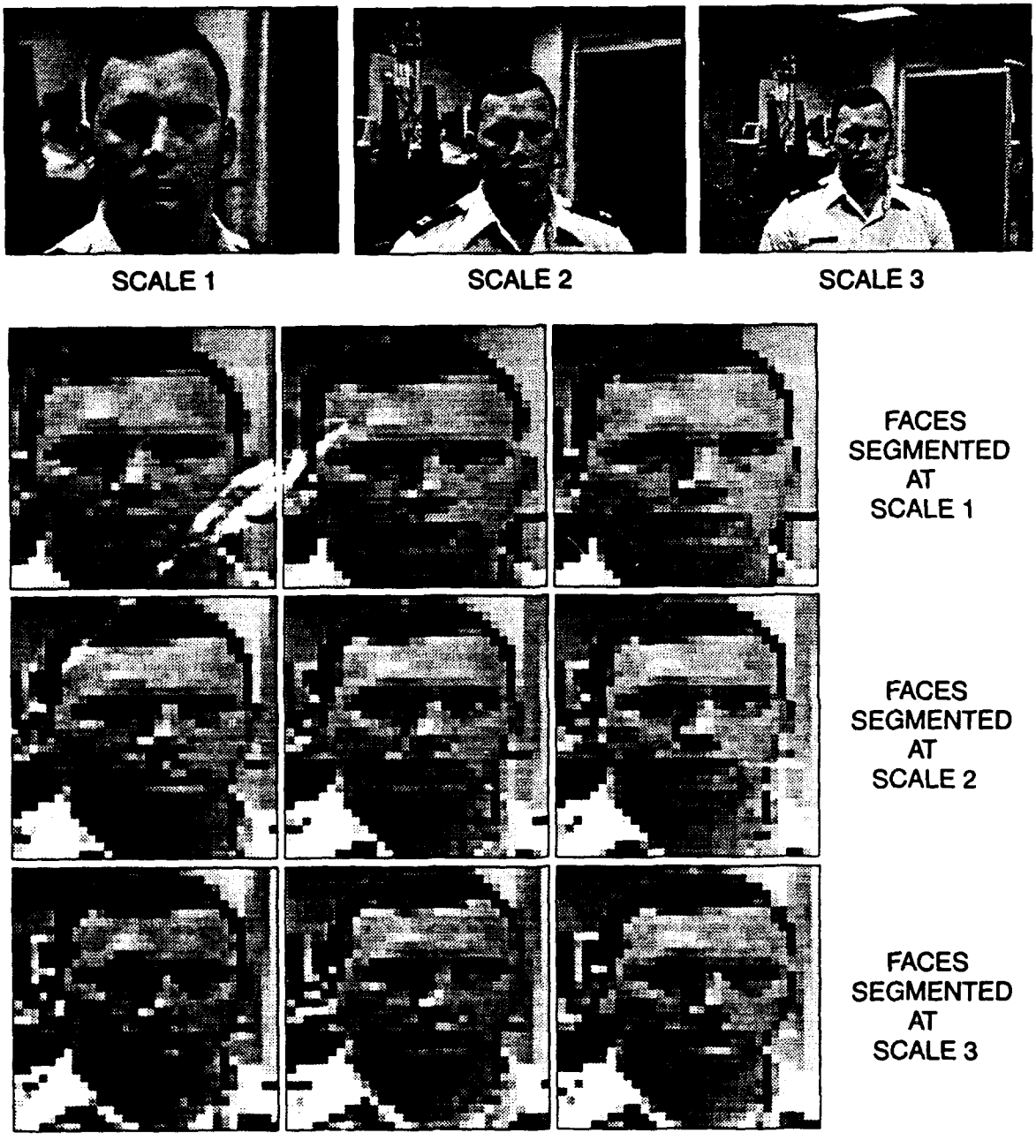


Figure 15. Segmentation Scale Test

Faces segmented using this algorithm may be further tested for their suitability an input to a face recognizer in a complementary thesis by Runyon (22).

4.5 Binarized Face Patterns

Figure 16 shows the 8-bit grey scale image used as the input for the "lambertized" images and the variation images which resulted from lambertizing the image using the local brightness variation box sizes shown and then thresholding the result. The grey areas are positive variations from the mean, and the black areas are the negative variations from the mean. Figure 17 is the result of lambertization and thresholding the same input image except it had been reduced from 640×480 pixels to 128×96 pixels.

The lambertized images vary according to the size of the box relative to the dimension of the input image, but the eyes, nose, and mouth appear consistently as black regions. Since these images are local variations, the lambertized images should be consistent over a large range of lighting conditions. Therefore, by either estimating the face size in the input image or lambertizing the image with several different boxes, or both, faces may be found by developing an algorithm which searches for the eyes, nose, and mouth regions.

Two ways this algorithm might be combined with the motion analysis technique are:

1. Use motion analysis to segment a possible face region, and then use use the lambertization technique to find facial features. This may better discriminate faces from non-faces, and may even reveal something of face orientation.
2. Use Lambertization to find possible faces, and then use the motion analysis to discriminate faces from non-faces.

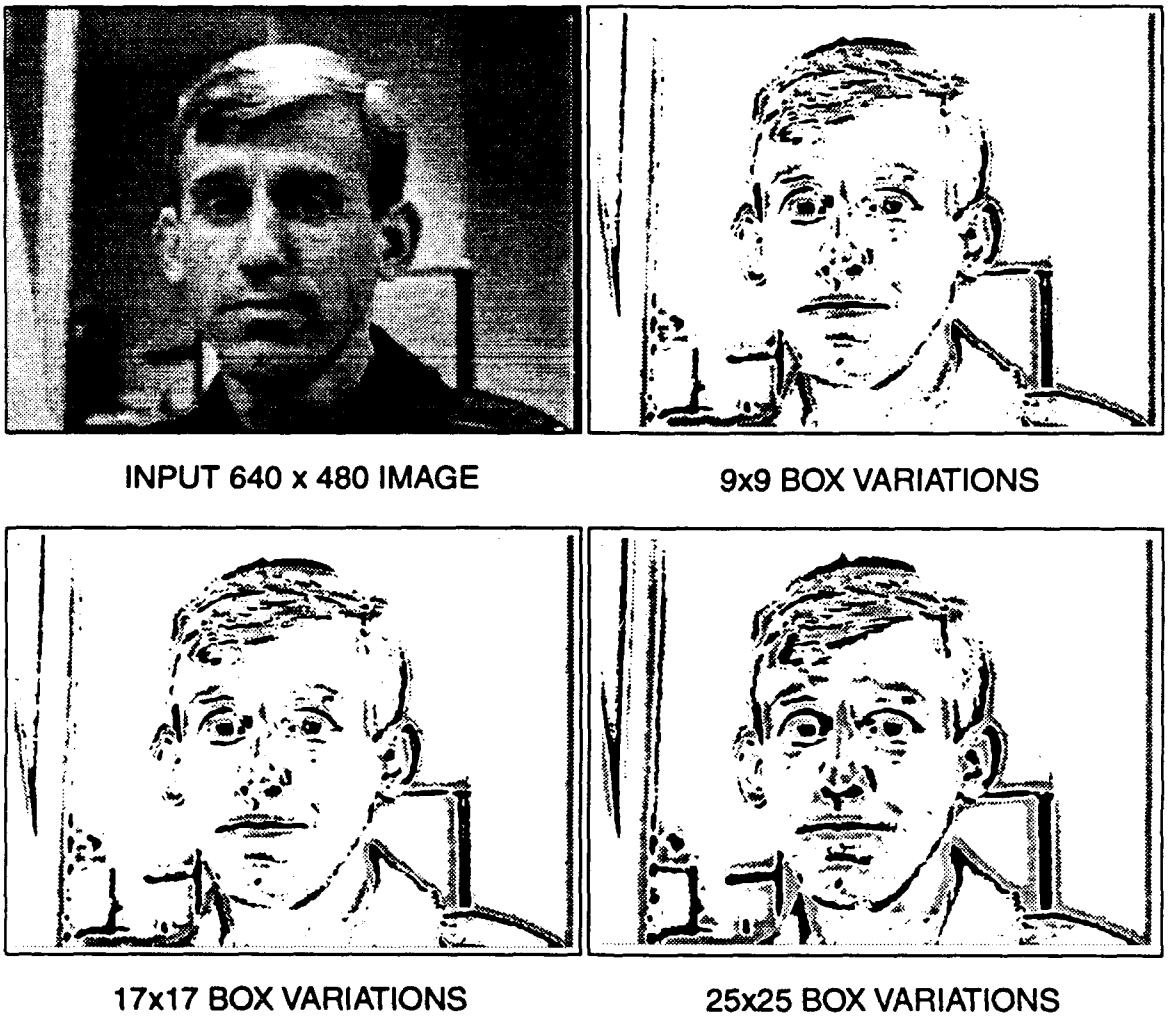


Figure 16. Lambertization Images

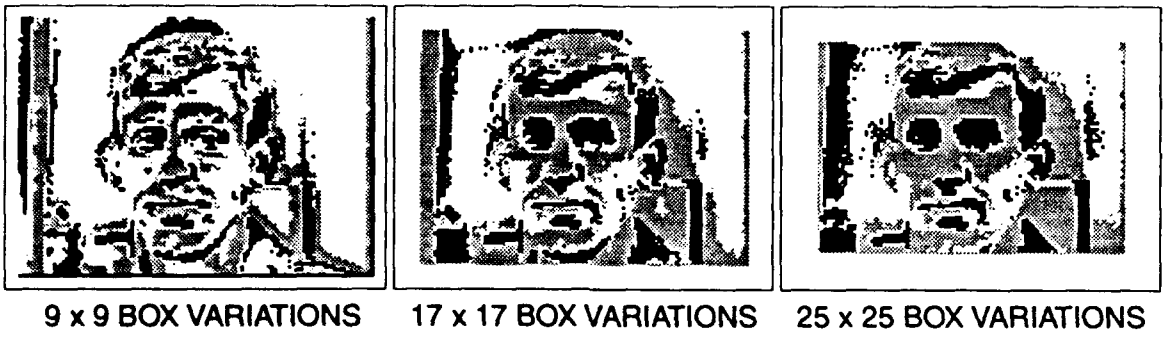


Figure 17. Reduced Input Lambertization Images

The advantages of the first method are it may be quicker and provide more information. The advantage of the second method is it may find faces even when motion may not be available.

V. Conclusions

This thesis demonstrates that motion analysis alone may provide sufficient segmentation for a face recognition system. Even though the motion outlines were not as consistent as expected, the system consistently found the heads in a scene and rescaled these heads to a new scale which was fairly consistent across all subjects.

This segmentation technique carries a low computational burden and is still relatively quick. As such, it may be a valuable technique to fuse with other segmentation techniques, such as searching for face patterns, to (a) reduce their search time, and (b) increase the consistency of discriminating face regions.

The only drawback in this thesis was the consistency of the motion images. Why the motion outlines were not consistently defined is not clear. Objects of similar grey scales values might be expected to obscure some boundaries, but this does not explain the results shown since the same objects and background produce outlines most of the time, and only occasionally do not. The more likely explanation is that the subjects did not move in direction of the plane of view.

With a better method of finding the motion image, the system could improve the scale standardization and perhaps even its ability to discriminate head regions from non-head regions. Although the scale of the segmented faces was consistent, it varied slightly depending on the width of the motion region. This is due to the fact that the motion outline did not find the edge of the head exactly. If the motion region found the outline of the head exactly, scale would be consistent for each person, and all background could be eliminated as well. Additionally, with a better motion image, a more discriminating analysis could be conducted on the motion image.

Appendix A. **VideoPix**

A.1 General

VideoPix is a tool which provides image grabbing and manipulation capability to Sun SPARCStations. The tool consists of (a) an electronics card inserted into the SPARCStation system, and (b) "C" software routines. There is a users' manual titled *Using VideoPix* which explains how to install the system and describes the software routines.

This appendix discusses some of the software routines and some of the problems associated with using these routines. There is no attempt to discuss installation, but to discuss the software, it is useful to have some understanding of some hardware capabilities. Appendix B has code which use these routines; this code may be useful as examples.

A.2 Hardware

The input video signal is decoded into 4:1:1 *YUV* data. This translates to two bytes per pixel, seven bits of Y (luminance) per pixel, and fourteen bits of UV (chrominance) per *four* pixels. The upper byte of each pixel word contains the seven Y data bits, and the lower byte contains either two or four of the fourteen UV data bits and it takes four pixel words to describe the UV for any one pixel.

The *YUV* image data is also based on non-square pixels; the decoder used by the tool was originally designed for televisions. Thus, to display the digitized images correctly, the data must be converted to square pixel data. The non-square data can be displayed, but it will appear distorted in the horizontal direction.

When an image is digitized, the images data is stored in two large FIFOs, one for each field of the video frame. Each FIFO read increments an internal pointer. Since the FIFO memory is serial access only, reading a specific pixel requires that

all pixels up to that location must be read as well. The dimensions of the digitized *YUV* data is 720×480.

A.3 Initialization

The key VideoPix initialization software routines are:

vfc_open() Opens the hardware and locks out other users.

vfc_set_port() Supposedly sets the hardware to look for a signal on the specified port (either either VFC_PORT1, VFC_PORT2, or VFC_SVIDEO). This routine did not appear to work. If only one signal is coming into the Sun SPARCStation, it will usually find that signal regardless of the port specified in this routine. With multiple active signals, the tool seems to default to port 1. The only consistent method for selecting a particular port seemed to be invoking the VideoPix vfctool prior to executing user created software, and previewing the video signal on the desired port. Then, after exiting the vfctool, VideoPix seems to always find the correct port when executing the user defined software.

vfc_set_format() Determines the format type from the incoming signal

An example of user software module which uses these software routines can be found in Appendix B, z_vfc_set_hw.c.

When the VideoPix hardware is no longer needed, the vfc_destroy() routine releases the hardware.

A.4 Grab and Convert

The key VideoPix routines to grab image data[1] and convert that data to 8-bit grey scale or rgb color are:

vfc_grab() Instructs the hardware to digitize the next complete frame. This puts the data into the FIFOs. Invoking vfc_yuvread_ntsc() is necessary to put the data into user memory where it can be manipulated.

vfc_yuvread_ntsc() Reads in digitized *YUV* image data from the VideoPix hardware into a memory block which has been allocated by the user. vfc_grab() and vfc_yuvread_ntsc() are usually executed together.

vfc_yuv2y8_ntsc() Converts the non-square pixel *YUV* data into square pixel, 8-bit grey scale data. Again, the user must allocate memory for the converted data (remember the *YUV* data is two bytes per pixel, the 8-bit grey scale is one byte per pixel). A point the user manual is not very clear on here is that since the *YUV* data is actually 7-bit of luminance, the 8-bit data is created by multiplying the 7-bit data by 2. VideoPix does this multiplication via look-up tables. As a consequence, vfc_init_lut() should be invoked prior to executing this routine to initialize the look-up tables. According to Sun, a colormap offset of zero is typical.

vfc_yuv2rgb_ntsc() Converts the non-square pixel *YUV* data into NTSC RGB color data. The color data is four bytes per pixel, and the data for each pixel is put into memory XGBR. The upper byte, X, which is evidently transparency information, was not used in this research. The memory allocation and look-up table comments given in the vfc_yuv2y8_ntsc() discussion apply here as well.

An example of user software modules which use these software routines can be found in Appendix B, z_grab_gra.c and z_grab_rgb.c.

[1] All functions which are format specific have both NTSC and PAL versions; the only format referred in this appendix will be NTSC, but it appears that PAL can replace NTSC in all cases.

Appendix B. Source Code

This appendix contains a listing of some of the source code used in performing this thesis. This code is presented as is, and no claims are made as to suitability for other applications.

B.1 segment2.c

```
/*
 * File:    segment2.c
 * Created:    July 1992
 * By:       Kevin Gay
 *
 * Purpose:
 *
 * Assumes:
 *
 * Modified:
 * By:
 * Why:
 */

#include <stdio.h>
#include <sys/types.h>
#include "vfc_lib.h"
#include "globals.h"

extern struct head_ptrs       *z_segment();
extern int            z_store_image();

main()
{
   u_char            *face, *ptr;
   register int       i;
   struct head_ptrs    *temp_ptr, *face_ptrs;
   char            tryagain[10],filename[30],command[80];
   int            count=1;


   sprintf(tryagain,"%s","Y");
   while((tryagain[0]=='Y')||(tryagain[0]=='y'))
      {
      printf("Please look at camera until you hear a beep.\n");
```

```
        face_ptrs = (struct head_ptrs *)z_segment();

        system("echo T");

        if(face_ptrs == NULL)
        {
        printf("Trouble getting images\n");
        exit(1);
        }
        else
        {
        while(face_ptrs ≠ NULL)
           {
           sprintf(filename,
              "%s%d%s","face",count,".sm");
           if(z_store_image(face_ptrs→head,filename,SM_SIZE)<0)
              fprintf(stderr,"Unable to write to file\n");
           sprintf(command,"graytorle -o hold.rle %d %d %s",
                    SM_WIDTH,SM_HEIGHT,filename);
           sprintf(filename,
              "%s%d%s","face",count,".rle");
           system(command);
           sprintf(command,"rleflip -v -o %s hold.rle",filename);
           system(command);
           system("rm hold.rle");
           sprintf(command,"xli -quiet -zoom 500 %s &",filename);
           system(command);
           temp_ptr = face_ptrs;
           face_ptrs = temp_ptr→next;
           free(temp_ptr→head);
           free(temp_ptr);
           count++;
           }
        }

        printf("Would you like to try again? (Y or N)");
        gets(tryagain);
        printf("\n");
        }
}
```

B.2 z_segment.c

```
/*****************************************************************************
 * File:    z_segment.c
 * Created:     August 1992
 * By:        Kevin Gay
 *
 * Purpose:    This code is a set of common routines to save time/typing.
 *        This code grabs all potential head regions in an image,
 *        reduces them (SM_SIZE) and returns these reduced head images
 *        in a structure; a null_ptr is returned if unsuccessful.
 *
 * Assumes:
 *
 * Modified:
 * By:
 * Why:
 *****************************************************************************/

#include <stdio.h>
#include <sys/types.h>
#include "vfc_lib.h"
#include "globals.h"

extern struct image_ptrs    *z_motion();
extern u_char            *z_seg_regions();
extern struct region        *z_outline();

struct head_ptrs *z_segment()
{
   u_char            *face;
   struct region        *head_regions, *face_ptr;
   struct image_ptrs    *im_ptrs;
   struct head_ptrs    *head_list=0, *temp_head, *null_ptr=0;


   im_ptrs = (struct image_ptrs *)z_motion();
   if(im_ptrs == NULL)
      {
      printf("Trouble getting images\n");
      return null_ptr;
      }

   head_regions=(struct region *)z_outline(im_ptrs->motion,
                VFC_NTSC_WIDTH,VFC_NTSC_HEIGHT);

   if(head_regions == NULL)
      printf("No head regions found\n");
```

```
    else
        {
        while(head_regions ≠ NULL)
            {
            face=(u_char *)z_seg_regions(im_ptrs→image2,
                    head_regions,VFC_NTSC_WIDTH,
                    VFC_NTSC_HEIGHT,SM_WIDTH,SM_HEIGHT);
            if(face == NULL)
                {
                printf("face ptr is null\n");
                }
            else
                {
                temp_head=(struct head_ptrs *)
                    malloc(sizeof(struct head_ptrs));
                if(temp_head==NULL)
                    {
                    printf("Problems adding to list\n");
                    return null_ptr;
                    }
                else
                    {
                    temp_head→next=head_list;
                    temp_head→head = face;
                    head_list = temp_head;
                    }
                }
            face_ptr = head_regions;
            head_regions = face_ptr→next;
            free(face_ptr);
            }
        }

    free(im_ptrs→image1);
    free(im_ptrs→image2);
    free(im_ptrs→motion);
    free(im_ptrs);
    free(head_regions);

    printf("z_segment complete\n\n");

    return head_list;
}
```

B.3 z_motion.c

```
/*****************************************************************************
 * File:    z_motion.c
 * Created:     August 1992
 * By:      Kevin Gay
 *
 * Purpose:    This code is a set of common routines to save time/typing.
 *      This code grabs two 8-bit grey images using z_grab_gra
 *      and finds the difference between the two images (to find
 *      out what was moving) using z_find_diff.
 *      The three images (both 8-bit grey and the difference)
 *      are put into a structure which is returned; a null_ptr
 *      is returned if unsuccessful.
 *      The images are all NTSC_SIZE (640x480) and 1bpp.
 *      The vfc hardware is set upon entry and released prior
 *      to departure from routine.
 *
 * Assumes:
 *
 * Modified:
 * By:
 * Why:
 *****************************************************************************/

#include <stdio.h>
#include <sys/types.h>
#include "vfc_lib.h"
#include "globals.h"

extern u_char        *z_set_vfc_hw();
extern u_char        *z_grab_gra();
extern u_char        *z_find_diff();
extern int       z_median();

struct image_ptrs *z_motion()
{
   u_char            *image[2], *motion, *ptr;
   register int       i;
   int           pixels_changed=0;
   struct hw_controls    *hw_ptrs;
   struct image_ptrs    *images, *null_ptr=0;


   hw_ptrs = (struct hw_controls *)z_set_vfc_hw();
   if(hw_ptrs == NULL)
      return null_ptr;
```

```
/*
 * Create a difference image and check to see if there is enough
 * movement to exceed threshold.
 */

while(pixels_changed<MOTION_THRESHOLD)
   {
   for(i=0; i<2; i++)
      {
        image[i]=(u_char *)z_grab_gra(hw_ptrs);
        if(image[i] == NULL)
        {
        printf("image ptr is null\n");
        return null_ptr;
        }
      if(i==1)
        {
        motion=(u_char *)z_find_diff(image[i-1],
                image[i], NTSC_SIZE);
          if(motion == NULL)
          {
          printf("motion ptr is null\n");
          return null_ptr;
          }
        }
      }
   for(i=1; i≤2; i++)
      if(z_median(motion,VFC_NTSC_WIDTH,VFC_NTSC_HEIGHT)<0)
        fprintf(stderr,"Median filtering error\n");
   ptr = motion;
   pixels_changed = 0;
   for(i=1; i≤(NTSC_SIZE*2/3); i++)
      if(*ptr++ == 0)
        pixels_changed++;
   }

images = (struct image_ptrs *)malloc(sizeof(struct image_ptrs));
if (images == NULL)
   {
   printf("Malloc error");
   return null_ptr;
   }

images→image1=image[0];
images→image2=image[1];
images→motion=motion;
```

```
    vfc_destroy(hw_ptrs->vfc_dev);
    free(hw_ptrs);

    printf("z_motion is complete\n\n");

    return images;
}
```

B.4 z_set_vfc_hw.c

```
/*****************************************************************************
 * File:    z_set_vfc_hw()
 * Created:     7 July 1992
 * By:        Kevin Gay
 *
 * This code was taken from an example written by Sun which came with the
 * VideoPix card.  The example was hw_setup.c which was labelled
 *      Copyright (c) 1990 by Sun Microsystems, Inc.
 *     #ident "@(#)hw_setup.c 1.5 90/12/12 SMI"
 *
 * Purpose:    The code is intended to initialize the VideoPix hardware.
 *        Returns hardware pointers if successful; null_ptr if not.
 *
 * Assumes:    Incoming signal is NTSC format; PORT set in global.h
 *
 * Modified:
 * By:
 * Why:
 *****************************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include "vfc_lib.h"
#include "globals.h"

struct hw_controls *z_set_vfc_hw()
{
   int          rc, format, v_format, wm_offset;
   VfcDev            *vfc_dev;
   struct hw_controls    *hw_ptrs, *null_ptr=0;

   /*
    * Open the hardware just use the default case
    * "/dev/vfc0" and test for success.
    * Lock the hardware to deny other programs access
    */

   /***** Open the vfc hardware and set a software lock ******/

   if((vfc_dev = vfc_open(NULL, VFC_LOCKDEV)) == NULL)
      {
      fprintf(stderr,"Could not open hardware\n");
      return null_ptr;
      }

   /** Set the input signal port (PORT defined in globals.h **/
```

```
    vfc_set_port(vfc_dev, PORT);

    /**** Determine format of incoming signal and lock on *****/

    rc = vfc_set_format(vfc_dev, VFC_AUTO, &format);
    if(rc < 0)
        {
        fprintf(stderr,"No video signal detected\n");
        return null_ptr;
        }

    if(format == NO_LOCK)
        {
        fprintf(stderr,"Unable to lock onto signal\n");
        return null_ptr;
        }
    v_format = vfc_video_format(format);

    if(v_format ≠ VFC_NTSC)
            {
        fprintf(stderr,"Warning : must be NTSC format\n");
            return null_ptr;
            }

    if(v_format == VFC_NTSC)
        vfc_adjust_hue(vfc_dev, 0);

    /***** Initialize colormap offset and look-up tables *****/

    wm_offset = 0; /* Initialize H/W colormap offset; usu. 0 */
    vfc_init_lut(wm_offset); /* initialize look-up tables    */

    /********* Assign harware control variables **************/

    hw_ptrs=(struct hw_controls *)malloc(sizeof(struct hw_controls));
    if(hw_ptrs == NULL)
        {
        fprintf(stderr,"malloc hw_ptrs error\n");
        exit(1);
        }
    hw_ptrs→vfc_dev = vfc_dev;
    hw_ptrs→colormap_offset = wm_offset;

    printf("z_set_vfc_hw complete\n\n");

    return hw_ptrs;
}
```

B.5 z_grab_gra.c

```
/*****************************************************************************
 * File:    z_grab_gra.c
 * Created:     3 June 1992
 * By:      Kevin Gay
 *
 * Purpose:   The code is intended to allocate memory for image data,
 *      grab YUV image data, and then
 *      convert that data to 8-bit grey, square pixel data.
 *      The code returns a pointer to the 8 bit grey data if
 *      successful, or a NULL if an error has occurred.
 *
 * Assumes:   z_set_vfc_hw.c has been executed.
 *      YUV data is 2 bytes per pixel (bpp), 8 bit gray 1 bpp.
 *
 * Modified:
 * By:
 * Why:
 *****************************************************************************/

#include <stdio.h>
#include <sys/types.h>
#include "vfc_lib.h"
#include "globals.h"

u_char *z_grab_gra(hw_ptrs)
struct hw_controls    *hw_ptrs;

{
   u_char       *yuv_data, *im_data, *mem_ptr, *null_ptr = NULL;
   register int    i;

   /*
    * Allocate space for images.
    */

   yuv_data = (u_char *)malloc(YUV_SIZE);
   if(yuv_data == NULL) {
      perror("malloc");
      return null_ptr;
   }

   im_data = (u_char *)malloc(NTSC_SIZE);
   if(im_data == NULL) {
      perror("malloc");
      free(yuv_data);
      return null_ptr;
```

```
    }

/*    printf("YUV and 8-bit image memories created\n");
*/
    /*
     * Do a grab and read in an image
     */

    if(vfc_grab(hw_ptrs→vfc_dev) < 0) {
        /*
         * If the grab fails the image read will
         * be garbage.
         */
        fprintf(stderr,"Warning, grab failed\n");
        free(yuv_data);
        free(im_data);
        return null_ptr;
    }
    if(vfc_yuvread_sq(hw_ptrs→vfc_dev, yuv_data, VFC_NTSC) < 0) {
        fprintf(stderr,"Warning, vfc_yuvread failed\n");
        free(yuv_data);
        free(im_data);
        return null_ptr;
    }
/*    printf("YUV image grabbed and read from the hardware\n");
*/

        /*
         * Convert the YUV data into 8-bit gray square pixel data.
         */

    if (hw_ptrs→colormap_offset ≠ 0)
          vfc_init_lut(hw_ptrs→colormap_offset);
        vfc_yuv2y8_ntsc(yuv_data, im_data);

/*        printf("YUV data converted to 8-bit gray data\n");
*/
    free(yuv_data);

    printf("z_grab_gra complete\n\n");

    return im_data;
}
```

B.6 z_find_diff.c

```
/*****************************************************************************
 * File:    z_find_diff.c
 * Created:     9 July 1992
 * By:      Kevin Gay
 *
 * Purpose:    The code is intended to allocate memory and store
 *      the pixel by pixel difference between two images.
 *      The resulting image is binarized (255 or 0) depending
 *      whether the difference exceeds the DIFFERENCE_THRESHOLD.
 *      Difference pixels exceeding the threshold are 0 -
 *      255 is white, 0 is black; difference is 0 to avoid
 *      a toner test when printing out difference images.
 *      The code return a pointer to the difference data; the
 *      pointer will point to NULL if an error has occurred.
 *
 * Assumes:    Both input images are 8 bit grey image data (1 byte/pixel)
 *      and the same size (difference data is 1bpp and same size
 *      as well).
 *
 * Modified:
 * By:
 * Why:
 *****************************************************************************/

#include <stdio.h>
#include <sys/types.h>
#include "vfc_lib.h"
#include "globals.h"

#define DIFFERENCE_THRESHOLD 10

u_char *z_find_diff(im1_data, im2_data, size)
u_char    *im1_data, *im2_data;
int    size;
{
   u_char       *im1_ptr, *im2_ptr, *diff_data, *diff_ptr,
         *null_ptr = 0;
   register int    i;

   diff_data = (u_char *)malloc(size);
   if(diff_data == NULL) {
      perror("malloc");
      return null_ptr;
   }

   im1_ptr = im1_data;
```

```
    im2_ptr = im2_data;
    diff_ptr = diff_data;
    for(i=0; i<(size); i++)
          {
          if((abs(*im1_ptr++ - *im2_ptr++)-DIFFERENCE_THRESHOLD)>0)
           *diff_ptr++ = 0;
         else
           *diff_ptr++ = 255;
         }

    printf("z_find_diff complete\n\n");

    return diff_data;
}
```

B.7 z_median.c

```
/*****************************************************************************
 * File:    z_median.c
 * Created:     10 July 1992
 * By:        Kevin Gay
 *
 * Purpose:    The code is intended to median filter an input image.
 *        The code creates a temp memory, and filters as follows:
 *         - check if 1st row, last row, or other
 *           - check if 1st column, last column, or other
 *             - set pixel to majority value of pixels which
 *               "surround" that pixel.
 *        Returns 1 if successful, -1 if not.
 *
 * Assumes:    The input image data is binarized (either 255 or 0)
 *         8 bit grey image data (1 byte/pixel).
 *
 * Modified:
 * By:
 * Why:
 *****************************************************************************/

#include <stdio.h>
#include <sys/types.h>
#include "vfc_lib.h"
#include "globals.h"

int z_median(im_data, width, height)
u_char    *im_data;
int    width, height;
{
   u_char        *med_data, *im1_ptr, *im2_ptr;
   register int    i, j;
   int        sum;

   med_data = (u_char *)malloc(width*height);
   if(med_data == NULL) {
      perror("malloc");
      return -1;
   }

   im1_ptr = im_data;
   im2_ptr = med_data;
   for(i=1; i≤height; i++)        /*******************/
      {              /* Since 0 is "on"  */
      for(j=1; j≤width; j++)        /* let ties goes to */
         {           /* 0 (best 5 of 9)  */
```

```
        if(i==1)        /*******************/
            {
/**** First Row ****/       if(j==1)
/* - first column  */       sum = *im1_ptr++ + *(im1_ptr+1)+*(im1_ptr+width)
/* - last column   */           +*(im1_ptr+width+1) + 255 + 255;
/* - other columns */       else if(j==width)
            sum = *im1_ptr++ + *(im1_ptr-1) +
               *(im1_ptr+width) + *(im1_ptr+width-1) +
               255 + 255;
            else
            sum = *im1_ptr++ + *(im1_ptr-1) + 255 +
               *(im1_ptr+width) + *(im1_ptr+width-1) +
               *(im1_ptr+width+1)+*(im1_ptr+1);
            }
        else if(i==height)
            {
/**** Last Row *****/       if(j==1)
/* - first column  */       sum = *im1_ptr++ + *(im1_ptr+1) +
/* - last column   */           *(im1_ptr-width) + *(im1_ptr-width+1) +
/* - other columns */           255 + 255;
            else if(j==width)
            sum = *im1_ptr++ + *(im1_ptr-1) +
               *(im1_ptr-width) + *(im1_ptr-width-1) +
               255 + 255;
            else
            sum = *im1_ptr++ +*(im1_ptr-1)+*(im1_ptr+1)+
               *(im1_ptr-width) + *(im1_ptr-width-1) +
               *(im1_ptr-width+1) + 255;
               }
        else
            {
/*** Other Rows ****/       if(j==1)
/* - first column  */       sum = *im1_ptr++ + *(im1_ptr+1) + 255 +
/* - last column   */        *(im1_ptr-width) + *(im1_ptr-width+1) +
/* - other columns */        *(im1_ptr+width) + *(im1_ptr+width+1);
            else if(j==width)
            sum = *im1_ptr++ + *(im1_ptr-1) + 255 +
               *(im1_ptr-width) + *(im1_ptr-width-1) +
               *(im1_ptr+width) + *(im1_ptr+width-1);
            else
            sum = *im1_ptr++ +*(im1_ptr-1)+*(im1_ptr+1)+
               *(im1_ptr-width) + *(im1_ptr-width-1) +
               *(im1_ptr-width+1) + *(im1_ptr+width) +
               *(im1_ptr+width-1) + *(im1_ptr+width+1);
            }
        if(sum≥(5*255))
            *im2_ptr++ = 255;
```

```
            else
                *im2_ptr++ = 0;
            }
        }
    im1_ptr = im_data;
    im2_ptr = med_data;
    for(i=0; i<(height*width); i++)
        *im1_ptr++ = *im2_ptr++;
    free(med_data);

    printf("z_median complete\n\n");

    return 1;
}
```

B.8 z_outline.c

```
/*****************************************************************************
 * File:    z_outline.c
 * Created:     August 1992
 * By:        Kevin Gay
 *
 * Purpose:    This code makes an outline image of moving regions by
 *        looking for the first motion pixel from the top in each
 *        column.  Some grouping is done to clean up the outline
 *        and some spikes are removed.  Then possible head regions
 *        are found by finding positive-zero-negative slope regions.
 *        The code to create a outline image remains, but it is
 *        commented out.
 *
 * Assumes:
 *
 * Modified:
 * By:
 * Why:
 *****************************************************************************/

#include <stdio.h>
#include <sys/types.h>
#include "vfc_lib.h"
#include "globals.h"

#define MAX_WIDTH    VFC_NTSC_WIDTH        /** 640 **/
#define MAX_HEIGHT    VFC_NTSC_HEIGHT        /** 480 **/
#define GROUPING 10
#define MIN_FACE_WIDTH 100

extern u_char    *z_store_image();

struct region *z_outline(motion, width, height)
u_char    *motion;
int    width, height;
{
   register int    i, j, slope_threshold;
   int        top[MAX_WIDTH], shorttop[MAX_WIDTH/GROUPING],
          top_slope[(MAX_WIDTH/GROUPING)-1],
          found, leave_loop, repeat, not_smaller,
          r_col, l_col, temp, remainder;
   u_char        *im_ptr, *outline;
   char        newname[30];
   struct region    *ptr, *ptr2, *poss_heads=0;
   static int    cnt=0;
```

```
if (width>MAX_WIDTH)
   {
   printf("Too wide for routine\n");
   return poss_heads;
   }

if (height>MAX_HEIGHT)
   {
   printf("Too high for routine\n");
   return poss_heads;
   }

/****** Find first motion pixels in each column (from top)
   (skip first ten rows in case of border and
    skip first/last ten columns in case of border) ********/

im_ptr = motion;
for(i=0; i<10; i++)
   top[i] = height;
for(i=10; i<(width-10); i++)
   {
   j=11;
   im_ptr = motion+(10*width)+i;
   found = 0;
   while((found==0)&&(j≤height))
      {
      if(*im_ptr == 0)
         {
         top[i] = j;
         found = 1;
         }
      else if (j++ == height)
         top[i] = height;
      im_ptr+=(width);
      }
   }
for(i=(width-10); i<width; i++)
   top[i] = height;

/********* Clean up top edges by grouping columns ***********/

for(i=0; i<(width/GROUPING); i++)
   {
   shorttop[i] = height;
   for(j=0; j<GROUPING; j++)
      if(top[(GROUPING*i)+j]<shorttop[i])
         shorttop[i] = top[(GROUPING*i)+j];
```

```
    }

/********** Find slopes from top **********/

for(i=0; i<((width/GROUPING)-1); i++)
    {
    top_slope[i]=((shorttop[i+1]-shorttop[i])/(GROUPING*2)*(-1));
    remainder = abs((shorttop[i+1]-shorttop[i])%(GROUPING*2));
    if((GROUPING-remainder)≥0)
        if(shorttop[i+1]>shorttop[i])
            top_slope[i]--;
        else
            top_slope[i]++;
    }

/********** Eliminate some spikes **********/

for(i=0; i<((width/GROUPING)-2); i++)
    if(abs(top_slope[i])≥1)
        {
        if((abs(top_slope[i]+top_slope[i+1])<top_slope[i])&&
            (abs(top_slope[i+1])≥1))
            {
            shorttop[i+1] = shorttop[i];
            top_slope[i] = 0;
            top_slope[i+1] = ((shorttop[i+2] -
                shorttop[i+1])/(GROUPING*2)*(-1));
            }
        else if((abs(top_slope[i]+top_slope[i-1])<top_slope[i])
            &&(i>0)&&(abs(top_slope[i-1])≥1))
            {
            shorttop[i] = shorttop[i-1];
            top_slope[i-1] = 0;
            top_slope[i] = ((shorttop[i+1] -
                shorttop[i])/(GROUPING*2)*(-1));
            }
        }
    /********** Find potential faces from the top **********/

for(slope_threshold=1; slope_threshold≤3; slope_threshold++)
    {
    l_col=0;
    while(l_col<((width/GROUPING)-3))
    {
    found = 0;
    if(top_slope[l_col]>slope_threshold)
        {
```

```
        r_col=l_col+1;
        leave_loop=0;
        while(leave_loop==0)
           {
/************************/    if(abs(top_slope[r_col])≤slope_threshold)
/* Beginning at leftmost */       r_col++;
/* column, find first col*/    else if(top_slope[r_col]>slope_threshold)
/* with pos slope (l_col)*/       {
/* above threshold.      */        l_col=r_col;
/* While searching for   */       r_col++;
/* neg slope above thres-*/        }
/* hold (r_col), update  */    else /*top_slope[r_col]<((-1)*slope_threshold)*/
/* l_col w/each pos slope*/        {
/* above threshold       */       temp=l_col-1;
/************************/        not_smaller=1;
              while((temp≥0)&&(not_smaller==1))
                {
                if(top_slope[temp]≥top_slope[l_col])
                 {
                 l_col=temp;
                 temp--;
                 }
                else
/************************/           not_smaller=0;
/* Once col w/neg slope  */         }
/* is found (r_col), push*/       temp=r_col+1;
/* l_col back until slope*/       while((temp<((width/GROUPING)-1))
/* is decreasing, and    */             &&(leave_loop==0))
/* push r_col forward    */         {
/* until slope increase  */          if(top_slope[temp]≤top_slope[r_col])
/************************/             {
                 r_col=temp;
                 temp++;
                 }
                else
                 leave_loop=1;
                }
              if(((r_col-l_col)*GROUPING)
                   ≥MIN_FACE_WIDTH)
/******************************/      {
/* If pos-neg slope region is */      ptr=(struct region *)
/* found, check for duplicate */            malloc(sizeof(struct region));
/* regions.  If not duplicate, */     ptr→x=(l_col*GROUPING)+1;
/* then add to linked list of */      ptr→width=(r_col+2)*
/* possible head regions      */            GROUPING-ptr→x;
/******************************/      ptr→y = shorttop[r_col];
                for(i=l_col; i<r_col; i++)
```

```
                    if(ptr→y > shorttop[i])
                      ptr→y = shorttop[i];
                  if((ptr→y+ptr→width*3/2)>height)
                    ptr→height=(height−ptr→y);
                  else
                    ptr→height=ptr→width*3/2;
                  repeat=0;
                  ptr2=poss_heads;
                  while(ptr2≠NULL)
                  {
                  if((ptr→x==ptr2→x)&&
                    (ptr→y==ptr2→y)&&
                    (ptr→width==ptr2→width))
                     repeat=1;
                  ptr2=ptr2→next;
                  }
                  if(repeat==0)
                  {
                     ptr→next=poss_heads;
                     poss_heads = ptr;
                  }
                  else
                  {
                  free(ptr);
                  }
                  }
               leave_loop = 1;
               l_col=r_col;
               }
            if(r_col≥((width/GROUPING)−1))
               {
               l_col = r_col;
               leave_loop = 1;
               }
            }
         }
      else
         l_col++;
      }
      }

   /********** Create an image of the motion outline **********/

/*       cnt++;

      outline = (u_char *)malloc(width*height);
      if(outline == NULL)
```

```
        {
        perror("malloc");
        return;
        }
    im_ptr = outline;
    for(i=0; i<width; i++)
        {
        im_ptr=outline;
        im_ptr+=i;
        for(j=1; j<shorttop[i/10]; j++)
            {
            *im_ptr = 255;
            im_ptr+=(width);
            }
        for(j=1; j<=((height-shorttop[i/10])+1); j++)
            {
            *im_ptr = 0;
            im_ptr+=(width);
            }
        }

    sprintf(newname,"%s%d%s","outline",cnt,".gra");
    if(z_store_image(outline, newname, width*height) < 0)
        fprintf(stderr,"Unable to write %s to file\n",newname);

    free(outline);
*/
    printf("z_outline complete\n\n");

    return poss_heads;
}
```

B.9 z_seg_regions.c

```
/*****************************************************************************
 * File:    z_seg_regions.c
 * Created:     August 1992
 * By:      Kevin Gay
 *
 * Purpose:   This routine takes in an image and a region structure
 *      which specifies the rectangular region to be segmented
 *      from an image.
 *      The segmented portion is then reduced to some specified
 *      dimension and returned; null ptr is returned if error.
 *      The code for view_face is left in, but commented out.
 *      The code for view_face will create an image the same
 *      dimensions as the input image with only those pixels
 *      to be segmented assigned grey scale values. The
 *       remaining pixels are white (255).
 *
 * Assumes:
 *
 * Modified:
 * By:
 * Why:
 *****************************************************************************/

#include <stdio.h>
#include <sys/types.h>
#include "vfc_lib.h"
#include "globals.h"

extern u_char    *z_reduce();
extern int    z_store_image();

u_char *z_seg_regions(image, head, width, height, seg_width, seg_height)
u_char       *image;
struct region    *head;
int       seg_width, seg_height;
{
   u_char       *ptr, *ptr2, *face, *seg_face, *view_face,
         *null_ptr=0;
   char       newname[30];
   register int    i, j;
   static int    cnt=0; /* Allows multiple calls with unique */
            /* filenames.                    */

   /******* Segment the potential face from the image **********/

   face = (u_char *)malloc(head->width*head->height);
```

```
    if(face == NULL)
        {
        perror("malloc");
        return null_ptr;
        }

    ptr = image;
    ptr2 = face;
    ptr+=(((head→y−1)*width)+(head→x−1));
    for(j=0; j<head→height; j++)
        {
        for(i=0; i<(head→width); i++)
            *ptr2++ = *ptr++;
        ptr+=(width−head→width);
        }

    seg_face=(u_char *)z_reduce(face,head→width,head→height,
                    seg_width,seg_height);
    if(seg_face == NULL)
        {
        printf("seg_face ptr is null\n");
        return null_ptr;
        }

    /********************* Create view_face ********************/

/*  cnt++;

    view_face = (u_char *)malloc(width*height);
    if(view_face == NULL)
        {
        perror("malloc");
        return null_ptr;
        }

    ptr2 = view_face;
    for(i=0; i<(width*height); i++)
        *ptr2++ = 255;
    ptr = image;
    ptr2 = view_face;
    ptr+=(((head->y-1)*width)+(head->x-1));
    ptr2+=(((head->y-1)*width)+(head->x-1));
    for(j=0; j<head->height; j++)
        {
        for(i=0; i<head->width; i++)
            *ptr2++ = *ptr++;
        ptr+=(width-head->width);
```

```
        ptr2+=(width-head->width);
        }

    sprintf(newname,"%s%d%s","view",cnt,".gra");
    if(z_store_image(view_face, newname, width*height) < 0)
        fprintf(stderr,"Unable to write %s to file\n",newname);

    free(view_face);
*/
    /************ Free memory and return image ******************/

    free(face);

    printf("z_seg_regions complete\n\n");

    return seg_face;
}
```

B.10 z_reduce.c

```
/*****************************************************************************
 * File:    z_reduce.c
 * Created:     13 July 1992
 * By:      Kevin Gay
 *
 * Purpose:    The code is intended to reduce an input image, of some
 *      arbitrary height and width, to a reduced height
 *      and width, also an input.
 *      The reduction algorithm simply skips bits.
 *      The algorithm skips more bits on the side of the image
 *      to be reduced in cases where the width division is not even,
 *      and more bits are skipped on the bottom of the input image
 *      when the height division is not even.
 *      Returns a pointer to the memory block containing the
 *      reduced image data upon success; a null ptr if not.
 *
 * Assumes:    The input image data is 8 bit grey image data (1 byte/pixel).
 *
 * Modified:
 * By:
 * Why:
 *****************************************************************************/

#include <stdio.h>
#include <sys/types.h>
#include "vfc_lib.h"
#include "globals.h"

u_char *z_reduce(im_data, old_width, old_ht, new_width, new_ht)
u_char    *im_data;
int    old_width, old_ht, new_width, new_ht;
{
   u_char       *sm_data, *ptr1, *ptr2, *null_ptr = 0;
   register int    i, j;
   int       width_div, width_adj, left_adj, right_adj, ht_div,
          bottom_adj;

   sm_data = (u_char *)malloc(new_width*new_ht);
   if(sm_data == NULL) {
      perror("malloc");
      return null_ptr;
   }

   if ((old_width<new_width)||(old_ht<new_ht))
      {
      fprintf(stderr,"Too small to reduce.\n");
```

```
        return null_ptr;
        }

    width_div = old_width/new_width;
    width_adj = old_width%new_width;
    left_adj = width_adj/2;
    right_adj = new_width-left_adj-(width_adj%2);
    ht_div = ((old_ht/new_ht)-1);
    bottom_adj = new_ht-(old_ht%new_ht);

    ptr1 = sm_data;
    ptr2 = im_data;
    ptr2+=(width_div/2);
    for(j=0; j<new_ht; j++)
        {
        for(i=0; i<new_width; i++)
          {
            *ptr1++ = *ptr2;
          ptr2+=width_div;
          if((i<left_adj)||(i≥right_adj))
              ptr2++;
          }
      ptr2+=(old_width*ht_div);
      if(j≥bottom_adj)
          ptr2+=old_width;
        }

    printf("z_reduce complete\n\n");

    return sm_data;
}
```

B.11 z_store_image.c

```
/*************************************************************************
 * File:    z_store_image.c
 * Created:     8 July 1992
 * By:        Kevin Gay
 *
 * Purpose:    The code is intended to store image data in a file
 *        which is labelled <filename> which is pass in.
 *        Returns a 1 is successful; -1 if not.
 *
 * Assumes:
 *
 * Modified:
 * By:
 * Why:
 *************************************************************************/

#include <stdio.h>
#include <sys/types.h>
#include "vfc_lib.h"
#include "globals.h"

int z_store_image(image_ptr, filename, size)
u_char    *image_ptr;
char    *filename;
int    size;

{
    FILE          *dat_file;

      /*
       * Open image files for writing images
       * into memory; check to make sure they open okay.
       */

   if ((dat_file = fopen(filename,"w")) == NULL)
      {
      printf("I can't open the %s file\n",filename);
      return -1;
      }
/*       printf("Opened %s file\n",filename);
*/
      /*
       * Put the data into a file; close file.
       */

      if (fwrite(image_ptr, 1, size, dat_file) == 0){
```

```
        perror("write failed");
        return -1;;
    }
        fflush(stdout);
        fclose(dat_file);

    printf("z_store_image complete\n\n");

    return 1;
}
```

B.12 globals.h

```
/*
 * File:    globals.h
 * Created:     August 1992
 * By:      Kevin Gay
 *
 * Purpose:    Put all global variables and definitions in one place.
 *
 * Assumes:    vfc_lib.h is also included - all the vfc routines and
 *      vfc definitions are in vfc_lib.h.
 *
 * Modified:
 * By:
 * Why:
 */

#define PORT        "VFC_SVIDEO"
#define YUV_SIZE    VFC_NTSC_HEIGHT*VFC_YUV_WIDTH*2    /* 720 x 480 x 2bpp
*/
#define NTSC_SIZE     VFC_NTSC_WIDTH*VFC_NTSC_HEIGHT    /* 640 x 480 */
#define RED_WIDTH    128
#define RED_HEIGHT    96
#define RED_SIZE    RED_WIDTH*RED_HEIGHT
#define SM_WIDTH    32
#define SM_HEIGHT    32
#define SM_SIZE        SM_WIDTH*SM_HEIGHT
#define MOTION_THRESHOLD 3000

struct region
   {
   int        x;
   int        y;
   int        width;
   int        height;
   struct region    *next;
   };

struct hw_controls
   {
   VfcDev        *vfc_dev;
   int        colormap_offset;
   };

struct image_ptrs
   {
   u_char        *image1;
   u_char        *image2;
```

```
    u_char      *motion;
    };

struct head_ptrs
    {
    u_char          *head;
    struct head_ptrs    *next;
    };
```

B.13 grab.c

```
/*
 * File:    grab.c
 * Created:     8 July 1992
 * By:      Kevin Gay
 *
 * Purpose:    The code is intended to take in images in a loop.
 *          Each pass thru the loop takes in YUV image data,
 *      converts the data to 8-bit grey, square pixel data, and
 *      save the 8-bit data in a file labelled with person's name.
 *      The number of images (up to ten) and the root of the
 *      filenames are entered during execution.
 *
 * Assumes:
 *
 * Modified:
 * By:
 * Why:
 */

#include <stdio.h>
#include <sys/types.h>
#include "vfc_lib.h"
#include "globals.h"

extern VfcDev    *z_set_vfc_hw();
extern u_char    *z_grab_gra();
extern int    z_store_image();

void grab();

main()
{
char       file[30];
int        num_images;

   printf("Enter the filename root: ");
   gets(file);
   printf("\n");

   printf("Enter the number of images (10 max): ");
   scanf("%d",&num_images);
   printf("\n");

   grab(file, num_images);
   exit(0);
}
```

```
void
grab(name, num_loop)
char    name[20];
int    num_loop;
{
   u_char            *grey[10];
   register int       j;
   struct hw_controls    *hw_ptrs;
   char            filename[30];

   hw_ptrs = (struct hw_controls *)z_set_vfc_hw();

   if(num_loop>10)
      {
      num_loop = 10;
      printf("Sorry, limited to 10 images\n");
      }

   for(j=1; j≤num_loop; j++)
      {
      grey[j]=(u_char *)z_grab_gra(hw_ptrs);
      if(grey[j] == NULL)
         {
         printf("ptr is null\n");
         exit(1);
         }
      }

      /*
       * Create name for image data and store in file.
       */

   for(j=1; j≤num_loop; j++)
      {
      sprintf(filename,"%s%d%s",name,j,".gra");
      if(z_store_image(grey[j], filename, NTSC_SIZE) < 0)
         fprintf(stderr,"Unable to write %s to file\n",filename);
      }

      /*
    * Now destroy the hardware and free the memory.
    */

   vfc_destroy(hw_ptrs→vfc_dev);
   free(hw_ptrs);
   for(j=1; j≤num_loop; j++)
```

```
        free(grey[j]);

    return;
}
```

B.14 grabrgb.c

```
/*
 * File:    grabrgb.c
 * Created:     July 1992
 * By:      Kevin Gay
 *
 * Purpose:    The code is intended to take in images in a loop num_loop
 *          times.  Each pass thru the loop takes in YUV image data,
 *      converts the data to 8-bit color, square pixel data, and
 *      save the 8-bit data in a file labelled with person's name.
 *      Number of loops (num_loop) and person's name (person)
 *      are entered during execution.
 *
 * Assumes:    Signal coming in S-Video port, NTSC format
 *
 * Modified:
 * By:
 * Why:
 */

#include <stdio.h>
#include <sys/types.h>
#include "vfc_lib.h"

extern VfcDev    *z_set_vfc_hw();
extern u_char    *z_grab_rgb();
extern int    z_store_image();

extern int ntsc_size;

void grabrgb();

main()
{
char       file[30];
int        num_images;

   printf("Enter the filename root: ");
   gets(file);
   printf("\n");

   printf("Enter the number of images (10 max): ");
   scanf("%d",&num_images);
   printf("\n");

   grabrgb(file, num_images);
   exit(0);
```

```
}

void
grabrgb(name, num_loop)
char    name[20];
int    num_loop;
{
    u_char        *color[10];
    register int    j;
    VfcDev        *vfc_dev;
    char        filename[30];

    vfc_dev = (VfcDev *)z_set_vfc_hw();

    if(num_loop>10)
        {
        num_loop = 10;
        printf("Sorry, limited to 10 images\n");
        }

    for(j=1; j≤num_loop; j++)
        {
        color[j]=(u_char *)z_grab_rgb(vfc_dev);
        if(color[j] == NULL)
            {
            printf("ptr is null\n");
            exit(1);
            }
        }

        /*
         * Create name for image data and store in file.
         */

    for(j=1; j≤num_loop; j++)
        {
        sprintf(filename,"%s%d%s",name,j,".rgb");
        if(z_store_image(color[j], filename, (ntsc_size*4)) < 0)
            fprintf(stderr,"Unable to write %s to file\n",filename);
        }

        /*
      * Now destroy the hardware and free the memory.
     */

    vfc_destroy(vfc_dev);
    for(j=1; j≤num_loop; j++)
```

```
        free(color[j]);

    return;
}
```

B.15 motion.c

```
/*
 * File:    motion.c
 * Created:    July 1992
 * By:      Kevin Gay
 *
 * Purpose:
 *
 * Assumes:
 *
 * Modified:
 * By:
 * Why:
 */

#include <stdio.h>
#include <sys/types.h>
#include "vfc_lib.h"
#include "globals.h"

extern struct image_ptrs    *z_motion();
extern int          z_store_image();

main()
{
   u_char           *face, *ptr;
   register int      i;
   struct region        *head_regions, *face_ptr;
   struct image_ptrs    *im_ptrs;
   char          filename[30];


   im_ptrs = (struct image_ptrs *)z_motion();
   if(im_ptrs == NULL)
      {
      printf("Trouble getting images\n");
      exit(1);
      }

   sprintf(filename,"%s","motion.gra");
   if(z_store_image(im_ptrs- motion, filename, NTSC_SIZE) < 0)
      fprintf(stderr,"Unable to write %s to file\n",filename);

   sprintf(filename,"%s","image1.gra");
   if(z_store_image(im_ptrs→image1, filename, NTSC_SIZE) < 0)
      fprintf(stderr,"Unable to write %s to file\n",filename);
```

```
    sprintf(filename,"%s","image2.gra");
    if(z_store_image(im_ptrs→image2, filename, NTSC_SIZE) < 0)
        fprintf(stderr,"Unable to write %s to file\n",filename);

    free(im_ptrs→image1);
    free(im_ptrs→image2);
    free(im_ptrs→motion);
    free(im_ptrs);
    free(head_regions);
}
```

B.16 rgb_motion.c

```
/*
 * File:    rgb_motion.c
 * Created:    July 1992
 * By:      Kevin Gay
 *
 * Purpose:    The code is intended to take in images in a loop num_loop
 *          times.  Each pass thru the loop takes in YUV image data,
 *      converts the data to 8-bit color, square pixel data, and
 *      save the 8-bit data in a file labelled with person's name.
 *      Number of loops (num_loop) and person's name (person)
 *      are entered during execution.
 *
 * Assumes:    Signal coming in S-Video port, NTSC format
 *
 * Modified:
 * By:
 * Why:
 */

#include <stdio.h>
#include <sys/types.h>
#include "vfc_lib.h"
#include "globals.h"

#define DIFFERENCE_THRESHOLD 10

extern struct hw_controls    *z_set_vfc_hw();
extern u_char            *z_grab_rgb();
extern int           z_store_image();

main()
{
   u_char            *color[2], *im_ptr, *ptr2, *ptr3, *ptr4,
            *red[2], *green[2], *blue[2],
            *ptr2a, *ptr3a, *ptr4a, *motion;
   register int        j, k;
   struct hw_controls    *hw_ptrs;
   char            filename[30];

   hw_ptrs = (struct hw_controls *)z_set_vfc_hw();

   for(j=0; j<2; j++)
      {
      color[j]=(u_char *)z_grab_rgb(hw_ptrs);
      if(color[j] == NULL)
         {
```

```
        printf("ptr is null\n");
        exit(1);
        }
    }

    /*
     * Create name for image data and store in file.
     */

for(j=0; j<2; j++)
    {
    sprintf(filename,"%s%d%s","image",j,".rgb");
    if(z_store_image(color[j], filename, (NTSC_SIZE*4)) < 0)
        fprintf(stderr,"Unable to write %s to file\n",filename);
    }


for(j=0; j<2; j++)
    {
    blue[j] = (u_char *)malloc(NTSC_SIZE);
    if(blue[j] == NULL)
        {
        perror("malloc");
        exit(1);
        }

    green[j] = (u_char *)malloc(NTSC_SIZE);
    if(green[j] == NULL)
        {
        perror("malloc");
        exit(1);
        }

    red[j] = (u_char *)malloc(NTSC_SIZE);
    if(red[j] == NULL)
        {
        perror("malloc");
        exit(1);
        }

    im_ptr = color[j];
    ptr2 = blue[j];
    ptr3 = green[j];
    ptr4 = red[j];
    for(k=0; k<NTSC_SIZE; k++)
        {
        im_ptr++;
```

```
            *ptr2++ = *im_ptr++;
            *ptr3++ = *im_ptr++;
            *ptr4++ = *im_ptr++;
            }
         sprintf(filename,"%s%d%s","blue",j,".gra");
         if(z_store_image(blue[j], filename, (NTSC_SIZE)) < 0)
            fprintf(stderr,"Unable to write %s to file\n",filename);
         sprintf(filename,"%s%d%s","green",j,".gra");
         if(z_store_image(green[j], filename, (NTSC_SIZE)) < 0)
            fprintf(stderr,"Unable to write %s to file\n",filename);
         sprintf(filename,"%s%d%s","red",j,".gra");
         if(z_store_image(red[j], filename, NTSC_SIZE) < 0)
            fprintf(stderr,"Unable to write %s to file\n",filename);

         }

/*    motion = (u_char *)malloc(NTSC_SIZE);
      if(motion==NULL)
         exit(1);
      im_ptr = motion;
      ptr2 = blue[0];
      ptr2a = blue[1];
      ptr3 = green[0];
      ptr3a = green[1];
      ptr4 = red[0];
      ptr4a = red[1];
      for(j=0; j<NTSC_SIZE; j++)
         {
         if((abs(*ptr2a++ - *ptr2++)>DIFFERENCE_THRESHOLD) ||
            (abs(*ptr3a++ - *ptr3++)>DIFFERENCE_THRESHOLD) ||
            (abs(*ptr4a++ - *ptr4++)>DIFFERENCE_THRESHOLD))
            *im_ptr++ = 0;
         else
            *im_ptr++ = 255;
         }
      if(z_store_image(motion,"rgb_motion.gra", NTSC_SIZE) < 0)
         fprintf(stderr,"Unable to write motion to file\n");
*/
      /*
       * Now destroy the hardware and free the memory.
       */

      vfc_destroy(hw_ptrs->vfc_dev);
      for(j=0; j<2; j++)
         {
         free(color[j]);
         free(red[j]);
```

```
        free(green[j]);
        free(blue[j]);
        }
    free(motion);

    return;
}
```

B.17 seg_test.c

```
/*
 * File:    seg_test.c
 * Created:     August 1992
 * By:      Kevin Gay
 *
 * Purpose:
 *
 * Assumes:
 *
 * Modified:
 * By:
 * Why:
 */

#include <stdio.h>
#include <sys/types.h>
#include "vfc_lib.h"
#include "globals.h"

extern struct image_ptrs    *z_motion();
extern struct region        *z_top_slopes();
extern u_char               *z_seg_regions();
extern u_char               *z_reduce();
extern int                  z_store_image();

void seg_test();

main()
{
   seg_test();
   exit(0);
}

void
seg_test()
{
   char             newname[30];
   u_char           *face, *ptr;
   struct region    *temp_ptr, *head_list;
   struct image_ptrs    *images;
   register int     i;
   int              count;
   FILE             *dat_file;

   for(i=1; i<=20; i++)
```

```
{
images=(struct image_ptrs *)z_motion();
if(images==NULL)
{
printf("Trouble getting images\n");
exit(1);
}

 head_list = (struct region *) z_top_slopes(images→motion,
        VFC_NTSC_WIDTH,VFC_NTSC_HEIGHT);
if(head_list == NULL)
printf("No head regions found.\n");
else
  {
  count=1;
  while(head_list ≠ NULL)
{
face = (u_char *)z_seg_regions(images→image2,head_list,
   VFC_NTSC_WIDTH,VFC_NTSC_HEIGHT,SM_WIDTH,SM_HEIGHT);
if(face == NULL)
   {
   printf("face ptr is null\n");
   }
else
   {
        /*
       * Create name for the face data and store in file.
        */

   sprintf(newname,"%s%d%s%d%s","face",i,"-",count,".sm");
   if(z_store_image(face, newname, SM_SIZE) < 0)
      fprintf(stderr,
      "Unable to write %s to file\n",newname);
   }
temp_ptr = head_list;
head_list = temp_ptr→next;
free(temp_ptr);
count++;
}
  }

   /*
    * Create name for the motion data and store in file.
    */

sprintf(newname,"%d%s%d%s",MOTION_THRESHOLD,"_",i,".gra");
if(z_store_image(images→motion, newname, NTSC_SIZE) < 0)
```

```
        fprintf(stderr,"Unable to write %s to file\n",newname);

        free(images→motion);

            /*
             * Create name for the image data and store in file.
             */

        sprintf(newname,"%s%d%s","image",i,".gra");
        if(z_store_image(images→image2, newname, NTSC_SIZE) < 0)
        fprintf(stderr,"Unable to write %s to file\n",newname);

        free(images→image1);
        free(images→image2);

            /*
             * Now free the memory.
             */

        free(images);
        free(face);
        }

    return;
}
```

B.18 test_lam.c

```
/*
 * File:    test_bin.c
 * Created:    8 July 1992
 * By:       Kevin Gay
 *
 * Purpose:    The code tests the z_lambert routine.
 *
 * Assumes:
 *
 * Modified:
 * By:
 * Why:
 */

#include <stdio.h>
#include <sys/types.h>
#include "vfc_lib.h"
#include "globals.h"

extern u_char    *z_lambert();
extern u_char    *z_grab_gra();
extern u_char    *z_reduce();
extern int    z_store_image();
extern int    z_median();

main()
{
   char        filename[30], newname[30];
   u_char        *image1, *image2, *red_data;
   register int    i;
   int        check;
   FILE        *dat_file;

   printf("Enter the raw 8-bit grey file (640x480) to lambertize: ");
   gets(filename);
   printf("\n");

   /*
    * Allocate memory, read in image, and reduce it.
    */

   image1 = (u_char *)malloc(NTSC_SIZE);
   if(image1 == NULL)
      {
      perror("malloc");
      exit(1);
```

```
    }

  if ((dat_file = fopen(filename,"r")) == NULL)
      {
      printf("I can't open the %s file\n",filename);
      exit(1);
      }

  if(check=fread(image1, 1, NTSC_SIZE, dat_file)≠NTSC_SIZE)
    printf("Error reading in file");
  else
    {
    for(i=4; i≤12; i+=4)
      {
      image2=(u_char *)z_lambert(image1,VFC_NTSC_WIDTH,
                VFC_NTSC_HEIGHT,i,i);
      if(image2 == NULL)
        {
        printf("lambert ptr is null\n");
        exit(1);
        }
/*      sprintf(newname,"%s%d%s%d%s","var",
              2*i+1,"x",2*i+1,".gra");
      if(z_store_image(image2, newname, NTSC_SIZE) < 0)
        printf("Unable to write %s to file\n",newname);
*/
      free(image2);
      }
    red_data=(u_char *)z_reduce(image1,VFC_NTSC_WIDTH,
          VFC_NTSC_HEIGHT,RED_WIDTH,RED_HEIGHT);
    if(red_data==NULL)
      {
      printf("reduction error\n");
      exit(1);
      }
    /*if(z_store_image(red_data,"im128X96.red",RED_SIZE)<0)
      fprintf(stderr,"Unable to write red_data to file\n");*/
    for(i=4; i≤12; i+=4)
      {
      image2=(u_char *)z_lambert(red_data,RED_WIDTH,
                RED_HEIGHT,i,i);
      if(image2 == NULL)
        {
        printf("lambert ptr is null\n");
        exit(1);
        }
/*      sprintf(newname,"%s%d%s%d%s","red_var",
```

```
                    2*i+1,"x",2*i+1,".red");
            if(z_store_image(image2, newname, (RED_SIZE)) < 0)
               printf("Unable to write %s to file\n",newname);
*/
            free(image2);
            }
         /*free(red_data);*/
         }

      fclose(dat_file);

   free(image1);

   return;
}
```

B.19 time_grab.c

```
/*
 * File:    time_grab.c
 * Created:     8 July 1992
 * By:      Kevin Gay
 *
 * Purpose:    The code is intended to take in images in a loop.
 *          Each pass thru the loop takes in YUV image data,
 *      converts the data to 8-bit grey, square pixel data, and
 *      save the 8-bit data in a file labelled with person's name.
 *      The number of images (up to ten) and the root of the
 *      filenames are entered during execution.
 *
 * Assumes:
 *
 * Modified:
 * By:
 * Why:
 */

#include <stdio.h>
#include <sys/types.h>
#include "vfc_lib.h"
#include "globals.h"

extern struct hw_controls    *z_set_vfc_hw();
extern u_char            *z_grab_gra();
extern int           z_store_image();

main()
{
   u_char            *grey[10];
   register int       j;
   struct hw_controls    *hw_ptrs;
   char           filename[30];

   hw_ptrs = (struct hw_controls *)z_set_vfc_hw();

   system("date");

   for(j=0; j<20; j++)
      {
      grey[j]=(u_char *)z_grab_gra(hw_ptrs);
      if(grey[j] == NULL)
         {
         printf("ptr is null\n");
         exit(1);
```

```
        }
      }

   system("date");

      /*
       * Create name for image data and store in file.
       */

/*    for(j=0; j<20; j++)
      {
      sprintf(filename,"%s%d%s","image",j,".gra");
      if(z_store_image(grey[j], filename, NTSC_SIZE) < 0)
         fprintf(stderr,"Unable to write %s to file\n",filename);
      }
*/
      /*
     * Now destroy the hardware and free the memory.
     */

   vfc_destroy(hw_ptrs→vfc_dev);
   free(hw_ptrs);
   for(j=0; j<20; j++)
      free(grey[j]);

   return;
}
```

B.20 z_binarize_gra.c

```
/*
 * File:    z_binarize_gra.c
 * Created:     9 July 1992
 * By:        Kevin Gay
 *
 * Purpose:    The code is intended to allocate memory and store
 *        binary image data (255 or 0 depending on the THRESHOLD).
 *        The code return a pointer to the binary data; the
 *        pointer will point to NULL if an error has occurred.
 *
 * Assumes:    The image is the same size as
 *        the 8 bit gray image data (both 1 byte/pixel).
 *
 * Modified:
 * By:
 * Why:
 */

#include <stdio.h>
#include <sys/types.h>
#include "vfc_lib.h"

#define THRESHOLD 75

u_char *z_binarize_gra(gray_ptr, size)
u_char    *gray_ptr;
int    size;
{
   u_char        *bin_ptr, *temp_ptr, *temp2_ptr, *null_ptr = 0;
   register int    i;

   bin_ptr = (u_char *)malloc(size); /* 8-bit gray and binary 1 bpp */
   if(bin_ptr == NULL)
      {
      perror("malloc");
      return null_ptr;
      }

   temp_ptr = gray_ptr;
   temp2_ptr = bin_ptr;
   for(i=0; i<(size); i++)
      {
        if (*temp_ptr++ < THRESHOLD)
         *temp2_ptr++ = 0;
       else
              *temp2_ptr++ = 255;
```

```
        }

    printf("z_binarize_gra complete\n\n");

    return bin_ptr;
}
```

B.21 z_grab_rgb.c

```
/*
 * File:    z_grab_rgb.c
 * Created:     3 June 1992
 * By:      Kevin Gay
 *
 * Purpose:   The code is intended to allocate memory for image data,
 *      grab YUV image data, and then
 *      convert that data to rgb color, square pixel data.
 *      The code return a pointer to the rgb color data
 *      which will point to NULL if an error has occurred.
 *
 * Assumes:   H/W has been initialized and signal is NTSC format.
 *      YUV data is 2 bytes per pixel (bpp), rgb color 1 bpp.
 *
 * Modified:
 * By:
 * Why:
 */

#include <stdio.h>
#include <sys/types.h>
#include "vfc_lib.h"

#define YUV_SIZE VFC_NTSC_HEIGHT*VFC_YUV_WIDTH*2  /* 720 x 480 x 2bpp */

extern int ntsc_size;
extern int colormap_offset;

u_char *z_grab_rgb(hw_ptr)
VfcDev    *hw_ptr;

{
   u_char       *yuv_data, *im_data, *null_ptr = NULL;
   register int    i;

   /*
    * Allocate space for images.
    */

   yuv_data = (u_char *)malloc(YUV_SIZE);
   if(yuv_data == NULL) {
      perror("malloc");
      return null_ptr;
   }

   im_data = (u_char *)malloc(ntsc_size*4);
```

```
    if(im_data == NULL) {
        perror("malloc");
        free(yuv_data);
        return null_ptr;
    }

/*    printf("YUV and color image memories created\n");
*/
    /*
     * Do a grab and read in an image
     */

    if(vfc_grab(hw_ptr) < 0) {
        /*
         * If the grab fails the image read will
         * be garbage.
         */
        fprintf(stderr,"Warning, grab failed\n");
        free(yuv_data);
        free(im_data);
        return null_ptr;
    }
    if(vfc_yuvread_sq(hw_ptr, yuv_data, VFC_NTSC) < 0) {
        fprintf(stderr,"Warning, vfc_yuvread failed\n");
        free(yuv_data);
        free(im_data);
        return null_ptr;
    }
/*    printf("YUV image grabbed and read from the hardware\n");
*/

        /*
         * Convert the YUV data into 8-bit gray square pixel data.
         */

    if (colormap_offset ≠ 0)
          vfc_init_lut(colormap_offset);
        vfc_yuv2rgb_ntsc(yuv_data, im_data);

/*        printf("YUV data converted to rgb color data\n");
*/
    free(yuv_data);

    printf("z_grab_rgb complete\n\n");

    return im_data;
}
```

B.22 z_lambert.c

```
/*****************************************************************************
 * File:    z_lambert.c
 * Created:     August 1992
 * By:      Kevin Gay
 *
 * Purpose:    This code finds local variations in a image (thereby
 *      discounting object brightness) with a variable size "local
 *      box".  The code moves a box around each pixel in an image,
 *      finds the mean of the pixel values in the box (for each
 *      pixel) and then subtracts the mean from that pixel which is
 *      surrounded by the box.  The box is specified all the pixels
 *      some number away from the pixel of interest in the positive
 *      and negative direction along both the horizontal and vertical
 *      axes; the number of pixels in the horizontal direction is
 *      passed in as hor; the vertical, vert.
 *      A threshold is then set for the variations to identify regions
 *      both negative and bright with regard to their surroundings.
 *      The code to create a grey scale image by adding 128 to the
 *      variations found is left in, but commented out.
 *
 * Assumes:
 *
 * Modified:
 * By:
 * Why:
 *****************************************************************************/

#include <stdio.h>
#include <sys/types.h>
#include "vfc_lib.h"
#include "globals.h"

extern u_char    *z_store_image();

u_char *z_lambert(im_data, width, height, hor, vert)
u_char    *im_data;
int    width, height, hor, vert;
{
   u_char        *lam_data, *im_ptr, *lam_ptr, *null_ptr=0,
         *variation_data, *variation_ptr,
         *mean_data, *mean_ptr;
   register int    i, j, k, m;
   int        sum, mean, check,
         col_sum[VFC_NTSC_HEIGHT][VFC_NTSC_WIDTH],
         neigh[VFC_NTSC_HEIGHT][VFC_NTSC_WIDTH];
   char        newname[30];
```

```
/*    lam_data = (u_char *)malloc(width*height);
    if(lam_data == NULL)
        {
        perror("malloc");
        return null_ptr;
        }
*/
    variation_data = (u_char *)malloc(width*height);
    if(variation_data == NULL)
        {
        perror("malloc");
        return null_ptr;
        }

    im_ptr=im_data;

    for(i=0; i<width; i++)
      {
      col_sum[0][i]=0;
      for(j=0; j<(2*vert+1); j++)
        col_sum[0][i]+=*(im_ptr+(width*j)+i);
      }

    for(i=0; i≤width−(2*hor+1); i++)
      {
      neigh[0][i]=0;
      for(j=0; j<(2*hor+1); j++)
        neigh[0][i]+=col_sum[0][j];
      }

    for(i=1; i≤(height−(2*vert+1)); i++)
      {
      for(j=0; j<width; j++)
        col_sum[i][j]=col_sum[i−1][j] − *(im_ptr+(width*(i−1))+j) +
              *(im_ptr+(width*(i+(2*vert)))+j);
      neigh[i][0]=0;
      for(j=0; j<(2*hor+1); j++)
        neigh[i][0]+=col_sum[i][j];
      for(j=1; j≤(width−(2*hor+1)); j++)
        neigh[i][j]=neigh[i][j−1]+col_sum[i][j+2*hor]−col_sum[i][j−1];
      }


    im_ptr=im_data;
    variation_ptr=variation_data;
/*    lam_ptr=lam_data;
```

```
*/
   for(i=1; i≤height; i++)
    for(j=1; j≤width; j++)
       {
       if((i>vert)&&(i≤(height−vert))&&(j>hor)&&(j≤(width−hor)))
       {
       mean=neigh[i−vert][j−hor]/((1+2*vert)*(1+2*hor));
       check = 128 + ((*im_ptr++) − mean);
       if(check>255)
          check = 255;
       else if(check<0)
          check = 0;
/*       *lam_ptr++ = check;
*/
       if(check<120)
          *variation_ptr++ = 0;
       else if(check>136)
          *variation_ptr++ = 128;
       else
          *variation_ptr++ = 255;
       }
        else
       {
/*       *lam_ptr++ = 255;
*/
       *variation_ptr++ = 255;
       im_ptr++;
       }
        }

/*   if((width==VFC_NTSC_WIDTH)&&(height==VFC_NTSC_HEIGHT))
       {
       sprintf(newname,"%s%d%s%d%s","lam",2*hor+1,"x",
                        2*vert+1,".gra");
       if(z_store_image(lam_data,newname, (NTSC_SIZE)) < 0)
          fprintf(stderr,"Unable to write var_data to file\n");
       }
   else if((width==RED_WIDTH)&&(height==RED_HEIGHT))
       {
       sprintf(newname,"%s%d%s%d%s","red_lam",2*hor+1,"x",
                        2*vert+1,".red");
       if(z_store_image(lam_data,newname, (RED_SIZE)) < 0)
          fprintf(stderr,"Unable to write var_data to file\n");
       }
   free(lam_data);
*/
   printf("z_lambert complete\n\n");
```

```
    return variation_data;
}
```

B.23 z_side_edges.c

```
/*****************************************************************************
 * File:    z_side_edges.c
 * Created:     August 1992
 * By:      Kevin Gay
 *
 * Purpose:
 *
 * Assumes:
 *
 * Modified:
 * By:
 *****************************************************************************/

#include <stdio.h>
#include <sys/types.h>
#include "vfc_lib.h"
#include "globals.h"

#define SLOPE_THRESHOLD 8

extern int          z_store_image();

struct region *z_side_edges(image, width, height)
u_char   *image;
int   width, height;
{
   register int   i, j, run, threshold;
   int       slope[VFC_NTSC_HEIGHT][VFC_NTSC_WIDTH-1],
         count=0, found, leave_loop, repeat,
         not_smaller, r_col, l_col, temp,total=0;
   u_char       *im_ptr, *im_ptr2, *edge_image;
   char       newname[30];
   struct region    *poss_faces=0, *ptr, *ptr2;

   if (width>VFC_NTSC_WIDTH)
      {
      printf("Too wide for routine\n");
      return poss_faces;
      }

   if (height>VFC_NTSC_HEIGHT)
      {
      printf("Too high for routine\n");
      return poss_faces;
      }
```

```
/******************** Find slopes from side ********************/

edge_image = (u_char *)malloc(width*height);
if(edge_image == NULL)
    {
    perror("malloc");
    return poss_faces;
    }

for(run=1; run<=1; run++)
    {
    im_ptr2=edge_image;
    for(i=0; i<height; i++)
        {
        im_ptr=image;
        im_ptr+=(width*i);
       for(j=0; j<(width-run); j++)
        {
        if((j+1)%40==0)
        fprintf(dat_file,"\n");
        slope[i][j]=*(im_ptr+run)-*im_ptr++;
        if(abs(slope[i][j])>SLOPE_THRESHOLD)
        *im_ptr2++ = 0;
        else
        *im_ptr2++ = 255;
        }
    for(j=(width-run); j<width; j++)
        *im_ptr2++ = 255;
        }
    sprintf(newname,"%s%d%s","edge",run,".gra");
    if(z_store_image(edge_image, newname, width*height) < 0)
    fprintf(stderr,"Unable to write %s to file\n",newname);
    }


free(edge_image);

printf("z_side_edges complete\n\n");

return poss_faces;
}
```

Appendix C. **Segmentation Test Image Set**

C.1 General

The following figures is the complete set of images which resulted from the segmentation testing. Each row of these figures begin with a grey scale image, the associated motion image, and all the potential face images segmented from the input image followed by a reduced image of just the segmented region. Thus, there are instances when a segmented region image with its reduced image are shown and there is no grey scale or motion image preceding these images in that row. These instances represent the case of multiple segmented regions from a single motion image. As such, the grey scale and motion images in the row immediately preceding the row where there is no grey scale or motion images are the associated grey scale and motion images for those segmented regions as well. A complete set of only the segmented regions for each test are provided for comparison at the end of this appendix.

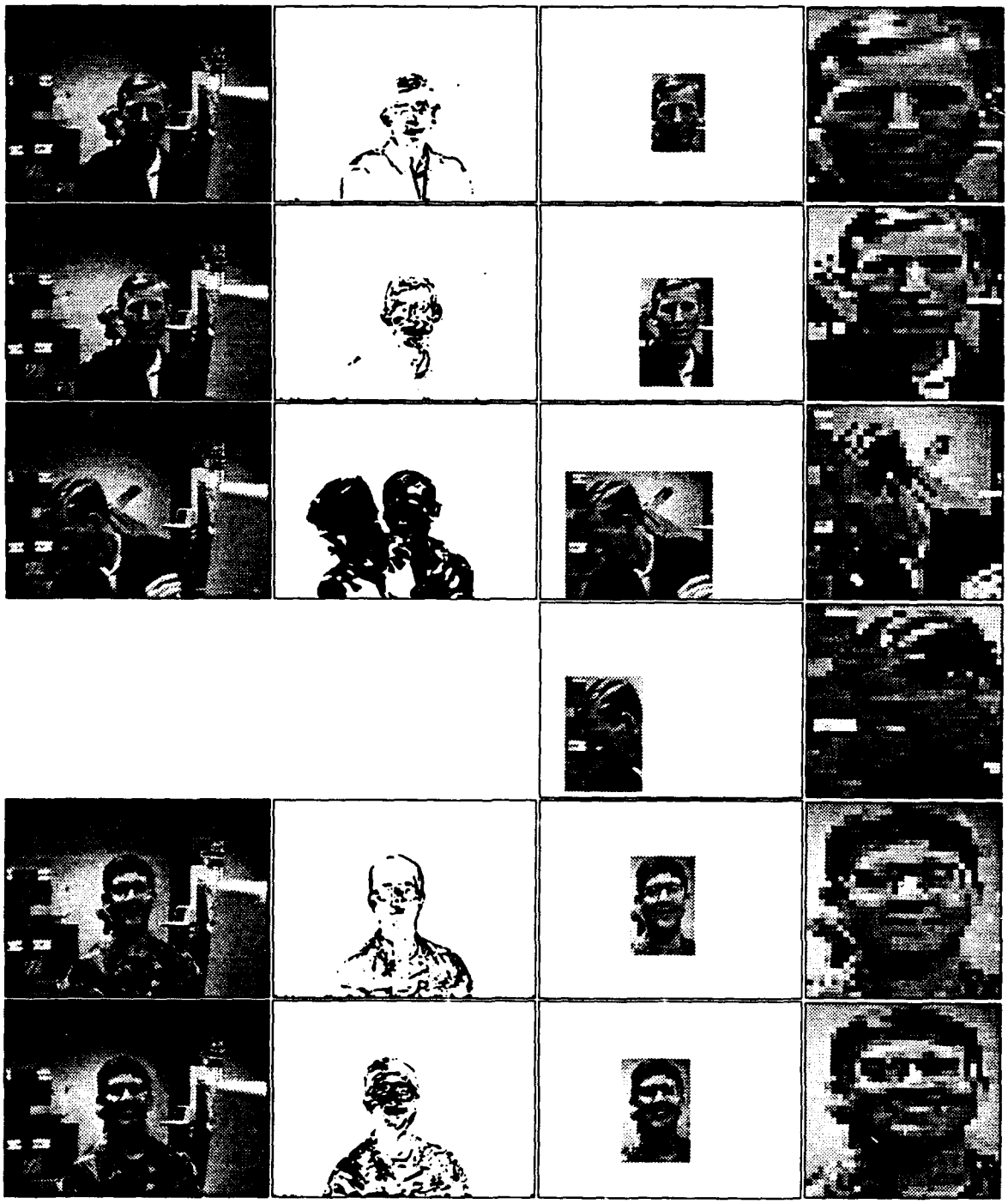


Figure 18. Segmentation Test I Images (1 of 4) [The Lowest Motion Threshold Tested]

Figure 19. Segmentation Test I Images (2 of 4)

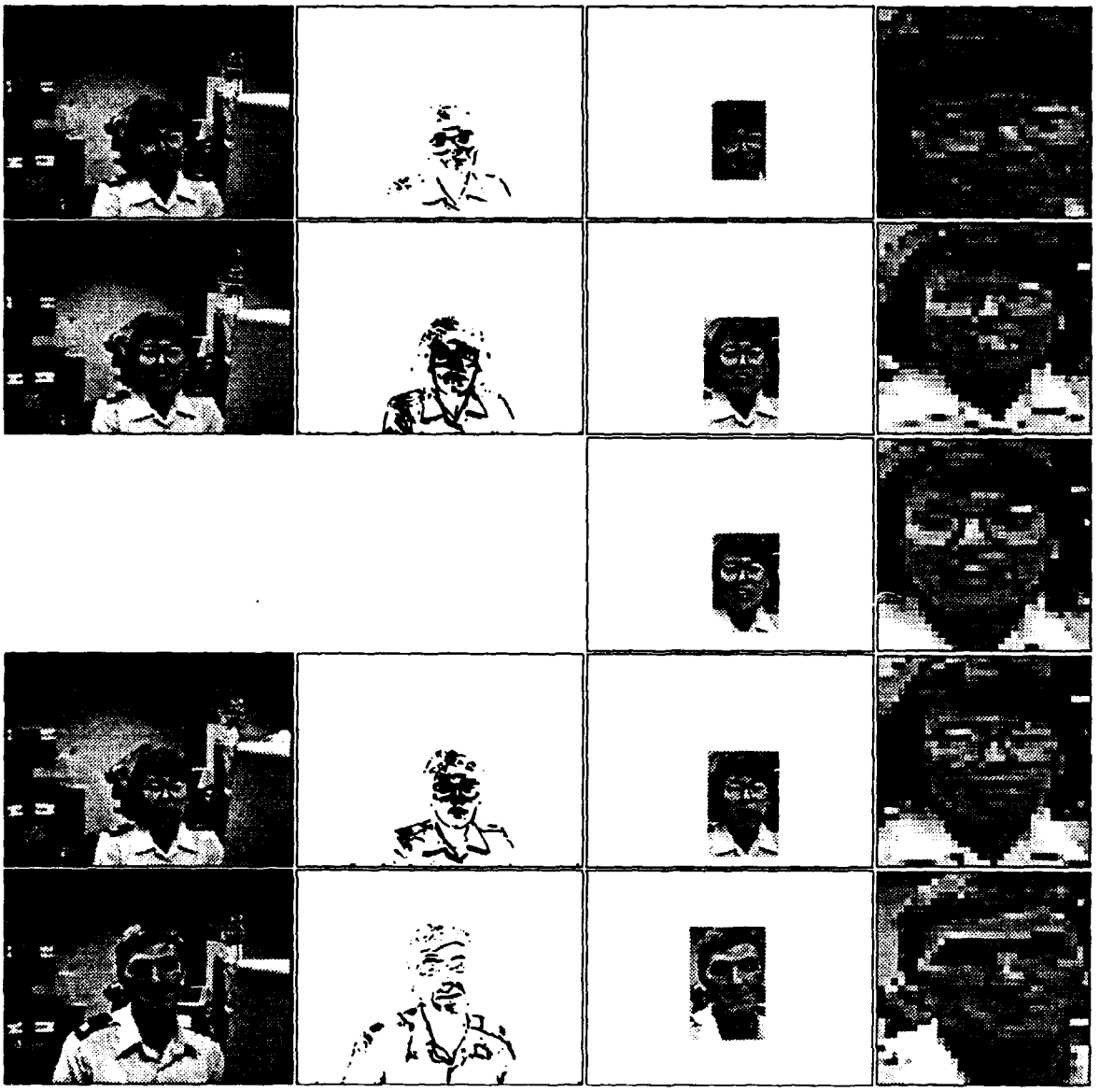


Figure 20. Segmentation Test I Images (3 of 4)

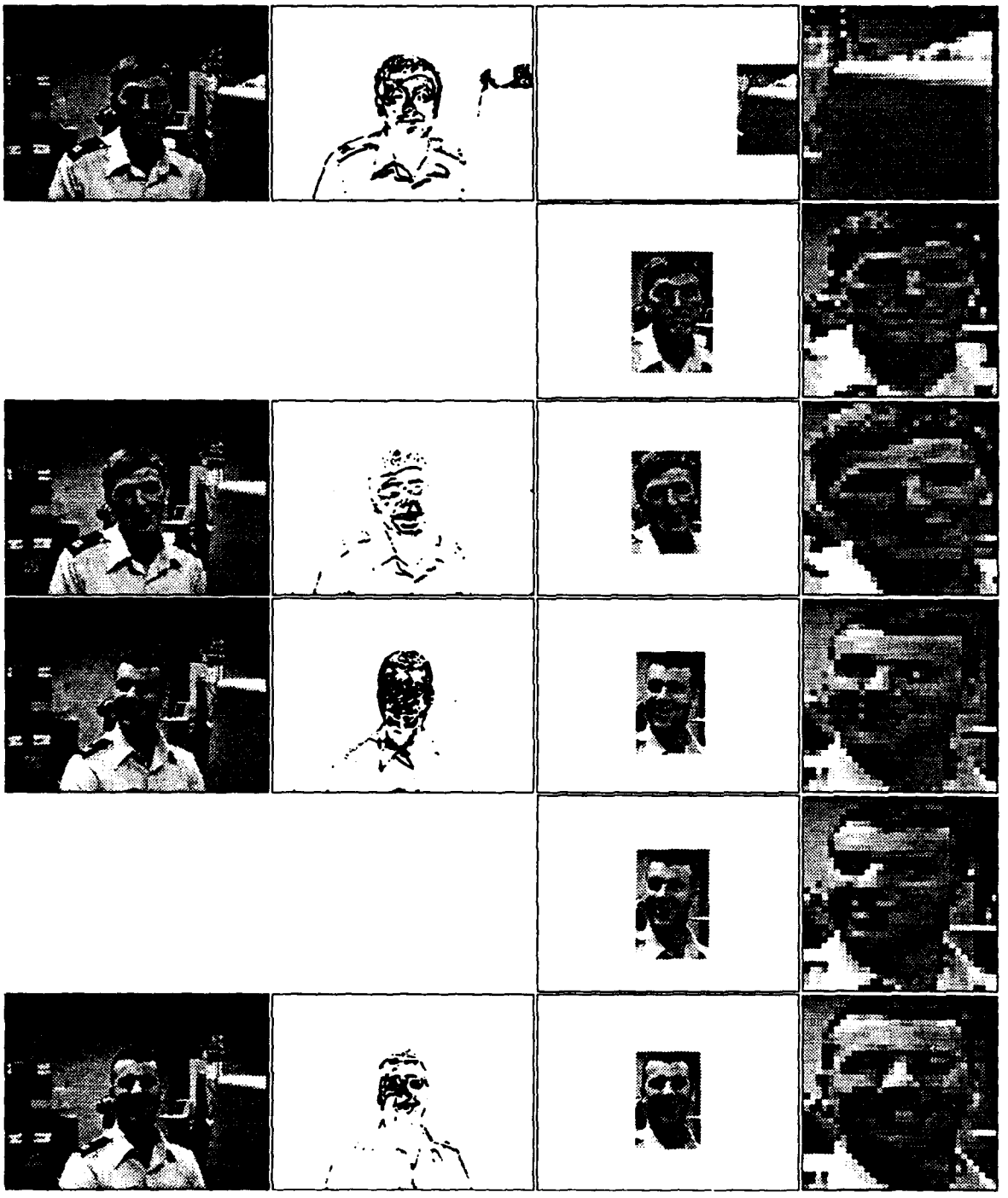


Figure 21. Segmentation Test I Images (4 of 4)

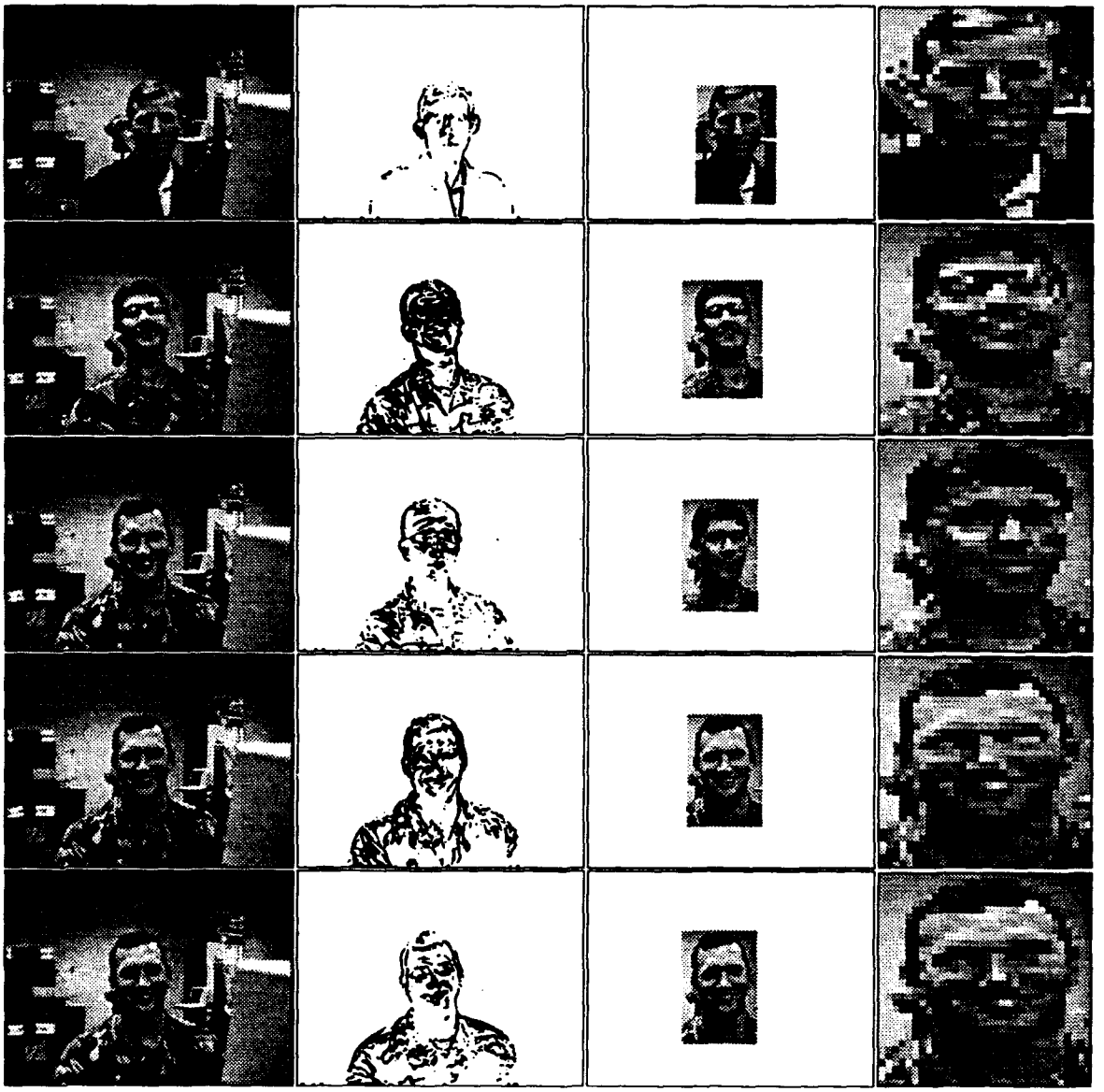


Figure 22. Segmentation Test II Images (1 of 4) [The Middle Motion Threshold Tested]

Figure 23. Segmentation Test II Images (2 of 4)

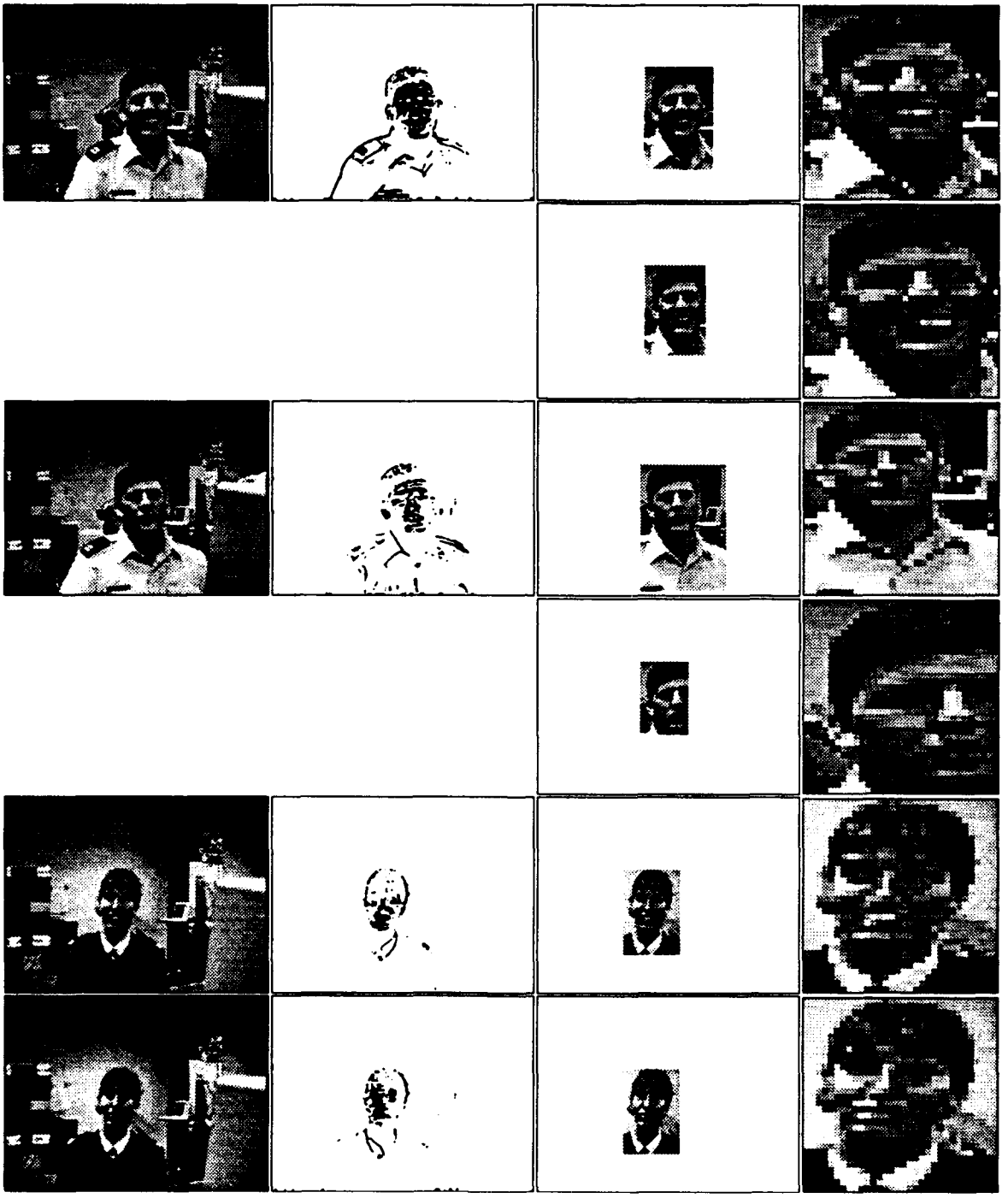


Figure 24. Segmentation Test II Images (3 of 4)

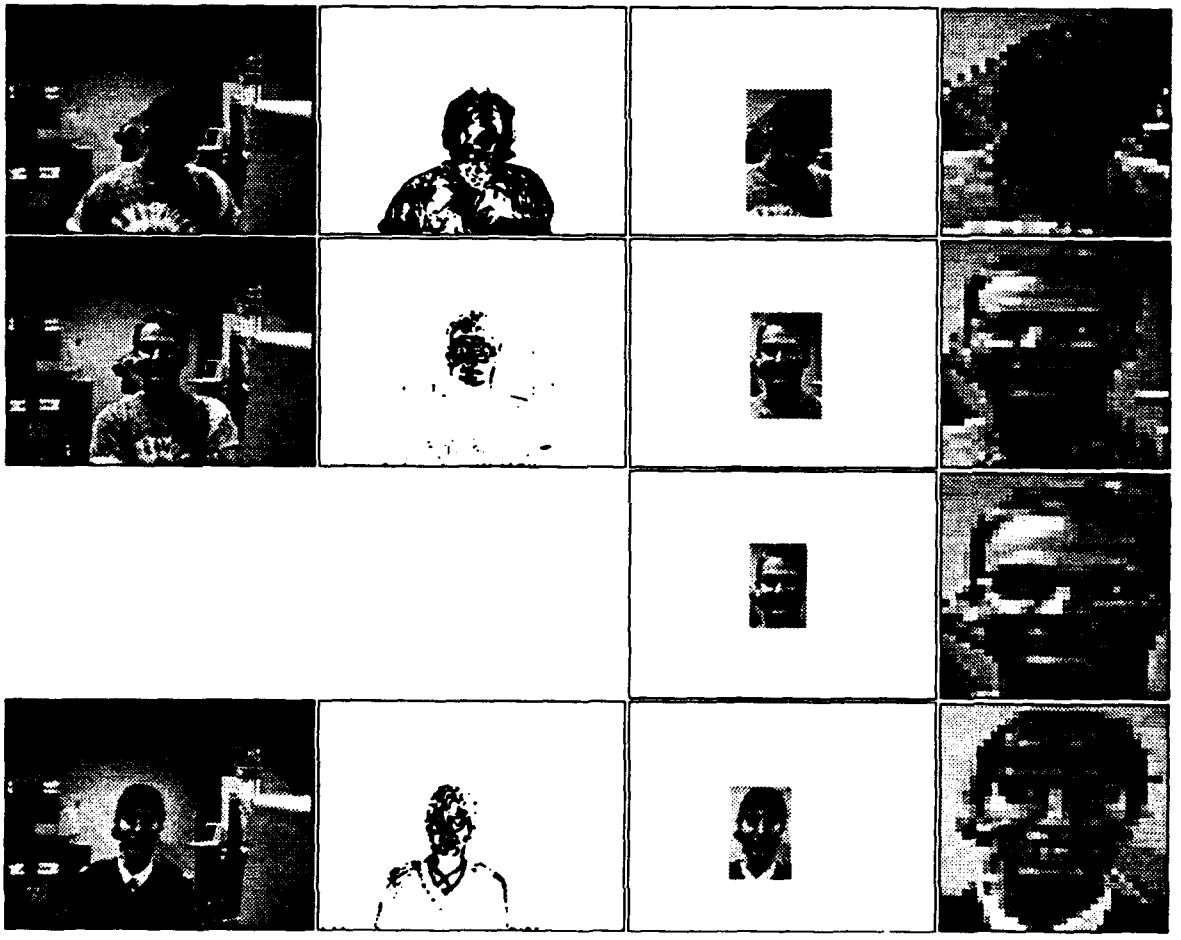


Figure 25. Segmentation Test II Images (4 of 4)

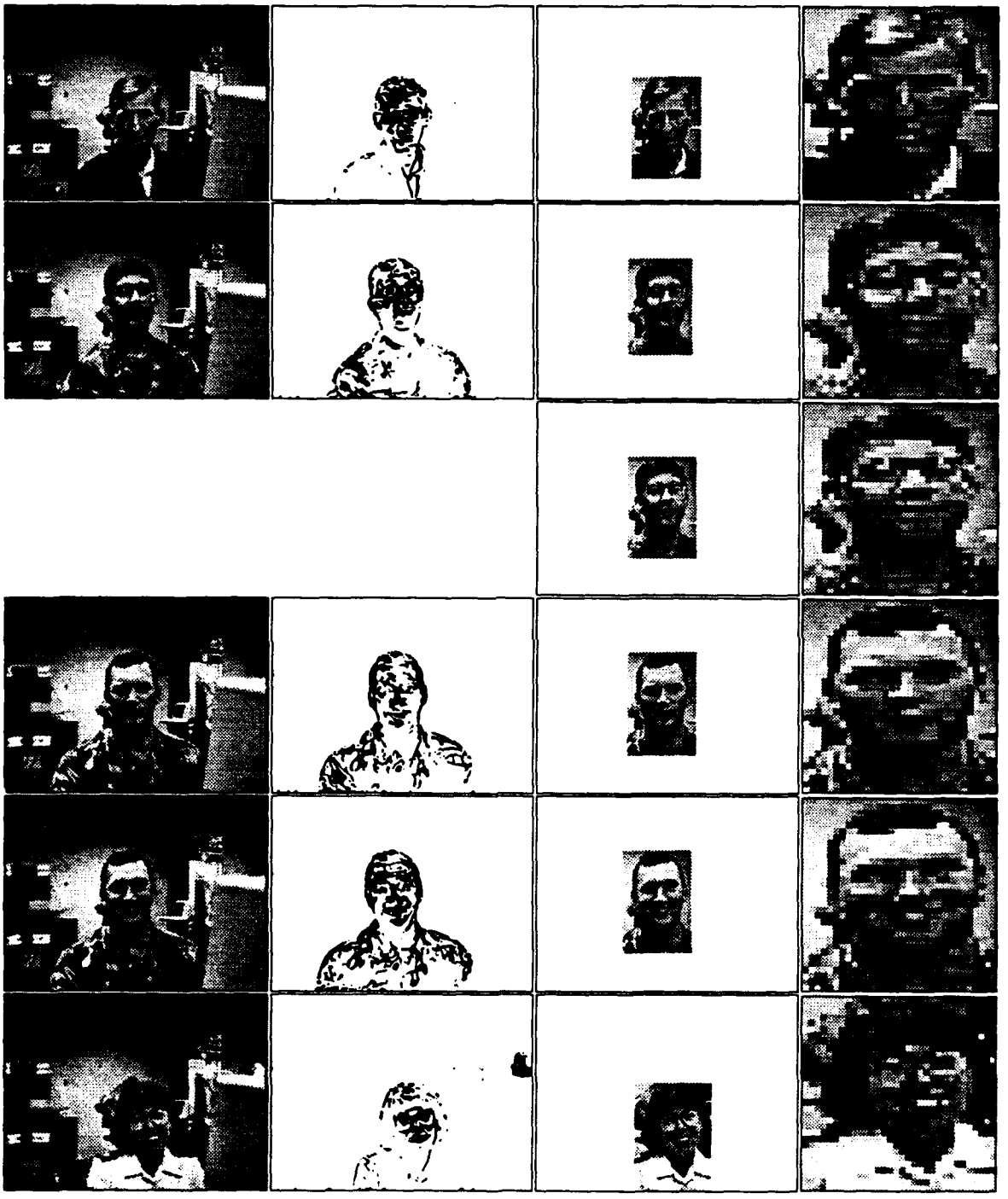


Figure 26. Segmentation Test III Images (1 of 4) [The Highest Motion Threshold Tested]

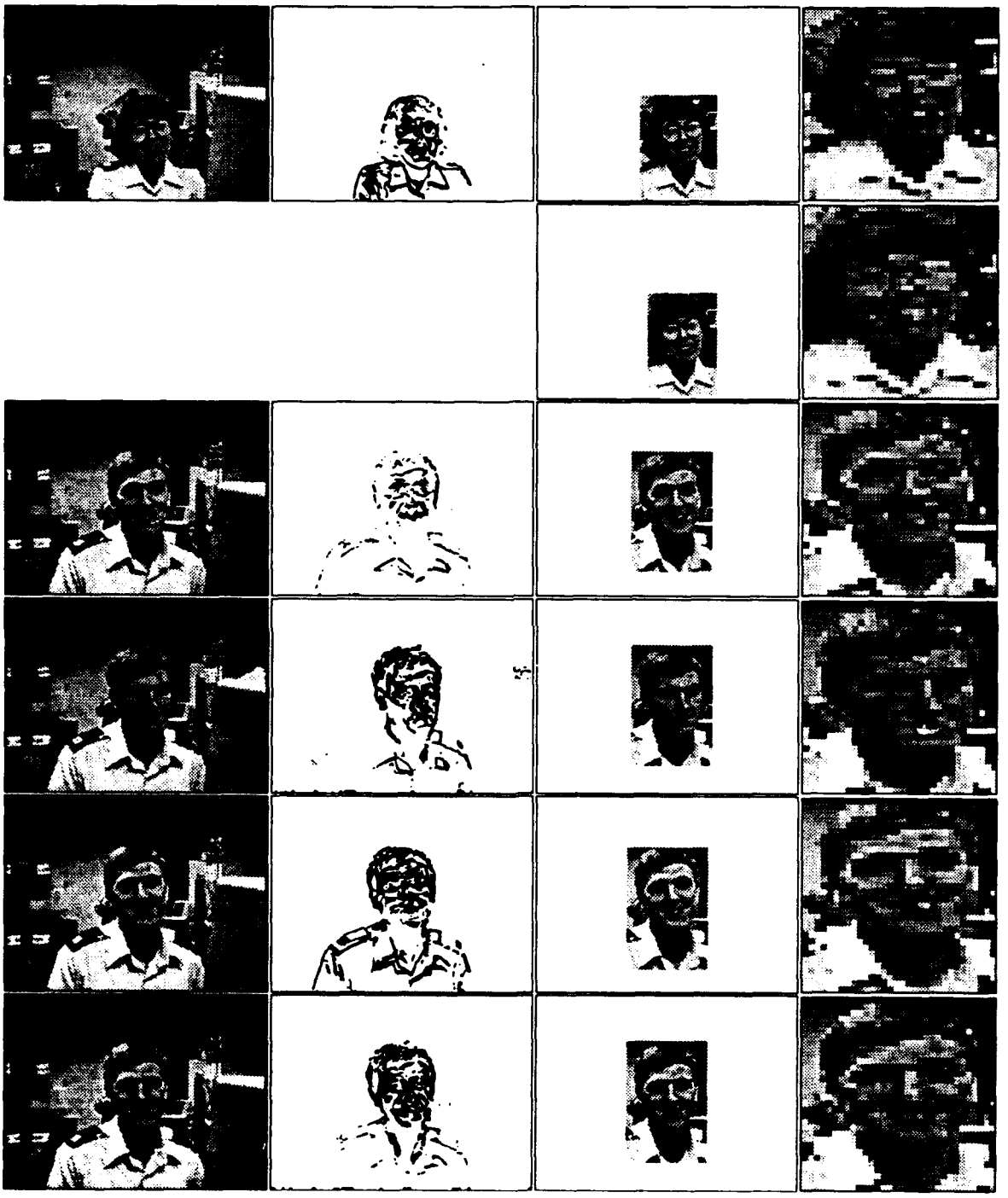


Figure 27. Segmentation Test III Images (2 of 4)

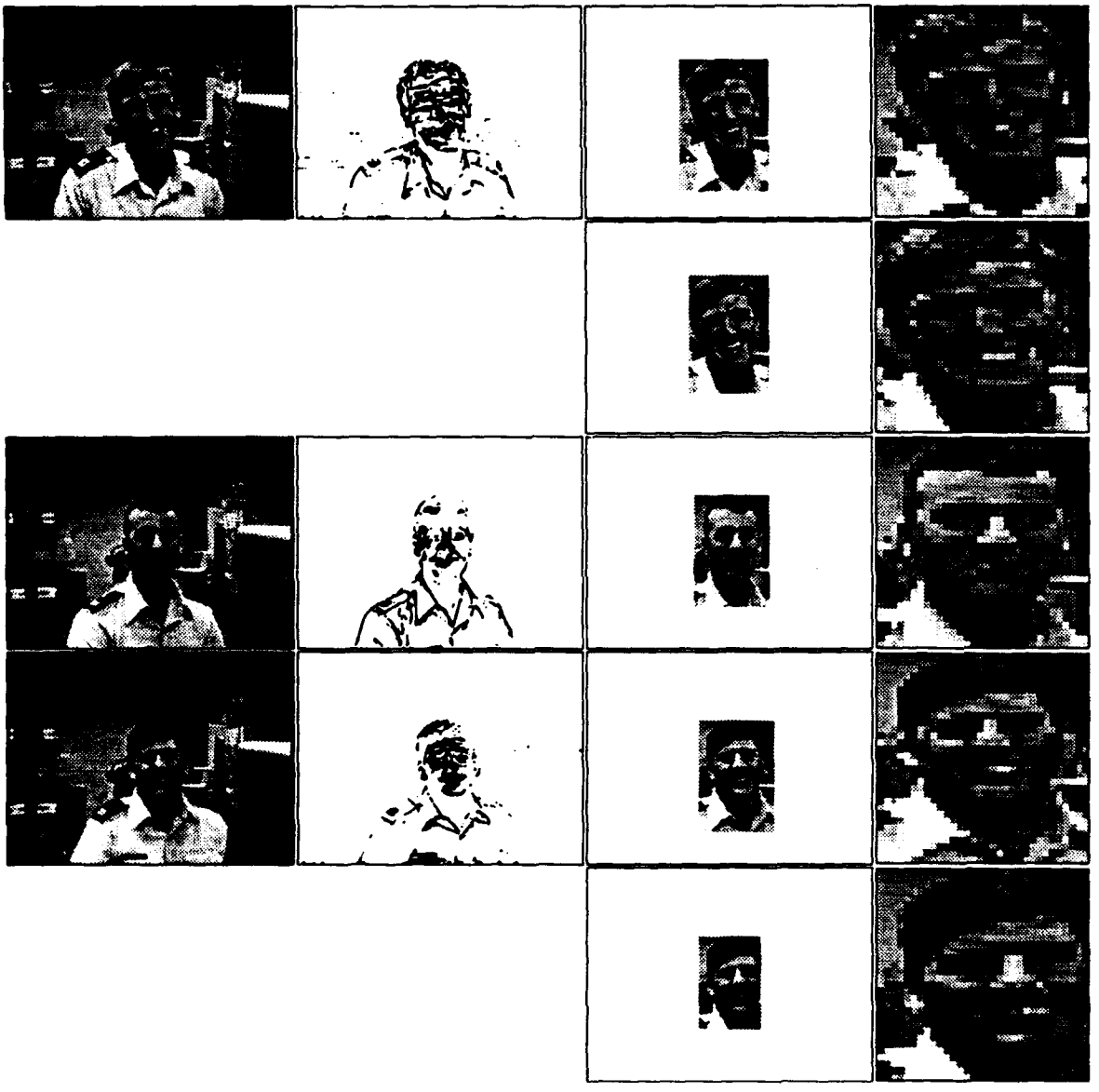


Figure 28. Segmentation Test III Images (3 of 4)

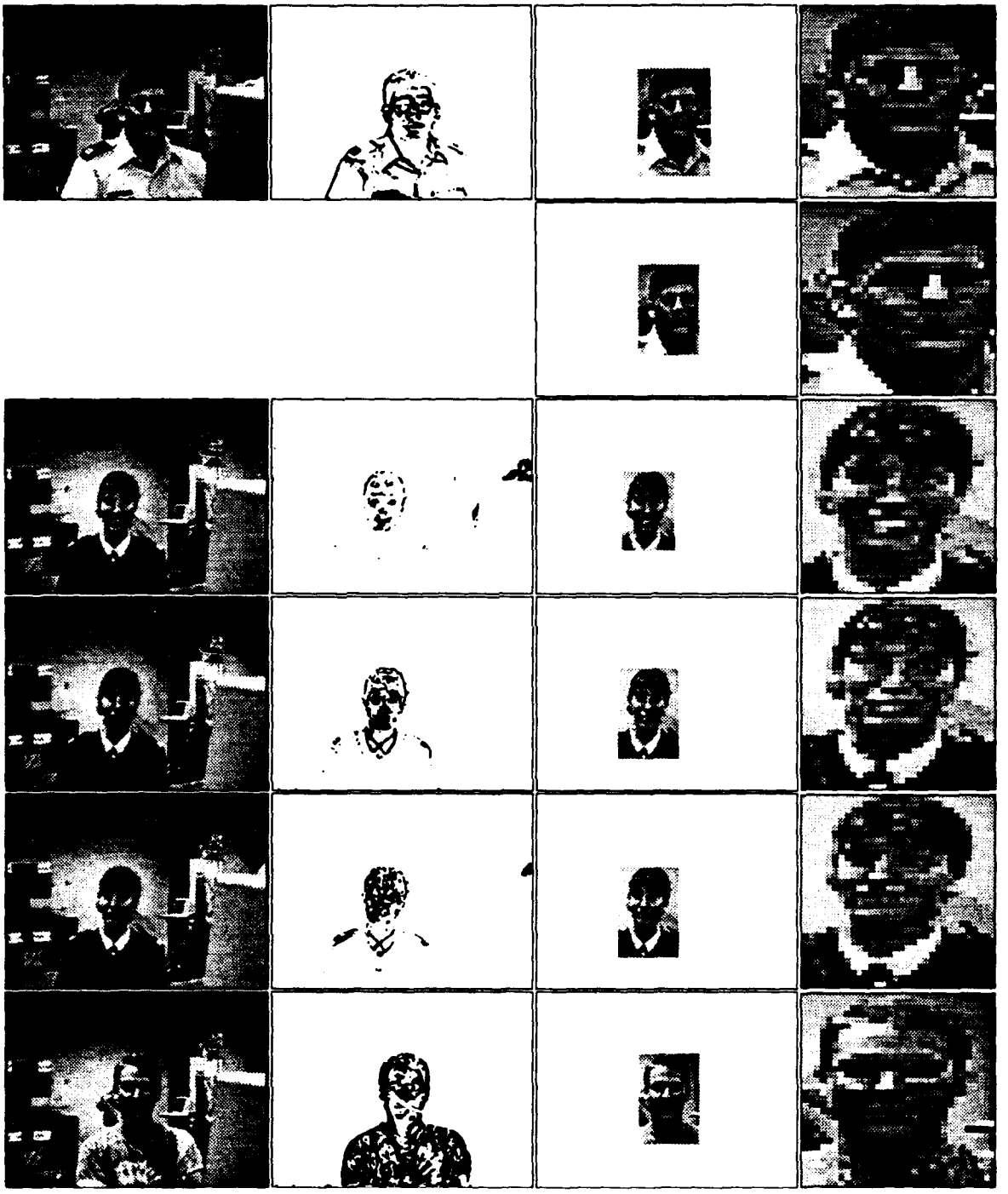


Figure 29. Segmentation Test III Images (4 of 4)

Figure 30. Segmented Regions I [The Lowest Motion Threshold Tested]

Figure 31. Segmented Regions II [The Middle Motion Threshold Tested]

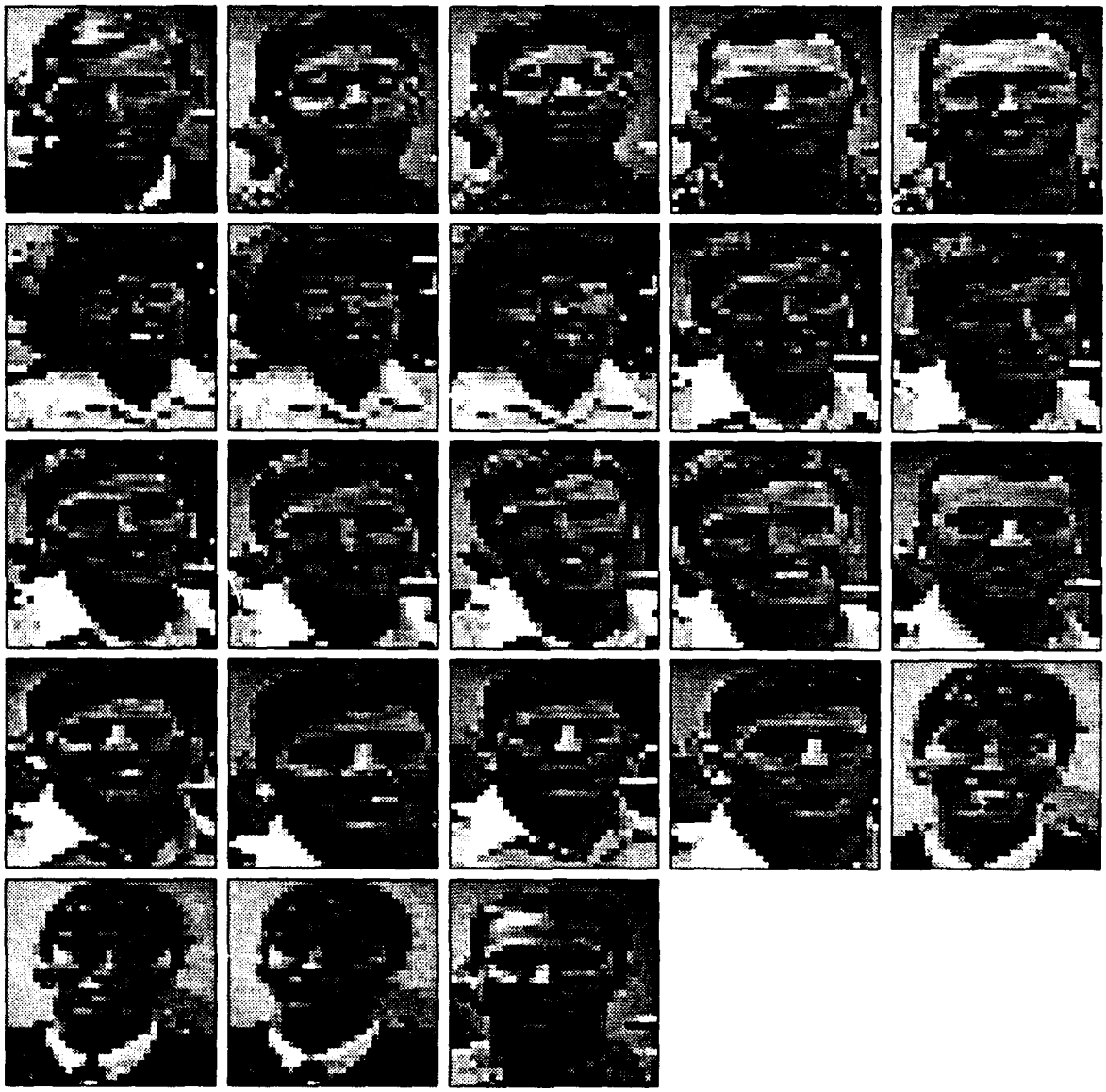


Figure 32. Segmented Regions III [The Highest Motion Threshold Tested]

Bibliography

1. Cannon, Scott, et al. "A Computer Vision System for Identification of Individuals," *IECON*, 347–351 (1986).

2. Carey, Susan and Rhea Diamond. "From Piecemeal to Configurational Representation of Faces," *Science*, *195*:312–314 (Jan 1977).

3. Damasio, Antonio R. "Prosopagnosia," *Trends in Neuroscience*, *8*:132–135 (1985).

4. Damasio, Antonio R. and others. "Prosopagnosia: Anatomic Basis and Behavioral Mechanisms," *Neurology*, 331–341 (April 1982).

5. Ellis, Hayden D. and others, editors. *Aspects of Face Processing*. The Netherlands: Martinus Nijhoff Publishers, 1986.

6. Fleming, Michael K. and Garrison W. Cottrell. "Categorization of Faces Using Unsupervised Feature Extraction," *IEEE International Joint Conference on Neural Networks*, 65–70 (1990).

7. Goble, James Robert. *Face Recognition using the Discrete Cosine Transform*. MS thesis, AFIT/GE/ENG/91D-21. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

8. Govindaraju, Venu, et al. "Locating Human Faces in Newspaper Photographs," *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 549–554 (June 1989).

9. Jeffreys, D. A. "A Face-Responsive Potential Recorded from the Human Scalp," *Experimental Brain Research*, *78*:193–202 (1989).

10. Jones, Judson P. and Larry A. Palmer. "The Two-Dimensional Spatial Structure of Simple Receptive Fields inCat Striate Cortex," *Journal of Neurophysiology*, *58*(6):1187–1211 (December 1987).

11. Kabrisky, Matthew, 1992. Human Factors Course Lecture.

12. Kendrick, Keith. "Through a Sheep's Eye," *New Scientist*, 62–65 (May 1990).

13. Kirby, M. and L. Sirovich. "Application of Karhunen-Loève Procedure for the Characterization of Human Faces." *IEEE Transaction on Pattern Analysis and Machine Intelligence 12*. 103 - 108. January 1990.

14. Krepp, Dennis. *Face Recognition With Neural Networks*. MS thesis, AFIT/GE/ENG/92D. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1992.

15. Lambert, Laurence C. *Evaluation and Enhancement of the AFIT Autonomous Face Recognition Machine*. MS thesis, AFIT/GE/ENG/87D-35. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1987.

16. Mannaert, Herwig and Andre Oosterlinck. "Self-Organizing System for Analysis and Identification of Human Faces," *SPIE, Applications of Digital Image Processing XIII*, *1349*:227–232 (1990).

17. McKinley, Richard, 1992. 3-D Sound Localization Lecture.

18. Meadows, J. C. "Varieties of Prosopagnosia," *Journal of Neurology, Neurosurgery, and Psychiatry*, 498–501 (1974).

19. Perrett, D. I., et al. "Visual Neurones Responsive to Faces in the Monkey Temporal Cortex," *Experimental Brain Research*, *47*:329–342 (1982).

20. Perrett, David I., et al. "Visual Neurones Responsive to Faces," *Trends in Neuroscience*, *10*:358–364 (1987).

21. Rogers, Steven K. *An Introduction to Biological and Artificial Neural Networks*. P.O. Box 10, Bellingham Washington, 98227-0010: SPIE Press, 1991.

22. Runyon, Kenneth. *Autonomous Face Recognition*. MS thesis, AFIT/GE/ENG/92D. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1992.

23. Smiley, Steven. *Image Segmentation Using Affine Wavelets*. MS thesis, AFIT/GE/ENG/91D. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

24. Suarez, Pedro F. *Face Recognition with the Karhunen-Loève Transform*. MS thesis, AFIT/GE/ENG/91D-54. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

25. Turk, Matthew A. and Alex P. Pentland. "Recognition in Face Space," *SPIE Intelligent Robots and Computer Vision IX: Algorithm and Techniques*, 43–54 (1990).

26. Wickenhauser and Coifman.

27. Wong, Dr. K. H., et al. "A System for Recognising Human Faces," *Proceedings from International Conference on Acoustics, Speech, and Signal Processing*, 1638–1641 (May 1989).

Vita

Kevin Gay was born in Fort Lauderdale, Florida on November 30, 1960. He lived in Hollywood, Florida until graduating from Cooper City High in 1978. He married his lovely and talented wife, Elizabeth, in August of 1981. Kevin enlisted in the U.S. Air Force in 1982 as an imagery interpreter. Upon completing his Bachelor of Science degree in Electrical Engineering at Brigham Young University, Kevin was commissioned in 1986. He served as an engineer in the gutless God-less liberal wasteland of Massachusetts at Electronics Systems Division (ESD) until he entered the Masters Program in the School of Engineering, Air Force Institute of Technology, in 1990. Along the way, he has accumulated five children, Amy, Daniel, Brenna, Elijah, and Samuel, and remains with the wife of his youth, the lovely Elizabeth. After graduation, Kevin will return to the warm sun of Florida and re-enter civilian life.

Permanent address: 6641 Douglas Street
Hollywood, Florida 33024

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1992	Master's Thesis

4. TITLE AND SUBTITLE
AUTONOMOUS FACE SEGMENTATION

5. FUNDING NUMBERS

6. AUTHOR(S)
Kevin P. Gay

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GE/ENG/92S-06

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Major Rodney Winter
DIR/NSA, R221
9800 Savage Road
Ft Meade, MD 20755-6000

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The purpose of this study was to implement an autonomous face segmentor as the front end to a face recognition system on a Sun SPARCStation2. Face recognition performance criteria, specifically, the capabilities to isolate and resize faces in an image to a consistent scale, were analyzed to determine current practical limitations. Face images were acquired using a S-VHS camcorder. Segmentation was accomplished using motion detection and pre-defined rules. Tests were run to determine the suitable of the autonomous segmentor as the front-end to a face recognition system. The segmentation system developed consistently located faces and rescaled those faces to a normalized scale for subsequent recognition.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Face Segmentation, Face Recognition, Segmentation	125
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL