An algorithm for facial expression recognition to assist handicapped individuals with eating disabilities

Anthony Rudolph De La Loza

AN ALGORITHM FOR FACIAL EXPRESSION RECOGNITION TO ASSIST HANDICAPPED INDIVIDUALS WITH EATING DISABILITIES

A Thesis

Presented to the

Faculty of

California State University,

San Bernardino

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

by

Anthony Rudolph De La Loza

March 2011

AN ALGORITHM FOR FACIAL EXPRESSION RECOGNITION TO ASSIST HANDICAPPED INDIVIDUALS WITH EATING DISABILITIES

A Thesis

Presented to the

Faculty of

California State University,

San Bernardino

---

by

Anthony Rudolph De La Loza

March 2011

Approved by:

[redacted]

Keith Evan Schubert, Chair, School of Computer Science and Engineering

3/15/11
Date

[redacted]

Ernesto Gomez

[redacted]

Haiyan Qiao

ABSTRACT

Assistive technology has allowed individuals with special needs to interact more actively and independently with their environment by allowing that individual to perform tasks that were impossible without it. In recent years, attempts at making efficient computer based assistive technologies have been made, which include speech recognition software and computerized wheelchairs. However, due to the complexity of the task, progress is very slow in the development of fast, efficient, and accurate computer based assistive technology. In addition, given that the needs of each individual vary, developing a device that meets the requirements of every individual is very difficult. As a result, this contributes to the slowing of development of such devices because the device would not be marketable if it was only made to handle a specific case. This thesis aims to present a solution to allow a special needs individual to eat more efficiently and foster independence, while providing a platform for further research in the area of feature detection to assist individuals with special needs. The proposed computer system aims to assist severely handicapped individuals with eating difficulties by using facial expression recognition to determine the individual's specific need that he or she wants to be performed in regards to the eating process. Upon determination of the facial expression made by the user, a specific feeding action would then performed by a spoon holding robotic arm. This solution was achieved by the use of five algorithms. The Canny edge detector algorithm was used to convert a webcam image of the user to an edge image. This was used to represent the bodily features of the user as simple edge lines. Then an edge connecting algorithm was used to modify the existing edge representation of the user's bodily features to a more well defined outline of those features. Once the edge connecting was complete, an edge walking algorithm was used to locate edges that had a high probability of representing the user's mouth. Then an image registration algorithm was used to locate the position of the user's mouth in a input image. Finally, an image matching algorithm was used to determine which mouth expression the user made based upon predefined mouth expression reference images. Based on this determination of the mouth expression made, a specific feeding action would then be carried out by a robotic arm.

ACKNOWLEDGEMENTS

I would like to thank all the faculty and staff at CSUSB that assisted me in achieving my educational goals.

DEDICATION

To my family and friends that have always been there to support me in all my endeavors.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

1.1 Overview of Thesis

This thesis in computer science is to develop a novel method to assist severely handicapped individuals with eating difficulties by using facial expression recognition to determine the individual's specific need that he or she wants to be performed in regards to the eating process. This specific need would then performed by a robotic arm. By classifying a given facial expression, and associating an action that should be performed with that facial expression, it becomes possible to allow an individual with no arm control to feed themselves independently. The feature that will be the main focus for this thesis and will be the main controller of the software system will be the mouth of the individual. In addition, it should be noted that even though the system was designed to control a robotic arm to assist individuals in eating, it could be adapted to control other devices or applications that special needs individuals may use.

1.2 Purpose of Thesis

The purpose of this thesis is to describe an algorithm and implement a software system based upon facial expression recognition that will accurately determine the specific need of a handicapped individual pertaining to the eating process. Then based upon that need, determine the appropriate action that should be executed. Examples of such actions include scooping food portions and delivering to the individual's mouth, and changing to different food to be scooped. The proposed system will consist of PC camera and laptop computer, where the laptop computer will process the video images from the PC camera. To verify that the final output is correct, a text of the action to be executed will be outputted, since there is currently no robotic arm or components to construct one that is strong enough for this system available to me.

The proposed system will allow the user to control a robotic arm that will assist the user in the process of eating. Consider this instance, if the user wants the robotic arm to scoop a large amount of food and bring the food portion close to user's mouth, the user just has to open his or her mouth all the way, which the system would recognize it as a "big mouth expression". Likewise, if user wants a small portion of food served to him or herself, the user just has to open his or her mouth partially, which the system would recognize it as a "small mouth expression". Finally if the user wants to halt execution of the current scooping and delivering process, closing of the mouth tightly should be made, which is recognized by the system as a "closed mouth expression".

1.3 Background

It has been estimated that there are about 48.9 million people in the United States that have a disability [1]. In addition, it has been estimated that about 9.2 million people in the United States over the age of 5 need personal assistance with one or more activities, which can include activities like "bathing, dressing, eating, [and] walking" [1]. So, the development of devices that assist in helping a disabled person accomplish everyday tasks is of great importance in improving their quality of life. For this research, the target group will be those individuals that have no or very little upper extremity control, more specifically those diagnosed with quadriplegia or cerebral palsy. Among this group it has been found that 60.7 percent of those that have this condition of "partial [or] complete paralysis of extremities" have an activity limitation [1].

According to the Department of Health and Human Services, it defines cerebral palsy as a condition that affects "a person's ability to move [their extremities] and to maintain balance and posture" [5]. This loss of ability is the result of part of the brain known as the cerebrum becoming damaged and no longer being able to control muscle tone [14]. Cerebral palsy is non-progressive, meaning that the individual's condition does not worsen, but their symptoms may change over time. At the current time, there is no known cure for cerebral palsy and the only approach to handle it is limited to treatment and prevention of problems that arise from the condition [14].

Quadriplegia is where an individual losses all or partial control of all their extremities as a result from damage to the brain or spinal cord. There are different forms of quadriplegia, but one of the most severe forms is spastic quadriplegia [28]. In addition to loss of muscle control, individuals with spastic quadriplegia suffer from hemiparetic tremors, "mental retardation, problems with muscles that control the mouth and tongue, and difficulty in speaking" [28].

1.4 Literature Survey of Feeding Methods

One method to assist individuals with eating difficulties is to have another individual spoon feed them. Spoon feeding is where a non-handicapped individual assists a handicapped individual in eating by placing the food item that is to be eaten on a spoon and delivering the food item to the handicapped individual's mouth, so that he or she can eat it. However, this method may not be suitable for some handicapped individuals because he or she may not have another individual to assist them in eating. In addition, a handicapped individual may seek methods that foster independence.

Another widely accepted method is to utilization a device called a percutaneous endoscopic gastrostomy tube (PEG tube). A PEG is "a procedure for placing a feeding tube directly into the stomach through a small incision in the abdominal wall using an instrument known as an endoscope" [7]. This is the preferred method to be utilized when a handicapped individual has difficulty swallowing food or is not able to consume food by mouth over long periods of time. In addition, "they [may] have muscle weakness," which can allow "food to leak [through] into the lungs when they swallow [food]" [7]. As a result, through the use of the PEG tube, the appropriate amount of fluids and nutrients can be placed directly into the stomach without the risk of facing respiratory problems from swallowing food through the mouth [7]. However, the disadvantage of this device is that a handicapped individual will need to have the assistance of another individual to use the device if he or she does not possess the necessary skill to use it properly. In addition, lower functioning handicapped individuals may repetitively attempt to remove the PEG tube, which can cause complications in the feeding process. These complications then would need to be addressed by the caretaker of that individual.

Another method that is used, but still undergoing continual improvements and development, is the utilization of a robotic arm to feed a handicapped individual. One mentionable device was developed in 1979 by R.L. Ramey et al. of the University of Virginia. They developed a system which allowed handicapped individuals to eat independently using a microcomputer-controlled manipulator [4]. By engaging a switch, the system would have the spoon holding robotic arm execute a pre-programmed path to the plate to scoop the food portion, and then follow a pre-programmed path to the user's mouth to deliver the scooped food portion [4]. Their preliminary results indicated that a handicapped individual could eat food independently and easily given that the food was in small portions [4]. In addition, it was noted that the dropping of food pieces occurred, but designing a spoon with the appropriate shape would hopefully solve this problem [4]. However, this problem can be overcome by using the concept of facial expression recognition proposed for this thesis. When the food portion is scooped, the system would recognize a certain facial expression that indicates the food portion was not scooped properly, and thus must re-execute the scooping procedure. This same technique can be applied to situations when food portion fall off the spoon on its path to the individual.

As technology improved and more research has been done on eating assist robots, other devices have been integrated into the robotic arm to improve the ease of use for the user while fostering more independence. One robotic arm system integrated the use of lights, a spoon, and a switch to allow a user to choose and scoop food, [15]. A series of lights would scan the columns that were behind a food portion, [15] The user would then press the switch when the light arrived to a column that had the food portion that the user wanted to eat, [15].

Another robotic arm system integrated the use of a PC display, a CCD (charge-coupled device) camera, a spoon, a puff switch, head pointing device, and a GSR (galvanic skin reflex) sensor to allow a user to scoop and serve food portions to oneself, [18]. A straw mouth tube served as the puff switch, where the user would puff into the straw in order to click items on the user interface, [18]. By wearing the head pointing device, the user then could control the robotic arm by his or her head movement, [18]. The user would move his or her head in order to move a pointer on a display and once that pointer reached the target position, the user would click using the puff switch [18]. The GSR sensor served in determining a safe speed for the robotic arm's movement for a given user [18].

Not only have there been single-user robotic arm systems been developed or proposed, but also multiple-user robotic arm systems. One robotic arm system was proposed that integrated the use of a vision system interface to allow one to four users to be fed [19]. The vision system would be used to recognize food, forks, spoons, and cups [19]. A specific user-interface was not proposed for this system, but mentioned that a user-interface that was "applicable in a dining environment that is susceptible to food or drink spills and be usable for the elderly with different upper-limb disabilities" would be the best choice [19].

1.5 Contributions

The main focus of this thesis is to develop a system that uses facial feature recognition to assist special needs individuals with little or no arm control with eating a meal. By classifying a certain mouth expression and classifying it to certain action to be performed, it becomes possible for a individual to control a device or complete a task that otherwise could not be accomplished on his or her own. The implementation of this system will provide a foundation for further research in the area of feature recognition systems that can assist special needs individuals. In addition, the use of such a system provides the opportunity to foster individual independence, while improving quality of life for that individual.

1.6 Assumptions Made in Developing the Computer System

There were several assumptions made for this thesis. The assumptions include:

1. The distribution of light throughout the room that the system is to be used in should be uniform.
2. The individual should be able to make the three defined mouth expressions. These mouth expressions were a small mouth expression, a big mouth expression, and a closed mouth expression.
3. The individual would remain about the same distance from the webcam throughout the feeding period.

1.7 Limitations of the Computer System

There were several limitations discovered throughout the development of the thesis. The first limitation noted was that the distribution of light in the room greatly effected how well the system performed. Too much light or too little amount of light from a certain direction caused the system to perform poorly. Another limitation was that as the number of mouth expressions that were searched for by the system increased, the amount of time greatly increased to find if any of these mouth expression reference images could be found in the captured webcam image. So the number of mouth expression images was limited to three different mouth expressions. These mouth expressions were a small mouth expression, a big mouth expression, and a closed mouth expression. The last limitation of the system is that the individual must remain about the same distance from the webcam throughout the feeding period. If the individual moves closer or moves farther away from the webcam, it makes it more difficult for the system to locate the mouth expressions.

1.8 Documentation Organization

1. Chapter 1 provides an overview of past developments in assistive technologies. In addition, it presents the purpose and contribution of this thesis, while noting the limitations discovered and assumptions that were made.

2. Chapter 2 provides background information on what needs to be considered when feeding a special needs individual and how precautions should be made to insure the overall safety of the individual.

3. Chapter 3 covers the procedure of how to convert an image captured from a webcam and converting it into an image made of edges.

4. Chapter 4 covers the procedure used to locate and collect edges of a certain size in an image . It also presents the idea how keeping certain edges and removing unwanted edges in an image is the basis for more accurate calculations in upcoming steps.

5. Chapter 5 covers the procedure used to connect separate, but in close proximity, edges together to form a single edge. It also presents the idea that image modification can enhance the image's content and provides a better basis to perform calculations upon.

6. Chapter 6 presents the concept on how performing calculations in the frequency domain rather than the spatial domain, can provide a more effective means of calculation. It covers how the use of discrete Fourier transforms provide a means to calculate the amount of pixel shift there is between two similar images.

7. Chapter 7 covers the procedure on how to determine how well a reference image matches to an input image, by inverting pixel values and using logical AND operations.

8. Chapter 8 provides and overviews the results from testing the system through the use of a sequence chart.

9. Chapter 9 concludes with presenting the contributions made from completing this thesis. It presents the idea how assistive technologies not only can help an individual complete a task, but also fosters independence. It also presents the

current problems faced when developing the system, such as the distribution of light in the room, and recommends ideas and areas to explore for further research.

2. CONSIDERATIONS WHEN FEEDING THE SPECIAL NEEDS INDIVIDUAL

2.1 Chapter Overview

This chapter provides background information on what needs to be considered when feeding a special needs individual and how precautions should be made to insure the overall safety of the individual.

1. Section 2.2 covers how the ability of the user to maintain an upright body position and to hold their position in front of the camera is essential in being able to use the system.

2. Section 2.3 covers how the ability of the user to position their head in front of the camera and to be able to make the appropriate mouth expression is essential in being able to use the system.

3. Section 2.4 covers how it is essential that the user possesses the mental capability that allow him or her to complete the tasks that are required by the system in order to use the system properly.

4. Section 2.5 covers how performing the correct procedure to feed a special needs individual is essential in ensuring the overall safety of the individual.

5. Section 2.6 covers how the manner in which light is distributed in the room greatly affects the overall performance of the system,and how certain types of

lighting can affect some individuals in dangerous ways.

2.2 Body Posture

The ability of the user to maintain an upright body position and to hold their position in front of the camera is essential in being able to use the system. In deciding whether an individual can utilize the system, the user must demonstrate the ability to maintain an upright body position in front of the camera for the duration of time it takes to be fed a meal. Failure to do so would limit the possibility to utilize the system and its functionality.

If the individual is unable to maintain an upright posture for extended periods of time, one possible solution would be to have the individual be placed in a chair with side supports to help him or her maintain an upright posture. If this maintaining of upright body posture cannot be achieved, then other feeding methods should be explored to better suit that individual's specific needs.

2.3 Head and Mouth Control

The ability of the user to position their head in front of the camera and to be able to make the appropriate mouth expression is essential in being able to use the system. In order to use the system, the user must be able to make the three different mouth expressions, and to be able to hold their head in front of the camera while making these mouth expressions. However, there are some individuals that do not have the ability or muscle control to complete these tasks, or find it very difficult to fully complete these tasks. As a result, this would prevent them from being able to use the system.

If the individual is unable to easily move their head in front of the camera, one possible solution is to position the individual in such a way that it results in little or no repositioning needed by the individual to place his or her head in front of the camera. If the individual has difficulty in making the different mouth expression, one possible solution is to attempt to find other mouth expressions that can be made easily by the individual. If such mouth expressions can be found, then these mouth expressions can replace the defaulted mouth expressions used by the system. If completing any of these tasks proves to be difficult and it is found the system cannot be used efficiently, then other feeding methods should be explored to better suit that individual's specific needs.

2.4 Mental Capability

It is essential that the user possesses the mental capability that allow him or her to complete the tasks that are required by the system in order to use the system properly. Such tasks as moving the individuals head in front of the camera, and making the necessary mouth expressions are required to use the system successfully. However, not all individuals have the mental capability to perform such tasks. For example, the individual may not understand commands such as to make a "big mouth expression" or to "move your head in front of the camera". Such inabilities will cause difficulties in using the system properly. In addition, the individual may not have the ability to understand that certain actions can cause dangerous situations that lead to harming himself or herself or the system itself. For example, an individual may grab the camera or the robotic arm with their hands out of curiosity, even though the individual should not touch it. So it is necessary that the individual that uses the system is capable of following and understanding commands.

If the individual is unable to understand the commands presented to him or her, one possible solution is to have another individual model the task to that special needs individual. By having another individual demonstrate the task, such as a making a big mouth expression, it may be possible for the special needs individual to correspond the command with the demonstrated task. However, such a solution is based on the special needs individual having the ability to learn new concepts in a timely manner. It may be helpful to teach the individual the different mouth expression before using the system. Once the individual has demonstrated an adequate understanding of the commands and expressions, then attempts can be made to see if the individual can successfully use the system. However, if it is determined that the individual does not possess the mental capability to use the system , then other feeding methods should be explored to better suit that individual's specific needs.

2.5 Safe Feeding Practices and Type of Food

Performing the correct procedure to feed a special needs individual is essential in ensuring the overall safety of the individual. The possibility of feeding the individual too quickly by the robotic arm is present, and the possibility of choking on the given food portion is possible.

Feeding too quickly can occur when the individual request another food portion before completely swallowing the food portion that is currently in his or her mouth.

This action can possibly cause a choking hazard and is likely to occur when the individual does not possess the mental capability to understand to swallow the current food portion before requesting another food portion. Also it could also occur when the individual has difficulty in exhibiting patience and wanting to eat quickly. One possible solution to avoid this from occurring is to have the special needs individual monitored by another individual. This individual can insure that the special needs individual eats at a safe rate and can assist the special needs individual in the event that he or she is choking.

The type of food should be considered that works best in being scooped by the robotic arm, while being safe to eat for the individual that has special needs. The type of food should be able to be scooped in the desired portion size easily by the robotic arm, and to be safely transferred to the plate to the individual's mouth. Food portions that are too large or a food that is of material that is difficult to chew and swallow can present a choking risk because some individuals may not possess the necessary muscle control in their mouth and tongue to efficiently chew and swallow certain types of food. So it is of great importance to use a type of food that is easy for the robotic arm to handle, yet safe for the special needs individual to eat.

2.6 Lighting

The manner in which light is distributed in the room greatly affects the overall performance of the system. Too much light, too little light, or more light coming from a certain direction than others, affect the systems performance in negative ways. However, not only does the configuration of the lighting in the room affect the system, it can affect some individuals in dangerous ways. Flashing light or damaged fluorescent light bulbs that have a low flicker rate can cause some individuals to have seizures. If the individual has a past history of having seizures, one alternative solution is to use natural lighting. By using a room that has windows that allow enough light to enter the room, the likelihood of triggering a seizure can be reduced.

3. IDENTIFYING EDGES IN IMAGES

3.1 Chapter Overview

This chapter covers the procedure of how to convert an image captured from a webcam and converting it into an image made of edges. An algorithm that converts a webcam image to an edge image is needed in this computer system because a edge representation of the user's bodily features provides a simpler means to perform computations upon. This conversion aims to make it easier to locate the user's mouth.

1. Section 3.2 provides a brief overview on why converting a image to an edged image can provide a better format to perform computations upon.
2. Section 3.3 provides a brief overview on why the Canny edge detection algorithm was chosen.
3. Section 3.4 covers the steps of the Canny edge detection algorithm
4. Section 3.5 covers how the Canny edge detection algorithm was used to convert webcam captured images into edge images for the system to use as a basis for performing computations upon.

3.2 Introduction

The process of converting the current image from the webcam into an edged image is essential in allowing the image to be processed in such a manner that it efficiently locates the users predefined mouth shapes. The data found in this two-dimensional edged image of the users environment will be the basis for calculation and manipulation, in order to find the users mouth given the predefined mouth shapes.

Even though the entire current image is converted to an edged image, a predefined square subarea located in the edged image will be the area in which calculations will be carried out on. This was done in order to reduce the amount of computations done because of the smaller number of pixels that are used for calculations. In addition, it gives a higher likelihood of the users mouth being present in that location of the edged image because it is located in the edged image in such a manner that is close to where the users mouth is most likely to be present, and if not, the user can easily navigate their mouth within that region. The method chosen to produce these edged images was the Canny edge detector.

3.3 Literature Survey of Canny Edge Detection Applications

Given that the Canny edge detection algorithm is commonly used in image processing, many research papers on applications that used the Canny edge detection algorithm were found. Many papers covered the implementation of the Canny edge detection algorithm [23] [25] [26]. However, other papers did a comparison study instead of doing an implementation.

In a comparison study performed by Raman Maini and Himanshu Aggarwal, it was discovered that the Canny edge detection algorithm performed the best under a variety of conditions [22]. They found that gradient-based edge detection algorithms had the disadvantage of being easily susceptible to noise [22]. In addition, since the kernel filter size and coefficients are a fixed value, they cannot be adapted to a certain image [22]. With the Canny edge detection algorithm they discovered that the algorithm provided a "robust solution that is adaptable to the varying noise levels," which in turn helped to "distinguish valid image contents from visual artifacts introduced by noise" [22]. Given that the Canny algorithm has adjustable parameters, it allows the user to adjust these parameters to better suit the given scenario presented, and thus generate a better quality edged image [22].

In another comparison study performed by Wang Luo, it was discovered that the Canny edge detector provided the best results on colony images [27]. It was found that edge detection methods "such as Roberts Cross, the Sobel Operator and Prewitt operator failed to perform adequately in such applications due to the noisy nature of remotely sensed data," while the Canny edge detector "presented the best performance both visually and quantitatively based on the measures such as mean square distance, error edge map and signal to noise ratio" [27].

Upon completion of reviewing these papers, it was decided that the Canny edge detection algorithm would be used to generate the edged images for this application. Given the positive results of using the Canny edge detector from these papers, and its ability to perform well under a variety of environment conditions, it presents itself as a valid choice for our application.

3.4 Canny Edge Detector Algorithm

The first step in the algorithm is to convert the image to gray scale if the image is not. Then the next step is to smooth the image using a Gaussian filter to eliminate noise [2]. The Gaussian filter is moved over the image, where it transforms the values of those pixels within that area. The result of this process is an array of smoothed data.

The next step is to calculate the gradient magnitude and orientation. The gradient of the smoothed array is calculated by using 2 x 2 first-difference approximations to produce two arrays for the x and y partial derivatives [2]. The gradient orientation and magnitude then be calculated by using rectangular-to-polar conversion formulas [2].

In the third step, nonmaxima suppression most be used to reduce the size of the ridges in the magnitude array to one pixel wide. This result is achieved by setting all values along the line of the gradient that are not peak values to zero [2]. These non-peak values are set to zero by moving a 3 x 3 neighborhood across the magnitude array [2]. The center element of the neighborhood is compared with its two neighbors along the line of the gradient, and if the value of the center is not greater than the values of its two neighbors its value is set to zero [2].

Thresholding is then applied in the fourth step to eliminate false edge fragments, and to produce an array of edges that were detected in the image [2]. However, since only one threshold value was used this can result in false edges because of a low threshold value, or missing portions of contours because the threshold value was too high. To lessen this problem, a thresholding algorithm that utilizes two thresholds should be applied [2]. By doing so, two thresholded edge images are created. The edges of the higher thresholded edge image are then linked into contours. Once a contour ends, the algorithm looks in the lower thresholded edge image for edges that can be linked to that specific contour [2]. This process is repeated until the gap is filled in and that edge now connects to another edge.

3.5 Canny Edge Detector Algorithm Use in the Computer System

Since the OpenCV library provided a function that implemented the Canny edge detection algorithm, it was decided to utilize this function to produce the needed edged images. The function has five parameters, which include an input image, an output image, two threshold hold values, and an aperture size. In the system, these two threshold values can be changed during runtime by the user in order to produce an edged image that is the best quality given the lighting and the users seating arrangement at the given time. It is essential for the user to set the threshold values to the proper values to ensure that the system performs at optimal performance. In Figure 3.2 it shows the result of applying the Canny edge detector function on Figure 3.1, with threshold values that were set by the user to produce a suitable edged image. It must be noted that the input image must be converted to grayscale before apply the Canny edge detector function.

By comparing Figure 3.2 with Figure 3.3, it is seen how it is essential to set the threshold values to the proper values because otherwise minor edges that do not correspond to physical features would be present in the image. More importantly, it makes it difficult to locate key features, in this case the mouth, in the output image.

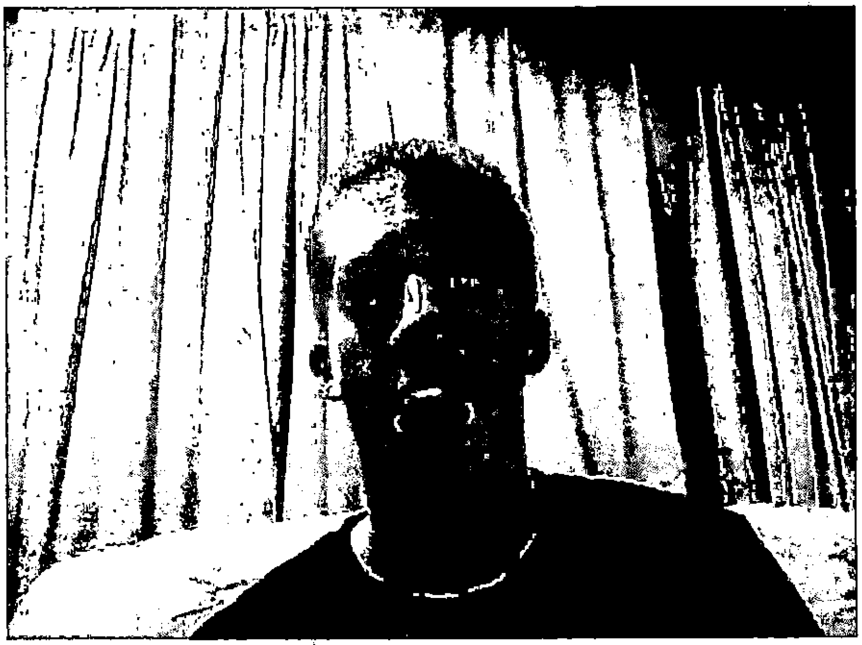

Fig. 3.1: **Input Image.**

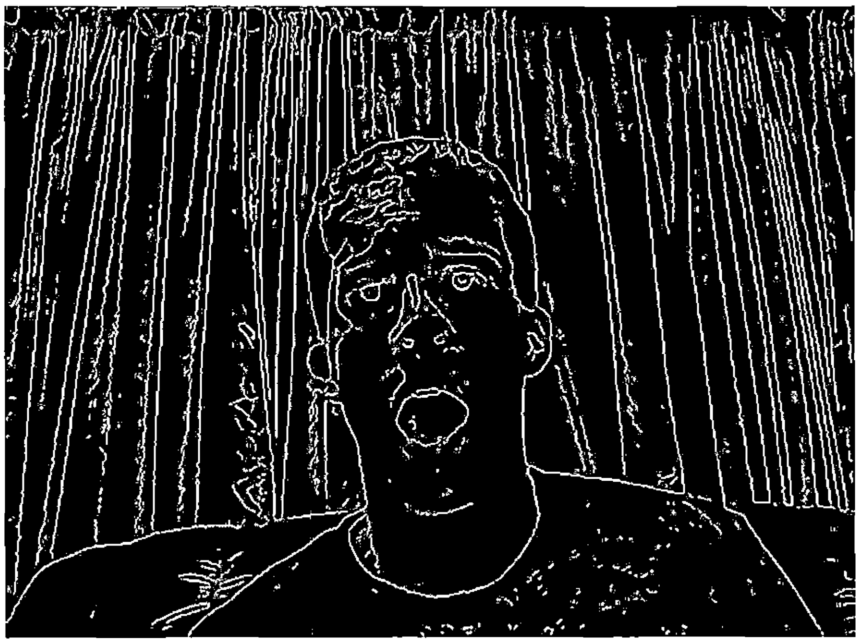

Fig. 3.2: The result of applying the OpenCV Canny edge detector function with a low threshold value of 20 and high threshold value of 20 to an input image.

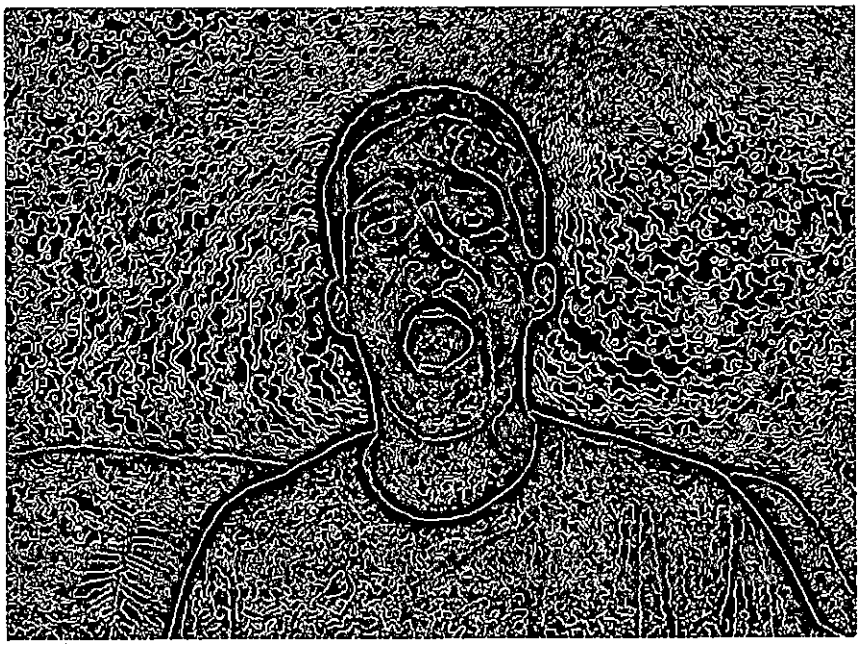

Fig. 3.3: A poor quality edged image by applying the OpenCV Canny edge detector function with a low threshold value of 0 and high threshold value of 0 to an web cam image.

4. DETERMINING WHICH EDGES IN THE IMAGES TO CONSIDER

4.1 Chapter Overview

This chapter covers the procedure used to locate and collect edges of a certain size in an image. It also presents the idea how keeping certain edges and removing unwanted edges in an image is the basis for more accurate calculations in upcoming steps. An edge walking algorithm is needed in this computer system because a means to locate edges that have a high probability of representing the user's mouth is essential. These found edges would then serve as the basis of for computation and classification in upcoming steps in determining the mouth expression made by the user.

1. Section 4.2 provides a brief overview on how it is important to realize that not all the edges present in the image are necessary in being used to locate key features, and how discarding certain edges in an image can provide more precise calculations in upcoming computational steps.

2. Section 4.3 covers the steps of how the system walks the edges in the image and then retains or discards the edges based on their pixel count.

3. Section 4.4 covers the use of the edge walking algorithm in the system and how it is used locate edges that are likely to define the user's mouth.

4.2 Introduction

It is important to realize that not all the edges present in the image are necessary in being used to locate key features, which in this case is the user's mouth. Large features normally are made up of very large pixel counts. A large feature may consist of a single edge or is made up a many large edges that define it. It is these large edges that should be considered for being used as a basis for calculation because of the higher likelihood that these edges define the shape of the user's mouth. Given this, there are two basic approaches that could be used to decide which edges to consider. The first is to only consider those edges that have a pixel count that falls between a minimum pixel count and a maximum pixel count. The second approach is to just consider the edge that has the largest pixel count among all the edges in the area that is being considered.

4.3 Edge Walking Algorithm Design

For the basic procedure that is performed to identify edges of a certain pixel count size, see Figure 4.1. The mentioned pixel neighbors that a given pixel can have is illustrated in Figure 4.2 and shows that the current pixel that is being examined can have at most eight pixel neighbors. Figure 4.3 shows an example of a possible configuration of neighboring pixels a pixel that is being examined can have. Upon examination of Figure 4.3, it is seen that the current pixel being examined has two pixel neighbors which are neighbor 3 and neighbor 7.

Figure 4.4 it provides an illustration of the steps that are performed in the algorithm. The illustration labeled (1) shows an input image that contains an edge to be walked. The illustration labeled (2) shows the procedures that are taken to perform the edge walking algorithm. The edge walking procedure starts at the pixel that is colored purple. It continues along it path, which is represented by the blue arrow. A backtrack pixel (green pixel) is found upon arrival at the red pixel. The backtrack pixel is stored and the edge continues to be walked until its end. In this case, since there is a backtrack pixel, the edge has not been completely walked. So edge walking resumes at the backtrack pixel (green pixel). The edge walking continues to the end of the edge (yellow arrows). Edge walking is complete once there is no longer any backtrack pixels remaining. In addition, the above algorithm can be slightly modified to only find the largest edge by changing the condition that an edge is valid if it has the largest pixel count among all the edges.

```
• Locate an edge pixel
• While edge not walked
    o Calculate neighbors of edge pixel
    o If pixel has no edge pixel neighbors
        ▪ If there are no backtrack pixels to consider
            • All pixels in edge have been found
            • If pixel count in edge meets size requirements
                o Add edge to the list of valid edges
            • Else pixel count in edge does not meet size requirement
                o Discard edge
        ▪ Else there are backtrack pixels to consider
            • Go to that pixel to continue walking edge
    o Else pixel has edge pixel neighbors
        ▪ Walk to a neighboring edge pixel
        ▪ If the edge pixel has more than 1 edge pixel neighbor add them to the list of
          backtrack pixels
```

Fig. 4.1: The basic steps needed to perform the edge walking procedure.

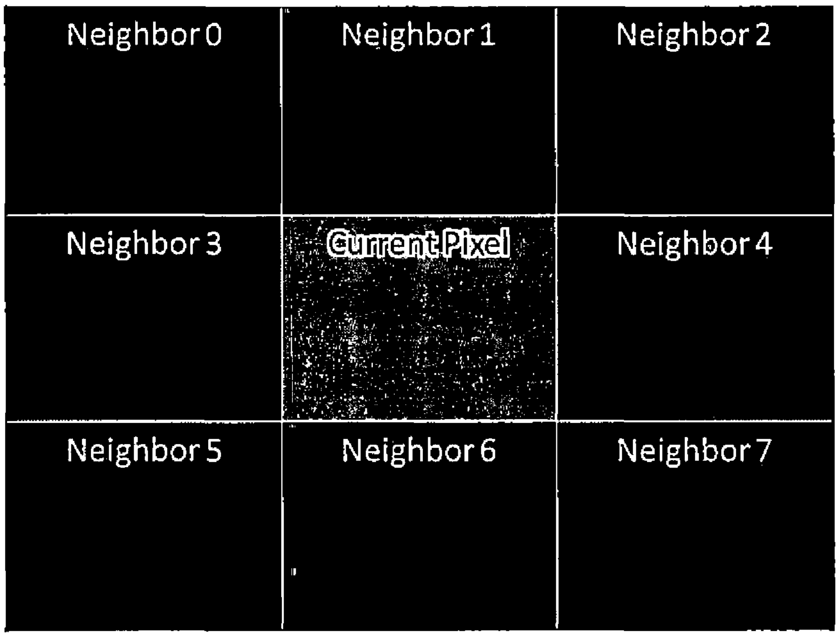

Fig. 4.2: The eight possible pixel neighbors a pixel can have.

4.4 Edge Walking Algorithm Use in the Computer System

The given basic edge walking algorithm was used to implement 2 different edge walking functions. The first function is named EdgeWalker, which takes ten parameters. These parameters include:

1. An input image
2. An x coordinate position in the image to start performing the procedure
3. A y coordinate position in the image to start performing the procedure
4. The minimum number of pixels that must be present in a single edge for it to be valid

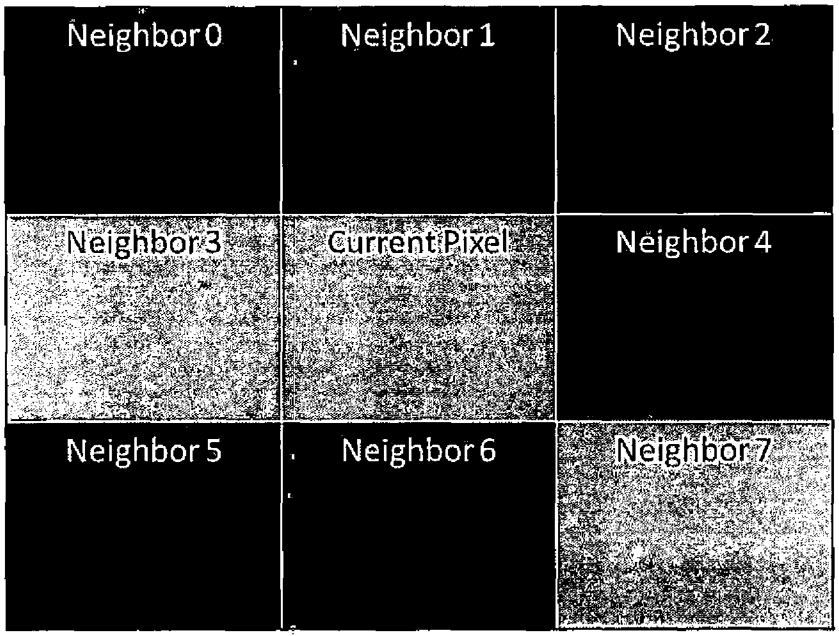

Fig. 4.3: A possible configuration of neighboring pixels a pixel that is being examined can have.

5. The maximum number of pixels that can be present in a single edge for it to be valid
6. The width of the area to perform the procedure in
7. The height of the area to perform the procedure in
8. A vector of "x" coordinates that represent valid edge pixels
9. A vector of corresponding "y" coordinates to the "x" coordinates that represent valid edge pixels
10. The number of pixels found that belong to valid edges

This function locates those edges that have a pixel count that falls between a minimum pixel count and a maximum pixel count. The result is an image that only

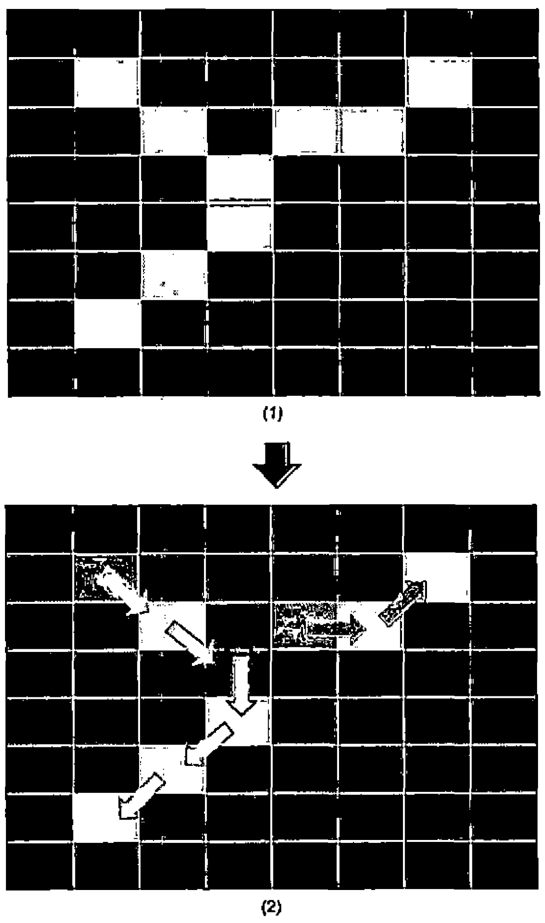

Fig. 4.4: An input image (1) and pixels walked in that image (2).

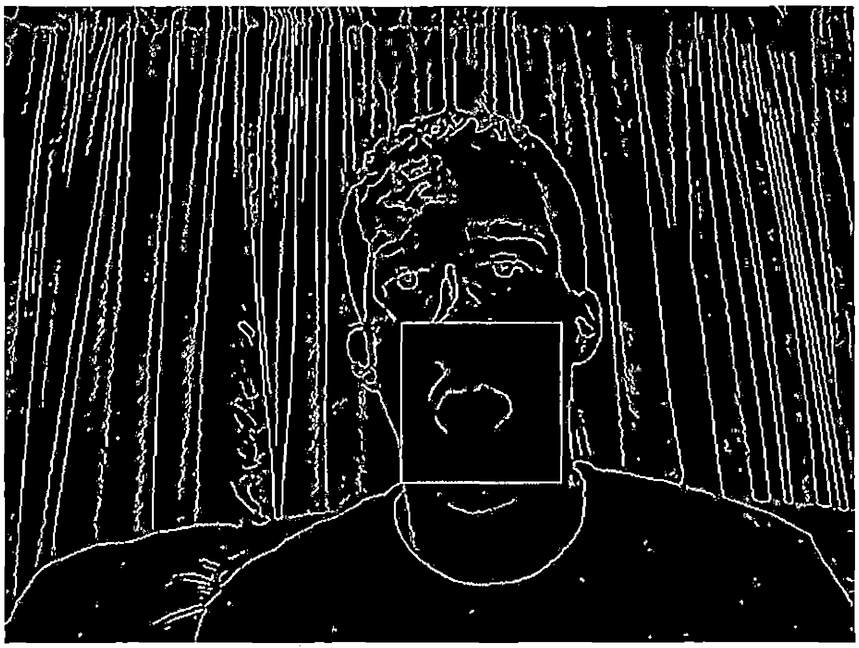

Fig. 4.5: The result of performing the edge walking procedure that finds the edges that consist of a pixel count that falls within the minimum pixel count size and maximum pixel count size.

has the edges that meet the size requirement remaining, See Figure 4.5. In Figure 4.5 the area of the image that contains these valid edges are found inside the bounded square region.

The second function is named FindLargestEdge, which takes eight parameters. These parameters include:

1. An input image
2. An x coordinate position in the image to start performing the procedure
3. A y coordinate position in the image to start performing the procedure
4. The minimum number of pixels that must be present in the largest edge for it to be valid

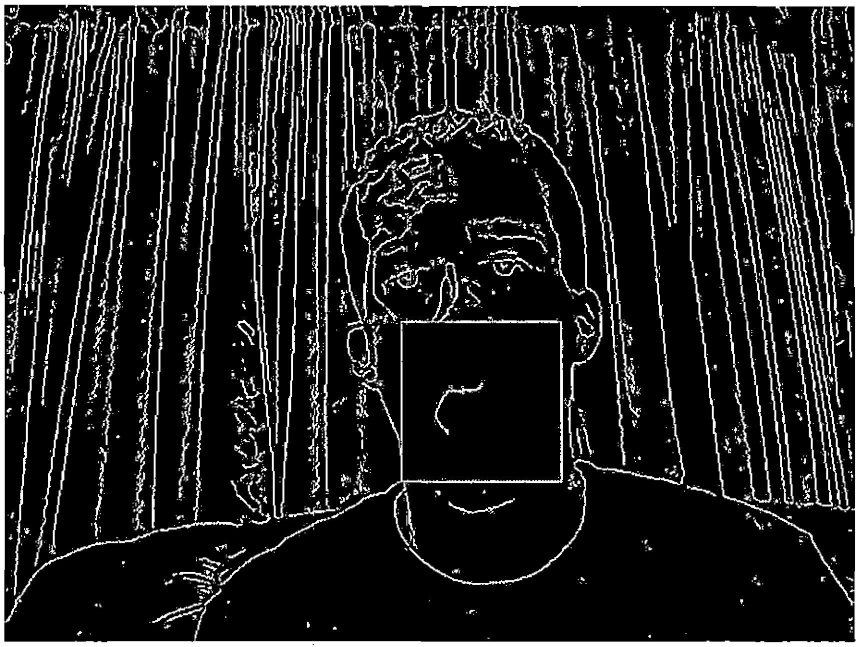

Fig. 4.6: The result performing the edge walking procedure that finds the edge with the largest pixel count.

5. The maximum number of pixels that can be present in the largest edge for it to be valid

6. The width of the area to perform the procedure in

7. The height of the area to perform the procedure in

8. The number of pixels present in the largest edge

This function locates the edge that has the largest pixel count among all the edges in the area that is being considered. The result is an image that only has the largest edge that meet the requirement remaining, see Figure 4.6. In Figure 4.6 the area of the image that contains these valid edges are found inside the bounded square region.

The advantage of using the EdgeWalker function is that it allows multiple edges to be present, which give more information in describing the shape of a given feature. The advantage of the FindLargestEdge function is that since it only finds the largest edge, it is more highly likely that the edge describes the shape of only one feature.

However, there are disadvantages to consider when choosing which method to utilize. If the EdgeWalker function is used, it is possible to have edges that belong to separate features present. For example, there can be edges that define the user's nose and other edges that define the user's mouth. Those edges that define the nose can thus possibly cause miscalculation in the location of the user's mouth and current mouth expression. Also by using the EdgeWalker function it is possible to exclude important smaller edges that don't meet the size requirement because a feature can be defined by a variety of edges of different sizes based on the given lighting configuration in the room and the user's distance from the camera.

The disadvantage of using the FindLargestEdge function was that there is a possibility that the largest edge found does not capture the entire shape of a feature. This can result from insufficient lighting in the room that may possible cast shadows. Thus causing these edges located in the shadow to not be detected.

Testing of the edge walking functions involved the subject to be tested in four settings (living room, classroom, kitchen, and bedroom), two lighting situations (standard fluorescent and incandescent aimed at the face), and three times of the day (morning, noon, and late afternoon) to cover plausible situations for the software might be used. In addition, only the thesis author was used as a subject because of human subject testing regulations. A method and setting combination was consid-

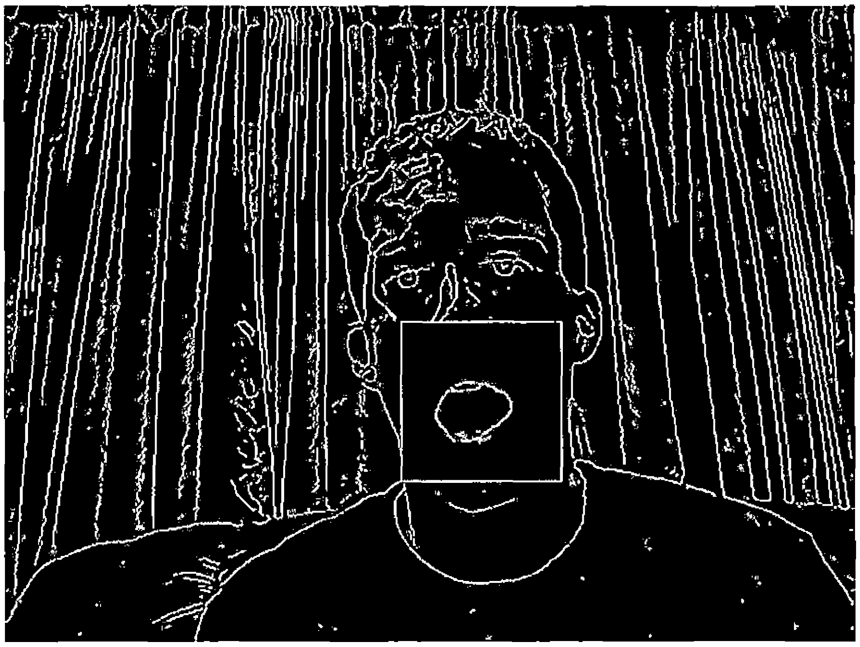

Fig. 4.7: The result of using the edge walking function (FindLargestEdge) and the gap filling function with a maximum allowed gap fill of one pixel.

ered passable if it provided passable if it provided a usable output in the 90 percent of trials overall. There were five runs for each test, and the basis for selection on what was considered passable output was how well a usable outline of a feature was generated. If the outlined generated defined approximately 80 percent of the feature, the output was deemed passable. Through the testing of the images by using the two different edge walking functions separately on each image, in combination with the gap filling function that is covered in Chapter 5, it was discovered that the FindLargestEdge function provided the best representation of the user's mouth expression, see Figure 4.7. The pixel gap parameter value that worked best for the gap filling function, in combination with the FindLargestEdge function, was to have it set to only connect edges with a one pixel gap between each other.

5. CONNECTING SEPARATE EDGES TOGETHER

5.1 Chapter Overview

This chapter covers the procedure used to connect separate, but in close proximity, edges together to form a single edge. It also presents the idea that image modification can enhance the image's content and provides a better basis to perform calculations upon. An edge connecting algorithm is needed in this computer system because a means to modify the existing edge representation of the user's bodily features to a more well defined outline of those features is essential in making the process of locating and classifying a given mouth expression made by the user easier.

1. Section 5.2 provides a brief overview on how image modification can provide a better basis for computation.
2. Section 5.3 covers the steps of how the system fills in the gaps between edges that are in close proximity.
3. Section 5.4 covers how the system uses the gap filling algorithm to construct a better defined representation of the user's mouth.

5.2 Introduction

Before an edged image can be used in order to locate the user's mouth, an attempt in producing a better quality edged image must be made. In some cases, a given input image may not be of optimal quality because given the contours of the user's face and mouth, it is possible that gaps between edges can be present, and can result in an edge to not be considered as part of a key feature. In order to link all of these separate edges that are in close proximity together to form more complete figure, it is essential to fill in gaps between these edges. In the image, these gaps are represented by black colored pixels, while an edge is represented by a gray colored pixel.

5.3 Edge Connecting Algorithm Design

For the the basic procedure that is performed to fill in the gaps between edges, see Figure 5.1. The first image, which is labeled (1), represents the given image of edge pixels. the second image, which is labeled (2), represents the found endpoints of those edges. The endpoints are represented as yellow pixels in the image. The third image, image labeled (3), represents the connections that were found for these edges. These new edge pixels that are added are the red pixels in the image. Upon examination of Figure 5.1, it is found that these images can be expanded and translated to more a precise algorithms that lends itself to easier implementation, see Figure 5.2.

An endpoint of an edge has at most eight possible areas that can be searched for other nearby edge pixels, see Figure 5.3. In Figure 5.3, the endpoint pixel is represented by the yellow pixel in the center, while each area that can be searched is represented by a different color. It should be noted that the figures and algorithm are assuming at most a three pixel gap, but the figures and algorithm can be generalized for a pixel gap sizes greater than three. Upon examination of Figure 5.3, it is found that each colored area can be broken down into their own separate area scanner, see Figure 5.4. The illustrations labeled (1) through (8) show the possible eight individual scanners that can be created from area of pixels surrounding an endpoint pixel as shown in Figure 5.3. The grey pixel represents the current endpoint pixel, while the blue pixels represent the pixels that can be searched to possible make a connection to. The black pixels represent pixels that are outside the search area for that scanner.

Upon examination of Figure 5.4, it is seen that there are two basic scanner structures with three levels of pixels common among them, see Figure 5.5. The brown pixels represent the first level to be scanned. The orange pixels represent the second level to be scanned. The green pixels represent the third level to be scanned. The yellow pixel serves as a check to make sure there is not already a connection made. In addition, the structure of the different levels in the scanner labeled (1) can be applied to those scanners labeled (1), (2), (3), and (4) in Figure 5.4 by doing a simple rotation. The structure of the different levels in the scanner labeled (2) can be applied to those scanners labeled (5), (6), (7), and (8) in Figure 5.4 by doing a simple rotation.

Referring back to Figure 5.4, if an edge pixel is found in the blue pixel area, then a connection can be made to that endpoint pixel. The connection that can be made is demonstrated in Figure 5.6. The connection illustrations labeled (1), (2), (3), (4), (5), (6), and (7) illustrate the possible seven connections that can be made for a three pixel gap using the scanner labeled (4) from Figure 5.4. The red pixels represent the pixels that connect together two edge pixels. These seven possible connections are the same for the scanners labeled (1), (2), and (3) from Figure 5.4 because the scanner's structure is the same and can be achieved by a simple rotation. The connection illustration labeled (8) illustrate the only possible connection that can be made for a three pixel gap using the scanner labeled (5) from Figure 5.4. In addition, if a smaller gap needs to be filled, the same concept applies like above, but now the "Found Pixel" replaces the red pixel in level 2 for a two pixel gap fill or the red pixel in level one for a one pixel gap fill.

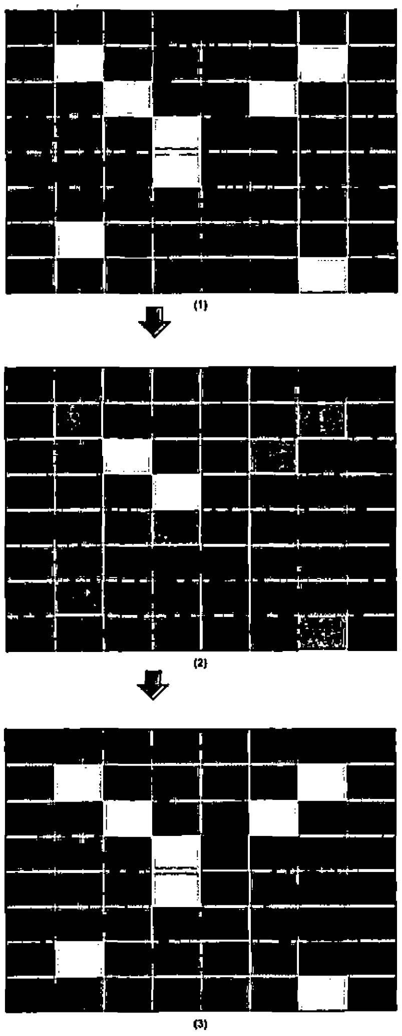

Fig. 5.1: An image of edges (1), an image of found endpoints (2), and an image of connections made (3).

- Locate an edge pixel
- While edge not walked
    - Calculate neighbors of edge pixel
    - Determine if that pixel is an endpoint or not
    - If pixel has no edge pixel neighbors
        - If there are no backtrack pixels to consider
            - All pixels in edge have been found
            - For all endpoints collected
                - Take an endpoint pixel that was stored and determine if it can be connected to another pixel using one of the eight different pixel connector functions
        - Else there are backtrack pixels to consider
            - Go to that pixel to continue walking edge
    - Else pixel has edge pixel neighbors
        - Walk to a neighboring edge pixel
        - If the edge pixel has more than 1 edge pixel neighbor add them to the list of backtrack pixels

Fig. 5.2: The basic procedure that is performed to fill in the gaps between nearby edges.

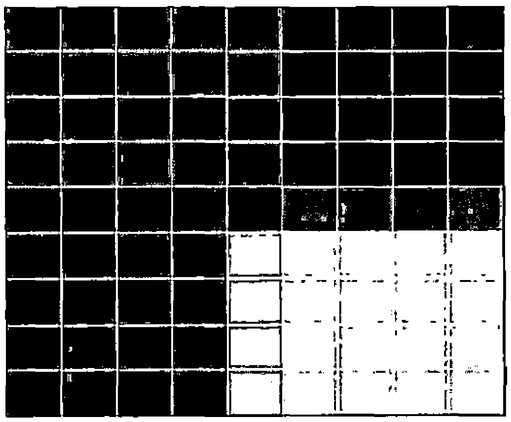

Fig. 5.3: The eight possible areas that can be searched from an endpoint pixel (yellow pixel).

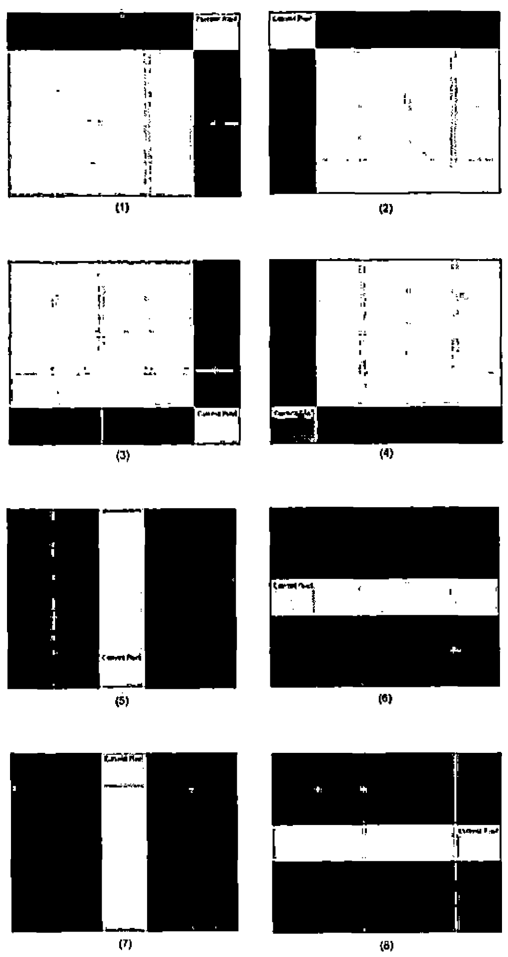

Fig. 5.4: The eight possible scanners that an endpoint pixel can utilize.

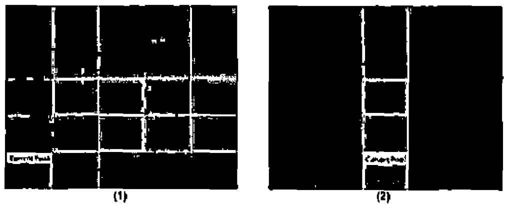

Fig. 5.5: The two basic scanner structures.

5.4 Edge Connecting Algorithm Use in the Computer System

The given gap filling algorithm was implemented by a function named GapFiller, which takes six parameters. These parameters include:

1. An input image
2. An x coordinate position in the image to start performing the procedure
3. A y coordinate position in the image to start performing the procedure
4. The maximum number of pixels to use to connect edges together
5. The height of the area to perform the procedure in
6. The width of the area to perform the procedure in

It was decided to limit the maximum number of pixels to be used to fill in a gap to be no more than three because this would eliminate likelihood of connecting edges that should not be connected to each other. For example, it is possible that an edge that represents the top portion of the lip to be connected to an edge that represents bottom portion of the nose. If this connection happened, it would result in the incorrect constructing of user's mouth, which would make it difficult for the system to locate the mouth in the upcoming processing steps.

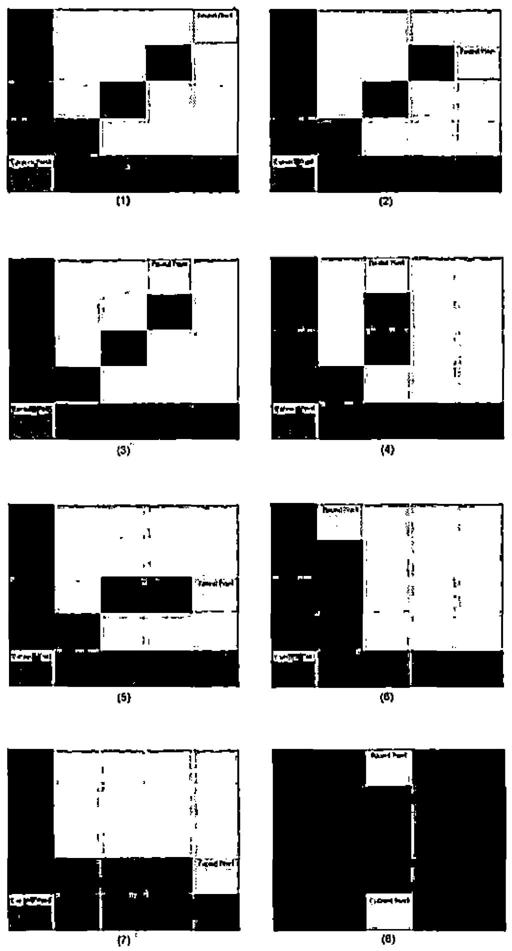

Fig. 5.6: The connection that is made for each scanner.

The gap filling function was tested and used in combination with either the edge walking function that found the largest edge or the edge walking function that returned edges that met the a set size requirement. Testing involved the subject to be tested in four settings (living room, classroom, kitchen, and bedroom), two lighting situations (standard fluorescent and incandescent aimed at the face), and three times of the day (morning, noon, and late afternoon) to cover plausible situations for the software might be used. In addition, only the thesis author was used as a subject because of human subject testing regulations.A method and setting combination was considered passable if it provided passable if it provided a usable output in the 90 percent of trials overall. There were five runs for each test, and the basis for selection on what was considered passable or usable output was how well all the edges of a given feature were connected together. This was verified through visual inspection of the images. In addition, if the majority of the edges were connected to other edges on different features, the output was deemed unusable. All testing was performed on input images similar to Figure 5.7. The pixels within the bounded square region are used for the basis of computation and manipulation.

The first step in testing was to start with a gap filling pixel size parameter of one pixel. In other words, edges that are one pixel apart would be connected together to form a single edge. Figure 5.8 shows the result of using the gap filling function with a gap filling pixel size parameter of one pixel, in combination with the edge walking function that finds the largest edge. While in Figure 5.11 shows the result of using the gap filling function with a gap filling pixel size parameter of one pixel, in combination with the edge walking function that finds edges that meet a set pixel size requirement. The pixel size requirement used for testing was edges were valid edges if they had a minimum pixel count size of 50, but less than or equal to a maximum pixel count size of 1000.

The next step in testing was to increment the gap filling pixel size parameter to two pixels. Figure 5.9 shows the result of using the gap filling function with a gap filling pixel size parameter of two pixels, in combination with the edge walking function that finds the largest edge. While in Figure 5.12 shows the result of using the gap filling function with a gap filling pixel size parameter of two pixels, in combination with the edge walking function that finds edges that meet a set pixel size requirement. The pixel size requirement used for testing was edges were valid edges if they had a minimum pixel count size of 50, but less than or equal to a maximum pixel count size of 1000.

The final step of testing was to increment the gap filling pixel size parameter to three pixels. Figure 5.10 shows the result of using the gap filling function with a gap filling pixel size parameter of three pixels, in combination with the edge walking function that finds the largest edge. While in Figure 5.13 shows the result of using the gap filling function with a gap filling pixel size parameter of three pixels, in combination with the edge walking function that finds edges that meet a set pixel size requirement. The pixel size requirement used for testing was edges were valid edges if they had a minimum pixel count size of 50, but less than or equal to a maximum pixel count size of 1000.

In addition, for comparison purposes, Figure 5.14 and Figure 5.15 show the resulting image if no gap filling is performed on an image. From these images it is seen how

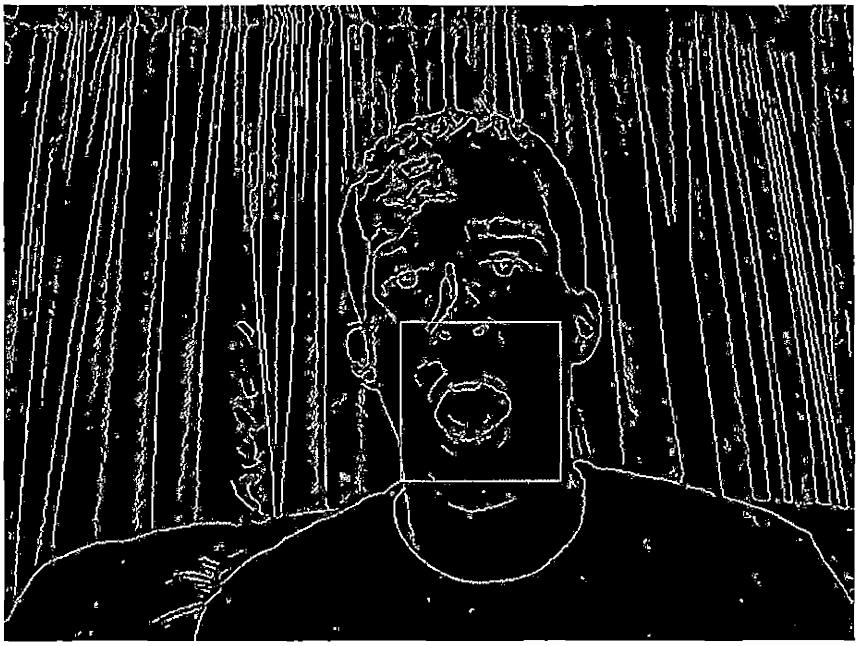

Fig. 5.7: Example of an edged input image that is used by the system.

important it is to perform a gap filling between the edges in order to construct an accurate representation of the user's mouth expression. Upon completion of testing, it was found that having the gap size parameter of the gap filling function set to a pixel gap size of one, and in combination of using the edge walking function that finds the largest edge, provided the best representation of just the user's mouth in the bounded region of the image. This was concluded because the gap filling pixel size parameter of one pixel provided a well defined outline of the user's mouth, while having a lesser likelihood of having edges be connected to other facial features. As the gap filling pixel size parameter was increased, the likelihood of edges being connected to other edges that should not have been connected together was increased.

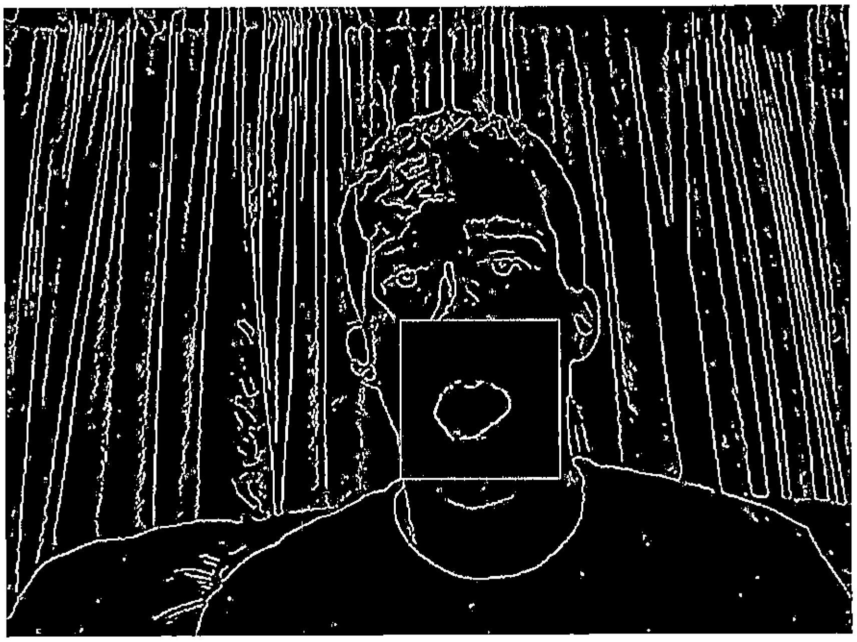

Fig. 5.8: Result of using the GapFiller function with gap filling pixel size parameter of one pixel and Find-LargestEdge function.

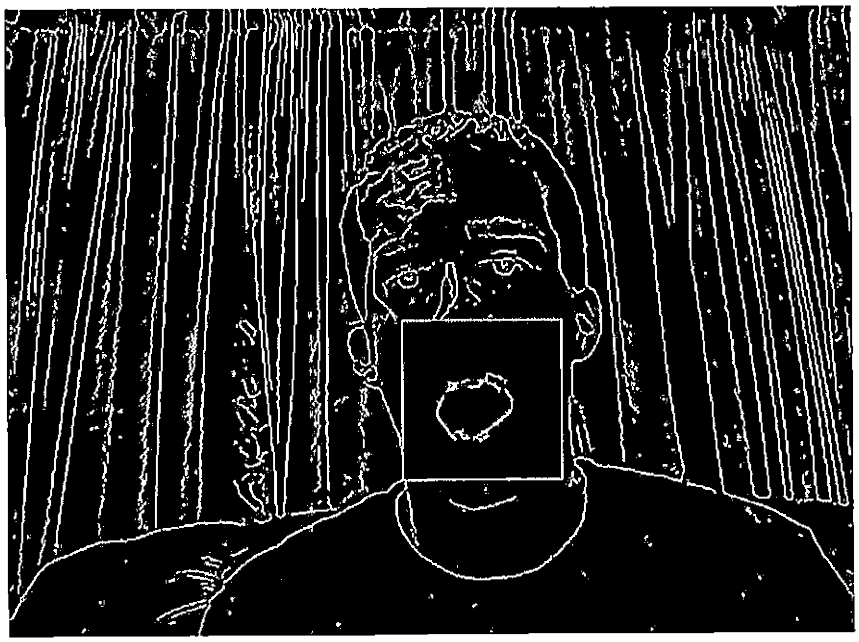

Fig. 5.9: Result of using the GapFiller function with gap filling pixel size parameter of two pixels and FindLargestEdge function.

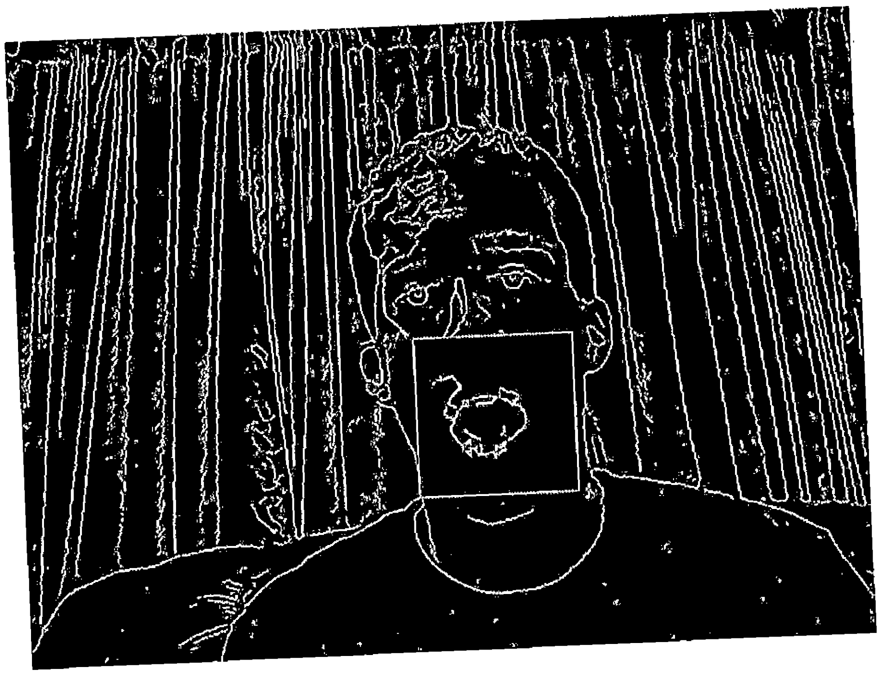

Fig. 5.10: Result of using the GapFiller function with gap filling pixel size parameter of three pixels and FindLargestEdge function.

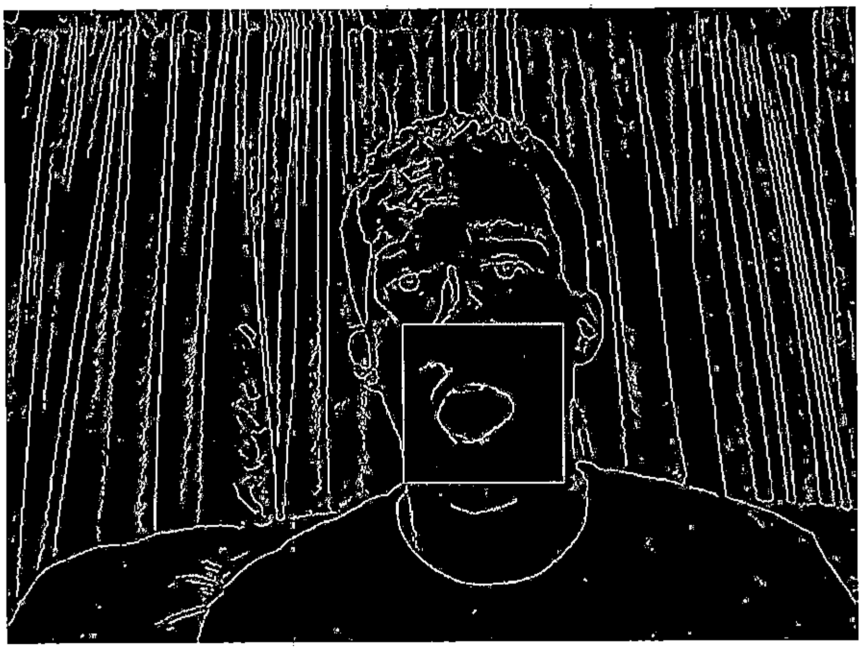

Fig. 5.11: Result of using the GapFiller function with gap filling pixel size parameter of one pixel and EdgeWalker function.

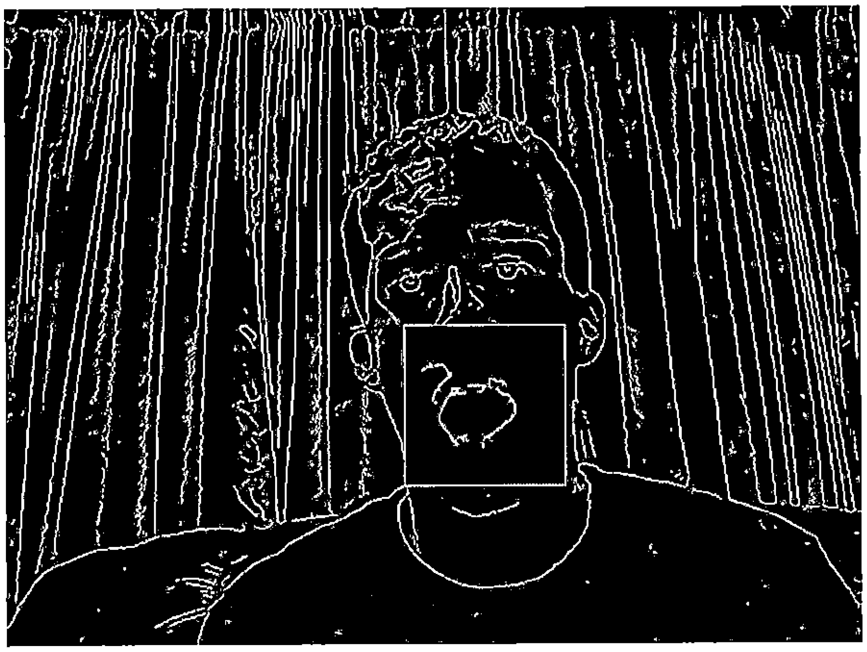

Fig. 5.12: Result of using the GapFiller function with gap filling pixel size parameter of two pixels and EdgeWalker function.

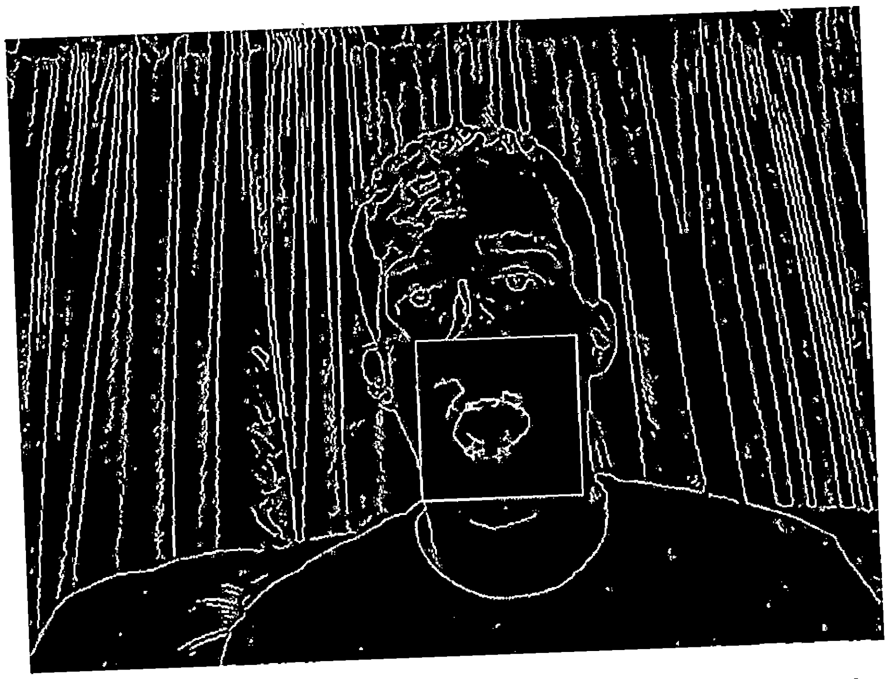

Fig. 5.13: Result of using the GapFiller function with gap filling pixel size parameter of three pixels and EdgeWalker function.

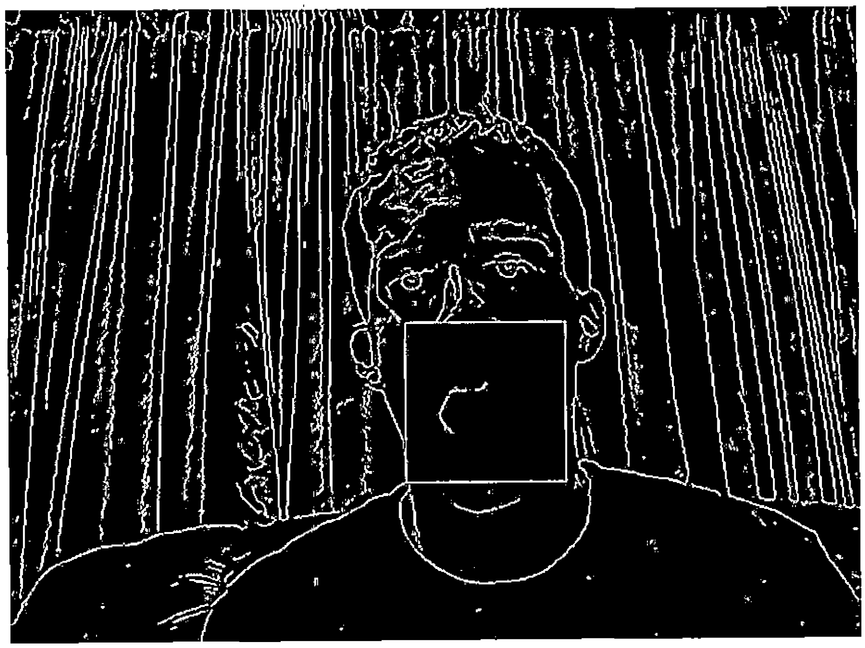

Fig. 5.14: The result of not using the GapFiller function, but using the FindLargestEdge function.

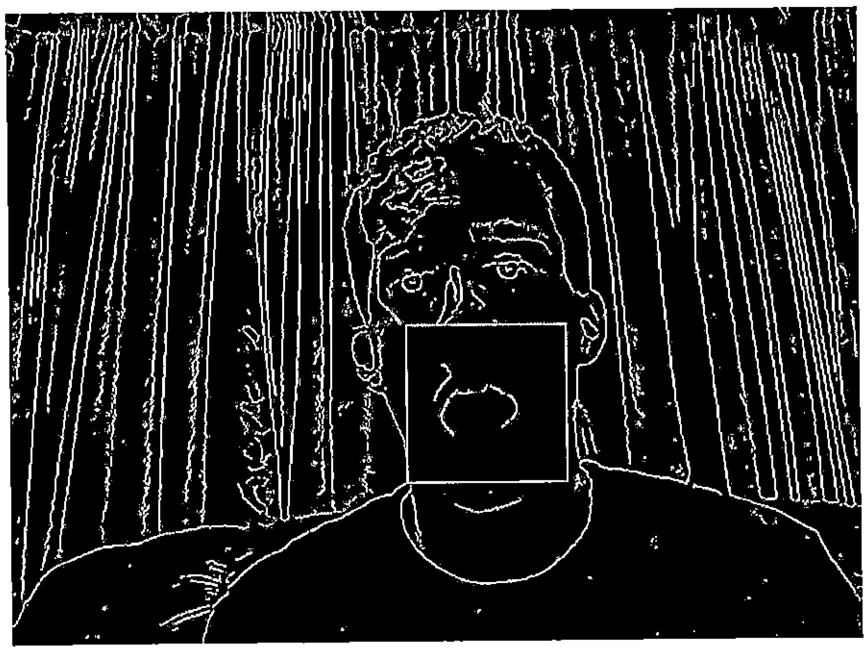

Fig. 5.15: The result of not using the GapFiller function, but using the EdgeWalker function.

6. USE OF FOURIER TRANSFORMS IN FEATURE LOCATION

6.1 Chapter Overview

This chapter covers the concept on how performing calculations in the frequency domain rather than the spatial domain, can provide a more effective means of calculation. It covers how the use of discrete Fourier transforms provide a means to calculate the amount of pixel shift there is between two similar images. An image registration algorithm is needed for this computer system because a means is needed to locate the position of the user's mouth in a input image.

1. Section 6.2 provides an overview on Fourier theory and the advantage of performing computations in the frequency space.

2. Section 6.3 covers the steps of how the system calculates the two-dimensional discrete Fourier transform of an image.

3. Section 6.4 covers how the system uses the two-dimensional discrete Fourier transform of two images to locate the user's mouth expression in the image.

4. Section 6.4.1 provides an overview of the first implementation of a two-dimensional discrete Fourier transform and its effectiveness in locating the user's mouth expression in the image.

5. Section 6.4.2 provides an overview of the second implementation of a two-dimensional

discrete Fourier transform and its effectiveness in locating the user's mouth expression in the image.

6.2 Introduction

The basic idea behind Fourier theory is that it shows that "it is possible to form any one-dimensional function $f(x)$ as a summation of a series of sine and cosine terms of increasing frequency" [11]. As a result, it becomes possible to take data that is in the spatial domain and transform it into the frequency domain [13]. The advantage to processing data in the frequency domain is that the data gets configured in such a manner that it allows the ability to perform operations that would be difficult to perform in the spatial domain. The Fourier transform, which is donated as $F(u)$, for the continuous function $f(x)$ is defined below [11].

$$F(u) = \int_{-\infty}^{\infty} f(x)e^{-2\pi iux}\mathrm{d}x \tag{6.1}$$

The corresponding inverse Fourier transform is defined below [11].

$$f(x) = \int_{-\infty}^{\infty} F(u)e^{2\pi iux}\mathrm{d}x \tag{6.2}$$

The benefit of having a forward and inverse Fourier transform is that it becomes possible to go from the spatial domain to the frequency domain, and then back to the spatial domain.

Given that we are working with images that are not continuous and are "limited by the finite spacing of the sampled points in the image," we must use a transform known as the discrete Fourier transform (DFT) [11]. In addition, since the images are two-dimensional and the above defined Fourier transform was for a one-dimensional function, we must make the appropriate substitutions so that the transform is for two-dimensional functions. The two-dimensional discrete Fourier transform is defined below [12].

$$F(k,l) = \frac{1}{N^2}\sum_{a=0}^{N-1}\sum_{b=0}^{N-1} f(a,b)e^{-i\theta} \tag{6.3}$$

Where θ is equal to:

$$\theta = 2\pi\left(\frac{ka}{N} + \frac{lb}{N}\right) \tag{6.4}$$

The corresponding inverse DFT is defined below [12].

$$f(a,b) = \frac{1}{N^2}\sum_{k=0}^{N-1}\sum_{l=0}^{N-1} F(k,l)e^{i\theta} \tag{6.5}$$

It should be noted that the terms in the above Fourier transform equations can be expanded upon, which can allow a more straight forward implementation of the transform. Consider Euler's formula below: [13]

$$e^{i\theta} = cos(\theta) + isin(\theta) \tag{6.6}$$

Notice that Euler's formula can be substituted in both the forward and inverse Fourier transform equations. Also consider that $f(a,b)$ is equal to a complex number in the form of $R + iI$, where R and I are real numbers and i is equal to $\sqrt{-1}$ [13]. Now by multiplying $R + iI$ by $cos(\theta) - isin(\theta)$ and making the substitution in the forward Fourier transform equation, the forward Fourier transform equation can now be written as: [13]

$$F(k,l) = \frac{1}{N^2}\sum_{a=0}^{N-1}\sum_{b=0}^{N-1}(Rcos(\theta) + Isin(\theta)) + i(Icos(\theta) - Rsin(\theta)) \tag{6.7}$$

Now by multiplying $R + iI$ by $cos(\theta) + isin(\theta)$ and making the substitution in the inverse Fourier transform equation, the inverse Fourier transform equation can now be written as:

$$f(a,b) = \frac{1}{N^2} \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} (Rcos(\theta) - Isin(\theta)) + i(Icos(\theta) + Rsin(\theta)) \qquad (6.8)$$

6.3 Image Registration Algorithm Design

The use of DFT's serve as an efficient means to calculate the amount of pixel shift that needs to be applied to a reference image in order for that image to match or closely resemble the input image. See Figure 6.4 for an example of a reference image and a shifted input image. Presented below is the algorithm that calculates the amount of pixel shift in the "x" and "y" directions that needs to be applied to a reference image in order for it to better match the input image. See Figure 6.3 for an illustration of the algorithm.

1. Perform a DFT on the input image
2. Perform a DFT on the reference image
3. Normalize the two images into a single image by divide the result of performing the DFT on the input image by the result of performing the DFT on the reference image
4. Perform an inverse DFT on the normalized image result
5. Locate the pixel with the largest value. The indices of this pixel serve as the amount of pixel shift to be done to the reference image
6. Shift the reference image by the calculated pixel shift values

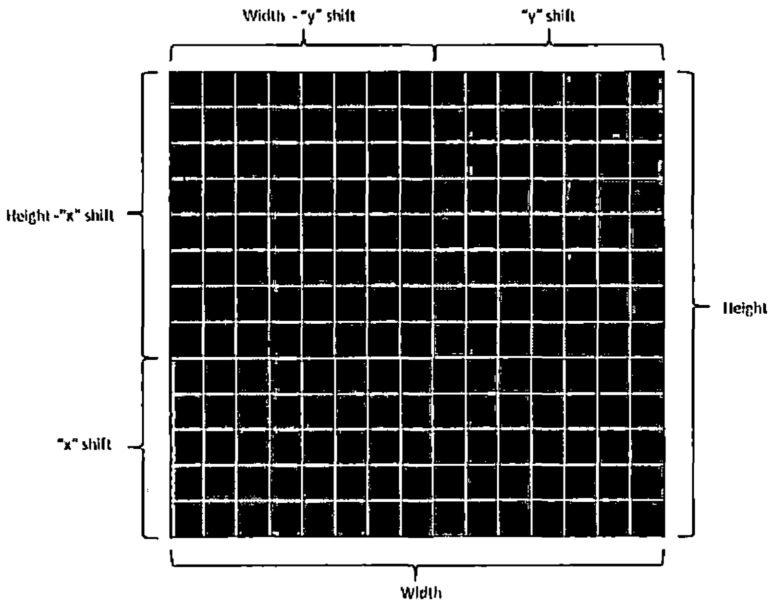

Fig. 6.1: The division of the image based on pixel shift values before shifting.

Once the pixel shift values in the "x" and "y" directions are calculated, the image is divided into the appropriate number of sections based on these pixel shift values. These sections are illustrated in Figure 6.1. Once the image is divided into sections, these sections are rearranged in the image to match the configuration that is illustrated by Figure 6.2.

6.4 Image Registration Algorithm Use in the Computer System

The basic algorithm for calculating the amount of pixel shift between two images presented above was implemented and used in the system in order to shift the mouth expression reference images by the appropriate pixel shift amount so to better match the current location of the user's mouth in the input image. This step is essential in preparing the images for the upcoming step where the system will determine how

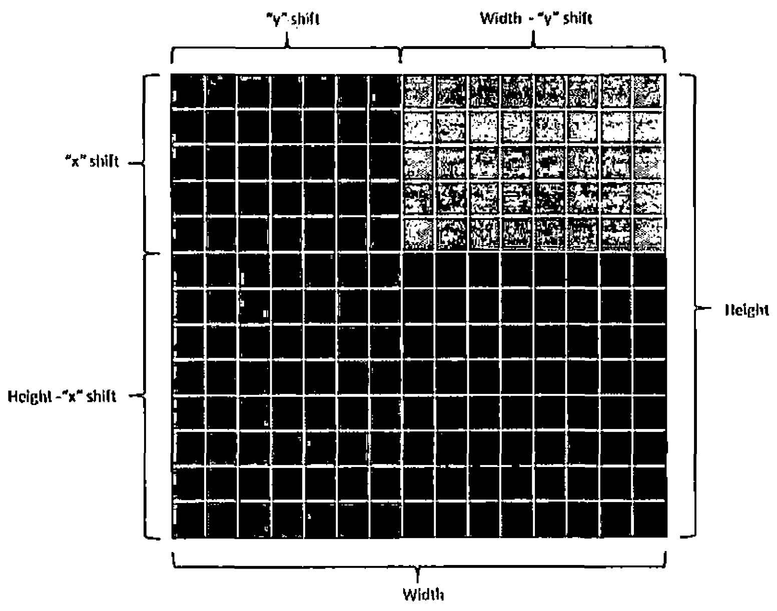

Fig. 6.2: The rearrangement of the image based on pixel shift values.

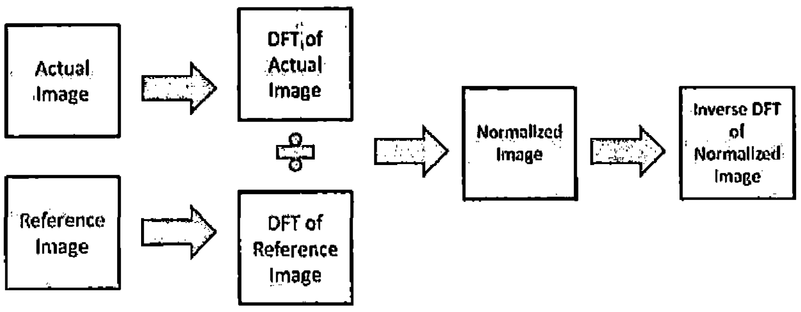

Fig. 6.3: Calculation of pixel shift between two images.

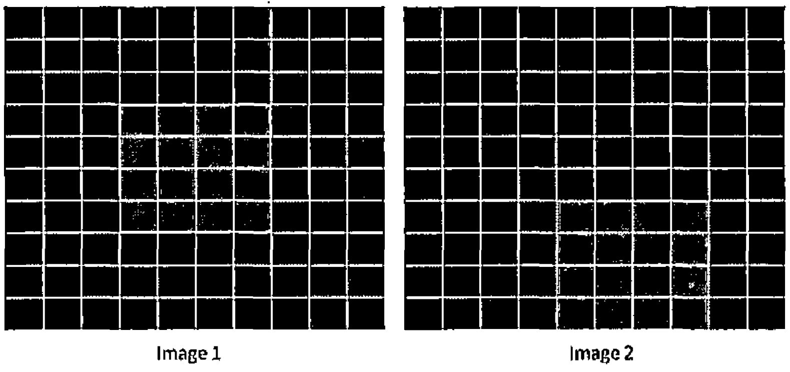

Fig. 6.4: A reference image, labeled "Image 1", and an input image, labeled "Image 2".

well the reference image fits to the input image.

6.4.1 First Implementation

The first implementation involved coding the above mentioned algorithm. The majority of the implementation involved implementing the function that calculated the two-dimensional DFT of an input image. The implementation for the forward DFT function was based on the equation below:

$$F(k,l) = \frac{1}{N^2}\sum_{a=0}^{N-1}\sum_{b=0}^{N-1}(Rcos(\theta) + Isin(\theta)) + i(Icos(\theta) - Rsin(\theta)) \tag{6.9}$$

The implementation for the inverse DFT function was based on the equation below:

$$f(a,b) = \frac{1}{N^2}\sum_{k=0}^{N-1}\sum_{l=0}^{N-1}(Rcos(\theta) - Isin(\theta)) + i(Icos(\theta) + Rsin(\theta)) \tag{6.10}$$

Testing involved 5 runs each for every room, lighting, and time of day combination. However, upon testing of the functions, it was found that the execution time was far too slow for practical use in the system. The overall time to complete all procedures including edge processing and the calculation using DFT's for the amount of shift needed given one reference image and one input image was on average 64 seconds. Only the edge processing procedures took on average .1 seconds. This shows that the calculation for the amount of shift needed using this DFT implementation consumed most of the overall execution time. Since the number of reference images that are going to be used by the system is three, the total execution time for one input image and three reference images would be roughly three times as long using this implementation. Given that the time for execution is too long to be practical, another approach needed to be developed.

6.4.2 Second Implementation

Since the first implementation of the algorithm was far too slow in execution, specifically the DFT implementation portion, an alternative solution had to be found. Upon further research, an alternative solution was found. The solution was to use a free software called FFTW that provided C routines to compute discrete Fourier transform. The figure below illustrates a basic example how to use the FFTW routines to compute a two-dimensional DFT. Before using the routines, memory must be allocated for the input and output arrays. Then a plan must be created that contains all the information needed to compute the DFT. See Figure 6.5 for an example of how to set up and execute a plan to compute a two dimensional DFT. Once the plan is created it is executed by calling the function fftw_execute. Finally the plan is destroyed and the allocated memory for the input and output arrays are deallocated. Testing

```
#include "fftw3.h"
...
#define SIZE 120
int _tmain(int argc, _TCHAR* argv[])
{
 ...
 fftw_complex *in, *out;
 fftw_plan p;
 in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * SIZE * SIZE);
 out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * SIZE * SIZE);
 p = fftw_plan_dft_2d(SIZE, SIZE, in, out, FFTW_FORWARD, FFTW_ESTIMATE);
 fftw_execute(p);
 fftw_destroy_plan(p);
 fftw_free(in);
 fftw_free(out);
 ...
}
```

Fig. 6.5: Example of basic usage of FFTW functions.

involved 5 runs each for every room, lighting, and time of day combination. After extensive testing it was found that this implementation provided the best performance. This implementation executed quick enough to not hinder the overall performance of the system. It was found that up to three reference images, which included the big mouth expression, the small mouth expression, and the closed mouth expression, still provided an adequate execution time. Anymore than three would cause the execution time for each input image to increase to a point that the system would run too slow to be effective.

The overall time to complete all procedures including edge processing and the calculation for the amount of shift needed given one reference image and one input image was on average .1 seconds. The edge processing procedures took on average .09 seconds, while the procedures to calculate the shift amount took a time of on average .01 seconds. With three reference images and one input image, the total time was on average 0.14 seconds. From these results, it is shown that this implementation was a vast improvement over the first implementation, and thus would be the implementation that would be used by the system.

7. MATCHING THE REFERENCE IMAGE TO THE INPUT IMAGE

7.1 Chapter Overview

This chapter covers the procedure on how to determine how well a reference image matches to an input image. This is accomplished by inverting pixel values and using logical AND operations. An image matching algorithm is needed for this computer system because a means is needed to determine which mouth expression the user made based upon predefined mouth expression reference images. It is essential that this matching process is accurate because based on this determination of the mouth expression made, a specific feeding action would then be carried out by a robotic arm.

1. Section 7.2 provides an overview of the idea that since it is unlikely that an input image would match a reference image perfectly, calculations should be performed to determine how well an input image matches to a reference image.

2. Section 7.3 covers the steps of how the system calculates how well an input image matches to a reference image.

3. Section 7.4 covers how the system uses this matching algorithm to determine which mouth expression the user's was making.

7.2 Introduction

In order to determine the given mouth expression made by the user, the use of reference images can be used. Each reference image would have a single mouth expression of the user. By comparing different reference images to the input image, it can be discovered which mouth expression was made by the user in that given moment. This determination is done by comparing how well the pixels in reference image and the input image match up to each other. Based on the how well the match is, it can be determined which is the most likely mouth expression that was made by the user.

Given that it is very unlikely that a user can make a mouth expression that exactly matches a given reference image, it is important to increase the edge thickness of the reference image to make it possible that a given user's mouth expression has the better possibility of fitting to a reference image. This procedure of inserting new edge pixels around existing edge pixels is going to be referred to as "thickening" the reference image.

In order to determine if a pixel in the reference image matches a pixel in the input image, a logical AND operation is performed. But before performing the AND operation between a pixel in the input image and a pixel in the reference image, the pixel values in the reference image must be inversed. That is if a pixel is an edge pixel, it must be changed to a nonedge pixel. However, if a pixel is a nonedge pixel, it must be changed to an edge pixel. This inverse procedure is important in ensuring that if two pixels match that they are removed from final image once the AND operation is performed. Upon completion of performing logical AND operation between the pixels in the two images, a final image remains with only the pixels that do not match up in reference image and input image. It is this unmatched pixel count and total pixel count for the images that can then be used to determine how well the reference image fits to the input image.

7.3 Image Matching Algorithm Design

Below is the basic procedure that is performed to determine if a given reference image is a good match to the input image captured:

1. Thicken the reference image by inserting new edge pixels around existing edge pixels, see Figure 7.1.
2. Inverse pixel values in reference image (1 to 0 and 0 to 1) where 0 represents a unlighted pixel (nonedge pixel) and 1 represents a lighted pixel (edge pixel), see Figure 7.2.
3. Perform an AND operation with the reference image's pixels with the input image's pixels, see Figure 7.3.
4. Count the number of remaining lighted pixels. The number of remaining pixels represent the number of pixels that do not match up with the reference image.
5. Determine if input image is either a big mouth expression, small mouth expression, closed mouth expression, or none of the earlier mentioned based on the percentage of pixels from the input image that match the pixels found in the reference image

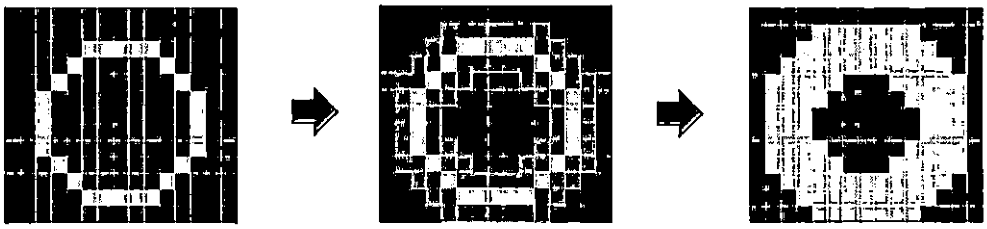

Fig. 7.1: Inserting new edge pixels around existing edge pixels in the reference image.

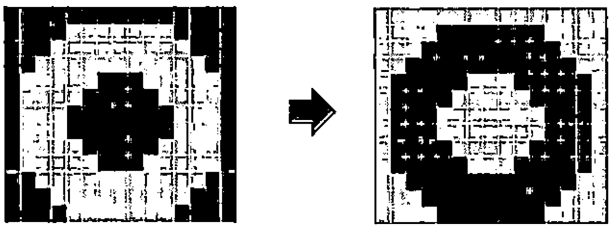

Fig. 7.2: Inversing pixel values in the reference image.

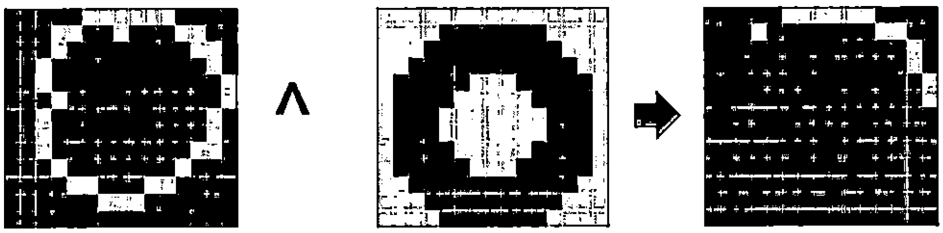

Fig. 7.3: Performing an AND operation with the input image's pixels with the reference image's pixels.

7.4 Image Matching Algorithm Use in the Computer System

The given basic image matching algorithm was implemented in order to match one of the three predefined mouth expressions to a user's current mouth expression, see Figure 7.4. The correct matching of the user's current mouth expression is essential in determining which feeding action the robotic arm should execute.

In order to determine which mouth expression had the best fit, the calculation of the percentage of pixels that matched between the reference image and input image would be used. If the percentage value for that reference image met a set minimum percentage, it was then decided that the reference image was a good enough fit to the input image.

Testing involved the subject to be tested in four settings (living room, classroom, kitchen, and bedroom), two lighting situations (standard fluorescent and incandescent aimed at the face), and three times of the day (morning, noon, and late afternoon) to cover plausible situations for the software might be used. In addition, only the thesis author was used as a subject because of human subject testing regulations.A method and setting combination was considered passable if it provided passable if it provided a usable output in the 90 percent of trials overall. There were five runs for each test, and the basis for selection on what was considered passable or usable output was if the system matched the input image to one of the reference images correctly. The minimum percentage value was either incremented or decremented based on how well the system matched the input image to one of the reference images until an optimal minimum percentage value was found. This was verified through visual inspection of the images.

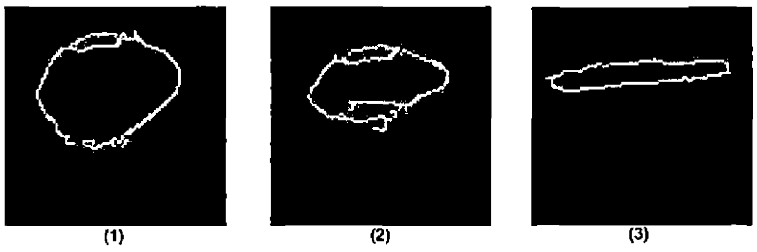

Fig. 7.4: One mouth expression is a big mouth expression (1). The second mouth expression is a small mouth expression (2). The last mouth expression is a closed mouth expression (3).

Through testing it was found that a value 70 percent was an adequate minimum percentage that should be met in determining the correct mouth expression. In other words, if 70 percent of the pixels in the input image were matched, and 70 percent of the pixels in the reference image were matched, then that reference image would be considered a good match. In Figure 7.5, it shows a user making a big mouth expression, and the system matching the correct big mouth expression reference image to the input image. This correct matching is represented by having the reference image of the big mouth expression overlapping the input image of the user's big mouth expression.

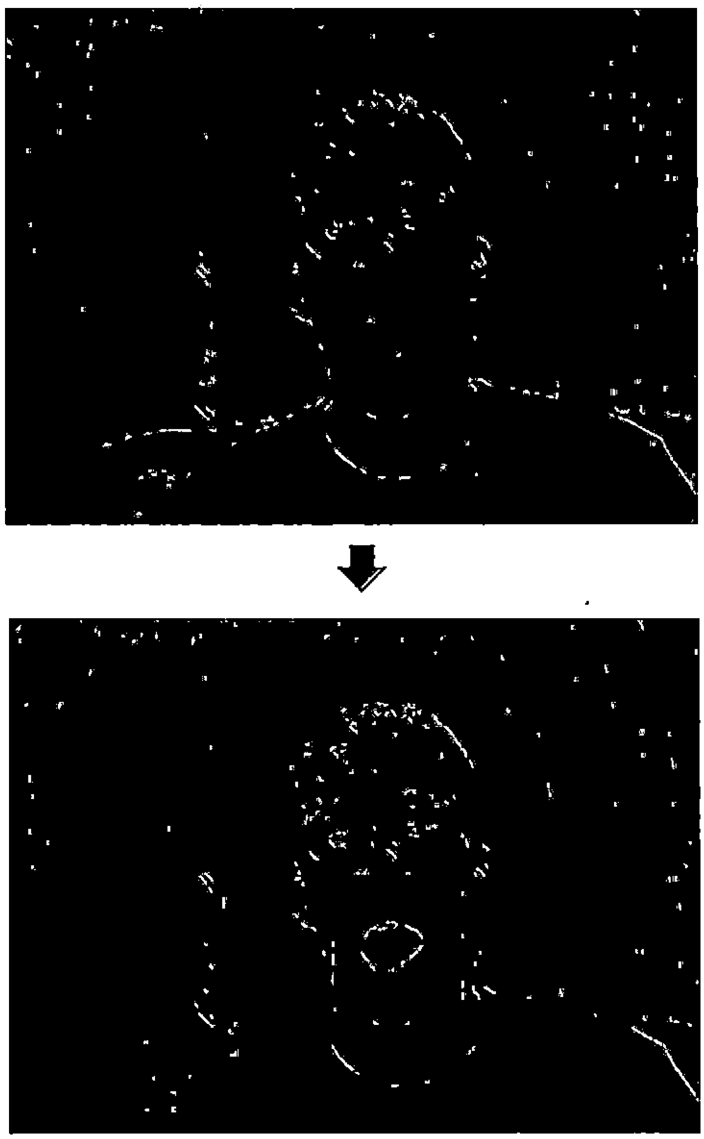

Fig. 7.5: An example of correctly matching a mouth expression reference image to an mouth expression input image

8. RESULTS

8.1 Chapter Overview

This chapter provides and overviews the results from testing the system through the use of a sequence chart. Section 8.2 provides the results on how well the system was able to locate the three defined mouth expressions. In addition, it provides an analysis of these results on why the system performed in the manner it did.

8.2 Mouth Expression Testing Results

In order to test the accuracy of the system, a sequence chart was constructed and utilized. The sequence chart consisted of a random sequence of mouth expressions to be made by the user. The mouth expressions that were included in the chart were the big mouth expression, the small mouth expression, and the closed mouth expression.

Testing involved the subject to be tested in four settings (living room, classroom, kitchen, and bedroom), two lighting situations (standard fluorescent and incandescent aimed at the face), and three times of the day (morning, noon, and late afternoon) to cover plausible situations for the software might be used. In addition, only the thesis author was used as a subject because of human subject testing regulations. An output was deemed passable if it correctly matched the output listed on the sequence chart. A total of 60 mouth expressions were tested throughout the session. Upon

Tab. 8.1: Results of Sequence Chart

Type of Mouth Expression	Number of True Positives	Number of False Positives	Number of False Negatives
Big Mouth Expression	20	0	0
Small Mouth Expression	17	2	1
Closed Mouth Expression	17	0	3

completion of making all the mouth expressions in the sequence chart, the results were tallied and percentages were calculated. Table 8.1 summarizes the results from the sequence chart to show the number of matches and non-matches made by the system for each mouth expression. The parameters that were run during the test are listed below:

1. Distance from camera to individual's face: approximately 1.5 feet
2. Low threshold value: 20
3. High threshold value: 20
4. Edge mode: Largest edge
5. Gap size for constructing reference image: 1 pixel gap
6. Gap size for constructing input image: 1 pixel gap

After tallying the results from the sequence chart, it was found that 54 out of the 60 mouth expressions, or 90 percent of the mouth expressions, made were matched correctly. Upon further review of the chart it is seen that 20 counts of each mouth expression made. Out of the 20 big mouth expressions made throughout the session, 20 were matched correctly. This shows that the system matched 100 percent of the big mouth expressions correctly. Out of the 20 small mouth expressions made throughout the session, 17 were matched correctly. This shows that the system matched 85 percent of the small mouth expressions correctly. Finally, out of the 20 closed mouth expressions made throughout the session, 17 were matched correctly. This shows that the system matched 85 percent of the closed mouth expressions correctly.

From the results, it is seen that the big mouth expression was the most frequent correctly matched mouth expression. This maybe due to the fact that if the user makes the biggest opening with his or her mouth possible, it serves as a natural stopping point because the jaw cannot be extended any further. Thus making it possible to better match the big mouth expression reference image and doing it repetitively with a high rate of accuracy. In addition, since in the edge processing step, the technique of only keeping the largest edge present was used, it presents an advantage for the big mouth expression. Since the big mouth expression usually consists of a large number of pixels that defines it, there usually is no edges that define other features around the mouth that have a greater pixel count than the big mouth expression. As a result, the edge that defines the big mouth expression, is more likely to show up in every image, thus leading to a higher rate of detection.

Given that the detection rate for the small mouth expression and closed mouth expression were the same, it can not be concluded on which one had the worst detection rate. However, there were observations made that could possible explain the much lower detection rate than the big mouth expression. The lower rate of detection for the small mouth expression could be due to the fact that it can be difficult to repetitively make the a small mouth expression that closely matches the small mouth reference image. Since when attempting to make the small mouth expression, there is no natural stopping point to help in making a consistently shaped small mouth expression, this can possible lead to making a small mouth expression that does not match the small mouth expression reference image well. This also leads to the possibility of making an incorrect mouth expression match. This would be were an attempted small mouth expression is made, but the system classifies it as a big mouth expression. Since there is no natural stopping point for making a small mouth expression, the possibility of extending the mouth opening beyond what is considered a small mouth expression is possible. Thus leading the system to incorrectly match the mouth expression. Lastly, the low detection rate can be due to the fact that since in the edge processing step, the technique of only keeping the largest edge present was used, there is a possibility that there is another edge that defined another feature that had a larger pixel count than the edge that defines small mouth expression, thus causes the edge that defines the small mouth expression to be removed from the image. This possibility is likely to occur, because of a bad lighting configuration in the room.

The lower rate of detection for the closed mouth expression was observed to be due to the fact that since in the edge processing step, the technique of only keeping the largest edge present was used, and in most cases the closed mouth expression was defined by an edge that had a low pixel count, other edges that defined other features usually had larger pixel counts than the edge that defined the closed mouth expression. As a result, the edge that defined that other feature would remain in the image, thus leading to the system in being unable to detect the closed mouth expression. Compared to the other mouth expression, the pixel count of the closed mouth expression was hugely dependent on the lighting configuration in the room. If the light was not distributed evenly throughout the room, it was very difficult to make a well defined closed mouth expression to show up in the input image (input image) so that it can be used as a reference image. If a well defined closed mouth expression reference image could not be made, it made it more difficult to detect the user's closed mouth expression.

In addition to testing the accuracy of the computer system on matching the user's mouth expression to one of the mouth expression reference images, the amount of time taken to find a mouth expression that matched the user's mouth expression was recorded. When the computer system was set to only locate one mouth expression out of the three possible mouth expressions, on average it took .2 seconds to calculate the user's mouth expression that matched that single mouth expression reference image. When the computer system was set to only locate two mouth expressions out of the three possible mouth expressions, on average it took .25 seconds to calculate the user's mouth expression that matched one of the two mouth expression reference images. When the computer system was set to locate all three mouth expressions, on average it took .28 seconds to calculate the user's mouth expression that matched one of the three mouth expression reference images. From these results, it is seen how the calculation time increased as the number of different mouth expressions that were searched for was increased. Given this, it is possible for the computer system's response time to become too great for a given user to use the computer system effectively and comfortably. So decreasing the number of mouth expressions that are searched for should be attempted to see if greater performance and overall comfort for the user is achieved.

In the end, it is seen that the system has a overall high detection rate for the different mouth expressions. However, this accuracy rate is entirely dependent on the current lighting configuration in the room, and how well the user is able to consistently make these mouth expressions. Inability to meet these requirements would result in a loss of accuracy and overall performance of the system.

9. CONCLUSION

9.1 Chapter Overview

This chapter concludes with presenting the contributions made from completing this thesis. It presents the idea how assistive technologies not only can help an individual complete a task, but also fosters independence. It also presents the current problems faced when developing the system, such as the distribution of light in the room, and recommends ideas and areas to explore for further research.

1. Section 9.2 provides an overview of the contribution made upon completion of this thesis.

2. Section 9.3 covers the current problem encountered upon development and testing of the system. Also presents the limitations that exist with the current system.

3. Section 9.3.1 covers the effects that lighting has on the system.

4. Section 9.3.2 covers the how the distance from which the user is seated from the camera can effect the systems performance.

5. Section 9.4 covers how several possible areas for further research were discovered and noted.

6. Section 9.4.1 presents the idea that different facial features to control the system should be explored.

7. Section 9.4.2 presents how achieving a speedup in the number of frames executed per second would enhance the system's performance.

8. Section 9.4.3 presents the idea that the system could be adapted to allow a user to control other applications and devices.

9.2 Contribution

In this thesis it was shown how it can be possible for severely handicapped individuals with eating difficulties to be assisted by using facial expression recognition to determine the individual's specific need that he or she wants to be performed in regards to eating a meal. By using the user's current mouth expression and matching it to one of the three mouth expression reference images, it becomes possible to classify a mouth expression with 90 percent or better accuracy, and then correspond it to a specific action to be performed by a robotic arm. Such a system will foster the development in independence and provide a basis in further research of feature recognition system to assist the special needs individual.

9.3 Current Problems and Limitations of the Computer System

Throughout the development of this thesis, several problems and limitations were discovered and are noted below.

9.3.1 Effects of Lighting

The manner in which the light is distributed throughout the room that the system was in greatly affected how well the system performed overall, specifically, how well the system was able to classify the current mouth expression. If the light was not distributed evenly throughout room and more light was coming in from a certain direction, it would cause poorly constructed edged images. Since the images would then be of poor quality, it would cause the system to inaccurately identify the mouth expression or to not be able to identify the mouth expression at all. In addition, there was the possibility that the light would cast shadows which would cause unwanted edges in the image which would cause misidentification of the current mouth expression.

9.3.2 Distance from Camera to Individual's Face

The distance from the individual is from the camera did affect how well the system performed. The best results were achieved when the individual maintained roughly the same distance from the camera throughout the execution of the program. The distance that provided the best results was to have the individual's head about 1.5 feet away from the camera. If the individual moved closer or further away from the where the reference images were taken during the execution of the program, the system had difficulty in locating any of the defined mouth expressions. For example, if the individual moved further away from the camera, the mouth expression would be smaller in size. However, even though it may have the general shape of a mouth expression reference image, the system has no means to scale the image to the size of the reference image. As a result, it is most likely that the system will not recognize the mouth expression.

9.4 Future Directions

Throughout the development of this thesis, several possible areas for further research were discovered and noted.

9.4.1 Exploring Other Facial Features to Control the System

Even though the mouth provided an adequate amount of different expressions to cover the basic feeding actions to be performed, other features should be explored to see if they provide better performance. Using other features like the eyes may provide better and more natural control of the system, and possible provide a wider array of actions that can performed. However, more detailed images would need to be captured because the current images created by the system do not provide much detail of the human eye to provide for a wide array of different eye expressions.

9.4.2 Increasing the Number of Frames Executed Per Second

Currently, the method used to classify the different mouth expressions is the DFT. However, as the number of mouth expressions searched for in each image increases, the overall execution time for each image increases. It is extremely important that the number of mouth expression reference images is not so great that it causes the system to run at a speed that makes it unpractical and dangerous. The system should be done with classifying a certain mouth expression before the user makes another expression because if the system was still performing the classification for a mouth expression when the user made another mouth expression, the system would not be able to perform the action that the user wanted at that moment. The current number of mouth expression reference images used by the system is three. However, if more mouth expression reference images are to be added, other methods of classification should be considered.

9.4.3 Using the System to Control Other Devices

Even though in the system was designed to control a robotic arm to assist individuals in eating, it could be generalized to control other devices that special needs individuals may use. With a slight modification, other expressions can be defined to control a device and carry out a specific action. For example, the system could control the lights in a room or control a television. Further research would have to be done to see if applying the system to other devices in the individual's life would be practical and further benefit the individual.

APPENDIX A

SOURCE CODE

A.1 Functions Source Code

```
#include <vector>
#include <math.h>
using namespace std;
#define PI 3.141592653
void DFT(double ***img_space, double ***fspace, int height,
        int width, double n)
{
 double angle;
 double R,I,x,y,a,b;
 for(int xi = 0; xi < height; xi++)
 {
  for(int yi = 0; yi < width; yi++)
  {
   for(int ai = 0; ai < height; ai++)
   {
    for(int bi = 0; bi < width; bi++)
    {
     x = xi; y = yi; a = ai; b = bi;
     angle = 2*PI*((x*a/n) + (y*b/n));
     R = img_space[0][ai][bi];
     I = img_space[1][ai][bi];
     fspace[0][xi][yi] = fspace[0][xi][yi] + (R*cos(angle) + I*sin(angle));
     I = img_space[1][ai][bi];
     R = img_space[0][ai][bi];
     fspace[1][xi][yi] = fspace[1][xi][yi] + (I*cos(angle) - R*sin(angle));
    }
   }
   fspace[0][xi][yi] = fspace[0][xi][yi]*(1/(n*n));
```

```
    fspace[1][xi][yi] = fspace[1][xi][yi]*(1/(n*n));
   }
  }
 }
void iDFT(double ***img_space, double ***fspace, int height,
          int width, double n)
{
 double angle;
 double R,I,x,y,a,b;
 for(int xi = 0; xi < height; xi++)
 {
  for(int yi = 0; yi < width; yi++)
  {
   for(int ai = 0; ai < height; ai++)
   {
    for(int bi = 0; bi < width; bi++)
    {
     x = xi; y = yi; a = ai; b = bi;
     angle = 2*PI*((x*a/n) + (y*b/n));
     R = img_space[0][ai][bi];
     I = img_space[1][ai][bi];
     fspace[0][xi][yi] = fspace[0][xi][yi] + (R*cos(angle) - I*sin(angle));
     I = img_space[1][ai][bi];
     R = img_space[0][ai][bi];
     fspace[1][xi][yi] = fspace[1][xi][yi] + (I*cos(angle) + R*sin(angle));
    }
   }
   fspace[0][xi][yi] = fspace[0][xi][yi]*(1/(n*n));
   fspace[1][xi][yi] = fspace[1][xi][yi]*(1/(n*n));
```

```
    }
   }
  }
  void ArrayToFFTWFormat(double ***img, fftw_complex *dest,
                          i_height, int i_width)
  {
   for(int i = 0; i < i_height; i++)
   {
    for(int j = 0; j < i_width; j++)
    {
     for(int k = 0; k < 2; k++)
     {
      dest[j + i_width * i][k] = img[k][i][j];
     }
    }
   }
  }
  void GetMaxValueFFTW(fftw_complex *dest, int &index_x,
                        int &index_y, int i_height, int i_width)
  {
   double current_max = 0;
   double value = 0;
   for(int i = 0; i < i_height; i++)
   {
    for(int j = 0; j < i_width; j++)
    {
     value = dest[j + i_width * i][0];
     if(value > current_max)
     {
        current_max = value;
        index_x = i;
        index_y = j;
      }
    }
  }
}
void NormalizeFFTW(fftw_complex *fspace1, fftw_complex *fspace2,
                   fftw_complex *norfspace, int f_height,
                   int f_width)
{
 double real_value, real_value2, img_value, img_value2;
 int x, y;
 double num_real, num_img;
 double dem;
 for(x = 0; x < f_height; x++)
 {
  for(y = 0; y < f_width; y++)
  {
   real_value = fspace1[y + f_width * x][0];
   real_value2 = fspace2[y + f_width * x][0];
   img_value = fspace1[y + f_width * x][1];
   img_value2 = fspace2[y + f_width * x][1];
   num_real = real_value*real_value2 + img_value*img_value2;
   num_img = real_value2*img_value - real_value*img_value2;
   dem = pow(real_value2,2) + pow(img_value2, 2);
   if(dem == 0)
   {
    norfspace[y + f_width * x][0] = 0;
```

```
     norfspace[y + f_width * x][0] = 0;
    }
    else
    {
     norfspace[y + f_width * x][0] = num_real/dem;
     norfspace[y + f_width * x][1] = num_img/dem;
    }
   }
  }
}
void revisedDFT(double ***img_space, double ***fspace, int height,
                int width, double n, vector<int> &x_points,
                vector<int> &y_points )
{
 double angle;
 double R,I,x,y,a,b;
 int x_point, y_point;
 for(int xi = 0; xi < height; xi++)
 {
  for(int yi = 0; yi < width; yi++)
  {
   for(int index = 0; index < x_points.size(); index++)
   {
    x = xi; y = yi;
    a = x_points[index];
    b = y_points[index];
    x_point = x_points[index];
    y_point = y_points[index];
    angle = 2*PI*((x*a/n) + (y*b/n));
```

```
      R = img_space[0][x_point][y_point];
      I = img_space[1][x_point][y_point];
      fspace[0][xi][yi] = fspace[0][xi][yi] + (R*cos(angle) + I*sin(angle));
      I = img_space[1][x_point][y_point];
      R = img_space[0][x_point][y_point];
      fspace[1][xi][yi] = fspace[1][xi][yi] + (I*cos(angle) - R*sin(angle));
     }
     fspace[0][xi][yi] = fspace[0][xi][yi]*(1/(n*n));
     fspace[1][xi][yi] = fspace[1][xi][yi]*(1/(n*n));
    }
   }
   x_points.clear();
   y_points.clear();
  }
  void PadImage(double ***padded, double ***image, int pad_height,
                int pad_width, int image_height, int image_width)
  {
   int pad_x = (pad_height/2) - (image_height/2);
   int pad_y = (pad_width/2) - (image_width/2);
   for(int i = 0; i < image_height; i++)
   {
    for(int j = 0; j < image_width; j++)
    {
     padded[0][pad_x][pad_y] = image[0][i][j];
     if(pad_y == (pad_width/2) - (image_width/2) + image_width - 1)
     {
      pad_x++;
      pad_y = (pad_width/2) - (image_width/2);
     }
    else
    {
     pad_y++;
    }
   }
  }
 }
void OverLapImage(double ***image, IplImage * source,
                  int startx, int starty, int image_height,
                  int image_width)
{
 CvScalar pixel;
 int tempx = startx;
 int tempy = starty;
 int c, c2;
 for(c = 0; c < image_height; c++)
 {
  for(c2 = 0; c2 < image_width; c2++)
  {
   if(image[0][c][c2] >= 100)
   {
    pixel.val[0] = 255;
    cvSet2D(source, tempx, tempy, pixel);
   }
   if(tempy == starty + image_width - 1)
   {
    tempx++;
    tempy = starty;
   }
```

```
    else
    {
     tempy++;
    }
   }
  }
 }
void ShiftImage(double ***source, double ***shifted,
                 int shift_x, int shift_y, int source_height,
                 int source_width, int &cx, int &cy)
{
 cx = source_height/2;
 cy = source_width/2;
 bool done = false;
 int sx_counter = 0;
 int sy_counter = 0;
 if(shift_x == 0 && shift_y == 0)
 {
  //do nothing
 }
 else if(shift_x > 0 && shift_y == 0)
 {
  sx_counter = shift_x;
  sy_counter = 0;
  for(int x = 0; x < source_height - shift_x; x++)
  {
   for(int y = 0; y < source_width; y++)
   {
    shifted[0][sx_counter][sy_counter] = source[0][x][y];
    if(x == cx && y == cy && done == false)
    {
     cx = sx_counter;
     cy = sy_counter;
     done = true;
    }
    if(sy_counter == source_width-1)
    {
     sx_counter++;
     sy_counter = 0;
    }
    else
    {
     sy_counter++;
    }
   }
  }
  sx_counter = 0;
  sy_counter = 0;
  for(int x = shift_x; x < source_height; x++)
  {
   for(int y = 0; y < source_width; y++)
   {
    shifted[0][sx_counter][sy_counter] = source[0][x][y];
    if(x == cx && y == cy && done == false)
    {
     cx = sx_counter;
     cy = sy_counter;
     done = true;
    }
    if(sy_counter == source_width-1)
    {
     sx_counter++;
     sy_counter = 0;
    }
    else
    {
     sy_counter++;
    }
   }
  }
 }
 else if(shift_x == 0 && shift_y > 0)
 {
  sx_counter = 0;
  sy_counter = shift_y;
  for(int x = 0; x < source_height; x++)
  {
   for(int y = 0; y < source_width - shift_y; y++)
   {
    shifted[0][sx_counter][sy_counter] = source[0][x][y];
    if(x == cx && y == cy && done == false)
    {
     cx = sx_counter;
     cy = sy_counter;
     done = true;
    }
    if(sy_counter == source_width-1)
   {
    sx_counter++;
    sy_counter = shift_y;
   }
   else
   {
    sy_counter++;
   }
  }
 }
 sx_counter = 0;
 sy_counter = 0;
 for(int x = shift_x; x < source_height; x++)
 {
  for(int y = 0; y < source_width - shift_y; y++)
  {
   shifted[0][sx_counter][sy_counter] = source[0][x][y];
   if(x == cx && y == cy && done == false)
   {
    cx = sx_counter;
    cy = sy_counter;
    done = true;
   }
   if(sy_counter == source_width -1)
   {
    sx_counter++;
    sy_counter = shift_y;
   }
   else
```

```
     {
      sy_counter++;
     }
    }
   }
  }
  else
  {
   //section a
   sx_counter = shift_x;
   sy_counter = shift_y;
   for(int x = 0; x < source_height - shift_x; x++)
   {
    for(int y = 0; y < source_width - shift_y; y++)
    {
     shifted[0][sx_counter][sy_counter] = source[0][x][y];
     if(x == cx && y == cy && done == false)
     {
      cx = sx_counter;
      cy = sy_counter;
      done = true;
     }
     if(sy_counter == source_width-1)
     {
      sx_counter++;
      sy_counter = shift_y;
     }
     else
     {
```

```
   sy_counter++;
  }
 }
}
//section b
sx_counter = shift_x;
sy_counter = 0;
for(int x = 0; x < source_height - shift_x; x++)
{
 for(int y = source_width - shift_y; y < source_width; y++)
 {
  shifted[0][sx_counter][sy_counter] = source[0][x][y];
  if(x == cx && y == cy && done == false)
  {
   cx = sx_counter;
   cy = sy_counter;
   done = true;
  }
  if(sy_counter == shift_y - 1)
  {
   sx_counter++;
   sy_counter = 0;
  }
  else
  {
   sy_counter++;
  }
 }
}
```

```
//section c
sx_counter = 0;
sy_counter = shift_y;
for(int x = source_height - shift_x; x < source_height; x++)
{
 for(int y = 0; y < source_width - shift_y; y++)
 {
  shifted[0][sx_counter][sy_counter] = source[0][x][y];
  if(x == cx && y == cy && done == false)
  {
   cx = sx_counter;
   cy = sy_counter;
   done = true;
  }
  if(sy_counter == source_width - 1)
  {
   sx_counter++;
   sy_counter = shift_y;
  }
  else
  {
   sy_counter++;
  }
 }
}
//section d
sx_counter = 0;
sy_counter = 0;
for(int x = source_height - shift_x; x < source_height; x++)
```

```
   {
    for(int y = source_width - shift_y; y < source_width; y++)
    {
     shifted[0][sx_counter][sy_counter] = source[0][x][y];
     if(x == cx && y == cy && done == false)
     {
      cx = sx_counter;
      cy = sy_counter;
      done = true;
     }
     if(sy_counter == shift_y - 1)
     {
      sx_counter++;
      sy_counter = 0;
     }
     else
     {
      sy_counter++;
     }
    }
   }
  }
 }

void MakeBorders(int &top, int &bottom, int &left,
                 int &right, int s_height, int s_width,
                 int p_height, int p_width, int centerx,
                 int centery)
{
 bottom = centerx + s_height/2;
 if(bottom > p_height - 1)
 {
  bottom = p_height - 1;
 }
 top = centerx - s_height/2;
 if(top < 0)
 {
  top = 0;
 }
 left = centery - s_width/2;
 if(left < 0)
 {
  left = 0;
 }
 right = centery + s_width/2;
 if(right > p_width - 1)
 {
  right = p_width - 1;
 }
}

void ImageToArray(IplImage * source, double ***fspace,
                  int startx, int starty, int i_height, int i_width)
{
 CvScalar pixel;
 int m = 0;
 int n = 0;
 for(int a = startx; a < startx + i_height; a++)
```

```
  {
   for(int b = starty; b < starty + i_width; b++)
   {
    pixel = cvGet2D(source, a, b);
    fspace[0][m][n] = pixel.val[0];
    fspace[1][m][n] = 0;
    if(n == i_width - 1)
    {
     m++;
     n = 0;
    }
    else
    {
     n++;
    }
   }
  }
 }
void GetMaxValue(double ***img_array, int &index_x,
                 int &index_y, int i_height, int i_width)
{
 double current_max = 0;
 double value = 0;
 for(int a = 0; a < i_height; a++)
 {
  for(int b = 0; b < i_width; b++)
  {
   value = img_array[0][a][b];
   if(value > current_max)
```

```
    {
     current_max = value;
     index_x = a;
     index_y = b;
    }
   }
  }
  cout << index_x << " " << index_y << endl;
 }
void TakeSnapShot(IplImage * image, CvArr * shot,
                  int startx, int starty, int shot_height,
                  int shot_width)
{
 CvScalar pixel_retrieved;
 int c1, c2;
 int tempx = startx;
 int tempy = starty;
 for(c1 = 0; c1 < shot_height; c1++)
 {
  for(c2 = 0; c2 < shot_width; c2++)
  {
   pixel_retrieved = cvGet2D(image, tempx, tempy);
   cvSet2D(shot, c1, c2, pixel_retrieved);
   if(tempy == starty + shot_width - 1)
   {
    tempx++;
    tempy = starty;
   }
   else
   {
    tempy++;
   }
  }
 }
}
void Normalize(double ***fspace1, double ***fspace2,
               double ***norfspace, int f_height, int f_width)
{
 double real_value, real_value2, img_value, img_value2;
 int x, y;
 double num_real, num_img;
 double dem;
 for(x = 0; x < f_height; x++)
 {
  for(y = 0; y < f_width; y++)
  {
   real_value = fspace1[0][x][y];
   real_value2 = fspace2[0][x][y];
   img_value = fspace1[1][x][y];
   img_value2 = fspace2[1][x][y];
   num_real = real_value*real_value2 + img_value*img_value2;
   num_img = real_value2*img_value - real_value*img_value2;
   dem = pow(real_value2,2) + pow(img_value2, 2);
   if(dem == 0)
   {
    norfspace[0][x][y] = 0;
    norfspace[0][x][y] = 0;
   }
```

```
    else
    {
     norfspace[0][x][y] = num_real/dem;
     norfspace[1][x][y] = num_img/dem;
    }
   }
  }
}
void FormatFilter(IplImage * image, int startx, int starty,
                  int filter_height, int filter_width)
{
 CvScalar pixel, pixelchange;
 for(int a = startx; a < startx + filter_height; a++)
 {
  for(int b = starty; b < starty + filter_width; b++)
  {
   pixel = cvGet2D(image, a, b);
   if(pixel.val[0] > 99)
   {
    pixelchange.val[0] = 100;
    cvSet2D(image, a, b, pixelchange);
   }
   else
   {
    pixelchange.val[0] = 0;
    cvSet2D(image, a, b, pixelchange);
   }
  }
 }
```

```
}
void EdgeWalker(IplImage * image, int startx, int starty,
                int mingroupsize, int maxgroupsize,
                int filter_height, int filter_width,
                                vector<int> &x_pts, vector<int> &y_pts,
                int &size)
{
  double **subframe;
  subframe = new double*[filter_height];
  for(int i = 0; i < filter_height; i++)
  {
   subframe[i] = new double[filter_width];
  }
  int borderx1 = 0;
  int borderx2 = 0 + filter_height - 1;
  int bordery1 = 0;
  int bordery2 = 0 + filter_width - 1;
  int icounter = 0;
  int jcounter = 0;
  int m = 0;
  int n = 0;
  int c,c2,c3,o,p;
  int tx, ty;
  int tn, tm;
  int tempx, tempy;
  int curposx = 0;
  int curposy = 0;
  int finalcounter = 0;
  int walk_to;
int backtrack_pixel;
double n0, n1, n2, n3, n4, n5, n6, n7;
CvScalar pixel, pixelchange;
vector<double> neighbors; //all neighbors and pixel values
vector<int> w_neighbors; // all white pixel neighbors
vector<int> neighbor_id;
vector<int> x_coords;
vector<int> y_coords;
vector<int> x_backtrack;
vector<int> y_backtrack;
vector<int> x_final;
vector<int> y_final;
bool edged_walked = false;
//copy subframe pixels into array
for(int a = startx; a < startx + filter_height; a++)
 {
  for(int b = starty; b < starty + filter_width; b++)
  {
   pixel = cvGet2D(image, a, b);
   subframe[m][n] = pixel.val[0];
   if(n == filter_width - 1)
   {
    m++;
    n = 0;
   }
   else
   {
    n++;
   }
   }
  }
m = 0;
n = 0;
for( o = 0; o < filter_height; o++)
{
 for(p = 0; p < filter_width; p++)
 {
  if(subframe[o][p] == 100) //not a black pixel
  {
   m = o;
   n = p;
   while(edged_walked != true)
   {
    if(m != borderx1 && m != borderx2
       && n != bordery1 && n != bordery2 ) //inner pixels
    {
     //calculate neigbor pixel values
     n0 = subframe[m-1][n-1];
     n1 = subframe[m-1][n];
     n2 = subframe[m-1][n+1];
     n3 = subframe[m][n-1];
     n4 = subframe[m][n+1];
     n5 = subframe[m+1][n-1];
     n6 = subframe[m+1][n];
     n7 = subframe[m+1][n+1];
     neighbors.push_back(n0);
     neighbors.push_back(n1);
     neighbors.push_back(n2);
 neighbors.push_back(n3);
 neighbors.push_back(n4);
 neighbors.push_back(n5);
 neighbors.push_back(n6);
 neighbors.push_back(n7);
 neighbor_id.push_back(0);
 neighbor_id.push_back(1);
 neighbor_id.push_back(2);
 neighbor_id.push_back(3);
 neighbor_id.push_back(4);
 neighbor_id.push_back(5);
 neighbor_id.push_back(6);
 neighbor_id.push_back(7);
}
else if (m != borderx1 && m != borderx2
         && n == bordery1) //vertical left side pixels
{
 n1 = subframe[m-1][n];
 n2 = subframe[m-1][n+1];
 n4 = subframe[m][n+1];
 n6 = subframe[m+1][n];
 n7 = subframe[m+1][n+1];
 neighbors.push_back(n1);
 neighbors.push_back(n2);
 neighbors.push_back(n4);
 neighbors.push_back(n6);
 neighbors.push_back(n7);
 neighbor_id.push_back(1);
 neighbor_id.push_back(2);
 neighbor_id.push_back(4);
 neighbor_id.push_back(6);
 neighbor_id.push_back(7);
}
else if (m == borderx2 && n == bordery1) // bottom left pixel
{
 n1 = subframe[m-1][n];
 n2 = subframe[m-1][n+1];
 n4 = subframe[m][n+1];
 neighbors.push_back(n1);
 neighbors.push_back(n2);
 neighbors.push_back(n4);
 neighbor_id.push_back(1);
 neighbor_id.push_back(2);
 neighbor_id.push_back(4);
}
else if (m == borderx2 && n != bordery1
         && n != bordery2) //horizontal bottom side pixels
{
 n0 = subframe[m-1][n-1];
 n1 = subframe[m-1][n];
 n2 = subframe[m-1][n+1];
 n3 = subframe[m][n-1];
 n4 = subframe[m][n+1];
 neighbors.push_back(n0);
 neighbors.push_back(n1);
 neighbors.push_back(n2);
 neighbors.push_back(n3);
 neighbors.push_back(n4);
 neighbor_id.push_back(0);
 neighbor_id.push_back(1);
 neighbor_id.push_back(2);
 neighbor_id.push_back(3);
 neighbor_id.push_back(4);
}
else if (m == borderx2 && n == bordery2) // bottom right corner pixel
{
 n0 = subframe[m-1][n-1];
 n1 = subframe[m-1][n];
 n3 = subframe[m][n-1];
 neighbors.push_back(n0);
 neighbors.push_back(n1);
 neighbors.push_back(n3);
 neighbor_id.push_back(0);
 neighbor_id.push_back(1);
 neighbor_id.push_back(3);
}
else if (m != borderx1 && m != borderx2
         && n == bordery2) // vertical right side pixels
{
 n0 = subframe[m-1][n-1];
 n1 = subframe[m-1][n];
 n3 = subframe[m][n-1];
 n5 = subframe[m+1][n-1];
 n6 = subframe[m+1][n];
 neighbors.push_back(n0);
 neighbors.push_back(n1);
 neighbors.push_back(n3);
 neighbors.push_back(n5);
 neighbors.push_back(n6);
 neighbor_id.push_back(0);
 neighbor_id.push_back(1);
 neighbor_id.push_back(3);
 neighbor_id.push_back(5);
 neighbor_id.push_back(6);
}
else if (m == borderx1 && n == bordery2) // top right pixel
{
 n3 = subframe[m][n-1];
 n5 = subframe[m+1][n-1];
 n6 = subframe[m+1][n];
 neighbors.push_back(n3);
 neighbors.push_back(n5);
 neighbors.push_back(n6);
 neighbor_id.push_back(3);
 neighbor_id.push_back(5);
 neighbor_id.push_back(6);
}
else if (m == borderx1 && n != bordery1
        && n != bordery2) //horizontal top side pixels
{
 n3 = subframe[m][n-1];
 n4 = subframe[m][n+1];
 n5 = subframe[m+1][n-1];
 n6 = subframe[m+1][n];
 n7 = subframe[m+1][n+1];
 neighbors.push_back(n3);
 neighbors.push_back(n4);
 neighbors.push_back(n5);
 neighbors.push_back(n6);
 neighbors.push_back(n7);
 neighbor_id.push_back(3);
 neighbor_id.push_back(4);
 neighbor_id.push_back(5);
 neighbor_id.push_back(6);
 neighbor_id.push_back(7);
}
else if ( m == borderx1 && n == bordery1) //top left pixel
{
 n4 = subframe[m][n+1];
 n6 = subframe[m+1][n];
 n7 = subframe[m+1][n+1];
 neighbors.push_back(n4);
 neighbors.push_back(n6);
 neighbors.push_back(n7);
 neighbor_id.push_back(4);
 neighbor_id.push_back(6);
 neighbor_id.push_back(7);
}
//find all pixel neighbors that are white
for(c = 0; c < neighbors.size(); c++)
{
 if(neighbors[c] == 100)
 {
  w_neighbors.push_back(neighbor_id[c]);
 }
```

```
}
if(w_neighbors.empty()) //if no connecting pixel
{
 subframe[m][n] = 0; //black out pixel
 x_coords.push_back(m);
 y_coords.push_back(n);
 if(x_backtrack.empty())
 {
  edged_walked = true;
  //check to see if number of points in group meets min.
  if(x_coords.size() >= mingroupsize
     && x_coords.size() <= maxgroupsize)
  {
   for(int y = 0; y < x_coords.size(); y++)
   {
    x_final.push_back(x_coords[y]);
    y_final.push_back(y_coords[y]);
    x_pts.push_back(x_coords[y]);
    y_pts.push_back(y_coords[y]);
   }
  }
  x_coords.clear();
  y_coords.clear();
 }
 else // backtrack not empty
 {
  m = x_backtrack.back();
  n = y_backtrack.back();
  x_backtrack.pop_back();
  y_backtrack.pop_back();
 }
}
else // w_neighbors not empty, connecting pixel
{
 subframe[m][n] = 0;
 x_coords.push_back(m);
 y_coords.push_back(n);
 walk_to = w_neighbors.back();
 w_neighbors.pop_back();
 tm = m;
 tn = n;
 if(walk_to == 7)
 {
  m = m+1;
  n = n+1;
 }
 else if(walk_to == 6)
 {
  m = m+1;
  n = n;
 }
 else if(walk_to == 5)
 {
  m = m+1;
  n = n-1;
 }
 else if(walk_to == 4)
 {
```

```
 m = m;
 n = n+1;
}
else if(walk_to == 3)
{
 m = m;
 n = n-1;
}
else if(walk_to == 2)
{
 m = m-1;
 n = n+1;
}
else if(walk_to == 1)
{
 m = m-1;
 n = n;
}
else if(walk_to == 0)
{
 m = m-1;
 n = n-1;
}
for(int h = 0; h < w_neighbors.size(); h++)
{
 backtrack_pixel = w_neighbors[h];
 if(backtrack_pixel == 7)
 {
  x_backtrack.push_back(tm+1);
 y_backtrack.push_back(tn+1);
 subframe[tm+1][tn+1] = 0;
}
else if(backtrack_pixel == 6)
{
 x_backtrack.push_back(tm+1);
 y_backtrack.push_back(tn);
 subframe[tm+1][tn] = 0;
}
else if(backtrack_pixel == 5)
{
 x_backtrack.push_back(tm+1);
 y_backtrack.push_back(tn-1);
 subframe[tm+1][tn-1] = 0;
}
else if(backtrack_pixel == 4)
{
 x_backtrack.push_back(tm);
 y_backtrack.push_back(tn+1);
 subframe[tm][tn+1] = 0;
}
else if(backtrack_pixel == 3)
{
 x_backtrack.push_back(tm);
 y_backtrack.push_back(tn-1);
 subframe[tm][tn-1] = 0;
}
else if(backtrack_pixel == 2)
{
        x_backtrack.push_back(tm-1);
        y_backtrack.push_back(tn+1);
        subframe[tm-1][tn+1] = 0;
       }
       else if(backtrack_pixel == 1)
       {
        x_backtrack.push_back(tm-1);
        y_backtrack.push_back(tn);
        subframe[tm-1][tn] = 0;
       }
       else if(backtrack_pixel == 0)
       {
        x_backtrack.push_back(tm-1);
        y_backtrack.push_back(tn-1);
        subframe[tm-1][tn-1] = 0;
       }
      }//for loop
     }
     w_neighbors.clear();
     neighbors.clear();
     neighbor_id.clear();
    }//while loop
    x_backtrack.clear();
    y_backtrack.clear();
   }//if statement, white pixel
   edged_walked = false;
  }//inner for loop
 } //outer for loop
 size = x_final.size();
```

```
while(x_final.size() != 0)
{
 tx = x_final.back();
 ty = y_final.back();
 subframe[tx][ty] = 100;
 x_final.pop_back();
 y_final.pop_back();
}
x_final.clear();
y_final.clear();
tempx = startx;
tempy = starty;
for(c2 = 0; c2 < filter_height; c2++)
{
 for(c3 = 0; c3 < filter_width; c3++)
 {
  pixelchange.val[0] = subframe[c2][c3];
  cvSet2D(image, tempx, tempy, pixelchange);
  if(tempy == starty + filter_width - 1)
  {
   tempx++;
   tempy = starty;
  }
  else
  {
   tempy++;
  }
 }
}
```

```
  for(int i = 0; i < filter_height; i++)
  {
   delete[] subframe[i];
  }
  delete[] subframe;
}
void FindLargestEdge(IplImage * image, int startx, int starty,
                     int filter_height, int filter_width, int min,
                     int max, int &size)
{
  double **subframe;
  subframe = new double*[filter_height];
  for(int i = 0; i < filter_height; i++)
  {
   subframe[i] = new double[filter_width];
  }
  int borderx1 = 0;
  int borderx2 = 0 + filter_height - 1;
  int bordery1 = 0;
  int bordery2 = 0 + filter_width - 1;
  int icounter = 0;
  int jcounter = 0;
  int m = 0;
  int n = 0;
  int c,c2,c3,o,p;
  int tx, ty;
  int tn, tm;
  int tempx, tempy;
  int curposx = 0;
int curposy = 0;
int finalcounter = 0;
int walk_to;
int backtrack_pixel;
double n0, n1, n2, n3, n4, n5, n6, n7;
CvScalar pixel , pixelchange;
vector<double> neighbors; //all neighbors and pixel values
vector<int> w_neighbors; // all white pixel neighbors
vector<int> neighbor_id;
vector<int> x_coords;
vector<int> y_coords;
vector<int> x_backtrack;
vector<int> y_backtrack;
vector<int> x_final;
vector<int> y_final;
bool edged_walked = false;
//copy subframe pixels into array
for(int a = startx; a < startx + filter_height; a++)
 {
  for(int b = starty; b < starty + filter_width; b++)
  {
   pixel = cvGet2D(image, a, b);
   subframe[m][n] = pixel.val[0];
   if(n == filter_width - 1)
   {
    m++;
    n = 0;
   }
   else
```

```
      {
       n++;
      }
     }
    }
   m = 0;
   n = 0;
   for( o = 0; o < filter_height; o++)
   {
    for(p = 0; p < filter_width; p++)
    {
     if(subframe[o][p] == 100) //not a black pixel
     {
      m = o;
      n = p;
      while(edged_walked != true)
      {
       if( m != borderx1 && m != borderx2
          && n != bordery1 && n != bordery2 ) //inner pixels
       {
        //calculate neigbor pixel values
        n0 = subframe[m-1][n-1];
        n1 = subframe[m-1][n];
        n2 = subframe[m-1][n+1];
        n3 = subframe[m][n-1];
        n4 = subframe[m][n+1];
        n5 = subframe[m+1][n-1];
        n6 = subframe[m+1][n];
        n7 = subframe[m+1][n+1];
```

```
 neighbors.push_back(n0);
 neighbors.push_back(n1);
 neighbors.push_back(n2);
 neighbors.push_back(n3);
 neighbors.push_back(n4);
 neighbors.push_back(n5);
 neighbors.push_back(n6);
 neighbors.push_back(n7);
 neighbor_id.push_back(0);
 neighbor_id.push_back(1);
 neighbor_id.push_back(2);
 neighbor_id.push_back(3);
 neighbor_id.push_back(4);
 neighbor_id.push_back(5);
 neighbor_id.push_back(6);
 neighbor_id.push_back(7);
}
else if (m != borderx1 && m != borderx2
         && n == bordery1) //vertical left side pixels
{
 n1 = subframe[m-1][n];
 n2 = subframe[m-1][n+1];
 n4 = subframe[m][n+1];
 n6 = subframe[m+1][n];
 n7 = subframe[m+1][n+1];
 neighbors.push_back(n1);
 neighbors.push_back(n2);
 neighbors.push_back(n4);
 neighbors.push_back(n6);
```

```
  neighbors.push_back(n7);
  neighbor_id.push_back(1);
  neighbor_id.push_back(2);
  neighbor_id.push_back(4);
  neighbor_id.push_back(6);
  neighbor_id.push_back(7);
}
else if (m == borderx2 && n == bordery1) // bottom left pixel
{
  n1 = subframe[m-1][n];
  n2 = subframe[m-1][n+1];
  n4 = subframe[m][n+1];
  neighbors.push_back(n1);
  neighbors.push_back(n2);
  neighbors.push_back(n4);
  neighbor_id.push_back(1);
  neighbor_id.push_back(2);
  neighbor_id.push_back(4);
}
else if (m == borderx2 && n != bordery1
         && n != bordery2) //horizontal bottom side pixels
{
  n0 = subframe[m-1][n-1];
  n1 = subframe[m-1][n];
  n2 = subframe[m-1][n+1];
  n3 = subframe[m][n-1];
  n4 = subframe[m][n+1];
  neighbors.push_back(n0);
  neighbors.push_back(n1);
```

```
 neighbors.push_back(n2);
 neighbors.push_back(n3);
 neighbors.push_back(n4);
 neighbor_id.push_back(0);
 neighbor_id.push_back(1);
 neighbor_id.push_back(2);
 neighbor_id.push_back(3);
 neighbor_id.push_back(4);
}
else if (m == borderx2 && n == bordery2) // bottom right corner pixel
{
 n0 = subframe[m-1][n-1];
 n1 = subframe[m-1][n];
 n3 = subframe[m][n-1];
 neighbors.push_back(n0);
 neighbors.push_back(n1);
 neighbors.push_back(n3);
 neighbor_id.push_back(0);
 neighbor_id.push_back(1);
 neighbor_id.push_back(3);
}
else if (m != borderx1 && m != borderx2
         && n == bordery2) // vertical right side pixels
{
 n0 = subframe[m-1][n-1];
 n1 = subframe[m-1][n];
 n3 = subframe[m][n-1];
 n5 = subframe[m+1][n-1];
 n6 = subframe[m+1][n];
```

```
  neighbors.push_back(n0);
  neighbors.push_back(n1);
  neighbors.push_back(n3);
  neighbors.push_back(n5);
  neighbors.push_back(n6);
  neighbor_id.push_back(0);
  neighbor_id.push_back(1);
  neighbor_id.push_back(3);
  neighbor_id.push_back(5);
  neighbor_id.push_back(6);
}
else if (m == borderx1 && n == bordery2) // top right pixel
{
  n3 = subframe[m][n-1];
  n5 = subframe[m+1][n-1];
  n6 = subframe[m+1][n];
  neighbors.push_back(n3);
  neighbors.push_back(n5);
  neighbors.push_back(n6);
  neighbor_id.push_back(3);
  neighbor_id.push_back(5);
  neighbor_id.push_back(6);
}
else if (m == borderx1 && n != bordery1
        && n != bordery2) //horizontal top side pixels
{
  n3 = subframe[m][n-1];
  n4 = subframe[m][n+1];
  n5 = subframe[m+1][n-1];
```

```
 n6 = subframe[m+1][n];
 n7 = subframe[m+1][n+1];
 neighbors.push_back(n3);
 neighbors.push_back(n4);
 neighbors.push_back(n5);
 neighbors.push_back(n6);
 neighbors.push_back(n7);
 neighbor_id.push_back(3);
 neighbor_id.push_back(4);
 neighbor_id.push_back(5);
 neighbor_id.push_back(6);
 neighbor_id.push_back(7);
}
else if ( m == borderx1 && n == bordery1) //top left pixel
{
 n4 = subframe[m][n+1];
 n6 = subframe[m+1][n];
 n7 = subframe[m+1][n+1];
 neighbors.push_back(n4);
 neighbors.push_back(n6);
 neighbors.push_back(n7);
 neighbor_id.push_back(4);
 neighbor_id.push_back(6);
 neighbor_id.push_back(7);
}
//find all pixel neighbors that are white
for(c = 0; c < neighbors.size(); c++)
{
 if(neighbors[c] == 100)
 {
  w_neighbors.push_back(neighbor_id[c]);
 }
}
if(w_neighbors.empty()) //if no connecting pixel
{
 subframe[m][n] = 0; //black out pixel
 x_coords.push_back(m);
 y_coords.push_back(n);
 if(x_backtrack.empty())
 {
  edged_walked = true;
  if(x_coords.size() >= min && x_coords.size() <= max)
  {
   if(x_coords.size() > x_final.size())
   {
    x_final.clear();
    y_final.clear();
    for(int y = 0; y < x_coords.size(); y++)
    {
     x_final.push_back(x_coords[y]);
     y_final.push_back(y_coords[y]);
    }
   }
  }
  x_coords.clear();
  y_coords.clear();
 }
 else // backtrack not empty
 {
  m = x_backtrack.back();
  n = y_backtrack.back();
  x_backtrack.pop_back();
  y_backtrack.pop_back();
 }
}
else // w_neighbors not empty, connecting pixel
{
 subframe[m][n] = 0;
 x_coords.push_back(m);
 y_coords.push_back(n);
 walk_to = w_neighbors.back();
 w_neighbors.pop_back();
 tm = m;
 tn = n;
 if(walk_to == 7)
 {
  m = m+1;
  n = n+1;
 }
 else if(walk_to == 6)
 {
  m = m+1;
  n = n;
 }
 else if(walk_to == 5)
 {
  m = m+1;
 n = n-1;
}
else if(walk_to == 4)
{
 m = m;
 n = n+1;
}
else if(walk_to == 3)
{
 m = m;
 n = n-1;
}
else if(walk_to == 2)
{
 m = m-1;
 n = n+1;
}
else if(walk_to == 1)
{
 m = m-1;
 n = n;
}
else if(walk_to == 0)
{
 m = m-1;
 n = n-1;
}
for(int h = 0; h < w_neighbors.size(); h++)
{
```

```
backtrack_pixel = w_neighbors[h];
if(backtrack_pixel == 7)
{
 x_backtrack.push_back(tm+1);
 y_backtrack.push_back(tn+1);
 subframe[tm+1][tn+1] = 0;
}
else if(backtrack_pixel == 6)
{
 x_backtrack.push_back(tm+1);
 y_backtrack.push_back(tn);
 subframe[tm+1][tn] = 0;
}
else if(backtrack_pixel == 5)
{
 x_backtrack.push_back(tm+1);
 y_backtrack.push_back(tn-1);
 subframe[tm+1][tn-1] = 0;
}
else if(backtrack_pixel == 4)
{
 x_backtrack.push_back(tm);
 y_backtrack.push_back(tn+1);
 subframe[tm][tn+1] = 0;
}
else if(backtrack_pixel == 3)
{
 x_backtrack.push_back(tm);
 y_backtrack.push_back(tn-1);
```

```
        subframe[tm][tn-1] = 0;
       }
       else if(backtrack_pixel == 2)
       {
        x_backtrack.push_back(tm-1);
        y_backtrack.push_back(tn+1);
        subframe[tm-1][tn+1] = 0;
       }
       else if(backtrack_pixel == 1)
       {
        x_backtrack.push_back(tm-1);
        y_backtrack.push_back(tn);
        subframe[tm-1][tn] = 0;
       }
       else if(backtrack_pixel == 0)
       {
        x_backtrack.push_back(tm-1);
        y_backtrack.push_back(tn-1);
        subframe[tm-1][tn-1] = 0;
       }
      }//for loop
     }
     w_neighbors.clear();
     neighbors.clear();
     neighbor_id.clear();
    }//while loop
    x_backtrack.clear();
    y_backtrack.clear();
   }//if statement, white pixel
```

```
  edged_walked = false;
 }//inner for loop
} //outer for loop
size = x_final.size();
while(x_final.size() != 0)
{
 tx = x_final.back();
 ty = y_final.back();
 subframe[tx][ty] = 100;
 x_final.pop_back();
 y_final.pop_back();
}
x_final.clear();
y_final.clear();
tempx = startx;
tempy = starty;
for(c2 = 0; c2 < filter_height; c2++)
{
 for(c3 = 0; c3 < filter_width; c3++)
 {
  pixelchange.val[0] = subframe[c2][c3];
  cvSet2D(image, tempx, tempy, pixelchange);
  if(tempy == starty + filter_width - 1)
  {
   tempx++;
   tempy = starty;
  }
  else
  {
      tempy++;
     }
    }
   }
   for(int i = 0; i < filter_height; i++)
   {
    delete[] subframe[i];
   }
   delete[] subframe;
}
void EndpointVerifier(double **sframe, int walkable_size,
                      int nonwalkable_size, vector<int> &endpx,
                      vector<int> &endpy, int x, int y,
                      int f_height, int f_width)
{
 if(nonwalkable_size + walkable_size > 1)
 {
  if(x != f_height-1 && x != 0 && y != f_width-1 && y != 0)
  {
   if(((sframe[x][y-1] != 105 || sframe[x][y-1] != 100) &&
       (sframe[x][y+1] != 105 || sframe[x][y+1] != 100)) && //3,4
      ((sframe[x-1][y] != 105 || sframe[x-1][y] != 100) &&
       (sframe[x+1][y] != 105 || sframe[x+1][y] != 100)) && //1,6
      ((sframe[x-1][y+1] != 105 || sframe[x-1][y+1] != 100) &&
       (sframe[x+1][y-1] != 105 || sframe[x+1][y-1] != 100)) && //2,5
      ((sframe[x-1][y-1] != 105 || sframe[x-1][y-1] != 100) &&
       (sframe[x+1][y+1] != 105 || sframe[x+1][y+1] != 100)) && //0,7
      ((sframe[x-1][y] != 105 || sframe[x-1][y] != 100) &&
       (sframe[x][y-1] != 105 || sframe[x][y-1] != 100)) && //1,3
```

```
((sframe[x-1][y] != 105 || sframe[x-1][y] != 100) &&
 (sframe[x][y+1] != 105 || sframe[x][y+1] != 100)) && //1,4
((sframe[x][y-1] != 105 || sframe[x][y-1] != 100) &&
 (sframe[x+1][y] != 105 || sframe[x+1][y] != 100)) && //3,6
((sframe[x-1][y-1] != 105 || sframe[x-1][y-1] != 100) &&
 (sframe[x-1][y+1] != 105 || sframe[x-1][y+1] != 100)) && //0,2
((sframe[x+1][y-1] != 105 || sframe[x+1][y-1] != 100) &&
 (sframe[x+1][y+1] != 105 || sframe[x+1][y+1] != 100)) && //5,7
((sframe[x-1][y-1] != 105 || sframe[x-1][y-1] != 100) &&
 (sframe[x+1][y-1] != 105 || sframe[x+1][y-1] != 100)) && //0,5
((sframe[x-1][y+1] != 105 || sframe[x-1][y+1] != 100) &&
 (sframe[x+1][y+1] != 105 || sframe[x+1][y+1] != 100)) && //7,2
((sframe[x-1][y-1] != 105 || sframe[x-1][y-1] != 100) &&
 (sframe[x+1][y] != 105 || sframe[x+1][y] != 100)) && //0,6
((sframe[x-1][y+1] != 105 || sframe[x-1][y+1] != 100) &&
 (sframe[x+1][y] != 105 || sframe[x+1][y] != 100)) && //2,6
((sframe[x-1][y] != 105 || sframe[x-1][y] != 100) &&
 (sframe[x+1][y-1] != 105 || sframe[x+1][y-1] != 100)) && //1,5
((sframe[x-1][y] != 105 || sframe[x-1][y] != 100) &&
 (sframe[x+1][y+1] != 105 || sframe[x+1][y+1] != 100)) && //1,7
((sframe[x][y-1] != 105 || sframe[x][y-1] != 100) &&
 (sframe[x-1][y+1] != 105 || sframe[x-1][y+1] != 100)) && //3,2
((sframe[x][y-1] != 105 || sframe[x][y-1] != 100) &&
 (sframe[x+1][y+1] != 105 || sframe[x+1][y+1] != 100)) && //3,7
((sframe[x][y+1] != 105 || sframe[x][y+1] != 100) &&
 (sframe[x-1][y-1] != 105 || sframe[x-1][y-1] != 100)) && //4,0
((sframe[x][y+1] != 105 || sframe[x][y+1] != 100) &&
 (sframe[x+1][y-1] != 105 || sframe[x+1][y-1] != 100)) && //4,5
((sframe[x][y+1] != 105 || sframe[x][y+1] != 100) &&
      (sframe[x+1][y] != 105 || sframe[x+1][y] != 100)) //4,6
     )
    {
     endpx.push_back(x);
     endpy.push_back(y);
    }
   }
  }
  else //1 pixel neighbor or zero
  {
   if(x != f_height-1 && x != 0 && y != f_width-1 && y != 0)
   {
    endpx.push_back(x);
    endpy.push_back(y);
   }
  }
}
bool RightScannerForward(double **sframe, int sx_c, int sy_c,
                         int levels_of_scan, int f_height, int f_width)
{
 int swidth_c;
 int act_width;
 int found_x, found_y;
 bool status = false;
 bool found_conn0 = false;
 bool found_conn = false;
 bool found_conn2 = false;
 sy_c = sy_c+1;
 //Check to make sure in bounds
```

```
if(sy_c > f_width - 2)
{
 return status;
}
//Check to see if already has pixel connected
if(sframe[sx_c][sy_c] == 105)
{
 status = true;
 return status;
}
swidth_c = sy_c + 3;
act_width = 4;
if(swidth_c > f_width - 1)
{
 act_width = 4 - (swidth_c - (f_width - 1));
 swidth_c = f_width - 1;
 if(act_width == 1)
 {
  return status;
 }
}
if(levels_of_scan >= 1)
{
 if(act_width >= 2)
 {
  if(sframe[sx_c][sy_c+1] == 105 || sframe[sx_c][sy_c+1] == 100)
  {
   found_x = sx_c;
   found_y = sy_c+1;
```

```
    found_conn0 = true;
    status = true;
   }
   if(found_conn0 == true)
   {
    sframe[sx_c][sy_c] = 105;
    return status;
   }
  }
 }
 if(levels_of_scan >= 2)
 {
  if(act_width >= 3)
  {
   if(sframe[sx_c][sy_c+2] == 105 || sframe[sx_c][sy_c+2] == 100)
   {
    found_x = sx_c;
    found_y = sy_c+2;
    found_conn = true;
    status = true;
   }
   if(found_conn == true)
   {
    sframe[sx_c][sy_c] = 105;
    sframe[sx_c][sy_c+1] = 105;
    return status;
   }
  }
 }
 if(levels_of_scan == 3)
 {
  if(act_width == 4)
  {
   if(sframe[sx_c][sy_c+3] == 105 || sframe[sx_c][sy_c+3] == 100)
   {
    found_x = sx_c;
    found_y = sy_c+3;
    found_conn2 = true;
    status = true;
   }
   if(found_conn2 == true)
   {
    sframe[sx_c][sy_c] = 105;
    sframe[sx_c][sy_c+1] = 105;
    sframe[sx_c][sy_c+2] = 105;
    return status;
   }
  }
 }
 return status;
}
bool UpScanner(double **sframe, int sx_c, int sy_c,
               int levels_of_scan, int f_height, int f_width)
{
 int sheight_c;
 int act_height;
 int found_x, found_y;
 bool status = false;
bool found_conn0 = false;
bool found_conn = false;
bool found_conn2 = false;
sx_c = sx_c -1;
//Check to make sure in bounds
if(sx_c < 1)
{
 return status;
}
//Check to see if already has pixel connected
if(sframe[sx_c][sy_c] == 105)
{
 status = true;
 return status;
}
sheight_c = sx_c - 3;
act_height = 4;
if(sheight_c < 0)
{
 act_height = 4 + sheight_c;
 sheight_c = 0;

 if(act_height == 1)
 {
  return status;
 }
}
if(levels_of_scan >= 1)
{
  if(act_height >= 2)
  {
   if(sframe[sx_c -1][sy_c] == 105 || sframe[sx_c -1][sy_c] == 100)
   {
    found_x = sx_c -1;
    found_y = sy_c;
    found_conn0 = true;
    status = true;
   }
   if(found_conn0 == true)
   {
    sframe[sx_c][sy_c] = 105;
    return status;
   }
  }
 }
 if(levels_of_scan >= 2)
 {
  if(act_height >= 3)
  {
   if(sframe[sx_c -2][sy_c] == 105 || sframe[sx_c -2][sy_c] == 100)
   {
    found_x = sx_c -2;
    found_y = sy_c;
    found_conn = true;
    status = true;
   }
   if(found_conn == true)
   {
      sframe[sx_c][sy_c] = 105;
      sframe[sx_c -1][sy_c] = 105;
      return status;
     }
    }
   }
   if(levels_of_scan == 3)
   {
    if(act_height == 4)
    {
     if(sframe[sx_c -3][sy_c] == 105 || sframe[sx_c -3][sy_c] == 100)
     {
      found_x = sx_c -3;
      found_y = sy_c;
      found_conn2 = true;
      status = true;
     }
     if(found_conn2 == true)
     {
      sframe[sx_c][sy_c] = 105;
      sframe[sx_c -1][sy_c] = 105;
      sframe[sx_c -2][sy_c] = 105;
      return status;
     }
    }
   }
   return status;
  }
  bool DownScanner(double **sframe, int sx_c, int sy_c,
                   int levels_of_scan, int f_height, int f_width)
{
 int sheight_c;
 int act_height;
 int found_x, found_y;
 bool status = false;
 bool found_conn0 = false;
 bool found_conn = false;
 bool found_conn2 = false;
 sx_c = sx_c+1;
 //Check to make sure in bounds
 if(sx_c > f_height - 2)
 {
  return status;
 }
 //Check to see if already has pixel connected
 if(sframe[sx_c][sy_c] == 105)
 {
  status = true;
  return status;
 }
 sheight_c = sx_c + 3;
 act_height = 4;
 if(sheight_c > f_height - 1)
 {
  act_height = 4 - (sheight_c - (f_height - 1));
  sheight_c = f_height - 1;
  if(act_height == 1)
  {
   return status;
  }
 }
 if(levels_of_scan >= 1)
 {
  if(act_height >= 2)
  {
   if(sframe[sx_c+1][sy_c] == 105 || sframe[sx_c+1][sy_c] == 100)
   {
    found_x = sx_c+1;
    found_y = sy_c;
    found_conn0 = true;
    status = true;
   }
   if(found_conn0 == true)
   {
    sframe[sx_c][sy_c] = 105;
    return status;
   }
  }
 }
 if(levels_of_scan >= 2)
 {
  if(act_height >= 3)
  {
   if(sframe[sx_c+2][sy_c] == 105 || sframe[sx_c+2][sy_c] == 100)
   {
    found_x = sx_c+2;
    found_y = sy_c;
```

```
     found_conn = true;
     status = true;
    }
    if(found_conn == true)
    {
     sframe[sx_c][sy_c] = 105;
     sframe[sx_c+1][sy_c] = 105;
     return status;
    }
   }
  }
  if(levels_of_scan == 3)
  {
   if(act_height == 4)
   {
    if(sframe[sx_c+3][sy_c] == 105 || sframe[sx_c+3][sy_c] == 100)
    {
     found_x = sx_c+3;
     found_y = sy_c;
     found_conn2 = true;
     status = true;
    }
    if(found_conn2 == true)
    {
     sframe[sx_c][sy_c] = 105;
     sframe[sx_c+1][sy_c] = 105;
     sframe[sx_c+2][sy_c] = 105;
     return status;
    }
  }
 }
 return status;
}
bool LeftScannerForward(double **sframe, int sx_c, int sy_c,
                        int levels_of_scan, int f_height, int f_width)
{
 int swidth_c;
 int act_width;
 int found_x, found_y;
 bool status = false;
 bool found_conn0 = false;
 bool found_conn = false;
 bool found_conn2 = false;
 sy_c = sy_c -1;
 //Check to make sure in bounds
 if(sy_c < 1)
 {
  return status;
 }
 //Check to see if already has pixel connected
 if(sframe[sx_c][sy_c] == 105)
 {
  status = true;
  return status;
 }
 swidth_c = sy_c - 3;
 act_width = 4;
 if(swidth_c < 0)
```

```
{
 act_width = 4 + swidth_c;
 swidth_c = 0;

 if(act_width == 1)
 {
  return status;
 }
}
if(levels_of_scan >= 1)
{
 if(act_width >= 2)
 {
  if(sframe[sx_c][sy_c-1] == 105 || sframe[sx_c][sy_c-1] == 100)
  {
   found_x = sx_c;
   found_y = sy_c-1;
   found_conn0 = true;
   status = true;
  }
  if(found_conn0 == true)
  {
   sframe[sx_c][sy_c] = 105;
   return status;
  }
 }
}
if(levels_of_scan >= 2)
{
 if(act_width >= 3)
 {
  if(sframe[sx_c][sy_c-2] == 105 || sframe[sx_c][sy_c-2] == 100)
  {
   found_x = sx_c;
   found_y = sy_c-2;
   found_conn = true;
   status = true;
  }
  if(found_conn == true)
  {
   sframe[sx_c][sy_c] = 105;
   sframe[sx_c][sy_c-1] = 105;
   return status;
  }
 }
}
if(levels_of_scan == 3)
{
 if(act_width == 4)
 {
  if(sframe[sx_c][sy_c-3] == 105 || sframe[sx_c][sy_c-3] == 100)
  {
   found_x = sx_c;
   found_y = sy_c-3;
   found_conn2 = true;
   status = true;
  }
  if(found_conn2 == true)
```

```
    {
     sframe[sx_c][sy_c] = 105;
     sframe[sx_c][sy_c-1] = 105;
     sframe[sx_c][sy_c-2] = 105;
     return status;
    }
   }
  }
  return status;
 }
 bool RightScannerUp(double **sframe, int sx_c, int sy_c,
                     int levels_of_scan, int f_height, int f_width)
 {
  int sheight_c, swidth_c;
  int act_width, act_height;
  int found_x, found_y;
  int range;
  bool status = false;
  bool found_conn0 = false;
  bool found_conn = false;
  bool found_conn2 = false;
  bool level_zero_ver = true;
  bool level_zero_hor = true;
  bool level_one_ver = true;
  bool level_one_hor = true;
  bool level_two_ver = true;
  bool level_two_hor = true;
  sx_c = sx_c-1;
  sy_c = sy_c+1;
```

```
//Check to make sure in bounds
if(sx_c < 0 || sy_c > f_width - 1)
{
 return status;
}
//Check to see if already has pixel connected
if(sframe[sx_c][sy_c] == 105)
{
 status = true;
 return status;
}
swidth_c = sy_c + 3;
sheight_c = sx_c - 3;
act_width = 4;
act_height = 4;
if(swidth_c > f_width - 1)
{
 act_width = 4 - (swidth_c - (f_width - 1));
 swidth_c = f_width - 1;
 if(act_width == 1)
 {
  level_zero_ver = false;
  level_one_ver = false;
  level_two_ver = false;
 }
 else if(act_width == 2)
 {
  level_one_ver = false;
  level_two_ver = false;
 }
 else if(act_width == 3)
 {
  level_two_ver = false;
 }
}
if(sheight_c < 0)
{
 act_height = 4 + sheight_c;
 sheight_c = 0;
 if(act_height == 1)
 {
  level_zero_hor = false;
  level_one_hor = false;
  level_two_hor = false;
 }
 else if(act_height == 2)
 {
  level_one_hor = false;
  level_two_hor = false;
 }
 else if(act_height == 3)
 {
  level_two_hor = false;
 }
}
if(levels_of_scan >= 1)
{
 //scan level zero
```

```
if(level_zero_ver == true)//vertical
{
 range = sx_c -1;
 if(act_height < 2)
 {
  range = sheight_c;
 }
 for(int a1 = sx_c; a1 >= range; a1--)
 {
  if(sframe[a1][sy_c+1] == 105 || sframe[a1][sy_c+1] == 100)
  {
   found_x = sx_c -1;
   found_y = sy_c;
   found_conn0 = true;
   status = true;
   break;
  }
 }//for
}//if
//scan level zero
if(level_zero_hor == true && found_conn0 == false)//horizontal
{
 if(sframe[sx_c -1][sy_c] == 105 || sframe[sx_c -1][sy_c] == 100)
 {
  found_x = sx_c -1;
  found_y = sy_c;
  found_conn0 = true;
  status = true;
 }
 }//if
 if(found_conn0 == true)
 {
  sframe[sx_c][sy_c] = 105;
  return status;
 }//if
}
if(levels_of_scan >= 2)
{
 //scan level one
 if(level_one_ver == true)//vertical
 {
  range = sx_c-2;
  if(act_height < 3)
  {
   range = sheight_c;
  }
  for(int a1 = sx_c; a1 >= range; a1--)
  {
   if(sframe[a1][sy_c+2] == 100 || sframe[a1][sy_c+2] == 105)
   {
    found_x = a1;
    found_y = sy_c+2;
    found_conn = true;
    status = true;
    break;
   }
  }//for
 }//if
```

```
if(level_one_hor == true && found_conn == false)//horizontal
{
 range = sy_c+1;
 if(act_width < 3)
 {
 range = swidth_c;
 }

 for(int a1 = sy_c; a1 <= range; a1++)
 {
  if(sframe[sx_c -2][a1] == 100 || sframe[sx_c -2][a1] == 105)
  {
   found_x = sx_c -2;
   found_y = a1;
   found_conn = true;
   status = true;
   break;
  }
 }//for
}//if
if(found_conn == true)
{
 if(found_x == sx_c)
 {
  sframe[sx_c][sy_c+1] = 105;
  sframe[sx_c][sy_c] = 105;
 }
 else if(found_y == sy_c)
```

```
    {
     sframe[sx_c -1][sy_c] = 105;
     sframe[sx_c][sy_c] = 105;
    }
    else
    {
     sframe[sx_c -1][sy_c+1] = 105;
     sframe[sx_c][sy_c] = 105;
    }
    return status;
   }//if
  }

  if(levels_of_scan == 3)
  {
   //scan level two
   if(level_two_ver == true)//vertical
   {
    for(int a1 = sx_c; a1 >= sheight_c; a1--)
    {
     if(sframe[a1][sy_c+3] == 100 || sframe[a1][sy_c+3] == 105)
     {
      found_x = a1;
      found_y = sy_c+3;
      found_conn2 = true;
      status = true;
      break;
     }
    }//for
```

```
}//if
if(level_two_hor == true && found_conn2 == false)//horizontal
{
 for(int a1 = sy_c; a1 <= swidth_c; a1++)
 {
  if(sframe[sx_c -3][a1] == 100 || sframe[sx_c -3][a1] == 105)
  {
   found_x = sx_c -3;
   found_y = a1;
   found_conn2 = true;
   status = true;
   break;
  }
 }//for
}//if
if(found_conn2 == true)
{
 if(found_x == sx_c)
 {
  sframe[sx_c][sy_c+1] = 105;
  sframe[sx_c][sy_c+2] = 105;
  sframe[sx_c][sy_c] = 105;
 }
 else if(found_y == sy_c)
 {
  sframe[sx_c -1][sy_c] = 105;
  sframe[sx_c -2][sy_c] = 105;
  sframe[sx_c][sy_c] = 105;
 }
```

```
      else if(found_x == sx_c -1)
      {
       sframe[sx_c -1][sy_c+1] = 105;
       sframe[sx_c -1][sy_c+2] = 105;
       sframe[sx_c][sy_c] = 105;
      }
      else if(found_y == sy_c+1)
      {
       sframe[sx_c -1][sy_c+1] = 105;
       sframe[sx_c -2][sy_c+1] = 105;
       sframe[sx_c][sy_c] = 105;
      }
      else
      {
       sframe[sx_c -1][sy_c+1] = 105;
       sframe[sx_c -2][sy_c+2] = 105;
       sframe[sx_c][sy_c] = 105;
      }
      return status;
     }
    }
   return status;
  }
  bool RightScannerDown(double **sframe, int sx_c, int sy_c,
                        int levels_of_scan, int f_height, int f_width)
  {
   int sheight_c, swidth_c;
   int act_width, act_height;
   int found_x, found_y;
```

```
int range;
bool status = false;
bool found_conn0 = false;
bool found_conn = false;
bool found_conn2 = false;
bool level_zero_ver = true;
bool level_zero_hor = true;
bool level_one_ver = true;
bool level_one_hor = true;
bool level_two_ver = true;
bool level_two_hor = true;
sx_c = sx_c+1;
sy_c = sy_c+1;
//Check to make sure in bounds
if(sx_c > f_height - 1 || sy_c > f_width - 1)
{
 return status;
}
//Check to see if already has pixel connected
if(sframe[sx_c][sy_c] == 105)
{
 status = true;
 return status;
}
swidth_c = sy_c + 3;
sheight_c = sx_c + 3;
act_width = 4;
act_height = 4;
if(swidth_c > f_width - 1)
```

```
{
 act_width = 4 - (swidth_c - (f_width - 1));
 swidth_c = 79;
 if(act_width == 1)
 {
  level_zero_ver = false;
  level_one_ver = false;
  level_two_ver = false;
 }
 else if(act_width == 2)
 {
  level_one_ver = false;
  level_two_ver = false;
 }
 else if(act_width == 3)
 {
  level_two_ver = false;
 }
}
if(sheight_c > f_height - 1)
{
 act_height = 4 - (sheight_c - (f_height - 1));
 sheight_c = f_height - 1;
 if(act_height == 1)
 {
  level_zero_hor = false;
  level_one_hor = false;
  level_two_hor = false;
 }
```

```
  else if(act_height == 2)
  {
   level_one_hor = false;
   level_two_hor = false;
  }
  if(act_height == 3)
  {
   level_two_hor = false;
  }
 }
 if(levels_of_scan >= 1)
 {
  //scan level zero
  if(level_zero_ver == true)//vertical
  {
   range = sx_c+1;
   if(act_height < 2)
   {
    range = sheight_c;
   }
   for(int a1 = sx_c; a1 <= range; a1++)
   {
    if(sframe[a1][sy_c+1] == 105)
    {
     found_x = a1;
     found_y = sy_c+1;
     found_conn0 = true;
     status = true;
     break;
     }
    }//for
   }//if
   //scan level zero
   if(level_zero_hor == true && found_conn0 == false)//horizontal
   {
    if(sframe[sx_c+1][sy_c] == 105 || sframe[sx_c+1][sy_c] == 100)
    {
     found_x = sx_c+1;
     found_y = sy_c;
     found_conn0 = true;
     status = true;
    }
   }//if
   if(found_conn0 == true)
   {
    sframe[sx_c][sy_c] = 105;
    return status;
   }//if
  }
  if(levels_of_scan >= 2)
  {
   //scan level one
   if(level_one_ver == true)//vertical
   {
    range = sx_c+2;
    if(act_height < 3)
    {
     range = sheight_c;
  }
  for(int a1 = sx_c; a1 <= range; a1++)
  {
   if(sframe[a1][sy_c+2] == 100 || sframe[a1][sy_c+2] == 105)
   {
    found_x = a1;
    found_y = sy_c+2;
    found_conn = true;
    status = true;
    break;
   }
  }//for
 }//if
 if(level_one_hor == true && found_conn == false)//horizontal
 {
  range = sy_c+1;
  if(act_width < 3)
  {
   range = swidth_c;
  }
  for(int a1 = sy_c; a1 <= range; a1++)
  {
   if(sframe[sx_c+2][a1] == 100 || sframe[sx_c+2][a1] == 105)
   {
    found_x = sx_c+2;
    found_y = a1;
    found_conn = true;
    status = true;
    break;
    }
   }//for
  }//if
  if(found_conn == true)
  {
   if(found_x == sx_c)
   {
    sframe[sx_c][sy_c+1] = 105;
    sframe[sx_c][sy_c] = 105;
   }
   else if(found_y == sy_c)
   {
    sframe[sx_c+1][sy_c] = 105;
    sframe[sx_c][sy_c] = 105;
   }
   else
   {
    sframe[sx_c+1][sy_c+1] = 105;
    sframe[sx_c][sy_c] = 105;
   }
   return status;
  }//if
 }
 if(levels_of_scan == 3)
 {
  //scan level two
  if(level_two_ver == true)//vertical
  {
   for(int a1 = sx_c; a1 <= sheight_c; a1++)
 {
  if(sframe[a1][sy_c+3] == 100 || sframe[a1][sy_c+3] == 105)
  {
   found_x = a1;
   found_y = sy_c+3;
   found_conn2 = true;
   status = true;
   break;
  }
 }//for
}//if
if(level_two_hor == true && found_conn2 == false)//horizontal
{
 for(int a1 = sy_c; a1 <= swidth_c; a1++)
 {
  if(sframe[sx_c+3][a1] == 100 || sframe[sx_c+3][a1] == 105)
  {
   found_x = sx_c+3;
   found_y = a1;
   found_conn2 = true;
   status = true;
   break;
  }
 }//for
}//if
if(found_conn2 == true)
{
 if(found_x == sx_c)
 {
```

```
 sframe[sx_c][sy_c+1] = 105;
 sframe[sx_c][sy_c+2] = 105;
 sframe[sx_c][sy_c] = 105;
}
else if(found_y == sy_c)
{
 sframe[sx_c+1][sy_c] = 105;
 sframe[sx_c+2][sy_c] = 105;
 sframe[sx_c][sy_c] = 105;
}
else if(found_x == sx_c+1)
{
 sframe[sx_c+1][sy_c+1] = 105;
 sframe[sx_c+1][sy_c+2] = 105;
 sframe[sx_c][sy_c] = 105;
}
else if(found_y == sy_c+1)
{
 sframe[sx_c+1][sy_c+1] = 105;
 sframe[sx_c+2][sy_c+1] = 105;
 sframe[sx_c][sy_c] = 105;
}
else
{
 sframe[sx_c+1][sy_c+1] = 105;
 sframe[sx_c+2][sy_c+2] = 105;
 sframe[sx_c][sy_c] = 105;
}
return status;
```

```
  }
 }
 return status;
}
bool LeftScannerUp(double **sframe, int sx_c, int sy_c,
                   int levels_of_scan, int f_height, int f_width)
{
 int sheight_c, swidth_c;
 int act_width, act_height;
 int found_x, found_y;
 int range;
 bool status = false;
 bool found_conn0 = false;
 bool found_conn = false;
 bool found_conn2 = false;
 bool level_zero_ver = true;
 bool level_zero_hor = true;
 bool level_one_ver = true;
 bool level_one_hor = true;
 bool level_two_ver = true;
 bool level_two_hor = true;
 bool done = false;
 sx_c = sx_c -1;
 sy_c = sy_c -1;
 //Check to make sure in bounds
 if(sx_c < 0 || sy_c < 0)
 {
  return status;
 }
```

```
//Check to see if already has pixel connected
if(sframe[sx_c][sy_c] == 105)
{
 status = true;
 return status;
}
swidth_c = sy_c - 3;
sheight_c = sx_c - 3;
act_width = 4;
act_height = 4;
if(swidth_c < 0)
{
 act_width = 4 + swidth_c;
 swidth_c = 0;
 if(act_width == 1)
 {
  level_zero_ver = false;
  level_one_ver = false;
  level_two_ver = false;
 }
 else if(act_width == 2)
 {
  level_one_ver = false;
  level_two_ver = false;
 }
 else if(act_width == 3)
 {
  level_two_ver = false;
 }
```

```
}
if(sheight_c < 0)
{
 act_height = 4 + sheight_c;
 sheight_c = 0;
 if(act_height == 1)
 {
  level_zero_hor = false;
  level_one_hor = false;
  level_two_hor = false;
 }
 else if(act_height == 2)
 {
  level_one_hor = false;
  level_two_hor = false;
 }
 else if(act_height == 3)
 {
  level_two_hor = false;
 }
}
if(levels_of_scan >= 1)
{
 //scan level zero
 if(level_zero_ver == true)//vertical
 {
  range = sx_c -1;
  if(act_height < 2)
  {
   range = sheight_c;
  }
  for(int a1 = sx_c; a1 >= range; a1--)
  {
   if(sframe[a1][sy_c -1] == 105 || sframe[a1][sy_c -1] == 100)
   {
    found_x = a1;
    found_y = sy_c -2;
    found_conn0 = true;
    status = true;
    break;
   }
  }//for
 }//if
 //scan level zero
 if(level_zero_hor == true && found_conn0 == false)//horizontal
 {
  if(sframe[sx_c -1][sy_c] == 105 || sframe[sx_c -1][sy_c] == 100)
  {
   found_x = sx_c;
   found_y = sy_c;
   found_conn0 = true;
   status = true;
  }
 }//if
 if(found_conn0 == true)
 {
  sframe[sx_c][sy_c] = 105;
  return status;
 }//if
}
if(levels_of_scan >= 2)
{
 //scan level one
 if(level_one_ver == true)//vertical
 {
  range = sx_c -2;
  if(act_height < 3)
  {
   range = sheight_c;
  }
  for(int a1 = sx_c; a1 >= range; a1--)
  {
   if(sframe[a1][sy_c -2] == 100 || sframe[a1][sy_c -2] == 105)
   {
    found_x = a1;
    found_y = sy_c -2;
    found_conn = true;
    done = true;
    status = true;
    break;
   }
  }//for
 }//if
 if(level_one_hor == true && found_conn == false)//horizontal
 {
  range = sy_c -1;
  if(act_width < 3)
 {
  range = swidth_c;
 }
 for(int a1 = sy_c; a1 >= range; a1--)
 {
  if(sframe[sx_c -2][a1] == 100 || sframe[sx_c -2][a1] == 105)
  {
   found_x = sx_c -2;
   found_y = a1;
   found_conn = true;
   done = true;
   status = true;
   break;
  }
 }//for
}//if
if(found_conn == true)
{
 if(found_x == sx_c)
 {
  sframe[sx_c][sy_c -1] = 105;
  sframe[sx_c][sy_c] = 105;
 }
 else if(found_y == sy_c)
 {
  sframe[sx_c -1][sy_c] = 105;
  sframe[sx_c][sy_c] = 105;
 }
 else
```

```
    {
     sframe[sx_c -1][sy_c -1] = 105;
     sframe[sx_c][sy_c] = 105;
    }
    return status;
  }//if
}
if(levels_of_scan == 3)
{
  //scan level two
  if(level_two_ver == true)//vertical
  {
    for(int a1 = sx_c; a1 >= sheight_c; a1--)
    {
      if(sframe[a1][sy_c -3] == 100 || sframe[a1][sy_c -3] == 105)
      {
        found_x = a1;
        found_y = sy_c -3;
        found_conn2 = true;
        status = true;
        break;
      }
    }//for
  }//if
  if(level_two_hor == true && found_conn2 == false)//horizontal
  {
    for(int a1 = sy_c; a1 >= swidth_c; a1--)
    {
      if(sframe[sx_c -3][a1] == 100 || sframe[sx_c -3][a1] == 105)
```

```
   {
    found_x = sx_c -3;
    found_y = a1;
    found_conn2 = true;
    status = true;
    break;
   }
  }//for
 }//if
 if(found_conn2 == true)
 {
  if(found_x == sx_c)
  {
   sframe[sx_c][sy_c -1] = 105;
   sframe[sx_c][sy_c -2] = 105;
   sframe[sx_c][sy_c] = 105;
  }
  else if(found_y == sy_c)
  {
   sframe[sx_c -1][sy_c] = 105;
   sframe[sx_c -2][sy_c] = 105;
   sframe[sx_c][sy_c] = 105;
  }
  else if(found_x == sx_c -1)
  {
   sframe[sx_c -1][sy_c -1] = 105;
   sframe[sx_c -1][sy_c -2] = 105;
   sframe[sx_c][sy_c] = 105;
  }
```

```
    else if(found_y == sy_c -1)
    {
     sframe[sx_c -1][sy_c -1] = 105;
     sframe[sx_c -2][sy_c -1] = 105;
     sframe[sx_c][sy_c] = 105;
    }
    else
    {
     sframe[sx_c -1][sy_c -1] = 105;
     sframe[sx_c -2][sy_c -2] = 105;
     sframe[sx_c][sy_c] = 105;
    }
    return status;
   }//if
  }
 return status;
}
bool LeftScannerDown(double **sframe, int sx_c, int sy_c,
                     int levels_of_scan, int f_height, int f_width)
{
 int sheight_c, swidth_c;
 int act_width, act_height;
 int found_x, found_y;
 int range;
 bool status = false;
 bool found_conn0 = false;
 bool found_conn = false;
 bool found_conn2 = false;
 bool level_zero_ver = true;
bool level_zero_hor = true;
bool level_one_ver = true;
bool level_one_hor = true;
bool level_two_ver = true;
bool level_two_hor = true;
sx_c = sx_c+1;
sy_c = sy_c-1;
//Check to make sure in bounds
if(sx_c > f_height - 1 || sy_c < 0)
{
 return status;
}
//Check to see if already has pixel connected
if(sframe[sx_c][sy_c] == 105)
{
 status = true;
 return status;
}
swidth_c = sy_c - 3;
sheight_c = sx_c + 3;
act_width = 4;
act_height = 4;
if(swidth_c < 0)
{
 act_width = 4 + swidth_c;
 swidth_c = 0;
 if(act_width == 1)
 {
  level_zero_ver = false;
```

```
   level_one_ver = false;
   level_two_ver = false;
  }
  else if(act_width == 2)
  {
   level_one_ver = false;
   level_two_ver = false;
  }
  else if(act_width == 3)
  {
   level_two_ver = false;
  }
 }
 if(sheight_c > f_height - 1)
 {
  act_height = 4 - (sheight_c - (f_height - 1));
  sheight_c = f_height - 1;
  if(act_height == 1)
  {
   level_zero_hor = false;
   level_one_hor = false;
   level_two_hor = false;
  }
  else if(act_height == 2)
  {
   level_one_hor = false;
   level_two_hor = false;
  }
  else if(act_height == 3)
 {
  level_two_hor = false;
 }
}

if(levels_of_scan >= 1)
{
 //scan level zero
 if(level_zero_ver == true)//vertical
 {
  range = sx_c+1;
  if(act_height < 2)
  {
   range = sheight_c;
  }
  for(int a1 = sx_c; a1 <= range; a1++)
  {
   if(sframe[a1][sy_c-1] == 105 || sframe[a1][sy_c-1] == 100)
   {
    found_x = a1;
    found_y = sy_c-1;
    found_conn0 = true;
    status = true;
    break;
   }
  }//for
 }//if
 //scan level zero
 if(level_zero_hor == true && found_conn0 == false)//horizontal
 {
  if(sframe[sx_c+1][sy_c] == 105 || sframe[sx_c+1][sy_c] == 100)
  {
   found_x = sx_c+1;
   found_y = sy_c;
   found_conn0 = true;
   status = true;
  }
 }//if
 if(found_conn0 == true)
 {
  sframe[sx_c][sy_c] = 105;
  return status;
 }//if
}
if(levels_of_scan >= 2)
{
 //scan level one
 if(level_one_ver == true)//vertical
 {
  range = sx_c+2;
  if(act_height < 3)
  {
   range = sheight_c;
  }
  for(int a1 = sx_c; a1 <= range; a1++)
  {
   if(sframe[a1][sy_c-2] == 100 || sframe[a1][sy_c-2] == 105)
   {
    found_x = a1;
    found_y = sy_c -2;
    found_conn = true;
    status = true;
    break;
   }
  }//for
 }//if
 if(level_one_hor == true && found_conn == false)//horizontal
 {
  range = sy_c -1;
  if(act_width < 3)
  {
   range = swidth_c;
  }
  for(int a1 = sy_c; a1 >= range; a1--)
  {
   if(sframe[sx_c+2][a1] == 100 || sframe[sx_c+2][a1] == 105)
   {
    found_x = sx_c+2;
    found_y = a1;
    found_conn = true;
    status = true;
    break;
   }
  }//for
 }//if
 if(found_conn == true)
 {
   if(found_x == sx_c)
   {
    sframe[sx_c][sy_c-1] = 105;
    sframe[sx_c][sy_c] = 105;
   }
   else if(found_y == sy_c)
   {
    sframe[sx_c+1][sy_c] = 105;
    sframe[sx_c][sy_c] = 105;
   }
   else
   {
    sframe[sx_c+1][sy_c-1] = 105;
    sframe[sx_c][sy_c] = 105;
   }
   return status;
  }//if
 }
 if(levels_of_scan == 3)
 {
  //scan level two
  if(level_two_ver == true)//vertical
  {
   for(int a1 = sx_c; a1 <= sheight_c; a1++)
   {
    if(sframe[a1][sy_c-3] == 100 || sframe[a1][sy_c-3] == 105)
    {
     found_x = a1;
     found_y = sy_c-3;
    found_conn2 = true;
    status = true;
    break;
   }
  }//for
 }//if
 if(level_two_hor == true && found_conn2 == false)//horizontal
 {
  for(int al = sy_c; al >= swidth_c; al--)
  {
   if(sframe[sx_c+3][al] == 100 || sframe[sx_c+3][al] == 105)
   {
    found_x = sx_c+3;
    found_y = al;
    found_conn2 = true;
    status = true;
    break;
   }
  }//for
 }//if
 if(found_conn2 == true)
 {
  if(found_x == sx_c)
  {
   sframe[sx_c][sy_c-1] = 105;
   sframe[sx_c][sy_c-2] = 105;
   sframe[sx_c][sy_c] = 105;
  }
  else if(found_y == sy_c)
```

```
    {
     sframe[sx_c+1][sy_c] = 105;
     sframe[sx_c+2][sy_c] = 105;
     sframe[sx_c][sy_c] = 105;
    }
    else if(found_x == sx_c+1)
    {
     sframe[sx_c+1][sy_c-1] = 105;
     sframe[sx_c+1][sy_c-2] = 105;
     sframe[sx_c][sy_c] = 105;
    }
    else if(found_y == sy_c-1)
    {
     sframe[sx_c+1][sy_c-1] = 105;
     sframe[sx_c+2][sy_c-1] = 105;
     sframe[sx_c][sy_c] = 105;
    }
    else
    {
     sframe[sx_c+1][sy_c-1] = 105;
     sframe[sx_c+2][sy_c-2] = 105;
     sframe[sx_c][sy_c] = 105;
    }
    return status;
   }//if
  }
  return status;
 }
 void GapFiller(IplImage * image, int startx, int starty,
                int levels, int filter_height, int filter_width)
{
  double **subframe;
  subframe = new double*[filter_height];
  for(int i = 0; i < filter_height; i++)
  {
   subframe[i] = new double[filter_width];
  }
  int borderx1 = 0;
  int borderx2 = 0 + filter_height - 1;
  int bordery1 = 0;
  int bordery2 = 0 + filter_width - 1;
  int icounter = 0;
  int jcounter = 0;
  int m = 0;
  int n = 0;
  int c,c2,c3,o,p;
  int tn, tm;
  int tempx, tempy;
  int curposx = 0;
  int curposy = 0;
  int finalcounter = 0;
  int walk_to;
  int backtrack_pixel;
  double n0, n1, n2, n3, n4, n5, n6, n7;
  CvScalar pixel, pixelchange;
  int sx,sy;
  int number_of_connections = 0;
  bool connected = false;
```

```
vector<int> endpoint_x;
vector<int> endpoint_y;
vector<double> neighbors; //all neighbors and pixel values
vector<int> w_neighbors; // all walkable pixel neighbors
vector<int> nw_neighbors; //all nonwalkable pixel neighbors
vector<int> neighbor_id;
vector<int> x_backtrack;
vector<int> y_backtrack;
bool edged_walked = false;
//copy subframe pixels into array
for(int a = startx; a < startx + filter_height; a++)
 {
  for(int b = starty; b < starty + filter_width; b++)
  {
   pixel = cvGet2D(image, a, b);
   subframe[m][n] = pixel.val[0];
   if(n == filter_width - 1)
   {
    m++;
    n = 0;
   }
   else
   {
    n++;
   }
  }
 }
 m = 0;
 n = 0;
```

```
for( o = 0; o < filter_height; o++)
{
 for(p = 0; p < filter_width; p++)
 {
  if(subframe[o][p] == 100) //not a black pixel
  {
   m = o;
   n = p;
   while(edged_walked != true)
   {
    if( m != borderx1 && m != borderx2
       && n != bordery1 && n != bordery2 ) //inner pixels
    {
     //calculate neigbor pixel values
     n0 = subframe[m-1][n-1];
     n1 = subframe[m-1][n];
     n2 = subframe[m-1][n+1];
     n3 = subframe[m][n-1];
     n4 = subframe[m][n+1];
     n5 = subframe[m+1][n-1];
     n6 = subframe[m+1][n];
     n7 = subframe[m+1][n+1];
     neighbors.push_back(n0);
     neighbors.push_back(n1);
     neighbors.push_back(n2);
     neighbors.push_back(n3);
     neighbors.push_back(n4);
     neighbors.push_back(n5);
     neighbors.push_back(n6);
 neighbors.push_back(n7);
 neighbor_id.push_back(0);
 neighbor_id.push_back(1);
 neighbor_id.push_back(2);
 neighbor_id.push_back(3);
 neighbor_id.push_back(4);
 neighbor_id.push_back(5);
 neighbor_id.push_back(6);
 neighbor_id.push_back(7);
}
else if (m != borderx1 && m != borderx2
         && n == bordery1) //vertical left side pixels
{
 n1 = subframe[m-1][n];
 n2 = subframe[m-1][n+1];
 n4 = subframe[m][n+1];
 n6 = subframe[m+1][n];
 n7 = subframe[m+1][n+1];
 neighbors.push_back(n1);
 neighbors.push_back(n2);
 neighbors.push_back(n4);
 neighbors.push_back(n6);
 neighbors.push_back(n7);
 neighbor_id.push_back(1);
 neighbor_id.push_back(2);
 neighbor_id.push_back(4);
 neighbor_id.push_back(6);
 neighbor_id.push_back(7);
}
```

```
else if (m == borderx2
         && n == bordery1) // bottom left pixel
{
 n1 = subframe[m-1][n];
 n2 = subframe[m-1][n+1];
 n4 = subframe[m][n+1];
 neighbors.push_back(n1);
 neighbors.push_back(n2);
 neighbors.push_back(n4);
 neighbor_id.push_back(1);
 neighbor_id.push_back(2);
 neighbor_id.push_back(4);
}
else if (m == borderx2 && n != bordery1
         && n != bordery2) //horizontal bottom side pixels
{
 n0 = subframe[m-1][n-1];
 n1 = subframe[m-1][n];
 n2 = subframe[m-1][n+1];
 n3 = subframe[m][n-1];
 n4 = subframe[m][n+1];
 neighbors.push_back(n0);
 neighbors.push_back(n1);
 neighbors.push_back(n2);
 neighbors.push_back(n3);
 neighbors.push_back(n4);
 neighbor_id.push_back(0);
 neighbor_id.push_back(1);
 neighbor_id.push_back(2);
 neighbor_id.push_back(3);
 neighbor_id.push_back(4);
}
else if (m == borderx2
        && n == bordery2) // bottom right corner pixel
{
 n0 = subframe[m-1][n-1];
 n1 = subframe[m-1][n];
 n3 = subframe[m][n-1];
 neighbors.push_back(n0);
 neighbors.push_back(n1);
 neighbors.push_back(n3);
 neighbor_id.push_back(0);
 neighbor_id.push_back(1);
 neighbor_id.push_back(3);
}
else if (m != borderx1 && m != borderx2
        && n == bordery2) // vertical right side pixels
{
 n0 = subframe[m-1][n-1];
 n1 = subframe[m-1][n];
 n3 = subframe[m][n-1];
 n5 = subframe[m+1][n-1];
 n6 = subframe[m+1][n];
 neighbors.push_back(n0);
 neighbors.push_back(n1);
 neighbors.push_back(n3);
 neighbors.push_back(n5);
 neighbors.push_back(n6);
 neighbor_id.push_back(0);
 neighbor_id.push_back(1);
 neighbor_id.push_back(3);
 neighbor_id.push_back(5);
 neighbor_id.push_back(6);
}
else if (m == borderx1
        && n == bordery2) // top right pixel
{
 n3 = subframe[m][n-1];
 n5 = subframe[m+1][n-1];
 n6 = subframe[m+1][n];
 neighbors.push_back(n3);
 neighbors.push_back(n5);
 neighbors.push_back(n6);
 neighbor_id.push_back(3);
 neighbor_id.push_back(5);
 neighbor_id.push_back(6);
}
else if (m == borderx1 && n != bordery1
        && n != bordery2) //horizontal top side pixels
{
 n3 = subframe[m][n-1];
 n4 = subframe[m][n+1];
 n5 = subframe[m+1][n-1];
 n6 = subframe[m+1][n];
 n7 = subframe[m+1][n+1];
 neighbors.push_back(n3);
 neighbors.push_back(n4);
 neighbors.push_back(n5);
 neighbors.push_back(n6);
 neighbors.push_back(n7);
 neighbor_id.push_back(3);
 neighbor_id.push_back(4);
 neighbor_id.push_back(5);
 neighbor_id.push_back(6);
 neighbor_id.push_back(7);
}
else if ( m == borderx1 && n == bordery1) //top left pixel
{
 n4 = subframe[m][n+1];
 n6 = subframe[m+1][n];
 n7 = subframe[m+1][n+1];
 neighbors.push_back(n4);
 neighbors.push_back(n6);
 neighbors.push_back(n7);
 neighbor_id.push_back(4);
 neighbor_id.push_back(6);
 neighbor_id.push_back(7);
}
//find all pixel neighbors that are white
for(c = 0; c < neighbors.size(); c++)
{
 if(neighbors[c] > 99)
 {
  if(neighbors[c] == 100)
  {
   w_neighbors.push_back(neighbor_id[c]);
  }
  else if (neighbors[c] == 105)
  {
   nw_neighbors.push_back(neighbor_id[c]);
  }
 }
}
EndpointVerifier(subframe, w_neighbors.size(), nw_neighbors.size(),
                 endpoint_x, endpoint_y, m, n, filter_height,
                 filter_width);
if(w_neighbors.empty()) //if no connecting pixel
{
 subframe[m][n] = 105; //non-walkable pixel
 if(x_backtrack.empty())
 {
  edged_walked = true;
  while(endpoint_x.size() != 0)
  {
   sx = endpoint_x.back();
   sy = endpoint_y.back();
   endpoint_x.pop_back();
   endpoint_y.pop_back();
   if((subframe[sx-1][sy-1] == 105 ||
       subframe[sx-1][sy-1] == 100) ||
      (subframe[sx][sy-1] == 105 ||
       subframe[sx][sy-1] == 100) ||
      (subframe[sx+1][sy-1] == 105 ||
       subframe[sx+1][sy-1] == 100)
     ) //right scanner
```

```
{
 if((subframe[sx-1][sy] == 105 ||
     subframe[sx-1][sy] == 100))
 {
  connected = DownScanner(subframe, sx, sy, levels,
                          filter_height, filter_width);
  if(connected == false)
  {
   connected = RightScannerDown(subframe, sx, sy, levels,
                                filter_height, filter_width);
  }
  if(connected == false)
  {
   connected = LeftScannerDown(subframe, sx, sy, levels,
                               filter_height, filter_width);
  }
  if(connected == false)
  {
   connected = RightScannerForward(subframe, sx, sy, levels,
                                   filter_height, filter_width);
  }
 }
 else if((subframe[sx+1][sy] == 105 ||
          subframe[sx+1][sy] == 100))
 {
  connected = UpScanner(subframe, sx, sy, levels,
                        filter_height, filter_width);
  if(connected == false)
  {
```

```
  connected = RightScannerUp(subframe, sx, sy, levels,
                             filter_height, filter_width);
 }
 if(connected == false)
 {
  connected = LeftScannerUp(subframe, sx, sy, levels,
                            filter_height, filter_width);
 }
 if(connected == false)
 {
  connected = RightScannerForward(subframe, sx, sy, levels,
                                  filter_height, filter_width);
 }
}
else
{
 connected = RightScannerForward(subframe, sx, sy, levels,
                                 filter_height, filter_width);
 if(connected == false)
 {
  connected = RightScannerUp(subframe, sx, sy, levels,
                             filter_height, filter_width);
 }
 if(connected == false)
 {
  connected = RightScannerDown(subframe, sx, sy, levels,
                               filter_height, filter_width);
 }
 if(connected == false)
```

```
   {
    connected = UpScanner(subframe, sx, sy, levels,
                              filter_height, filter_width);
   }
   if(connected == false)
   {
    connected = DownScanner(subframe, sx, sy, levels,
                                filter_height, filter_width);
   }
  }
} //right scanner
else if((subframe[sx-1][sy+1] == 105 ||
         subframe[sx-1][sy+1] == 100) ||
        (subframe[sx][sy+1] == 105 ||
         subframe[sx][sy+1] == 100) ||
        (subframe[sx+1][sy+1] == 105 ||
         subframe[sx+1][sy+1] == 100)
       ) //left scanner
{
 if((subframe[sx-1][sy] == 105 ||
     subframe[sx-1][sy] == 100))
 {
  connected = DownScanner(subframe, sx, sy, levels,
                              filter_height, filter_width);
  if(connected == false)
  {
   connected = RightScannerDown(subframe, sx, sy, levels,
                                    filter_height, filter_width);
  }
 if(connected == false)
 {
  connected = LeftScannerDown(subframe, sx, sy, levels,
                              filter_height, filter_width);
 }
 if(connected == false)
 {
  connected = LeftScannerForward(subframe, sx, sy, levels,
                                 filter_height, filter_width);
 }
}
else if((subframe[sx+1][sy] == 105 ||
         subframe[sx+1][sy] == 100))
{
 connected = UpScanner(subframe, sx, sy, levels,
                       filter_height, filter_width);
 if(connected == false)
 {
  connected = LeftScannerForward(subframe, sx, sy, levels,
                                 filter_height, filter_width);
 }
 if(connected == false)
 {
  connected = LeftScannerForward(subframe, sx, sy, levels,
                                 filter_height, filter_width);
 }
 if(connected == false)
 {
  connected = RightScannerUp(subframe, sx, sy, levels,
                              filter_height , filter_width);
 }
}
else
{
 connected = LeftScannerForward(subframe, sx, sy, levels,
                              filter_height , filter_width);
 if(connected == false)
 {
  connected = LeftScannerUp(subframe, sx, sy, levels,
                            filter_height , filter_width);
 }
 if(connected == false)
 {
  connected = LeftScannerDown(subframe, sx, sy, levels,
                              filter_height , filter_width);
 }
 if(connected == false)
 {
  connected = UpScanner(subframe, sx, sy, levels,
                        filter_height , filter_width);
 }
 if(connected == false)
 {
  connected = DownScanner(subframe, sx, sy, levels,
                          filter_height , filter_width);
 }
}
}//left scanner
```

```
else if((subframe[sx+1][sy] == 105 ||
        subframe[sx+1][sy] == 100) &&
        subframe[sx-1][sy] == 0) //up scanner
{
 connected = UpScanner(subframe, sx, sy, levels,
                      filter_height, filter_width);
 if(connected == false)
 {
  connected = LeftScannerUp(subframe, sx, sy, levels,
                           filter_height, filter_width);
 }
 if(connected == false)
 {
  connected = RightScannerUp(subframe, sx, sy, levels,
                            filter_height, filter_width);
 }
 if(connected == false)
 {
  connected = LeftScannerForward(subframe, sx, sy, levels,
                                filter_height, filter_width);
 }
 if(connected == false)
 {
  connected = RightScannerForward(subframe, sx, sy, levels,
                                 filter_height, filter_width);
 }
}
else if((subframe[sx-1][sy] == 105 ||
        subframe[sx-1][sy] == 100) &&
        subframe[sx+1][sy] == 0) //down scanner
{
 connected = DownScanner(subframe, sx, sy, levels,
                         filter_height, filter_width);
 if(connected == false)
 {
  connected = LeftScannerDown(subframe, sx, sy, levels,
                              filter_height, filter_width);
 }
 if(connected == false)
 {
  connected = RightScannerDown(subframe, sx, sy, levels,
                               filter_height, filter_width);
 }
 if(connected == false)
 {
  connected = LeftScannerForward(subframe, sx, sy, levels,
                                 filter_height, filter_width);
 }
 if(connected == false)
 {
  connected = RightScannerForward(subframe, sx, sy, levels,
                                  filter_height, filter_width);
 }
}
else if(subframe[sx-1][sy-1] == 0 && subframe[sx-1][sy] == 0 &&
        subframe[sx-1][sy+1] == 0 && subframe[sx][sy-1] == 0 &&
        subframe[sx][sy+1] == 0 && subframe[sx+1][sy-1] == 0 &&
        subframe[sx+1][sy] == 0 && subframe[sx+1][sy+1] == 0
        ) //up scan, single pixel
{
 connected = UpScanner(subframe, sx, sy, levels,
                        filter_height, filter_width);
 if(connected == true)
 {
  number_of_connections++;
 }
 if(connected == false || (connected == true
                        && number_of_connections < 2))
 {
  connected = LeftScannerUp(subframe, sx, sy, levels,
                        filter_height, filter_width);
  if(connected == true)
  {
   number_of_connections++;
  }
 }
 if(connected == false || (connected == true
                        && number_of_connections < 2))
 {
  connected = RightScannerUp(subframe, sx, sy, levels,
                        filter_height, filter_width);
  if(connected == true)
  {
   number_of_connections++;
  }
 }
 if(connected == false || (connected == true
                              && number_of_connections < 2))
   {
    connected = LeftScannerForward(subframe, sx, sy, levels,
                                  filter_height, filter_width);
    if(connected == true)
    {
     number_of_connections++;
    }
   }
   if(connected == false || (connected == true
                              && number_of_connections < 2))
   {
    connected = RightScannerForward(subframe, sx, sy, levels,
                                   filter_height, filter_width);
    if(connected == true)
    {
     number_of_connections++;
    }
   }
   number_of_connections = 0;
  }
 }//while loop
 endpoint_x.clear();
 endpoint_y.clear();
}
else // backtrack not empty
{
 m = x_backtrack.back();
 n = y_backtrack.back();
```

```
  x_backtrack.pop_back();
  y_backtrack.pop_back();
 }
}
else // w_neighbors not empty, connecting pixel
{
 subframe[m][n] = 105;
 walk_to = w_neighbors.back();
 w_neighbors.pop_back();
 tm = m;
 tn = n;
 if(walk_to == 7)
 {
  m = m+1;
  n = n+1;
 }
 else if(walk_to == 6)
 {
  m = m+1;
  n = n;
 }
 else if(walk_to == 5)
 {
  m = m+1;
  n = n-1;
 }
 else if(walk_to == 4)
 {
  m = m;
 n = n+1;
}
else if(walk_to == 3)
{
 m = m;
 n = n-1;
}
else if(walk_to == 2)
{
 m = m-1;
 n = n+1;
}
else if(walk_to == 1)
{
 m = m-1;
 n = n;
}
else if(walk_to == 0)
{
 m = m-1;
 n = n-1;
}
for(int h = 0; h < w_neighbors.size(); h++)
{
  backtrack_pixel = w_neighbors[h];
  if(backtrack_pixel == 7)
  {
    x_backtrack.push_back(tm+1);
    y_backtrack.push_back(tn+1);
 subframe[tm+1][tn+1] = 105;
}
else if(backtrack_pixel == 6)
{
 x_backtrack.push_back(tm+1);
 y_backtrack.push_back(tn);
 subframe[tm+1][tn] = 105;
}
else if(backtrack_pixel == 5)
{
 x_backtrack.push_back(tm+1);
 y_backtrack.push_back(tn-1);
 subframe[tm+1][tn-1] = 105;
}
else if(backtrack_pixel == 4)
{
 x_backtrack.push_back(tm);
 y_backtrack.push_back(tn+1);
 subframe[tm][tn+1] = 105;
}
else if(backtrack_pixel == 3)
{
 x_backtrack.push_back(tm);
 y_backtrack.push_back(tn-1);
 subframe[tm][tn-1] = 105;
}
else if(backtrack_pixel == 2)
{
 x_backtrack.push_back(tm-1);
```

```
        y_backtrack.push_back(tn+1);
        subframe[tm-1][tn+1] = 105;
       }
       else if(backtrack_pixel == 1)
       {
        x_backtrack.push_back(tm-1);
        y_backtrack.push_back(tn);
        subframe[tm-1][tn] = 105;
       }
       else if(backtrack_pixel == 0)
       {
        x_backtrack.push_back(tm-1);
        y_backtrack.push_back(tn-1);
        subframe[tm-1][tn-1] = 105;
       }
      }//for loop
     }
     w_neighbors.clear();
     nw_neighbors.clear();
     neighbors.clear();
     neighbor_id.clear();
    }//while loop
    x_backtrack.clear();
    y_backtrack.clear();
   }//if statement, white pixel
   edged_walked = false;
  }//inner for loop
 } //outer for loop
 tempx = startx;
```

```
    tempy = starty;
    for(c2 = 0; c2 < filter_height; c2++)
    {
     for(c3 = 0; c3 < filter_width; c3++)
     {
      pixelchange.val[0] = subframe[c2][c3];
      cvSet2D(image, tempx, tempy, pixelchange);
      if(tempy == starty + filter_width - 1)
      {
       tempx++;
       tempy = starty;
      }
      else
      {
       tempy++;
      }
     }
    } //end for loop
    for(int i = 0; i < filter_height; i++)
    {
     delete[] subframe[i];
    }
    delete[] subframe;
}
void ThickenImage(double ***subframe, int f_height, int f_width)
{
 int borderx1 = 0;
 int borderx2 = 0 + f_height - 1;
 int bordery1 = 0;
int bordery2 = 0 + f_width - 1;
int m = 0;
int n = 0;
int c, c2,c3,o,p;
double n0, n1, n2, n3, n4, n5, n6, n7;
vector<double> neighbors; //all neighbors and pixel values
vector<int> um_neighbors; // unmarked pixel neighbors
vector<int> neighbor_id;
m = 0;
n = 0;
for( o = 0; o < f_height; o++)
{
 for(p = 0; p < f_width; p++)
 {
  if(subframe[0][o][p] == 100)
  {
   m = o;
   n = p;
   if( m != borderx1 && m != borderx2  && n != bordery1
      && n != bordery2 ) //inner pixels
   {
    //calculate neigbor pixel values
    n0 = subframe[0][m-1][n-1];
    n1 = subframe[0][m-1][n];
    n2 = subframe[0][m-1][n+1];
    n3 = subframe[0][m][n-1];
    n4 = subframe[0][m][n+1];
    n5 = subframe[0][m+1][n-1];
    n6 = subframe[0][m+1][n];
 n7 = subframe[0][m+1][n+1];
 neighbors.push_back(n0);
 neighbors.push_back(n1);
 neighbors.push_back(n2);
 neighbors.push_back(n3);
 neighbors.push_back(n4);
 neighbors.push_back(n5);
 neighbors.push_back(n6);
 neighbors.push_back(n7);
 neighbor_id.push_back(0);
 neighbor_id.push_back(1);
 neighbor_id.push_back(2);
 neighbor_id.push_back(3);
 neighbor_id.push_back(4);
 neighbor_id.push_back(5);
 neighbor_id.push_back(6);
 neighbor_id.push_back(7);
}
else if (m != borderx1 && m != borderx2
         && n == bordery1) //vertical
                             //left side pixels
{
 n1 = subframe[0][m-1][n];
 n2 = subframe[0][m-1][n+1];
 n4 = subframe[0][m][n+1];
 n6 = subframe[0][m+1][n];
 n7 = subframe[0][m+1][n+1];
 neighbors.push_back(n1);
 neighbors.push_back(n2);
 neighbors.push_back(n4);
 neighbors.push_back(n6);
 neighbors.push_back(n7);
 neighbor_id.push_back(1);
 neighbor_id.push_back(2);
 neighbor_id.push_back(4);
 neighbor_id.push_back(6);
 neighbor_id.push_back(7);
}
else if (m == borderx2 &&
         n == bordery1) // bottom left pixel
{
 n1 = subframe[0][m-1][n];
 n2 = subframe[0][m-1][n+1];
 n4 = subframe[0][m][n+1];
 neighbors.push_back(n1);
 neighbors.push_back(n2);
 neighbors.push_back(n4);
 neighbor_id.push_back(1);
 neighbor_id.push_back(2);
 neighbor_id.push_back(4);
}
else if (m == borderx2 && n != bordery1
         && n != bordery2) //horizontal
                           //bottom side pixels
{
 n0 = subframe[0][m-1][n-1];
 n1 = subframe[0][m-1][n];
 n2 = subframe[0][m-1][n+1];
 n3 = subframe[0][m][n-1];
 n4 = subframe[0][m][n+1];
 neighbors.push_back(n0);
 neighbors.push_back(n1);
 neighbors.push_back(n2);
 neighbors.push_back(n3);
 neighbors.push_back(n4);
 neighbor_id.push_back(0);
 neighbor_id.push_back(1);
 neighbor_id.push_back(2);
 neighbor_id.push_back(3);
 neighbor_id.push_back(4);
}
else if (m == borderx2
         && n == bordery2) // bottom right corner pixel
{
 n0 = subframe[0][m-1][n-1];
 n1 = subframe[0][m-1][n];
 n3 = subframe[0][m][n-1];
 neighbors.push_back(n0);
 neighbors.push_back(n1);
 neighbors.push_back(n3);
 neighbor_id.push_back(0);
 neighbor_id.push_back(1);
 neighbor_id.push_back(3);
}
else if (m != borderx1 && m != borderx2
         && n == bordery2) //vertical
                           //right side pixels
```

```
{
 n0 = subframe[0][m-1][n-1];
 n1 = subframe[0][m-1][n];
 n3 = subframe[0][m][n-1];
 n5 = subframe[0][m+1][n-1];
 n6 = subframe[0][m+1][n];
 neighbors.push_back(n0);
 neighbors.push_back(n1);
 neighbors.push_back(n3);
 neighbors.push_back(n5);
 neighbors.push_back(n6);
 neighbor_id.push_back(0);
 neighbor_id.push_back(1);
 neighbor_id.push_back(3);
 neighbor_id.push_back(5);
 neighbor_id.push_back(6);
}
else if (m == borderx1 &&
         n == bordery2) // top right pixel
{
 n3 = subframe[0][m][n-1];
 n5 = subframe[0][m+1][n-1];
 n6 = subframe[0][m+1][n];
 neighbors.push_back(n3);
 neighbors.push_back(n5);
 neighbors.push_back(n6);
 neighbor_id.push_back(3);
 neighbor_id.push_back(5);
 neighbor_id.push_back(6);
```

```
}
else if (m == borderx1 && n != bordery1
         && n != bordery2) //horizontal
                           //top side pixels
{
 n3 = subframe[0][m][n-1];
 n4 = subframe[0][m][n+1];
 n5 = subframe[0][m+1][n-1];
 n6 = subframe[0][m+1][n];
 n7 = subframe[0][m+1][n+1];
 neighbors.push_back(n3);
 neighbors.push_back(n4);
 neighbors.push_back(n5);
 neighbors.push_back(n6);
 neighbors.push_back(n7);
 neighbor_id.push_back(3);
 neighbor_id.push_back(4);
 neighbor_id.push_back(5);
 neighbor_id.push_back(6);
 neighbor_id.push_back(7);
}
else if (m == borderx1 &&
         n == bordery1) //top left pixel
{
 n4 = subframe[0][m][n+1];
 n6 = subframe[0][m+1][n];
 n7 = subframe[0][m+1][n+1];
 neighbors.push_back(n4);
 neighbors.push_back(n6);
```

```
  neighbors.push_back(n7);
  neighbor_id.push_back(4);
  neighbor_id.push_back(6);
  neighbor_id.push_back(7);
 }
 //find all pixel neighbors that are white
 for(c = 0; c < neighbors.size(); c++)
 {
  if(neighbors[c] == 0)
  {
   um_neighbors.push_back(neighbor_id[c]);
  }
 }
 if(um_neighbors.size() > 0)
 {
  for(int h = 0; h < um_neighbors.size(); h++)
  {
   if(um_neighbors[h] == 0)
   {
    subframe[0][m-1][n-1] = 105;
   }
   else if(um_neighbors[h] == 1)
   {
    subframe[0][m-1][n] = 105;
   }
   else if(um_neighbors[h] == 2)
   {
    subframe[0][m-1][n+1] = 105;
   }
```

```
      else if(um_neighbors[h] == 3)
      {
       subframe[0][m][n-1] = 105;
      }
      else if(um_neighbors[h] == 4)
      {
       subframe[0][m][n+1] = 105;
      }
      else if(um_neighbors[h] == 5)
      {
       subframe[0][m+1][n-1] = 105;
      }
      else if(um_neighbors[h] == 6)
      {
       subframe[0][m+1][n] = 105;
      }
      else if(um_neighbors[h] == 7)
      {
       subframe[0][m+1][n+1] = 105;
      }
     }//for loop
    }
    um_neighbors.clear();
    neighbors.clear();
    neighbor_id.clear();
   }//if statement, white pixel
  }//inner for loop
 } //outer for loop
 for(c2 = 0; c2 < f_height; c2++)
 {
  for(c3 = 0; c3 < f_width; c3++)
  {
   if(subframe[0][c2][c3] == 105)
   {
    subframe[0][c2][c3] = 100;
   }
  }
 } //end for loop
}
void InvertPixelValues(double ***reference, int r_height,
                       int r_width)
{
 for(int i = 0; i < r_height; i++)
 {
  for(int j = 0; j < r_width; j++)
  {
   if(reference[0][i][j] == 0)
   {
    reference[0][i][j] = 100;
   }
   else if(reference[0][i][j] == 100)
   {
    reference[0][i][j] = 0;
   }
  }
 }
}
```

```
void AND_Images(double ***reference, double ***actual,
                double ***result, int r_height, int r_width,
                int &count, int &top, int &bottom, int &right,
                int &left, int &removed)
{
 removed = 0;
 count = 0;
 for(int i = 0; i < r_height; i++)
 {
  for(int j = 0; j < r_width; j++)
  {
   if((i < bottom && i > top) &&
       (j < right && j > left))
   {
    if(reference[0][i][j] == 0 ||
       actual[0][i][j] == 0)
    {
     result[0][i][j] = 0;
    }
    else if(reference[0][i][j] == 100 &&
            actual[0][i][j] == 100 )
    {
     result[0][i][j] = 100;
     count++;
    }
   }
   else
   {
    if(actual[0][i][j] == 100)
    {
     removed++;
    }
   }
  }
 }
}
void CountNonMatchingPixels(double ***result, int r_height,
                             int r_width, int &count)
{
 count = 0;
 for(int i = 0; i < r_height; i++)
 {
  for(int j = 0; j < r_width; j++)
  {
   if(result[0][i][j] == 100)
   {
    count++;
   }
  }
 }
}
bool DetermineMouthExpression(int unmatched_count, int edge_size,
                              int ref_size, bool &mouth_expression,
                              double min_percentage_ref,
                              double min_percentage_act)
{
 double matched = 0;
 double t_pixels = edge_size;
double percentage_act = 0;
double percentage_ref = 0;
if(edge_size == 0)
{
 mouth_expression = false;
 return false;
}
if(edge_size - unmatched_count == 0)
{
 mouth_expression = false;
 return false;
}
matched = t_pixels - unmatched_count;
percentage_act = 100*(matched/t_pixels);
if(percentage_act >= min_percentage_act)
{
 percentage_ref = 100*(matched/ref_size);
 if(percentage_ref >= min_percentage_ref)
 {
  mouth_expression = true;
  return true;
 }
 else
 {
  mouth_expression = false;
  return false;
 }
}
else
```

```
 {
  mouth_expression = false;
  return false;
 }
}
bool DrawRectangle(IplImage * source, int pointx,
                   int pointy, int recheight,
                   int recwidth, int grayscale)
{
 int imageheight;
 int imagewidth;
 imageheight = source->height;
 imagewidth  = source->width;
 CvScalar pixel;
 pixel.val[0] = grayscale;
 if(pointx < 0 || pointy < 0 ||
    pointx > imageheight - 1 || pointy > imagewidth - 1 ||
    pointx + recheight - 1 > imageheight ||
    pointy + recwidth - 1 > imagewidth)
 {
  cout << "Can't draw" << endl;
  return false;
 }
 for(int top = pointy; top < pointy + recwidth; top++)
 {
  cvSet2D(source, pointx, top, pixel);
 }
 for(int bottom = pointy; bottom < pointy + recwidth; bottom++)
 {
```

```
 cvSet2D(source, pointx + recheight - 1, bottom, pixel);
 }
 for(int left= pointx; left < pointx + recheight - 1; left++)
 {
  cvSet2D(source, left, pointy, pixel);
 }
 for(int right = pointx + 1; right < pointx + recheight - 1; right++)
 {
  cvSet2D(source, right, pointy + recwidth - 1, pixel);
 }
 return true;
}
void CountPixelsInRef(double ***result, int height, int width, int &count)
{
 count = 0;
 for(int i = 0; i < height; i++)
 {
  for(int j = 0; j < width; j++)
  {
   if(result[0][i][j] == 100)
   {
    count++;
   }
  }
 }
}
```

A.2 Main Program Source Code

```
#include ''stdafx.h"
```

```
#include <cv.h>
#include <cxcore.h>
#include <highgui.h>
#include <iostream>
#include <vector>
#include <string>
#include ''fftw3.h"
#include ''image_pro_functions.h"
using namespace std;
#define PADWIDTH 120
#define PADHEIGHT 120
#define SSWIDTH 90
#define SSHEIGHT 90
#define N 120.0
#define SIZE 120
int l_threshpos = 20;
int h_threshpos = 20;
int l_thresh = 20;
int h_thresh = 20;
void changeLThresh(int lpos)
{
 l_thresh = lpos;
}
void changeHThresh(int hpos)
{
 h_thresh = hpos;
}
int _tmain(int argc, _TCHAR* argv[])
{
```

```
int cex = 0;
int cey = 0;
int matchpercentage = 70;
int top_b = 0;
int top_c = 0;
int top_s = 0;
int bottom_b = 0;
int bottom_c = 0;
int bottom_s = 0;
int left_b = 0;
int left_c = 0;
int left_s = 0;
int right_b = 0;
int right_c = 0;
int right_s = 0;
int removed = 0;
int removed = 0;
int edge_max_size = 500;
int edge_min_size = 80;
int startx = 175;
int starty = 270;
int shift = 30;
int unmat_s_count = 0;
int unmat_b_count = 0;
int unmat_c_count = 0;
int b_total = 0;
int s_total = 0;
int c_total = 0;
int ref_size = 0;
int act_size = 0;
bool bigmouth = false;
bool smallmouth = false;
bool closedmouth = false;
bool locate_bigmouth = true;
bool locate_smallmouth = true;
bool locate_closedmouth = true;
string expression_made = ''no known mouth expression made";
vector<int> final_x;
vector<int> final_y;
char key;
CvCapture* capture = cvCaptureFromCAM( CV_CAP_ANY );
if( !capture )
{
  fprintf( stderr, ''Error with capture -> null \n" );
  getchar();
  return -1;
}
cvNamedWindow( ''Image", CV_WINDOW_AUTOSIZE );
cvNamedWindow( ''Edged Image",CV_WINDOW_AUTOSIZE );
cvNamedWindow( ''Big Mouth",0);
cvNamedWindow( ''Small Mouth",0);
cvNamedWindow( ''Closed Mouth",0);
cvCreateTrackbar(''Low Threshold", ''Edged Image", &l_threshpos,
                 100, changeLThresh);
cvCreateTrackbar(''High Threshold", ''Edged Image", &h_threshpos,
                 100, changeHThresh);
IplImage *gray = NULL;
IplImage *edges = NULL;
IplImage *original = NULL;
IplImage *snap_shot = cvCreateImage(cvSize(SSWIDTH, SSHEIGHT),
                                    IPL_DEPTH_8U, 1);
IplImage *frame = NULL;
IplImage *reference = NULL;
IplImage *reference2 = NULL;
IplImage *reference3 = NULL;
bool show_frame_data = false;
bool show_once1 = false;
bool show_once2 = false;
bool show_once3 = false;
bool show_once4 = false;
bool show_message1 = true;
bool show_message2 = false;
bool show_message3 = false;
bool shots_taken = false;
bool begin_execution = false;
bool calculations = false;
bool bigmouth_cal = false;
bool smallmouth_cal = false;
bool largest_edge = true;
bool live_image = true;
int height = 0;
int width = 0;
int pos_x = 0;
int pos_y = 0;
int pos_x2 = 0;
int pos_y2 = 0;
int pos_x3 = 0;
int pos_y3 = 0;
int gapnumber = 1;
fftw_complex *in, *in2, *in3, *in4, *out, *out2, *out3, *out4, *nor, *res;
fftw_plan p, p2, p3, p4, p5, p6, p7;
in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * SIZE * SIZE);
out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * SIZE * SIZE);
in2 = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * SIZE * SIZE);
out2 = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * SIZE * SIZE);
in3 = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * SIZE * SIZE);
out3 = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * SIZE * SIZE);
in4 = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * SIZE * SIZE);
out4 = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * SIZE * SIZE);
nor = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * SIZE * SIZE);
res = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * SIZE * SIZE);
double ***i2 = new double**[2];
double ***i3 = new double**[2];
double ***i4 = new double**[2];
for(int x = 0; x < 2; x++)
{
 i2[x] = new double*[SSHEIGHT];
 i3[x] = new double*[SSHEIGHT];
 i4[x] = new double*[SSHEIGHT];
 for(int y = 0; y < SSHEIGHT; y++)
 {
  i2[x][y] = new double[SSWIDTH];
  i3[x][y] = new double[SSWIDTH];
  i4[x][y] = new double[SSWIDTH];
  for(int z = 0; z < SSWIDTH; z++)
  {
     i2[x][y][z] = 0;
     i3[x][y][z] = 0;
     i4[x][y][z] = 0;
    }
  }
}
double ***i = new double**[2];
double ***pd = new double**[2];
double ***pd2 = new double**[2];
double ***pd3 = new double**[2];
double ***sf = new double**[2];
double ***map = new double**[2];
double ***map2 = new double**[2];
double ***map3 = new double**[2];
double ***and = new double**[2];
for(int x = 0; x < 2; x++)
{
  i[x] = new double*[PADHEIGHT];
  pd[x] = new double*[PADHEIGHT];
  pd2[x] = new double*[PADHEIGHT];
  pd3[x] = new double*[PADHEIGHT];
  sf[x] = new double*[PADHEIGHT];
  map[x] = new double*[PADHEIGHT];
  map2[x] = new double*[PADHEIGHT];
  map3[x] = new double*[PADHEIGHT];
  and[x] = new double*[PADHEIGHT];
  for(int y = 0; y < PADHEIGHT; y++)
  {
    i[x][y] = new double[PADWIDTH];
```

```
   pd[x][y] = new double[PADWIDTH];
   pd2[x][y] = new double[PADWIDTH];
   pd3[x][y] = new double[PADWIDTH];
   sf[x][y] = new double[PADWIDTH];
   map[x][y] = new double[PADWIDTH];
   map2[x][y] = new double[PADWIDTH];
   map3[x][y] = new double[PADWIDTH];
   and[x][y] = new double[PADWIDTH];
   for(int z = 0; z < PADWIDTH; z++)
   {
    i[x][y][z] = 0;
    pd[x][y][z] = 0;
    pd2[x][y][z] = 0;
    pd3[x][y][z] = 0;
    sf[x][y][z] = 0;
    map[x][y][z] = 0;
    map2[x][y][z] = 0;
    map3[x][y][z] = 0;
    and[x][y][z] = 0;
   }
  }
 }
  while( 1 )
  {
   frame = cvQueryFrame( capture );
   gray  = cvCreateImage(cvGetSize(frame), IPL_DEPTH_8U, 1);
   edges = cvCreateImage(cvGetSize(frame), IPL_DEPTH_8U, 1);
   original = cvCreateImage(cvGetSize(frame), IPL_DEPTH_8U, 1);
   if(show_frame_data == false)
```

```
{
 height = edges->height;
 width  = edges->width;
 printf(''The width %d\n", width);
 printf(''The height %d\n", height);
 show_frame_data = true;
}
if(show_message1 == true && show_once1 == false)
{
 cout << ''Make a big mouth expression and press the spacebar"
      << endl;
 cout << endl;
 show_once1 = true;
}
else if(show_message2 == true && show_once2 == false)
{
 cout << ''Make a small mouth expression and press the spacebar"
      << endl;
 cout << endl;
 show_once2 = true;
}
else if(show_message3 == true && show_once3 == false)
{
 cout << ''Make a closed mouth expression and press the spacebar"
      << endl;
 cout << endl;
 show_once3 = true;
}
else if(begin_execution == true && show_once4 == false)
```

```
{
 cout << ''Press the spacebar to begin execution" << endl;
 cout << endl;
 show_once4 = true;
}
if(!frame )
{
 fprintf( stderr, ''Error with frame -> null\n" );
 getchar();
 break;
}
cvCvtColor(frame, gray, CV_BGR2GRAY);
cvSmooth(gray, gray, CV_GAUSSIAN, 9, 9);
cvCanny( gray, edges, l_thresh, h_thresh, 3 );
original = cvCloneImage(edges);
if(shots_taken == false)
{
 FormatFilter(edges, startx,  starty ,SSHEIGHT, SSWIDTH);
 GapFiller(edges, startx , starty, gapnumber, SSHEIGHT, SSWIDTH);
 FormatFilter(edges, startx,  starty, SSHEIGHT, SSWIDTH);
 if(largest_edge == false)
 {
  EdgeWalker(edges, startx, starty, edge_min_size, edge_max_size,
             SSHEIGHT, SSWIDTH, final_x, final_y, ref_size);
  final_x.clear(); final_y.clear();
 }
 else
 {
  FindLargestEdge(edges, startx, starty, SSHEIGHT, SSWIDTH, 0,
                    SSHEIGHT*SSHEIGHT, ref_size);
}
if(show_message1 == false && show_once1 == true &&
                    bigmouth_cal == false)
{
 reference = cvLoadImage( "big_mouth_open.jpg", 0);
 FormatFilter(reference, 0, 0, SSHEIGHT, SSWIDTH);
 ImageToArray(reference, i2, 0, 0, SSHEIGHT, SSWIDTH);
 CountPixelsInRef(i2, SSHEIGHT, SSWIDTH, b_total);
 PadImage(pd, i2, PADHEIGHT, PADWIDTH, SSHEIGHT, SSWIDTH);
 ArrayToFFTWFormat(pd, in2, PADHEIGHT, PADWIDTH);
 p2 = fftw_plan_dft_2d(SIZE, SIZE, in2, out2, FFTW_FORWARD,
                    FFTW_ESTIMATE);
 fftw_execute(p2);
 fftw_destroy_plan(p2);
 bigmouth_cal = true;
}
if(show_message2 == false && show_once2 == true
   && smallmouth_cal == false)
{
 reference2 = cvLoadImage("small_mouth_open.jpg", 0);
 FormatFilter(reference2, 0, 0, SSHEIGHT, SSWIDTH);
 ImageToArray(reference2, i3, 0, 0, SSHEIGHT, SSWIDTH);
 CountPixelsInRef(i3, SSHEIGHT, SSWIDTH, s_total);
 PadImage(pd2, i3, PADHEIGHT, PADWIDTH, SSHEIGHT, SSWIDTH);
 ArrayToFFTWFormat(pd2, in3, PADHEIGHT, PADWIDTH);
 p4 = fftw_plan_dft_2d(SIZE, SIZE, in3, out3, FFTW_FORWARD,
                    FFTW_ESTIMATE);
 fftw_execute(p4);
```

```
  fftw_destroy_plan(p4);
  smallmouth_cal = true;
 }
 if(show_message3 == false && show_once3 == true
   && calculations == false)
 {
  reference3 = cvLoadImage("closed_mouth.jpg", 0);
  FormatFilter(reference3, 0, 0, SSHEIGHT, SSWIDTH);
  ImageToArray(reference3, i4, 0, 0, SSHEIGHT, SSWIDTH);
  CountPixelsInRef(i4, SSHEIGHT, SSWIDTH, c_total);
  PadImage(pd3, i4, PADHEIGHT, PADWIDTH, SSHEIGHT, SSWIDTH);
  ArrayToFFTWFormat(pd3, in4, PADHEIGHT, PADWIDTH);
  p6 = fftw_plan_dft_2d(SIZE, SIZE, in4, out4, FFTW_FORWARD,
                          FFTW_ESTIMATE);
  fftw_execute(p6);
  fftw_destroy_plan(p6);
  calculations = true;
 }
 DrawRectangle(edges, startx - 1, starty - 1, SSHEIGHT + 2,
                SSWIDTH + 2, 200.0);
}
else
{
 FormatFilter(edges, startx - shift, starty, PADHEIGHT,
              PADWIDTH);
 GapFiller(edges,startx - shift, starty, gapnumber,
            PADHEIGHT, PADWIDTH);
 FormatFilter(edges, startx - shift, starty, PADHEIGHT,
              PADWIDTH);
```

```
if(largest_edge == false)
{
 EdgeWalker(edges, startx - shift, starty, edge_min_size,
             edge_max_size, PADHEIGHT, PADWIDTH, final_x,
             final_y, act_size);
 final_x.clear();
 final_y.clear();
}
else
{
 FindLargestEdge(edges, startx - shift, starty, PADHEIGHT,
                  PADWIDTH, 0, 1000, act_size);
}
ImageToArray(edges, i, startx - shift, starty, PADHEIGHT,
              PADWIDTH);
ArrayToFFTWFormat(i, in, PADHEIGHT, PADWIDTH);
p = fftw_plan_dft_2d(SIZE, SIZE, in, out,FFTW_FORWARD,
                      FFTW_ESTIMATE);
fftw_execute(p);
//Big Mouth
if(locate_bigmouth == true)
{
 NormalizeFFTW(out, out2, nor, PADHEIGHT, PADWIDTH);
 p3 = fftw_plan_dft_2d(SIZE, SIZE, nor, res, FFTW_BACKWARD,
                        FFTW_ESTIMATE);
 fftw_execute(p3);
 GetMaxValueFFTW(res, pos_x, pos_y, PADHEIGHT, PADWIDTH);
 ShiftImage(pd, map, pos_x, pos_y, PADHEIGHT, PADWIDTH, cex, cey);
 MakeBorders(top_b, bottom_b, left_b, right_b, SSHEIGHT,
SSWIDTH, PADHEIGHT, PADWIDTH, cex, cey);
 for(int thicken = 0; thicken < 2; thicken++)
 {
  ThickenImage(map, PADHEIGHT, PADWIDTH);
 }
 InvertPixelValues(map, PADHEIGHT, PADWIDTH);
 AND_Images(map, i, and, PADHEIGHT, PADWIDTH, unmat_b_count,
            top_b, bottom_b, right_b, left_b, removed);
 DetermineMouthExpression(unmat_b_count, act_size-removed, b_total,
                           bigmouth, matchpercentage, matchpercentage);
 fftw_destroy_plan(p3);
}
//Small mouth
if(locate_smallmouth == true)
{
 NormalizeFFTW(out, out3, nor, PADHEIGHT, PADWIDTH);
 p5 = fftw_plan_dft_2d(SIZE, SIZE, nor, res, FFTW_BACKWARD,
                        FFTW_ESTIMATE);
 fftw_execute(p5);
 GetMaxValueFFTW(res, pos_x2, pos_y2, PADHEIGHT, PADWIDTH);
 ShiftImage(pd2, map2, pos_x2, pos_y2, PADHEIGHT, PADWIDTH, cex, cey);
 MakeBorders(top_s, bottom_s, left_s, right_s, SSHEIGHT,
             SSWIDTH, PADHEIGHT, PADWIDTH, cex, cey);
 for(int thicken = 0; thicken < 2; thicken++)
 {
  ThickenImage(map2, PADHEIGHT, PADWIDTH);
 }
 InvertPixelValues(map2, PADHEIGHT, PADWIDTH);
 AND_Images(map2, i, and, PADHEIGHT, PADWIDTH, unmat_s_count,
            top_s, bottom_s, right_s, left_s, removed);
 DetermineMouthExpression(unmat_s_count, act_size-removed, s_total,
                          smallmouth, matchpercentage, matchpercentage);
 fftw_destroy_plan(p5);
}
//Closed mouth
if(locate_closedmouth == true)
{
 NormalizeFFTW(out, out4, nor, PADHEIGHT, PADWIDTH);
 p7 = fftw_plan_dft_2d(SIZE, SIZE, nor, res, FFTW_BACKWARD,
                       FFTW_ESTIMATE);
 fftw_execute(p7);
 GetMaxValueFFTW(res, pos_x3, pos_y3, PADHEIGHT, PADWIDTH);
 ShiftImage(pd3, map3, pos_x3, pos_y3, PADHEIGHT, PADWIDTH,
            cex, cey);
 MakeBorders(top_c, bottom_c, left_c, right_c, SSHEIGHT, SSWIDTH,
             PADHEIGHT, PADWIDTH, cex, cey);
 for(int thicken = 0; thicken < 2; thicken++)
 {
  ThickenImage(map3, PADHEIGHT, PADWIDTH);
 }
 InvertPixelValues(map3, PADHEIGHT, PADWIDTH);
 AND_Images(map3, i, and, PADHEIGHT, PADWIDTH, unmat_c_count,
            top_c, bottom_c, right_c, left_c, removed);
 DetermineMouthExpression(unmat_c_count, act_size-removed, c_total,
                          closedmouth, matchpercentage, matchpercentage);
 fftw_destroy_plan(p7);
}
if(bigmouth == true && locate_bigmouth == true)
```

```
{
 ShiftImage(pd, sf, pos_x, pos_y, PADHEIGHT, PADWIDTH, cex, cey);
 OverLapImage(sf, edges, startx - shift, starty, PADHEIGHT,
              PADWIDTH);
 expression_made = ``big mouth expression";
 cout << expression_made << endl;
}
else if(smallmouth == true && locate_smallmouth == true &&
        bigmouth == false)
{
 ShiftImage(pd2, sf, pos_x2, pos_y2, PADHEIGHT, PADWIDTH, cex, cey);
 OverLapImage(sf, edges, startx - shift, starty, PADHEIGHT,
              PADWIDTH);
 expression_made = ``small mouth expression";
 cout << expression_made << endl;
}
else if(closedmouth == true && locate_closedmouth == true &&
        smallmouth == false && bigmouth == false)
{
 ShiftImage(pd3, sf, pos_x3, pos_y3, PADHEIGHT, PADWIDTH, cex, cey);
 OverLapImage(sf, edges, startx - shift, starty, PADHEIGHT,
              PADWIDTH);
 expression_made = ``closed mouth expression";
 cout << expression_made << endl;
}
unmat_b_count = false;
unmat_s_count = false;
unmat_c_count = false;
bigmouth = false;
```

```
 smallmouth = false;
 closedmouth = false;
 fftw_destroy_plan(p);
 DrawRectangle(edges, startx - shift - 1, starty - 1,
               PADHEIGHT + 2, PADWIDTH + 2, 200.0);
}
cvShowImage(''Edged Image", edges);
if(live_image == true)
    {
     cvShowImage(''Image", frame);
    }
    if(shots_taken == true)
    {
    cvShowImage(''Big Mouth", reference);
    cvShowImage(''Small Mouth", reference2);
    cvShowImage(''Closed Mouth", reference3);
    }
key = cvWaitKey(10);
if( key == 27 )
{
 break;
}
else if( key == 32 && show_message1 == true)
{
 TakeSnapShot(edges, snap_shot, startx, starty, SSHEIGHT,
              SSWIDTH);
 if(!cvSaveImage(''big_mouth_open.jpg", snap_shot))
 {
  cout << ''Image did not save, big_mouth_open.jpg"
```

```
        << endl;
  }
  show_message1 = false;
  show_message2 = true;
}
else if( key == 32 && show_message2 == true)
{
  TakeSnapShot(edges, snap_shot, startx, starty,
               SSHEIGHT, SSWIDTH);
  if(!cvSaveImage(''small_mouth_open.jpg", snap_shot))
  {
   cout << ''Image did not save, small_mouth_open.jpg"
        << endl;
  }
  show_message2 = false;
  show_message3 = true;
}
else if( key == 32 && show_message3 == true)
{
  TakeSnapShot(edges, snap_shot, startx, starty,
               SSHEIGHT, SSWIDTH);
  if(!cvSaveImage(''closed_mouth.jpg", snap_shot))
  {
   cout << ''Image did not save, closed_mouth.jpg"
        << endl;
  }
  show_message3 = false;
  begin_execution = true;
}
```

```
else if(key == 32 && begin_execution == true)
{
 shots_taken = true;
}
else if(key == 119 || key == 87) //Move frame up,
                                 //press w key
{
 if(startx - 10 > PADHEIGHT)
 {
  startx = startx - 10;
 }
}
else if(key == 100 || key == 68) //Move frame right ,
                                 //press d key
{
 if(starty + 10 < width - PADWIDTH)
 {
  starty = starty + 10;
 }
}
else if(key == 97 || key == 65) //Move frame left ,
                                //press a key
{
 if(starty - 10 >= 2)
 {
  starty = starty - 10;
 }
}
else if(key == 115 || key == 83) //Move frame down,
                                //press s key
{
 if(startx + 10 < height - PADHEIGHT)
 {
  startx = startx + 10;
 }
}
else if(key == 71 || key == 103) //Increase gap fill size,
                                //press g key
{
 if(gapnumber < 3)
 {
  gapnumber++;
  cout << ''Current Gap Size: " << gapnumber << endl;
 }
}
else if(key == 98 || key == 66) //Decrease gap fill size,
                               //press b key
{
 if(gapnumber > 1)
 {
  gapnumber--;
  cout << ''Current Gap Size: " << gapnumber << endl;
 }
}
else if(key == 85 || key == 117) //Increase match
                                //percentage, press
                                //u key
{
 if(matchpercentage < 100)
 {
  matchpercentage = matchpercentage + 10;
  cout << "Current Match Percentage: " <<
           matchpercentage << endl;
 }
}
else if(key == 89 || key == 121) //Decrease match
                                  //percentage, press
                                  //y key
{
 if(matchpercentage > 0)
 {
  matchpercentage = matchpercentage - 10;
  cout << "Current Match Percentage: " <<
           matchpercentage << endl;
 }
}
else if(key == 99 || key == 67) //Take edge capture of
                                 //screen and cam, press
                                 //c key
{
 if(!cvSaveImage(''original.jpg", original))
 {
  cout << ''Image did not save, original.jpg" << endl;
 }
 if(!cvSaveImage(''cam.jpg", frame))
 {
  cout << ''Image did not save, cam.jpg" << endl;
 }
}
else if(key == 118 || key == 86) //Take cam capture
                                 //of screen, press v key
{
 if(!cvSaveImage(''cam.jpg", frame))
 {
  cout << ''Image did not save, cam.jpg" << endl;
 }
}
else if(key == 69 || key == 101) //display mouth expressions
                                 //searched for, press e key
{
 cout << endl;
 cout << ''Current mouth expressions being searched for:" << endl;
 if(locate_bigmouth == true)
 {
  cout << ''big mouth expression" << endl;
 }
 if(locate_smallmouth == true)
 {
  cout << ''small mouth expression" << endl;
 }
 if(locate_closedmouth == true)
 {
  cout << ''closed mouth expression" << endl;
 }
 if(locate_bigmouth == false && locate_smallmouth == false &&
    locate_closedmouth == false)
 {
  cout << ''None'' << endl;
 }
 cout << endl;
}
else if(key == 74 || key == 106) //decrease min edge size,
                                 //press j key
{
 if(edge_min_size > 1 && largest_edge == false)
 {
  edge_min_size = edge_min_size - 10;
  cout << ''Minimum edge size: '' << edge_min_size << '' ''
 << ''Maximum edge size: '' << edge_max_size << endl;
 }
}
else if(key == 78 || key == 110) //increase min edge size,
                                 //press n key
{
 if(shots_taken == false && largest_edge == false)
 {
  if(edge_min_size + 10 < edge_max_size)
  {
   edge_min_size = edge_min_size + 10;
   cout << ''Minimum edge size: '' << edge_min_size << '' ''
        << ''Maximum edge size: '' << edge_max_size << endl;
  }
 }
 else if(shots_taken == true && largest_edge == false)
 {
  if(edge_min_size + 10 < edge_max_size)
  {
   edge_min_size = edge_min_size + 10;
   cout << ``Minimum edge size: " << edge_min_size << `` "
        << ``Maximum edge size: " << edge_max_size << endl;
  }
 }
}
else if(key == 75 || key == 107) //decrease max edge size,
                                  //press k key
{
 if(edge_max_size - 10 > edge_min_size
    && largest_edge == false)
 {
  edge_max_size = edge_max_size - 10;
  cout << ``Minimum edge size: " << edge_min_size << `` "
       << ``Maximum edge size: " << edge_max_size << endl;
 }
}
else if(key == 77 || key == 109) //increase max edge size,
                                  //press m key
{
 if(shots_taken == false && largest_edge == false)
 {
  if(edge_max_size + 10 <= SSHEIGHT*SSWIDTH)
  {
   edge_max_size = edge_max_size + 10;
   cout << ``Minimum edge size: " << edge_min_size << `` "
        << ``Maximum edge size: " << edge_max_size << endl;
  }
 }
 else if(shots_taken == true && largest_edge == false)
 {
  if(edge_max_size + 10 <= PADHEIGHT*PADWIDTH)
  {
   edge_max_size = edge_max_size + 10;
   cout << ''Minimum edge size: " << edge_min_size << '' "
        << ''Maximum edge size: " << edge_max_size << endl;
  }
 }
}
else if(key == 76 || key == 108) //switch between largest
                                 //edge or edges in range,
                                 //press l key
{
 if(largest_edge == false)
 {
  largest_edge = true;
  cout << ''Largest Edge Mode On" << endl;
 }
 else
 {
  largest_edge = false;
  cout << ''Edges in Certain Size Range Mode On" << endl;
 }
}
else if(key == 70 || key == 102) //switch on or off live image,
                                 //press f key
```

```
{
 if(live_image == false)
 {
  live_image = true;
  cvNamedWindow( ''Image", CV_WINDOW_AUTOSIZE );
 }
 else
 {
  live_image = false;
  cvDestroyWindow( ''Image" );
 }
}
else if(key == 82 || key == 114) //retake reference images,
                                 //press r key
{
 show_once1 = false;
 show_once2 = false;
 show_once3 = false;
 show_once4 = false;
 show_message1 = true;
 show_message2 = false;
 show_message3 = false;
 shots_taken = false;
 begin_execution = false;
 calculations = false;
 bigmouth_cal = false;
 smallmouth_cal = false;
 locate_bigmouth = true;
 locate_smallmouth = true;
 locate_closedmouth = true;
}
else if(key == 73 || key == 105) //turn on/off bigmouth
                                  //expression search,
                                  //press i key
{
 if(locate_bigmouth == true)
 {
  locate_bigmouth = false;
  cout << ``Big mouth expression search: off" << endl;
 }
 else
 {
  locate_bigmouth = true;
  cout << ``Big mouth expression search: on" << endl;
 }
}
else if(key == 79 || key == 111) //turn on/off smallmouth
                                  //expression search,
                                  //press o key
{
 if(locate_smallmouth == true)
 {
  locate_smallmouth = false;
  cout << ``Small mouth expression search: off" << endl;
 }
 else
 {
  locate_smallmouth = true;
```

```
    cout << ''Small mouth expression search: on" << endl;
   }
  }
  else if(key == 80 || key == 112) //turn on/off closedmouth
                                   //expression search,
                                   //press p key
  {
   if(locate_closedmouth == true)
   {
    locate_closedmouth = false;
    cout << ''Closed mouth expression search: off" << endl;
   }
   else
   {
    locate_closedmouth = true;
    cout << ''Closed mouth expression search: on" << endl;
   }
  }
  cvReleaseImage(&gray);
  cvReleaseImage(&edges);
  cvReleaseImage(&original);
}
  fftw_free(in); fftw_free(out); fftw_free(in2); fftw_free(out2);
  fftw_free(in3); fftw_free(out3); fftw_free(in4); fftw_free(out4);
  fftw_free(nor); fftw_free(res);
  for(int x = 0; x < 2; x++)
  {
   for(int y = 0; y < PADHEIGHT; y++)
   {
```

```
   delete [] i[x][y];
   delete [] pd[x][y];
   delete [] pd2[x][y];
   delete [] pd3[x][y];
   delete [] sf[x][y];
   delete [] map[x][y];
   delete [] map2[x][y];
   delete [] map3[x][y];
   delete [] and[x][y];
  }
  delete [] i[x];
  delete [] pd[x];
  delete [] pd2[x];
  delete [] pd3[x];
  delete [] sf[x];
  delete [] map[x];
  delete [] map2[x];
  delete [] map3[x];
  delete [] and[x];
 }
 delete [] i;
 delete [] pd;
 delete [] pd2;
 delete [] pd3;
 delete [] sf;
 delete [] map;
 delete [] map2;
 delete [] map3;
 delete [] and;
```

```
    for(int x = 0; x < 2; x++)
    {
     for(int y = 0; y < SSHEIGHT; y++)
     {
      delete [] i2[x][y];
      delete [] i3[x][y];
      delete [] i4[x][y];
     }
     delete [] i2[x];
     delete [] i3[x];
     delete [] i4[x];
    }
    delete [] i2;
    delete [] i3;
    delete [] i4;
    cvReleaseImage(&snap_shot);
    cvReleaseImage(&reference);
    cvReleaseImage(&reference2);
    cvReleaseImage(&reference3);
    cvReleaseCapture( &capture );
    cvDestroyAllWindows();
    return 0;
}
```

REFERENCES

[1] Access to Disability Data. http://www.infouse.com/disabilitydata/disability/, 2010.

[2] Jain R, Kasturi R, Schunck B. Machine Vision. *McGraw Hill*, 169-172, 1995.

[3] Kumar V, Rahman T, Krovi, V. Assistive Devices For People With Motor Disabilities. *Wiley Encyclopedia of Electrical and Electronics Engineering*, 1997.

[4] Ramey RL, Aylor JH, Williams RD. Microcomputer-Aided Eating for the Severely Handicapped. *Computer*, 12:54-61, 1979.

[5] Developmental Disabilities. http://www.cdc.gov/ncbddd/dd/cp2.htm, 2010.

[6] Spastic Quadripelgia. http://www.brainandspinalcord.org/cerebral-palsy/types/spastic-quadriplegia.html, 2010.

[7] Percutaneous Endoscopic Gastrostomy. http://www.gi.org/patients/gihealth/peg.asp, 2010.

[8] Ohara E, Ken'ichi Y, Horihata S, Aoki T, Nishimoto Y. Development of Tremor-Suppression Filter for Meal-Assist Robot. *Third Joint Eurohaptics Conference and Symposium on Haptic Interfaces*, 2009.

[9] Ju J, Shin Y, Kim E. Intelligent Wheelchair (IW) Interface using Face and Mouth recognition. *ACM*, 2009.

[10] Bergasa LM , Mazo M, Gardel A , Barea R, Boquete L. Commands Generation by Face Movements Applied to the Guidance of a Wheelchair for Handicapped People. *IEEE*, 2000.

[11] Russ, J. The Image Processing Handbook: Second Edition. *CRC Press*, 1995.

[12] Transforms: Fourier Transform. http://homepages.inf.ed.ac.uk/rbf/HIPR2/fourier.htm, 2003.

[13] Yoo, Y. Tutorial on Fourier Theory. http://www.cs.otago.ac.nz/cosc453/student_tutorials/fourier_analysis.pdf, 2001.

[14] Cerebral Palsy: Hope Through Research. http://www.ninds.nih.gov/disorders/cerebral_palsy/detail_cerebral_palsy.htm?css=print, 2010.

[15] Topping, M. HANDY 1: A Robotic System to Assist the Severely Disabled. *Knowledge Enterprise*, 2002.

[16] Regalbuto, M. Toward a practical mobile robotic aid system for people with severe physical disabilities. *Journal of Rehabilitation Research and Development*, 2002.

[17] Takahashi Y, Kobayashi T. Upper Limb Motion Assist Robot. *ICORR*, 1999.

[18] Takahashi Y, Hasegawa N. Human Interface Using PC Display With Head Pointing Device for the Eating Assist Robot and Emotional Evaluation by GSR Sensor. *IEEE*, 2001.

[19] Pourmohammadali H, Kofman J, Khajepour A. Design of a Multiple-user Intelligent Feeding Robot for Elderly and Disabled People. *University of Waterloo*, 2008.

[20] Xie H, Hicks N, Keller R, Huang H, Kreinovich V. An IDL/ENVI implementation of the FFT-based algorithm for automatic image registration. *Computers and Geosciences*, 2003.

[21] Banerjee R, Kundu M, Banerjee A. FEEDING for the child with cerebral palsy. *IICP*, 1995.

[22] Maini R, Aggarwal H. Study and Comparison of Various Image Detection Techniques. *International Journal of Image Processing.*

[23] Ding L, Goshtasby A. On the Canny edge detector. *Pattern Recognition*, 2001.

[24] Edge detection. http://www.cse.unr.edu/~bebis/CS791E/Notes/EdgeDetection.pdf

[25] Wang B, ShaoSheng F. An improved CANNY edge detection algorithm. *2009 Second International Workshop on Computer Science and Engineering*, 2009.

[26] Er-sen L, Shu-long Z, Bao-shan Z, Yong Z, Chao-gui X, Li-hua S. An Adaptive Edge-Detection Method Based on the Canny Operator. *2009 International Conference on Environmental Science and Information Application Technology*, 2009.

[27] Wang L. Comparison for Edge Detection of Colony Images. *IJCSNS International Journal of Computer Science and Network Security*, 2006.

[28] Quadriplegia and Cerebral Palsy. http://www.cerebralpalsysource.com/Types_of _CP/quadriplegia_cp/index.html, 2005.