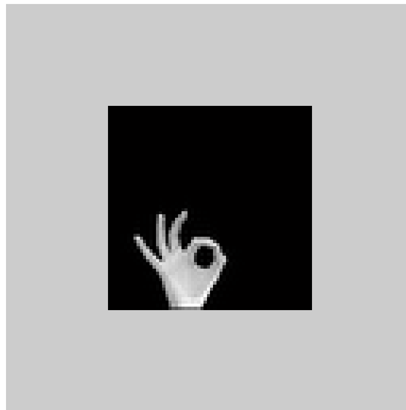
18-551:
Digital Communication and Signal Processing
System Design

Spring 2003

Handslation

A Translation of American Sign Language into Text

Submission Date: May 5, 2003



Group 3

Steve Benson (sbenson@)

Paul Harvin (pharvin@)

Emily Lauffer (elauffer@)

Kristen Matlock (kmatlock@)

Table of Contents

Background and Motivation 3

Previous Work 3

Development of Test Images, Training Images, and Filters 4

Special Letters 6

Filter Design 6

Algorithm Overview 7

Implementation 8

Code Reuse 8

Speed 9

Still Image Issues 9

Image Pre-Processing (Still Image Implementation) 10

Initial Still Image Results 11

Final Still Image Results 13

Possible Improvements 13

Video Capture Implementation 14

Image Pre-Processing for Video Capture Implementation 14

Special Letters and Video 16

EVM Optimizations 16

References 17

Background and Motivation

There are around 350,000 profoundly deaf individuals in the United States, making American Sign Language the fourth most used language in the country. While many people use sign language as their main form of communication, there are still many people who would be unable to understand sign language. Our group wanted to create a project that would be able to help the hearing impaired communicate with individuals who do not know sign language. Without being able to communicate with other people, a hearing impaired person would have great difficulty interacting with the rest of society.

While having the project be able to translate the entire American Sign Language dictionary would be too complex of a task to complete in one semester, we say dealing with just the 26 letters of the fingerspelling alphabet as a more realistic goal. The fingerspelling alphabet is composed of 26 unique hand positions to represent the 26 letters of the English alphabet. Two of the letters even involve some motion.

Our project would take in an image of a person's hand in one of these 26 positions, and translate it into the matching letter. While two of the letters do involve motion, we notice that their ending positions were unique and decided to use these as the position used in the translation.

In order to further make our project into a task that could be completed in one semester, we decided to use a standard black background for the images. Also, the camera would always be at the same angle relative to the signer. Our final constraint was that the system would be user dependent. One group member was chosen as the signer. All test and training images would be taken of her hand.

We decided we first needed to get our project running with still images as the input, but wanted to eventually have it be able to handle a video input with the results being produced in real time.

Previous Work

Our project uses image processing techniques such as fast fourier transforms and correlations like many previous 18-551 projects, but none of them are very similar to our project other than that.

We had trouble finding resources on the web as well. Many implementations of sign-language translation involved the use of a glove with sensors on the fingers, that measured the movements and angles of the hand

symbols in order to detect the sign. One researcher used 2-D images with a colored glove, where each finger was a different color. Then he used principal component analysis to determine the sign. We also found a report from a class similar to 18-551 at Cornell, where students tried to use skeletonizing algorithms and Sobel maps to interpret 2-D images of hand gestures. The students reported that their method was unsuccessful. Another attempt that we looked at used 3-D images to detect and classify sign language symbols. A number of different algorithms were tried, including Chamfer Distance, which is a type of Euclidean distance transform, edge orientation histograms, moment-based matching, and combinations of the afore mentioned. We could not find any sources that used 2-D images and correlations with filters to determine sign language symbols, therefore our project is very unique.

Development of Test Images, Training Images, and Filters

We realized the best way to make our translation system was to use a correlation filter. A correlation filter would not be sensitive to changes in the position of the hand in an image.

Since it seemed like the easiest way to distinguish between letters was by the edges in the image (or the high frequencies in its FFT) we looked into using either a MACE or MINACE filter.

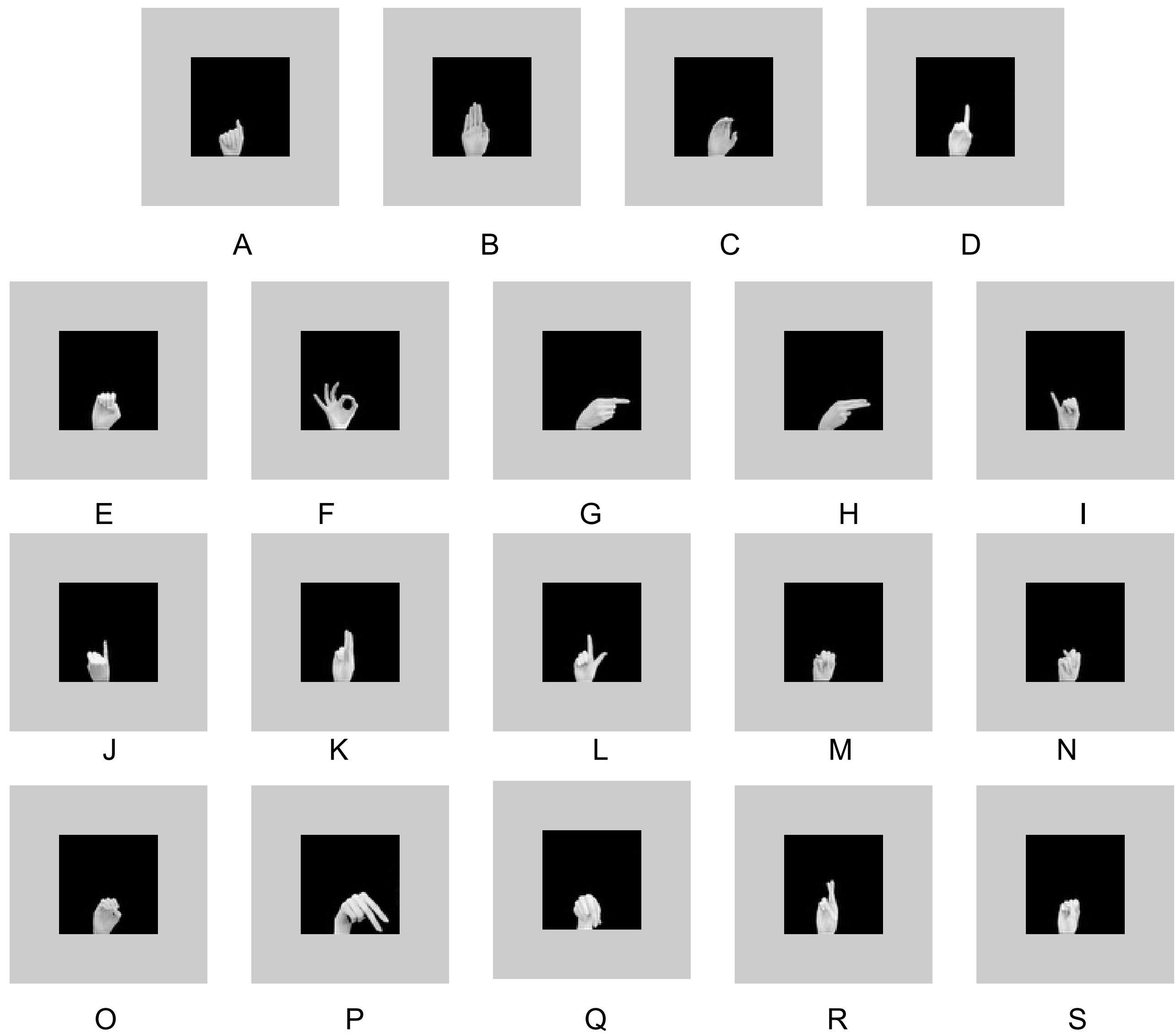
When an image is correlated with its filter, the output should be a clear peak at the location of the image. However, there are also side lobes along with this peak. The MACE or Minimum Average Correlation Energy filter attempts to minimize these side lobs by minimizing the correlation plane energy. However, it appeared that there was a problem with using the MACE filter. This filter is highly sensitive to the edges of the image, which is what we desired. In our input images, though, there were variations in the image that this filter could not handle. When a sign was made, the exact positions of each finger would not always be same. There would be slight variations even though it was the same letter. We needed a filter that could take into account this problem.

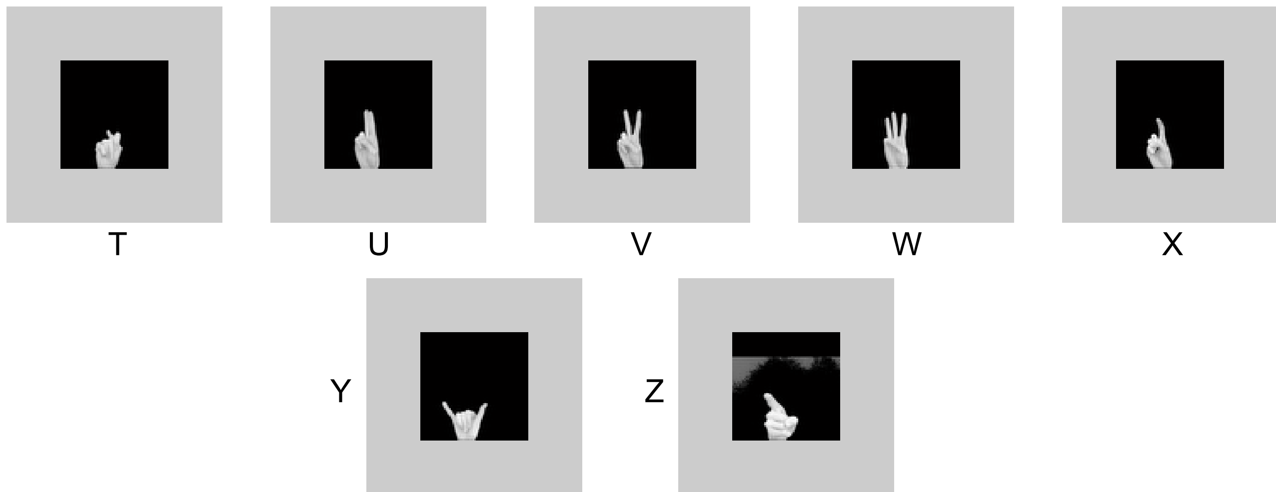
We realized that the MINACE (or Minimum Noise and Average Correlation plane Energy) filter would be far superior. This filter was sensitive to the edges, or high frequencies, or the images just like the MACE was, but it could also be set to handle various amounts of noise.

Our group decided to build 26 MINACE filters (one for each letter of the alphabet.) To do this, we first needed a set of training images. We chose to use

10 images of each letter in the training set. These images were taken using digital camera. The image was then downsampled to 64 x 64 pixels. We chose a number of pixels that was a factor of four so we could eventually take advantage of the radix4 FFT on the EVM.

The training images were taken in front of the black background. In between each image, the signer made sure to move her hand and then remake the sign in order to represent the slight variations that could occur when signing a letter. Below is a sample of images from our training set.





Additional images would then be taken to use as a test set. These images would not be used in building the filters, but would still be taken against the same black background. We took an additional 5 images of each sign to later use as the test set.

Special Letters

Two of the letters, j, and z, have movement. We were unsure of how to detect these letters because at each stage of their movements, a still image of the hand pose would look different. It would be a challenge to match it with a filter. We decided to solve the problem by using the final hand pose of each of the letters as that letter. In other words, the final hand pose for j was what we used to take all of the images for our j filter. These final hand poses worked because they were unique.

Filter Design

The first step to building the MINACE filter was to lexicographically order each of the FFTs of the ten training images, and then put each result into a column of a matrix X. First, we took the 2D FFT of each image, and then to lexicographically order the images, we arranged each column of the image one after another into a column matrix. Since each image was 64 x 64, and there were ten images, the final result was a matrix with 10 columns and 4096 rows.

After this was done, we needed to create a diagonal matrix with the average power spectrum of each training image along its diagonal. The average power spectrum is found by taking the average of the magnitude squared of each column of X. This value was then placed into the diagonal of a new 10 x 10 matrix D.

Next, we needed to create a vector N that was also a diagonal matrix. This one, however, contained along its diagonal the value of sigma squared that we wished to use as our white Gaussian noise model. We chose a sigma squared of 100.

The diagonal matrix to be used in the final algorithm would be a weighted sum of the diagonal matrices D and X. The would be weighted with a coefficient c using the following equation:

$$T = cD + (1 - c)N$$

Our group initially chose a c value of 0.7. This c value could later be changed after initial testing if the weighting of noise and average power spectrum did not produce the desired results.

The final component we needed to create our filters was a vector u which would contain the desired peak values in the output for each training images. Our vector was simply a column of 10 ones, since we wanted any training image correlated with the filter for that particular letter to produce a peak of one.

The equation for our filter would now be the following:

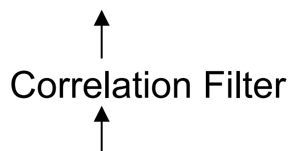
$$H = T^{-1} X (X^{+} T^{-1} X)^{-1} u$$

Where T is the weighted sum we just created, X is our lexicographically ordered training images, u is our vector of 10 ones, and H is our final filter. We then created a filter for each of the 26 letters of the alphabet.

Algorithm Overview

After our filters were complete, we then needed to test them by using the following algorithm:

Test Image → FFT → Multiply → IFFT → Correlation



Filter Design ← Training Images

Our filters were created in the frequency domain, so we took the FFT of the test image, multiplied the two, and took the IFFT. This correlated the test image with our filter. We then found the peak of the result. We chose the corresponding letter of the correlation that produced the highest peak to be our match. One of the weaknesses of our current project is that we did not account for the case where there is no match. We could have set some kind of threshold for a match, where if there was no peak high enough, no match would be produced.

Implementation

We used the EVM to FFT the incoming test image, multiply each filter with the FFT of the image, IFFT the multiply result, and finally to find the peak of each of the IFFT results. The entire 64 x 64 incoming test image was transferred on-chip, the FFT was performed, and the 64 x 128, complex interleaved, FFT result was stored off-chip (SBSRAM) out of necessity.

All 26 filters were also stored in the SBSRAM off-chip. We then asynchronously loaded alternate lines of the filter and the image FFT. We multiplied the two line by line, and stored the result line by line in the 64 x 128, complex interleaved, variable called multiplyresult. Multiplyresult was stored on-chip although it was extremely large, because we accessed its elements repeatedly. The IFFT was accomplished by using the same FFT code that was used to FFT the incoming image. The result of the IFFT was also stored in multiplyresult. The last step was to take the absolute value, which required an extra variable since the result would be a different size than multiplyresult. It is called Out, and is the only other variable that is stored off-chip (SBSRAM). We find the peak of Out by looking for its largest element. Each of the 26 peak values that correspond to each of the 26 letter filters are stored in the 26 element array, outspace, which is on-chip. Outspace was transferred back to the PC, where we found its largest element, converted the index of the largest element to a character, and outputed that character to a file as our match.

Code

We used the c-callable 1D FFT function given to us in lab two, and combined it with our own code to perform the 2D FFT and IFFT. We wrote our own complex interleaved multiply function, and code to find the peak of the IFFT. We used code similar to that used in lab 3, step 6, which utilized asynchronous DMA transfers. We also used the skeleton from Lab 3 for PC<->EVM communications.

Speed

We had two variables that were off-chip, which probably cost us some speed. PC<->EVM communications were the slowest component of our system. After optimizing our code size we were able to fit all of our EVM code on-chip, and our program runs quite fast. The profiler gave the following cycles for our EVM functions:

- 2D FFT: 224425 cycles
- 2D IFFT: 224425 cycles
- Filter Multiply: 37751 cycles
- Find Peak: 84270 cycles

Still Image Issues

Most of the issues we encountered on the EVM side of things involved not having enough on-chip memory storage space. We initially started out by having entire images and filters transferred asynchronously and calculated on, before we realized that we needed more room due to intermediate variables and complex interleaved variables. Another serious problem that we ran into was that the skeleton code that we were using from lab 3 read in files and stored them as character arrays. We needed to read in floating point values, which take up 4 times the number of bytes as characters do, contributing to our problem with memory storage space. Finally we figured out the right sizes to transfer in if we were using characters (4x more), and the correct method for casting (first to a long, then to a float). We also ran into testing issues because Matlab can store a complex number in one element, while our code stored a complex number in two elements, so that everything was complex interleaved. In order to make sure our C code was correct at intermediary points, we had to write similar code in Matlab, and could not use some of it's simplified functions. For example, we could not simply use the symbol * to multiply two arrays. We had to write a complex interleaved multiply function in Matlab and compare that to our C code. Many times we were not even sure if our Matlab code was correct. Also, one of our initial Matlab test files created a single filter and correlated it with 26 test images. This gave us accuracy rates as high as 100%. But that was not completely correct because our final C code correlated 26 filters with 1 test image. This is one of the reasons that we decided to abandon trying to integrate other methods, such as zooming or sub-grouping, with our filter method – we were already getting good results.

Image Pre-Processing (Still Image Implementation)

Images used for building the filters and all test images are run through a Matlab script that does the following image preprocessing operations:

- Converts the 24-bit RGB image to an 8-bit grayscale image
- Blackens the background
- Finds the hand in the frame
- Normalizes the image so that the hand is always in the same place and of the same proportions
- Downsamples the image to create a 64x64 pixel version to use in the correlation
- Modifies the full sized image to include guiding lines that will allow to know that their hand is being found properly and that will assist the user in repositioning their hand so that it can be found more easily

Grayscale Conversion

When we load an image into the Matlab code, we use Matlab's ind2gray function to convert a 24-bit RGB bitmap into a grayscale image. In this grayscale image, pixel values are in the range of 0 (black) to 1 (white).

Blackening the Background

Even though we provide a black background for video capture of the hand, lighting variations can still interfere with the correlation (background appears as different shades of black/gray). In order to prevent these interferences, we set all pixels that are below a lightness threshold (0.34) to 0 (black).

Find the Hand in the Frame

We find the hand in the frame by first examining the each row of the image and finding the width of the hand at each row. We find the width of the hand by comparing each pixel to a lightness threshold (0.34) and deciding whether a pixel is background or skin by whether it is below or above that threshold (0.34). We generate a width of the hand for each row by counting the number of pixels above the lightness threshold. These widths are stored in a vector.

To handle any small variations in the hand width due to the high resolution of the image, we smooth the vector of the widths by a moving averaging of adjacent pixels.

We find the location of the wrist by searching the width vector from the bottom of the image up (following the arm) until we see a sharp increase in the width. This sharp increase in width corresponds with the row at which the wrist is located.

Normalize the Image so the Hand is Always in the Same Place
Once the location of the wrist has been found, the image is "normalized" by:

- cropping the image below the wrist
- cropping or padding to make sure there is 2 * (wrist width) amount of space to the left of the wrist, 3 * (wrist width) amount of space to the right of the wrist, and 6*(wrist width) amount of space above the row of the wrist

This ensures that the image to be processed will be square, that the base (wrist) of the hand will always be in the same place in the image, that the entire hand will be in the image, and that the hand will be of the same proportions.

Downsample the Image to 64x64 Pixels

Downsampling is of the image is done using Matlab's resample function in both the horizontal and vertical directions on the matrix of pixel values

Initial Still Image Results:

These are the results that we obtained initially for our still images. Our accuracy was only 41%, which was disappointingly low. We realized that all of the images that were in the filters were very similar, which would explain why it would not match a test image of the same letter that had more variation. The entire purpose of the filter is to include multiple different images so that our test image will be recognized despite small variations. Otherwise we could simply convolve with a single image, which is essentially what our filter was doing, since all the pictures that comprised it were almost identical. Based on this, we guessed that if we exchanged some of the images that were already in the filters, for images with more variation, we might increase our accuracy. We were correct. Our accuracy went up to a respectable 76%.

Initial Still Image Results (Table):

Letter	Number Tested	Number Correct	
A	1	1	
B	1	1	
C	1	0	
D	2	1	
E	1	1	
F	1	1	
G	5	0	
H	1	1	
I	1	1	
J	1	1	
K	4	1	
L	6	1	
M	2	1	
N	2	1	
O	1	1	
P	3	1	
Q	3	1	
R	6	1	
S	1	1	
T	1	1	
U	1	1	
V	3	1	
W	1	1	
X	5	0	
Y	1	1	
Z	1	1	
Total	56	23	**41%**

Final Still Image Results:

These are our final still image results. None of our test images were included in the filters, which would automatically produce a match.

Letter	Number Tested	Number Correct	
A	2	2	
B	2	2	
C	1	1	
D	3	2	
E	2	2	
F	2	2	
G	4	4	
H	2	2	
I	2	2	
J	2	2	
K	3	3	
L	4	3	
M	3	1	
N	3	2	
O	2	2	
P	2	2	
Q	5	3	
R	6	3	
S	3	1	
T	3	2	
U	2	2	
V	4	3	
W	2	2	
X	4	1	
Y	2	2	
Z	2	2	
Total	72	55	**76.39%**

Possible Improvements:

Designing filters with more than 10 images, and more variation in those images is one way that we thought we could further improve our accuracy. We also realized from looking at a matching image (one that the filter detected), and a non-matching image, that there was very little difference between the two, at least to the naked eye. This leads us to believe that a higher resolution is necessary to improve performance.

Video Capture Implementation : Camera Interaction

In the video capture implementation of this project, we used a Logitech QuickCam VC camera that communicated with the computer via a parallel port interface. This camera communicated with our program via supplied drivers and via Microsoft's *Video for Windows* version 1.1e support software. Our C program was based on the framework of a program called *CapTest* that was provided in the *Video for Windows Development Kit* version 1.1e. This program provided the ability to capture AVI video and still frames to disk. It also provided a preview window so that the user could see what was seen by the camera in real time. We modified the *CapTest* program in so that each incoming video frame from the camera, on its way to the preview windows would be rerouted through our own image pre-processing:

- Convert the 24-bit RGB frame to an 8-bit grayscale image
- Blackening the background
- Find the hand in the frame
- Normalize the image so that the hand is always in the same place and of the same proportions
- Downsample the image to create a 64x64 pixel version to use in the correlation
- Modify the full sized image to include guiding lines that will allow to know that their hand is being found properly and that will assist the user in repositioning their hand so that it can be found more easily

Image Pre-Processing for Video Capture Implementation

Grayscale Conversion

When a video frame is received from the camera, it is stored in a frame buffer in the format of a 320 (width) x 240 (height) pixel frame buffer where each pixel is a 24-bit structure (8-bits for red, 8-bits for green, 8 bits for blue). We convert the image to grayscale by simply ignoring the red and blue components, as their effect on the grayscale version of the image is minimal. This provides use with a 320 x 240 pixel image at 8 bits per pixel.

Blackening the Background

Even though we provide a black background for video capture of the hand, lighting variations can still interfere with the correlation (background appears as different shades of black/gray). In order to prevent these interferences, we set all pixels that are below a lightness threshold to 0 (black).

Find the Hand in the Frame

We find the hand in the frame by first examining the each row of the image and finding the width of the hand at each row. We find the width of the hand by comparing each pixel to a lightness threshold and deciding whether a pixel is background or skin by whether it is below or above that threshold. For each row, we find the first pixel in the row that is above the threshold and store that as a location of the left edge of the hand for that row. For each row, we also find the last pixel in the row that is above the threshold and store that as a location of the right edge of the hand for that row.

Once we have the vectors of left and right edges of the hand, we find the location of the wrist by searching from the bottom of the image up (following the arm) until we see a sharp increase in the width. This sharp increase in width corresponds with the location of the wrist.

Normalize the Image so the Hand is Always in the Same Place

Once the location of the wrist has been found, the image is "normalized" by:

- cropping the image below the wrist
- cropping or padding to make sure there is 2 * (wrist width) amount of space to the left of the wrist, 3 * (wrist width) amount of space to the right of the wrist, and 6*(wrist width) amount of space above the row of the wrist

This ensures that the image to be processed will be square, that the base (wrist) of the hand will always be in the same place in the image, that the entire hand will be in the image, and that the hand will be of the same proportions.

Downsample the Image to 64x64 Pixels

Downsampling in the video capture implementation is done using a publicly available image resize algorithm that utilizes bilinear filtering in order to make sure new pixel values are generated using the correct old pixels and to minimize error caused by downsampling. The downsampling algorithm was posted by Christian Graus to the website, **The Code Project.** We use the algorithm to downsample the normalized square image to 64x64 pixels.

Modify the full sized image to include guiding lines

In the video capture implementation, the incoming frame, after being converted to grayscale is modified so that guiding lines are placed at where the image pre-processing code found the base of the wrist and at the points where the image would be cropped to the left, right and top by the normalizing code. After these lines are added, the image is converted back RGB (with only the green component non-zero) and stored again in the preview frame buffer.

Special Letters and Video

The cases of j and z are different with respect to movement. Z was not a problem because even though the hand moves, the hand pose is the same for that entire movement. Our code focused on the hand, caught an image of it, and normalized it. Therefore, it did not matter at what point in the movement the hand was in, only the hand pose. In this way z could be detected. J was a different story. J involved rotation of the hand. The beginnning pose looked like i, and the ending pose was what we had labeled "j". The original plan when we were intent on using motion detection to determine a change in letters, was that if we detected an I before a j, we would throw out the i and simply call it a j. When we ended up not being able to use motion detection, and used timing to determine when a new letter was being signed, Emily had to hold each symbol for a few seconds before moving to a new one. This allowed her to sign the j symbol and hold the final pose, which could be detected by the filters just like all the other letters.

EVM optimizations

The final frame rate we achieved was 1fps. This fell short of our goal of 6 fps, however we felt that this was OK since this gave the user enough time to sign a letter and wait for the image to be captured. If the frame rate was much faster we would have introduced more problems to deal with, such as determining the start and end of letters, and when to count double letters. Also, we would have had to do a better job at throwing out images we found in between letters.
We did use some optimization techniques to reach a 1 fps frame rate. First, we made sure our EVM code was stored on chip. By storing the code on chip instead of in SBSRAM, we were able to improve our frame rate by a factor of four. So, initially we were able to process 1 frame every 4 seconds, but by storing our EVM code on chip we were able to improve this to processing 1 frame every second. We also tried to minimize the amount of data we transferred between the EVM and the PC. When the program starts up, we load the filters onto the EVM so all that needs to be transferred between the EVM and PC is our 64x64 image and the result array.

References

"Advanced Distortion-Invariant MACE Filters," Applied Optics, Vol. 31, No. 8, 10 March 1992, pp. 1109-1116 (Casasent, Ravichandran).

· This paper describes a MACE filter. We used this in order to begin understanding the MINACE filter.

M. Savvides, B.V.K. Vijaya Kumar, and P. Khosla. Face Verification using Correlation Filters. http://www.ece.cmu.edu/~marlene/kumar/Biometrics_AutoID.pdf.

· This paper also describes the MACE filter.

"Minimum Noise and Correlation Energy (MINACE) Optical Correlation Filter," Applied Optics, Vol. 31, pp. 1823-1833, 10 April 1992 (Casasent, Ravichandran).

· This paper describes a MINACE filter.

D. Casasent. Advanced Distortion Invariant Filters. Class Notes.

· We used this paper to figure out how exactly to build the MINACE filter.

V. Athitsos and S. Sclaroff. An Appearance-Based Framework for 3D Hand Shape Classification and Camera Viewpoint Estimation. In *Fifth IEEE International Conference on Automatic Face and Gesture Recognition*, Washington, D.C., 2002.

· This paper talked about the chamfer distance, which after further research we have decided not to use.

Microsoft's *Video For Windows Development Kit* v1.1e

- ftp://ftp.microsoft.com/developr/drg/Multimedia/Jumpstart/VfW11e/.

Graus, Christian. Image Processing for Dummies with C# and GDI+ Part 4 - Bilinear Filters and Resizing. The Code Project, 15 April 2002.

- http://www.codeproject.com/cs/media/imageprocessing4.asp

Appendix A - Commented Image Pre-Processing Matlab Code for Still Image Implementation

```
function g = g3_imgnorm( g )
% G3_IMGNORM Normalize (Image Size, Hand Size, Hand Location) in a Grayscale Image
%   Inputs:
%       g = image matrix in grayscale format (pixels 0 to 1)
%
%   Ouput:
%       normalized grayscale image
%   Group 3 : 18-551 Spring 2003
%
%   $Log: g3_imgnorm.m,v $
%   Revision 1.4  2003/03/16 23:05:39  kmatlock
%   Changed the size of the image to 60x60
%
%   Revision 1.3  2003/03/08 20:17:53  pharvin
%   Added line before CVS logging
%
%   Revision 1.2  2003/03/08 20:16:20  pharvin
%   Edited g3_imgnorm.m to test CVS
%

% CONSTANTS
width_difference_ratio = .164;
lightness_threshold = 0.34;
norm_wrist = 10;
left_const = 2;     % When multiplied by norm_wrist, gives you the width of the
                    % margin between the left side of the image and the
                    % wrist
right_const = 3;    % When multiplied by norm_wrist, gives you the width of the
                    % margin between the right side of the image and the
                    % wrist
total_const = left_const + right_const + 1; % When multiplied by norm_wrist, gives
                                            % you the total width of the
                                            % image
height_const = 6;

fill = 0;

% GET VECTOR HORIZONTAL LOCATION DATA FOR LEFTMOST AND RIGHTMOST LIGHTLY
% COLORED PIXELS OF EACH LINE

[ M, N ] = size(g);

for j = 1:M
    for k = 1:N
        if ( g(j,k)>lightness_threshold )
            left_location(j) = k;
            break;
        end
    end
    for k = N:-1:1
        if( g(j,k)>lightness_threshold )
            right_location(j) = k;
            break;
        end
    end
end

%Make background completely black
for j=1:M
    for k=1:N
        if (g(j, k) < lightness_threshold)
            g(j, k) = 0;
        end
    end
end
```

```
% GET VECTOR CONTAINING NUMBER OF LIGHTLY COLORED PIXELS IN EACH LINE
[ M, N ] = size(g);
for j=1:M
    width(j)=0;
    for k=1:N
        if ( g(j,k)>lightness_threshold )
            width(j) = width(j) + 1;
        end
    end
end

% GET VECOR CONTAINING DIFFERNCES BETWEEN EACH ELEMENT IN with
for j = M-3:-1:2;
    diff_width(j) = width(j-1) - width(j);
end

% SMOOTH DIFFERENCE VECTOR
smooth_diff = diff_width;
for j = 5:length(diff_width)-4
    smooth_diff(j) = ( (  smooth_diff(j-3) + smooth_diff(j-2) + smooth_diff(j-1) +
smooth_diff(j) + smooth_diff(j+1) + smooth_diff(j+2) + smooth_diff(j+3) ) / 7 );
end
smooth_diff = round(smooth_diff);


% FIND THE ROW THAT IS THE BASE OF GESTURE/SIGN (OFTEN CORRESPONDS TO
% WRIST)
for jl=length(smooth_diff):-1:11
    if ( ( smooth_diff(jl) + smooth_diff(jl-2) + smooth_diff(jl-4) + smooth_diff(jl-6) +
smooth_diff(jl-8) + smooth_diff(jl-10) )/6 >=.9  )
        row = jl+3;
        break;
    end
end


% CROP IMAGE BELOW BASE ROW
g = g(1:row,:);


% MIND MINIMUM WRIST WIDTH AT BASE ROW
if ( ( row+20 < length(width) ) && ( row-20 > 1 ) )
    width_collection = [ width(row) ];
    % Get widths for leftmost point fixed, and vary the rightmost point
    % over 21 different rows
    for j = row-20:1:row+20
        width_collection = [ width_collection sqrt( (row-j)^2 + (left_location(row)-
right_location(j))^2 ) ];
        width_collection = [ width_collection sqrt( (row-j)^2 + (left_location(j)-
right_location(row))^2 ) ];
    end
    wrist_width = round( min( width_collection ) );
else
    wrist_width = width(row);
end


% RESAMPLE TO NORMALIZE OBJECT AREA => MAKE WRIST WIDTH EQUAL TO 13 PIXELS
g = resample( g, norm_wrist, wrist_width );
g = resample( g', norm_wrist, wrist_width )';


% NORMALIZE CANVAS SIZE (TOTAL SIZE) OF IMAGE AND OBJECT PLACEMENT IN
% IMAGE => APPEND OR CROP BACKGROUND
[ M, N ] = size(g);
% Find leftmost lightly colored pixel at base (wrist)
for j = 1:N
    if ( g(M,j)>lightness_threshold )
        left_pixel = j;
        break;
```

```
    end
end
% Ensure that there are 2*norm_wrist pixels to the left of the wrist
if ( (left_pixel-1) > (left_const*norm_wrist) )        % Too much space on left
    g = g( :, (left_pixel-left_const*norm_wrist):N );
    [ M, N ] = size(g);
elseif ( (left_pixel-1) < (left_const*norm_wrist) )    % Too little space on left
    g = [ fill.*ones(M,((left_const*norm_wrist)-(left_pixel-1))) g ];
    [ M, N ] = size(g);
end
% Find rightmost lightly colored pixel at base (wrist)
for j = N:-1:1
    if ( g(M,j)>lightness_threshold )
        right_pixel = j;
        break;
    end
end
% Ensure that image is total_const*norm_wrist wide
if ( N > total_const*norm_wrist )       % Too much space on right
    g = g( :, 1:(total_const*norm_wrist) );
    [ M, N ] = size(g);
elseif ( N < total_const*norm_wrist )   % Too little space on right
    g = [ g fill.*ones(M,((total_const*norm_wrist)-N)) ];
    [ M, N ] = size(g);
end
% Ensure that the image is 4*norm_wrist pixels in height
if ( M > (height_const*norm_wrist) )                    % Too much space above
    g = g( (M+1-(height_const*norm_wrist)):M, : );
    [ M, N ] = size(g);
elseif ( M < (height_const*norm_wrist) )                % Too little space above
    g = [ fill.*ones(((height_const*norm_wrist)-M),N); g ];
    [ M, N ] = size(g);
end
```

Appendix B - Commented Image Pre-Processing Code for Video Capture Implementation

```
// Image.c

#include "G3Image.h"
#include "stdio.h"
#include "math.h"

// Lightness threshold that decides if a pixel is part of the hand or part of the background
#define G3_LIGHTNESS_THRESHOLD                        ( 150 )
// Width of the frames coming in from the camera
#define G3_SOURCE_IMAGE_WIDTH                         320
// Height of the frames coming in from the camera
#define G3_SOURCE_IMAGE_HEIGHT                        240


int g3_imageNorm64x64( unsigned char *source, unsigned char *destination,
                                             unsigned char *display, int srcHeight, int srcWidth )
{
        // BEGIN CODE TO PROCESS EACH FRAME
        char *outfile;
        unsigned char *lefts, *rights, *widths, *blackened, *buffer1, *buffer2;
        int i,j,baseRow, baseRowWidth, value;
        int newHeight, newWidth, leftBound, rightBound, upperBound, lowerBound;
        int newHeight2, newWidth2;
        int leftDist, rightDist, downDist;

        lefts =                  (unsigned char *) malloc(240);
        rights =         (unsigned char *) malloc(240);
        widths =         (unsigned char *) malloc(240);
        blackened =              (unsigned char *) malloc((320*240));
        buffer1 =                (unsigned char *) malloc((512*512));
        buffer2 =                (unsigned char *) malloc((512*512));

        outfile = "test.out";


        /**** Prepare image for processing ****/
        // Convert RGB image to Grayscale image
        g3_convertRGB2Gray( source, buffer1, srcHeight, srcWidth );
        // Blacken the background of the image
        g3_blackenBackground( buffer1, blackened, srcHeight, srcWidth );


        /**** Make first guess at the location of the wrist in the image ****/
        // Get a vector of width of hand at each line
        g3_getHandWidths( blackened, lefts, rights, widths, srcHeight, srcWidth );
        // Find the baseRow of the image
        baseRow = 0;
        for ( i=50; i<=srcHeight-21; i++ )
        {
                 value = widths[i+20] - widths[i-20];
                 if ( value > 20 ) {
                         baseRow = i;
                         break;
                 }
        }
        // Get width of the baseRow
        baseRowWidth = widths[i];


        /**** Find edges of the image space we care about ****/
        // Find Left Boundary of Image
        leftDist = 3 * baseRowWidth;
        leftBound = ((int) lefts[baseRow]) - leftDist;
        if (leftBound<0) leftBound = 0;
        // Find Right Boundary of Image
        rightDist = 3 * baseRowWidth;
        rightBound = ((int) rights[baseRow]) + rightDist;
        if (rightBound>=srcWidth) rightBound = srcWidth-1;
        // Find Upper Boundary of Image
        upperBound = baseRow;
        // Find Lower Boundary of Image
        downDist = 7 * baseRowWidth;
        lowerBound = baseRow + downDist;
        if (lowerBound>=srcHeight) lowerBound = srcHeight - 1;
```

```
        /**** Resize the image ****/
        // Crop off parts of the image you know you do not need
        g3_cropImage( blackened, buffer1 , srcHeight, srcWidth, &newHeight, &newWidth,
                                  upperBound, lowerBound, leftBound, rightBound );
        // Pad the top of the image so it will be square
        g3_padImageTop( buffer1 , buffer2, newHeight, newWidth, &newHeight2, &newWidth2,
                (newWidth-newHeight) );
        // Shrink the square image down to 64x64
        g3_imageResizeNew( buffer2, destination, newHeight2, newWidth2, 64, 64 );
        // Output 64x64 grayscale image to comma delimited file
        g3_outputCommaFile2D( destination, outfile, 64, 64 );


        /**** Display image preview ****/
        // Mark boundaries for display image
        for ( i=0; i<srcHeight; i++ )
        {
                blackened[(i*srcWidth + leftBound)] = 255;
        }
        for ( i=0; i<srcHeight; i++ )
        {
                blackened[(i*srcWidth + rightBound)] = 255;
        }
        for ( j=0; j<srcWidth; j++ )
        {
                blackened[upperBound*srcWidth + j] = 255;
        }
        for ( i=0; i<srcWidth; i++ )
        {
                blackened[ lowerBound*srcWidth + i ] = 255;
        }
        // Generate version of image to display to screen
        g3_convertGray2RGB( blackened, display, srcHeight, srcWidth );


        /**** Clean up ****/
        free(lefts);
        free(rights);
        free(widths);
        free(blackened);
        free(buffer1 );
        free(buffer2);


        return (newWidth);
}

int g3_getHandWidths( unsigned char *source, unsigned char *lefts, unsigned char *rights, unsigned
char *widths, int imgHeight, int imgWidth )
{
        // Local variables
        int i,j;

        // Get vector containing location of leftmost light pixel in each line
        for ( i = 0; i<imgHeight; i++ )
        {
                for ( j = 0; j<imgWidth; j++ )
                {
                        if ( source[(i*imgWidth+j)] > G3_LIGHTNESS_THRESHOLD )
                        {
                                lefts[i] = j;
                                break;
                        }
                }
        }

        // Get vector containing location of rightmost light pixel in each line
        for ( i = 0; i<imgHeight; i++ )
        {
                for ( j = imgWidth-1; j>=0; j-- )
                {
                        if ( source[(i*imgWidth+j)] > G3_LIGHTNESS_THRESHOLD )
                        {
                                rights[i] = j;
                                break;
                        }
                }
        }


        // Get vector containing width of hand at each line
```

```
        for ( i = 0; i<imgHeight; i++ )
        {
                widths[i] = rights[i] - lefts[i];
        }
        return 0;
}


// Code to output a comma delimited file of a one-dimentional vector
int g3_outputCommaFile1D( const unsigned char *source, char *fileName, int length )
{
        FILE *outputFile;
        int i;

        outputFile = fopen( fileName,"w+");
        if( outputFile == NULL )
        {
                        printf("ERROR: Can not open %s\n", outputFile);
                        return -1;
        }
        for( i=0; i<length; i++ )
        {
                fprintf(outputFile,"%i,", source[i]);
        }
        fprintf(outputFile, "\n");

        fflush(outputFile);
        fclose(outputFile);
        return 0;
}


// Code to output a space delimited file for a two-dimensional vector
int g3_outputCommaFile2D( const unsigned char *source, char *fileName, int height, int width )
{
        FILE *outputFile;
        int i,j;

        outputFile = fopen(fileName,"w+");
        if( outputFile == NULL )
        {
                        printf("ERROR: Can not open %s\n", outputFile);
                        return -1;
        }
        for ( i=0; i<height; i++ )
        {
                for( j=0; j<width; j++ )
                {
                        fprintf(outputFile,"%i ", source[((i*width)+j)]);
                }
                fprintf(outputFile, "\n");
        }
        fflush(outputFile);
        fclose(outputFile);
        return 0;
}

/*
 * Since we will be working with grayscale images, we need to convert RGB 24 bit images to grayscale
 * 8 bit images.  We will do this by merely taking the green component of the image.  This saves
 * computation time and has minimal detremental effects because the green portion of the image
contains
 * most of the light information.
 */
int g3_convertRGB2Gray( unsigned char *source, unsigned char *destination, int height, int width )
{
        // Local variables
        int i, j;
        unsigned char *src;
        unsigned char *dest;

        // Make a copy of the pointer to the RGB pixel array to work with
        src = source;
        // Make a copy of the pointer to the grayscale array to work with
        dest = destination;

        // Generate new pixel values - just copy over the green values and throw away the rest
        for ( i=0; i<height; i++ ) {
                for ( j=0; j<width; j++ ) {
                        src++;
                        *dest++ = (char) *src;
```

```
                        src += 2;
                }
        }
        return 0;
}

// Converts a grayscale image to RGB by making the grayscale intensity values the values for the
// green component of the RGB image and leaving the red and blue components zero
int g3_convertGray2RGB( unsigned char *source, unsigned char *destination, int height, int width )
{
        // Local variables
        int i, j;
        unsigned char *src;
        unsigned char *dest;

        // Make a copy of the pointer to the grayscale array to work with
        src = source;
        // Make a copy of the pointer to the RGB pixel array to work with
        dest = destination;

        // Generate new pixel values
        for ( i=0; i<height; i++ ) {
                for ( j=0; j<width; j++ ) {
                        *dest++ = 0;
                        *dest++ = (char) *src++;
                        *dest++ = 0;
                }
        }
        return 0;
}



// Code to flip the image vertically
int g3_flipUD( unsigned char *source, unsigned char *destination, int height, int width )
{
        // Local variables
        int i, j;
        unsigned char *src;
        unsigned char *dest;

        // Make a copy of the pointer to the source array to work with
        src = source;
        // Make a copy of the pointer to the destination array to work with
        dest = destination;

        // Make the bottom row of pixels the top row, and so on...
        for ( i=height-1; i>=0; i-- ) {
                for ( j=0; j<width; j++ ) {
                        *dest++ = src[(i*320+j)];
                }
        }
        return 0;
}

// Code to blacken (zero) the pixel values that fall below a certain threshold,
// otherwise, leave them as they are
int g3_blackenBackground( unsigned char *source, unsigned char *destination,
                                                int height, int width )
{
        unsigned char *dst;
        int i,j;

        dst = destination;

        for ( i=0; i<height; i++ )
        {
                for ( j=0; j<width; j++ )
                {
                        if ( source[ i*width + j ] <= G3_LIGHTNESS_THRESHOLD )
                        {
                                *dst++ = 0;
                        }
                        else
                        {
                                *dst++ = source[ i*width + j ];
                        }
                }
        }

        return 0;
```

```
}


// Code to crop an image to within specified bounds.
int g3_cropImage( unsigned char *source, unsigned char *destination,
                                    int srcHeight, int srcWidth, int *dstHeight, int *dstWidth,
                                    int upperBound, int lowerBound, int leftBound, int rightBound )
{
        unsigned char *dst;
        int i,j;

        dst = destination;

        *dstHeight = (lowerBound - upperBound);
        *dstWidth = (rightBound - leftBound);

        for ( i=upperBound; i<lowerBound; i++ )
        {
                for ( j=leftBound; j<rightBound; j++ )
                {
                        *dst++ = source[i*srcWidth+j];
                }
        }
        return 0;
}

// Code to resize an image to an arbitrary new size using bilinear filtering.
int g3_imageResizeNew( unsigned char *source, unsigned char *destination,
                                                int srcHeight, int srcWidth, int dstHeight, int dstWidth
)
{
        double nXFactor = (double)srcWidth/(double)dstWidth;
        double nYFactor = (double)srcHeight/(double)dstHeight;

        double fraction_x, fraction_y, one_minus_x, one_minus_y;
        int ceil_x, ceil_y, floor_x, floor_y;
        unsigned char c1;
        unsigned char c2;
        unsigned char c3;
        unsigned char c4;
        unsigned char gray;
        unsigned char b1, b2;
        int x, y;

        for (x = 0; x < dstWidth; ++x)
    {
                for (y = 0; y < dstHeight; ++y)
        {
                        // Setup
            floor_x = (int) floor((double)x * nXFactor);
            floor_y = (int) floor((double)y * nYFactor);
            ceil_x = floor_x + 1;
            if (ceil_x >= srcWidth) ceil_x = floor_x;
            ceil_y = floor_y + 1;
            if (ceil_y >= srcHeight) ceil_y = floor_y;
            fraction_x = x * nXFactor - floor_x;
            fraction_y = y * nYFactor - floor_y;
            one_minus_x = 1.0 - fraction_x;
            one_minus_y = 1.0 - fraction_y;

            c1 = source[(floor_x + floor_y*srcWidth)];
            c2 = source[(ceil_x + floor_y*srcWidth)];
            c3 = source[(floor_x + ceil_y*srcWidth)];
            c4 = source[(ceil_x + ceil_y*srcWidth)];

            b1 = (unsigned char)(one_minus_x * c1 + fraction_x * c2);
            b2 = (unsigned char)(one_minus_x * c3 + fraction_x * c4);
            gray = (unsigned char)(one_minus_y * (double)(b1) + fraction_y * (double)(b2));

            destination[(x + y*dstWidth)] = gray;
                }
        }
        return 0;
}


// Code used to pad an image if it is too small
int g3_padImage( unsigned char *source, unsigned char *destination,
                                    int srcHeight, int srcWidth, int *dstHeight, int *dstWidth,
                                    int baseRow, int baseRowWidth, int baseRowLeft, int baseRowRight,
                                    int downDist, int leftDist, int rightDist )
```

```
{
        // Declare local variables
        int padAmount;
        unsigned char *buffer1;
        int height1, width1;
        unsigned char *buffer2;
        int height2, width2;
        int i;
        unsigned char *tempBuffer;

        // Allocate buffer memory
        buffer1 = (unsigned char *) malloc((512*512));
        buffer2 = (unsigned char *) malloc((512*512));

        // Copy information into buffer
        for ( i=0; i<srcHeight*srcWidth; i++ )
        {
                buffer1[i] = source[i];
        }
        height1 = srcHeight;
        width1 = srcWidth;


        /**** Pad left side if nessesary ****/
        if ( baseRowLeft < leftDist )
        {
                padAmount = leftDist - baseRowLeft;
                g3_padImageLeft( buffer1, buffer2, height1, width1, &height2, &width2, padAmount );

        }
        else
        {
                // SWAP BUFFER POINTERS
                tempBuffer = buffer1;
                buffer1 = buffer2;
                buffer2 = tempBuffer;
                // TRANSFER STATISTICS
                height2 = height1;
                width2 = width1;
        }

        /**** Pad right side if nessesary ****/
        if ( ((srcWidth-baseRowRight) - 1) < rightDist )
        {
                padAmount = rightDist - ((srcWidth-baseRowRight) - 1);
                g3_padImageRight( buffer2, buffer1, height2, width2, &height1, &width1, padAmount );
        }
        else
        {
                // SWAP BUFFER POINTERS
                tempBuffer = buffer1;
                buffer1 = buffer2;
                buffer2 = tempBuffer;
                // TRANSFER STATISTICS
                height1 = height2;
                width1 = width2;
        }

        /**** Pad vertically if nessesary ****/
        if ( (srcHeight-baseRow) < downDist )
        {
                padAmount = downDist - (srcHeight-baseRow);
                g3_padImageTop( buffer1, destination, height1, width1, dstHeight, dstWidth, padAmount
);
        }
        else
        {
                // COPY INFORMATION TO DESTINATION BUFFER
                for ( i=0; i<height1*width1; i++ )
                {
                        destination[i] = buffer1[i];
                }
                // TRANSFER STATISTICS TO DESTINATION
                *dstHeight = height1;
                *dstWidth = width1;
        }

        free(buffer1);
        free(buffer2);
        return 0;
}
```

```
// Code to pad only the top of the image if it is too short
int g3_padImageTop( unsigned char *source, unsigned char *destination,
                                     int srcHeight, int srcWidth, int *dstHeight, int *dstWidth,
                                     int padAmount )
{
        unsigned char *dst;
        int i,j;
        dst = destination;

        *dstHeight = srcHeight + padAmount;
        *dstWidth = srcWidth;

        for ( i=0; i<srcHeight; i++ )
        {
                for ( j=0; j<srcWidth; j++ )
                {
                        *dst++ = source[i*srcWidth + j];
                }
        }
        for ( i=0; i<padAmount*srcWidth; i++ )
        {
                *dst++ = 0;
        }
        return 0;
}

// Code to pad only the right side of the image
int g3_padImageRight( unsigned char *source, unsigned char *destination,
                                     int srcHeight, int srcWidth, int *dstHeight, int *dstWidth,
                                     int padAmount )
{
        unsigned char *dst;
        int i,j;
        dst = destination;

        *dstHeight = srcHeight;
        *dstWidth = srcWidth + padAmount;

        for ( i=0; i<srcHeight; i++ )
        {
                for ( j=0; j<srcWidth; j++ )
                {
                        *dst++ = source[i*srcWidth + j];
                }
                for ( j=0; j<padAmount; j++ )
                {
                        *dst++ = 0;
                }
        }
        return 0;
}


// Code to pad only the left side of the image
int g3_padImageLeft( unsigned char *source, unsigned char *destination,
                                     int srcHeight, int srcWidth, int *dstHeight, int *dstWidth,
                                     int padAmount )
{
        unsigned char *dst;
        int i,j;
        dst = destination;

        *dstHeight = srcHeight;
        *dstWidth = srcWidth + padAmount;

        for ( i=0; i<srcHeight; i++ )
        {
                for ( j=0; j<padAmount; j++ )
                {
                        *dst++ = 0;
                }
                for ( j=0; j<srcWidth; j++ )
                {
                        *dst++ = source[i*srcWidth + j];
                }
        }
        return 0;
}
```