CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

REAL-TIME COLOR IMAGE INTERPOLATION

A graduate project submitted in partial fulfillment of the requirements
For the degree of Master of Science
In Electrical Engineering

By

Aaron Santee

May 2012

The graduate project of Aaron Santee is approved:

Deborah van Alphen, Ph.D.	Date
Sharlene Katz, Ph.D.	Date
Xiyi Hang, Ph.D, Chair	Date

California State University, Northridge

ACKNOWLEDGEMENT

I would like to thank my wonderful wife, Karen, and my two beautiful daughters, Emma and Lauren, for their love and support through these past several years. It has been a long journey and I couldn't have made it without them.

TABLE OF CONTENTS

SIGNATURE PAGE ii

ACKNOWLEDGEMENT iii

ABSTRACT v

INTRODUCTION 1

Image Interpolation 2

Color Image Background 2

Demosaicing 4

COLOR IMAGE INTERPOLATION METHODS 5

Simple 2x2 5

Nearest Neighbor 6

Bilinear 7

Color Balanced 3x3 9

SYSTEM SET-UP AND IMPLEMENTATION 11

RESULTS 16

Visual 16

PSNR – Peak Signal to Noise Ratio 19

CONCLUSION 21

REFERENCES 22

APPENDIX A: MATLAB CODE 23

APPENDIX B: Code Composer Studio Code 30

ABSTRACT

REAL-TIME COLOR IMAGE INTERPOLATION

By

Aaron Santee

Master of Science in Electrical Engineering

This graduate project explores the real-time implementation of non-adaptive color image interpolation algorithms using a Texas Instrument TMS320C6416T DSP module. Code Compose Studio 4, a product of Texas Instruments and built on the Eclipse platform, is used to implement the algorithms using C language and is the graphical interface that allows the code to be processed through the DSP module. The four interpolation algorithms that will be examined are simple 2x2, nearest neighbor, bilinear and color balanced 3x3. MATLAB is used to deconstruct and reconstruct the RGB matrices of a specified JPEG image as Code Composer Studio is utilized for the interpolation process. The Peak Signal to Noise Ratio (PSNR) will be used as a means of image analysis to compare the value of each red pixel data of the original image to the corresponding red pixel of the interpolated image. The highest value of the PSNR indicates a more accurate result and thus determines the best interpolation method.

INTRODUCTION

Color image interpolation is widely used in digital cameras. The image sensors of a digital camera are known as CCDs (charge couple devices), in which a CFA (color filter array) acts as a mask across the sensors. The visual interpretation is stored in memory as an n by m array as it gets converted into electrical signals. The most common filter used in digital cameras is a Bayer RGB pattern. The following figure is an example of a 6 x 6 Bayer RGB pattern [2][5].

G11	R12	G13	R14	G15	R16
B21	G22	B23	G24	B25	G26
G31	R32	G33	R34	G35	R36
B41	G42	B43	G44	B45	G46
G51	R52	G53	R54	G55	R56
B61	G62	B63	G64	B55	G66

Figure 1

As seen in the above figure, the G and R pixels alternate horizontally as do the B and G pixels. Consequently, the G and R pattern alternate vertically with the B and G pattern. This RGB pattern means that each pixel has only one data value that represents a specific primary color. Three data values representing each primary color is needed to signify a valid RGB color space. This is accomplished by estimating the two missing pixel values from the available surrounding color values based on the specified algorithms.

Image Interpolation

The basic idea of interpolation is to utilize existing data to estimate values between and around the values that have already been determined. For example, if there are two points in space with specific coordinates, the value of a point at the halfway mark could be estimated by using linear interpolation. This is achieved by connecting the two existing points with a straight line. The coordinates of any successive points placed on the line can now be estimated. The concept of image interpolation is very similar. Instead of estimating points, pixel values are estimated. To find the value of an unknown pixel, depending on the algorithm used, the pixel is estimated by taking a weighted average of the surrounding known pixel values. Image interpolation is widely used in digital photography software to resize, distort and rotate images.

Color Image Background

Each pixel of a RGB color image corresponds to a triplet of red, blue and green color values. When an image is classified as `uint8` (unsigned 8-bit integer), the range of each

color value is [0,255]. The value 255 is determined by calculating 2^8. Since each pixel contains 3 color components, an 8-bit image is 24 bits deep. A fully saturated color in a 24-bit RGB color space is represented by the value 255. At the other extreme, the value 0 denotes no color. For the class `uint8`, each pixel is represented by 1 byte. Other classes exist, such as `uint16`, but for the purpose of this project, the class `uint8` will be used. In the field of image processing, `uint8` is significant because this class is common when dealing with image files in the format of a JPEG or TIFF. The following RGB color cube, which has been normalized, represents the RGB color space [1].

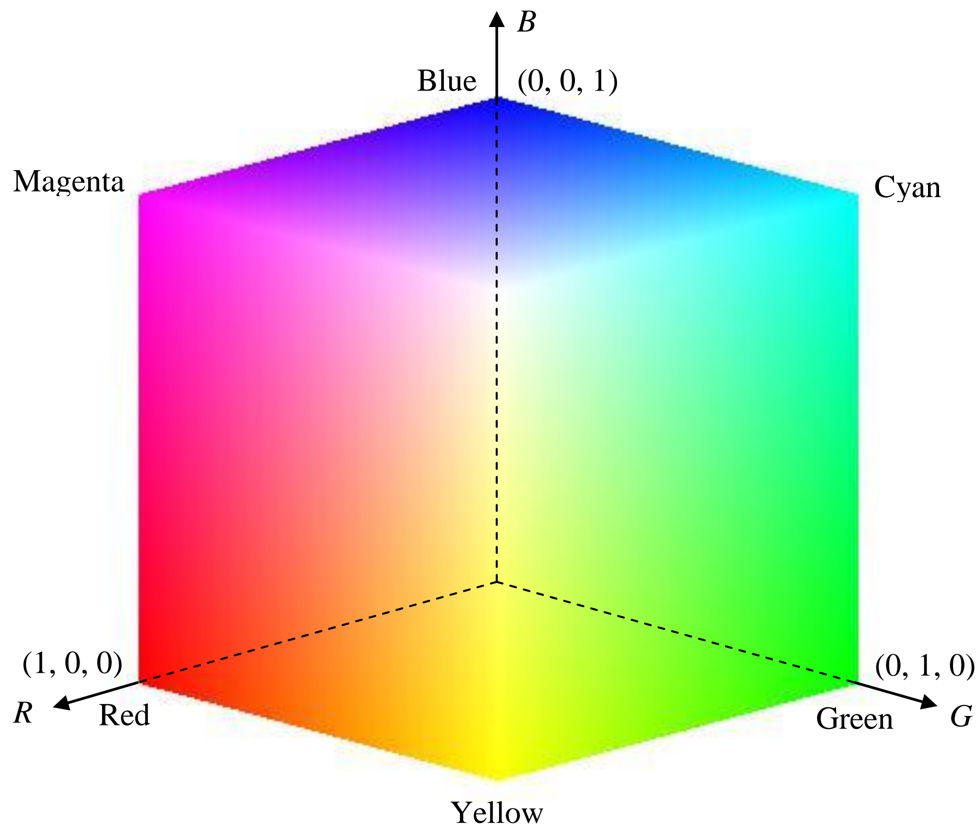

Figure 2

Demosaicing

Demosaicing is the process in which the missing color values at each pixel are interpolated to generate a complete color image from the Bayer RGB pattern [4][5]. The four types of interpolation methods that will be discussed are simple 2 x 2, nearest neighbor, bilinear and color balanced 3 x 3. The process, in which the pixels are estimated, will be further explained for each specific interpolation method.

An 8-bit color JPEG will be used as the original image, in which a Bayer RGB pattern will be applied and subsequently demosaiced using each of the four interpolation methods to produce a high resolution image. The results will be compared to the original image by evaluating the Peak Signal to Noise Ratio. The higher PSNR, the more accurate the result and thus indicates the more efficient and useful interpolation method. It is important to point out that because each pixel in the entire Bayer RGB pattern is interpolated, one by one, the load that this exerts on the processor can be very high. As a result, the interpolation method that is determined to be the most accurate must be weighed for its efficiency as well as for its speed.

COLOR IMAGE INTERPOLATION METHODS

Simple 2x2

The simple 2 x 2 method basically splits the pixels of the image into 2 x 2 matrices and interpolates the missing green pixel data with the average of the existing green pixels in that specific matrix. The values of the Red and Blue pixels are determined from the existing Red and Blue pixels within that specific 2 x 2 matrix. This can be better explained by imagining a 2 x 2 window sliding horizontally across the image (left to right), pixel by pixel, row by row, and interpolating color data for each specific pixel [4][5] .

$Rd = R$

$Gr = mean(Gr + Gb)$

$Bl = B$

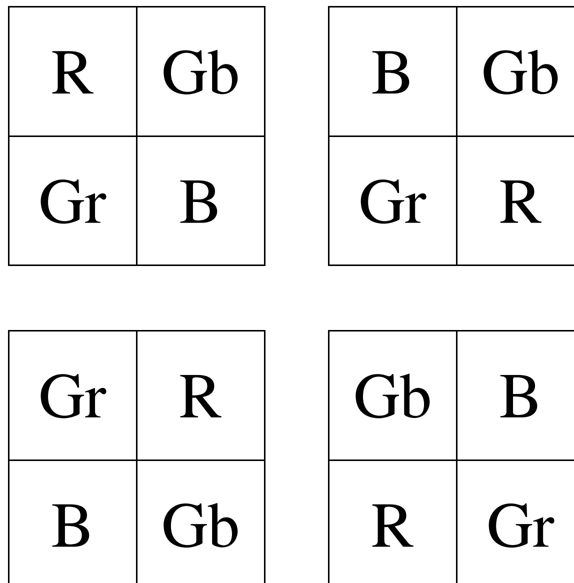

Figure 3

Nearest Neighbor

The nearest neighbor method interpolates the missing red, green or blue pixel data for a specific pixel by assuming the color data from an adjacent pixel. The nearest neighbor method was also modified so that there was a slight variation as to which pixels were used to interpolate the color data. Instead of interpolating the specific pixel value from the pixel to the left, the pixel to the right was used [2][4].

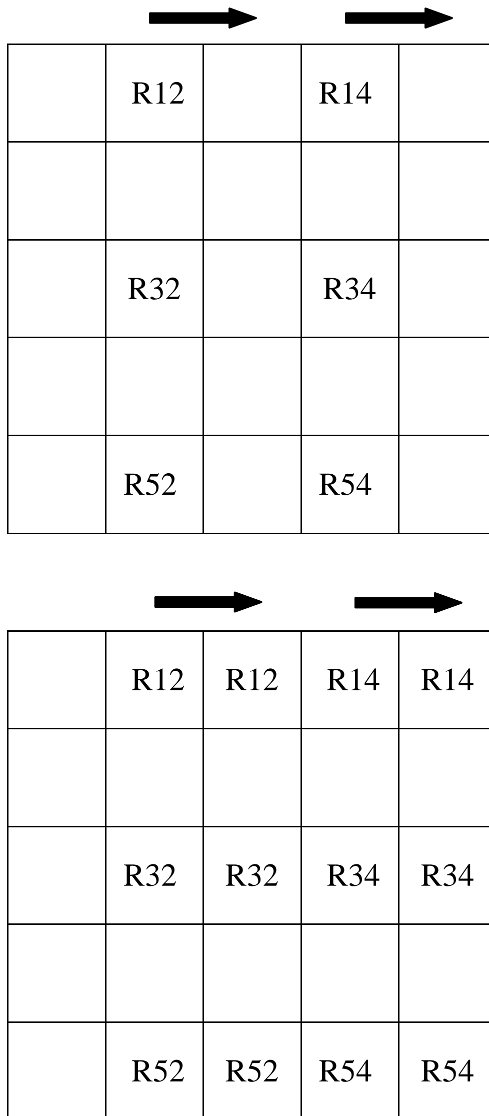

Figure 4

Bilinear

In the bilinear method, which is a 3 x 3 interpolation scheme, when a blue pixel is at the center of the matrix, the missing green data is calculated by taking the average of the surrounding green pixels. The red data is also determined by calculating the average of the surrounding red pixels. The blue center pixel determines the missing blue data for the other pixels in the matrix. The process is the same as the color of the center pixel changes. As a result, there are four different possibilities for this specific interpolation method [4][5].

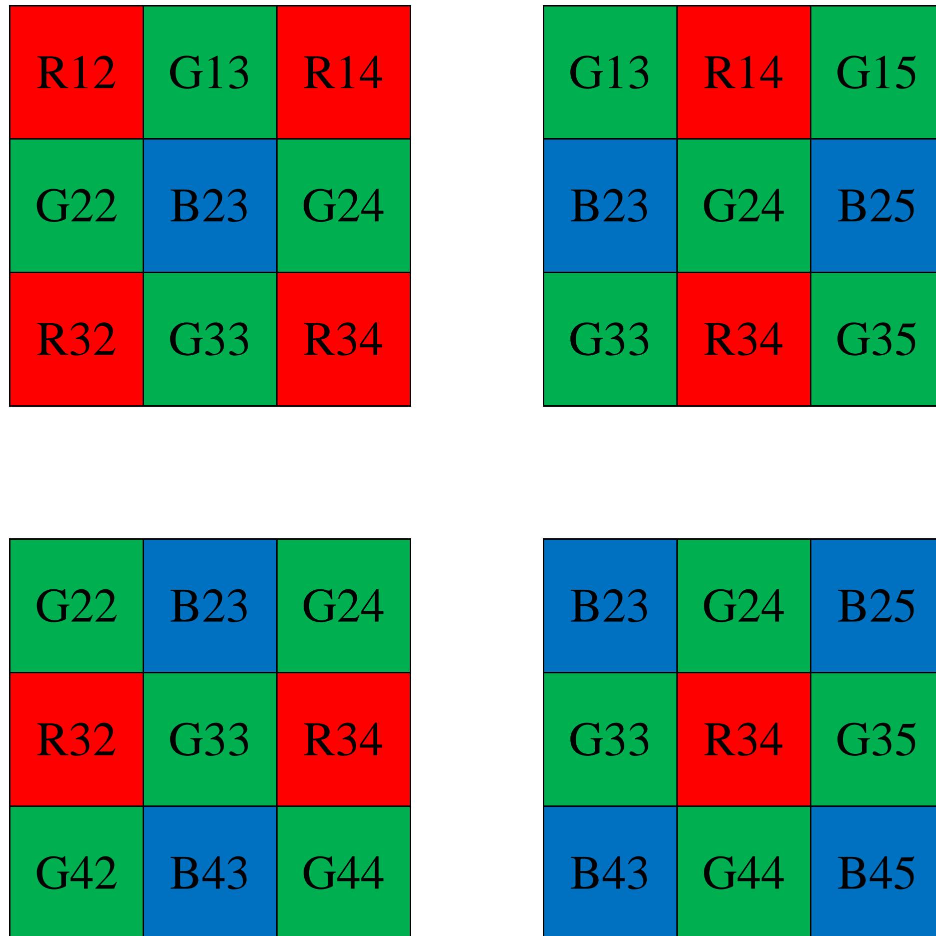

Figure 5

Like the Simple 2 x 2, to understand the concept of this method, imagine a 3 x 3 window sliding across the image pixel by pixel. Depending on the specified center pixel, the values of the other color pixels are determined by the following formulation.

B23 pixel at center of matrix:

$Bl = B23$

$Rd = mean(R12 + R14 + R32 + R34)$

$Gr = mean(G13 + G33 + G22 + G24)$

G24 pixel at center of matrix:

$Gr = G24$

$Rd = mean(R14 + R34)$

$Bl = mean(B23 + B25)$

G33 pixel at center of matrix:

$Gr = G33$

$Rd = mean(R32 + R34)$

$Bl = mean(B23 + B43)$

R34 pixel at center of matrix:

$Rd = R34$

$Gr = mean(G24 + G44 + G33 + G35)$

$Bl = mean(B23 + B25 + B43 + B45)$

Color Balanced 3x3

The Color Balanced method is also a 3x3 interpolation scheme and is similar to bilinear, where missing color data is based on the average of the surrounding pixels. In this case, though, the center pixel value is subtracted from the sum of the surrounding pixels before the average is taken. In effect, all the color data is used to determine the missing pixel colors. The center pixel still determines the missing said color data for the surrounding pixels in the matrix. Again, the result is four possible scenarios for this interpolation method based on the specific 3 x 3 matrix [4].

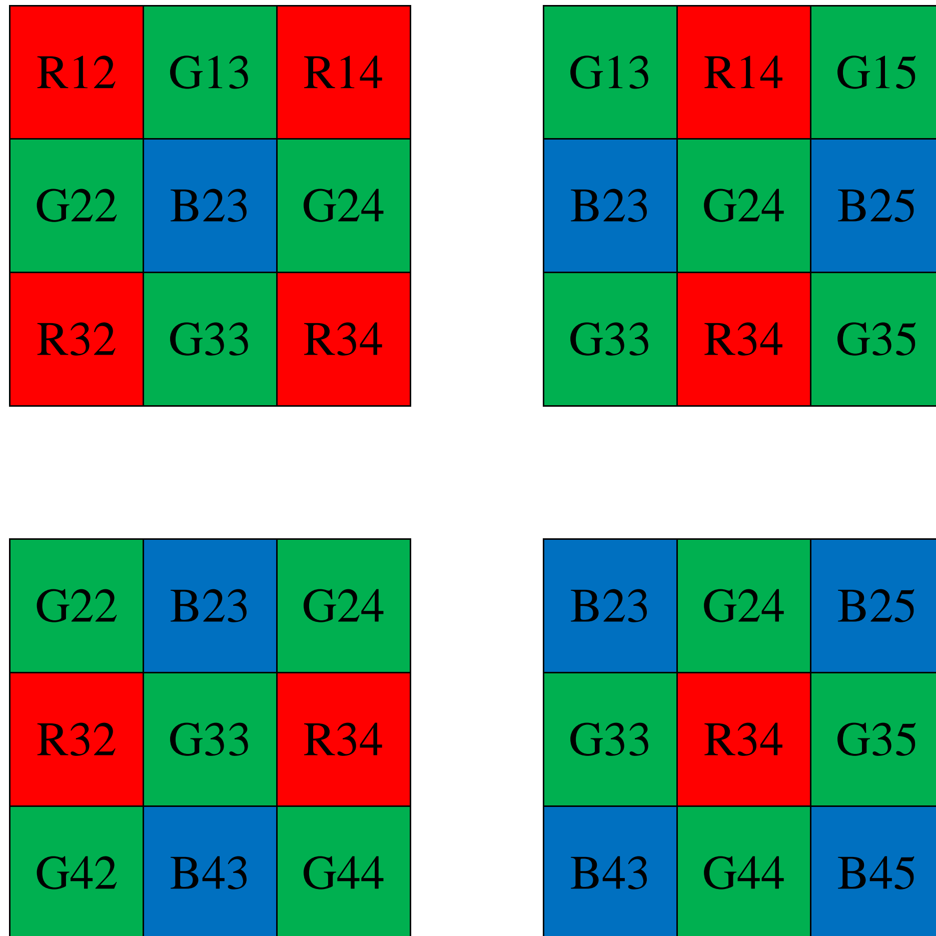

Figure 6

R34 pixel at center of matrix:

$$Rd = R34$$

$$Gr = ((G24 + G44 - R34) + (G33 + G35 - R34))/4$$

$$Bl = (B23 + B25 + B43 + B45 - R34)/4$$

G24 pixel at center of matrix:

$$Gr = ((G13 + G15 + G33 + G35 - G24)/4) + G24)/2$$

$$Rd = (R14 + R34 - G24)/2$$

$$Bl = (B23 + B25 - G24)/2$$

G33 pixel at center of matrix:

$$Gr = ((G22 + G24 + G42 + G44 - G33)/4) + G33)/2$$

$$Rd = (R32 + R34 - G33)/2$$

$$Bl = (B23 + B43 - G33)/2$$

B23 pixel at center of matrix:

$$Bl = B23$$

$$Rd = (R12 + R14 + R32 + R34 - B23)/4$$

$$Gr = ((G13 + G33 - B23) + (G22 + G24 - B23))/4$$

As it will be subsequently observed in the results, because all of the color data is involved in the interpolation process for achieving an estimated pixel value, the resulting image becomes more saturated and has an interesting effect on the calculated PSNR.

SYSTEM SET-UP AND IMPLEMENTATION

The specific color components of an image can be extracted using MATLAB:

$RGB = uint8\big(imread(image)\big);$

$Rd = RGB(:,:,1);$

$Gr = RGB(:,:,1);$

$Bl = RGB(:,:,3);$

The Rd, Gr and Bl layers are separated with varying pixel data based on the intensity of that color at the specific pixel of the image, again with a value in the range [0,255]. Equally, the image can be reconstructed using the MATLAB operator `cat`:

$RGB_image = \; cat\big(3, unit8(Rd), uint8(Gr), uint8(Bl)\big);$

The three layers, Rd, Gr, and Bl, are essentially stacked onto each other to produce a single color pixel, based on the value and intensity of the specific pixel on each layer. It is important to recognize that the classification of each layer of color is 8 bits. Array indexing is also an important aspect of color image processing. The image, as well as its separated color components, is represented by indexed arrays. The value of a color pixel corresponds to a specific index within that array [1]. For demonstration purposes, take the example of array “a”:

$a = [4\ 9\ 2\ 0\ 8\ 5\ 0\ 4];$

The value at index 5 of the array (in MATLAB) would be:

$a(5) = 8;$

This particular concept is integral when implementing the interpolation algorithms. Each application of the interpolation methods will be referencing a specific index from each of the three separate color component arrays, producing an estimated color pixel value.

The initial step to this entire system is to simulate the application of the Bayer RGB pattern on the image, *tutles.jpg*. This is initiated in the parent.m script and is achieved through the missing_data.m function created in MATLAB [Appendix A]. The basic concept of this function is that it sets the color values Red, Green and Blue to 0, based on the location of the pixel and in accordance with the RGB Bayer pattern. The following figure shows the initial JPEG image as it is read through MATLAB. The successive figures show the image after the color data has been removed, based on the Bayer RGB pattern. Note the de-saturated color value and intensity of the image. The zoomed image with the pattern applied shows a closer detail of the simulation. The alternating green and red / blue and green patterns are apparent through the close-up of the image.

Figure 7 - turtles.jpg

Figure 8 - Bayer RGB pattern

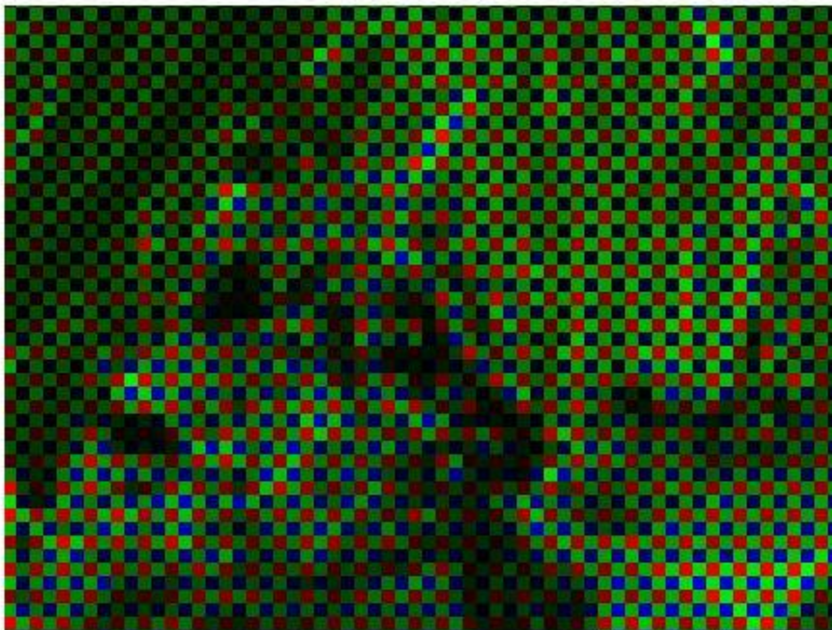

Figure 9 - Bayer RGB pattern, zoomed

The parent.m file then initiates several important processes [Appendix A]. After it reads in the 8 bit turtle.jpg file and passes through the missing_data.m function, the simulated RGB Bayer pattern image is then deconstructed into each individual Red, Green and Blue matrices, separating the three color components. This is performed by utilizing the extraction method previously explained. These matrices are then written out to the text files, Red.txt, Green.txt and Blu.txt, using the MATLAB function, dlmwrite. This function delimits, or separates, the values of each pixel within each color matrix using commas. The format2Darray C program [Appendix B] uses the delimited text files as the input and adds a series of curly brackets, commas and semicolons to create and write out header files containing the readable 2D indexed arrays for each color matrix. The header files will be used as the 2D array input for the C interpolation programs in Code Composer Studio. It is important to point out that indexed arrays in MATLAB begin with 1 and the arrays in C begin with 0. This minute detail needs to be accommodated for accordingly when dealing with arrays across each of the programs.

Each interpolation project in Code Composer Studio consists of these files:

main.c - Main program file

Rd.h - Red data header file

Gr.h - Green data header file

Bl.h - Blue data header file

c64xx.ccxml - Target configuration file for c6416

dsk6416.cmd - Linker file for c6416

The figure below depicts the general structure of an interpolation project.

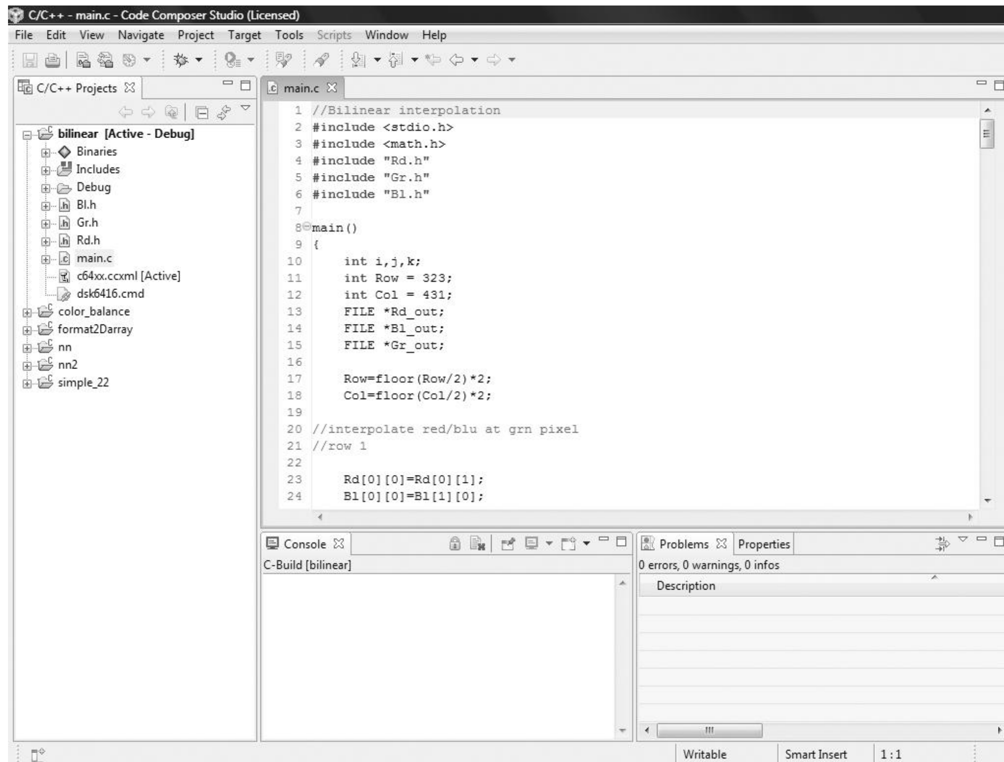

Figure 10

The interpolation program for each specific method is executed on separate occasions and creates an out file that is loaded onto the c6416. After the out file is run and the designated interpolation completes, each main.c is set to write out three files, Rd_*out.txt, Gr_*out.txt and Bl_*out.txt, depending on the specified method, to c:/../"interpolation method"/Debug. For example, using the nearest neighbor interpolation method, the output text file would be Rd_NNout.txt and so on. The fromC.m script in MATLAB, copies the text files to the MATLAB folder located in c:/../MatLab and uses dlmread to convert the text file to a readable color matrix. It then uses the `cat` operator to reconstruct, "stack", the RGB color matrices into a readable format for MATLAB. The image can subsequently be displayed by using imshow( ).

RESULTS

Visual

The following figures display the results of the discussed interpolation methods processed through the c6416 DSP module.

Figure 11 shows the simple 2 x 2. The aliasing is most noticeable along the edges of the leaves and rock, but the algorithm does a decent job of reproducing the saturated color of the original image.

Figures 12 and 13, the nearest neighbor and its modification, nearest neighbor 2, are almost identical to Figure 11. Both display the aliased edges, but the color reproduction of the original is very close.

Although some of the brightness of the original image was lost, the bilinear displayed in Figure 14 shows the increased softening of the edges, anti-aliasing. The image overall has a greener tone because only data from two color pixels were used when determining the average for the missing color pixels.

The color balance showed in Figure 15 is similar to Figure 14, in that, the aliasing has been significantly reduced and the edges appear to be softer. In this case, the overall tone of the image is much redder than the previous image. This is primarily due to the red color data being factored in when determining the green color pixels and vice versa for calculating the red color pixels.

Figure 11 - Simple 2 x 2

Figure 12 - Nearest Neighbor

Figure 13 - Nearest Neighbor, Modified

Figure 14 - Bilinear

Figure 15 - Color Balanced

PSNR – Peak Signal to Noise Ratio

The quality of the interpolated images is mathematically calculated by using the Peak Signal to Noise Ratio, measured in dB [5]. The Root Mean Square Error must first be determined.

$$RMSE = \sqrt{\frac{1}{\text{MxN}}\sum(\text{interpolatedValue} - \text{actualValue})^2}$$

$$PSNR = 20log_{10}\frac{255}{RMSE}$$

The value 255 is used because it is the maximum pixel value for an 8-bit image.

The results of the PSNR calculation for each type of interpolation scheme are arranged in the following table:

Interpolation Method	Red_RMSE	Red_PSNR
Simple 2x2	16.9999	23.5219
Nearest Neighbor	16.9999	23.5219
Nearest Neighbor, mod	16.9999	23.5219
Bilinear	16.6354	23.7101
Color Balance	50.8395	14.0068

Clearly, the Simple 2x2, Nearest Neighbor and Nearest Neighbor, modified, have identical RMSEs and PSNRs. The Bilinear PSNR is slightly higher than the aforementioned interpolation methods. The Color Balance PSNR should be almost identical, if not better, than the Bilinear PSNR. In this case, it is significantly lower and points out the fact that measuring the PSNR is not always an accurate means for determining image quality. For this particular situation, the case can be made that the contributing factor for the decreased PSNR is again the use of other color values in the estimated calculation of another color pixel for this interpolation method.

CONCLUSION

The Simple 2x2 and Nearest Neighbor are both basic interpolation algorithms. They achieve decent color reproduction but lack the complexity to overcome the aliasing effect along the edges. The Bilinear and Color Balance algorithms are more sophisticated in interpolating color pixels, where it considers a broader range of adjacent pixels in its calculation. Although the color reproduction of the original image is not completely accurate, it significantly reduces the degree of aliasing on the edges, producing a much softer, feathery effect. As a result of being more complex in its calculation, the Bilinear and Color Balanced will require more processing power. Based on the visual and mathematical analysis, the consensus points to the Bilinear as being the best interpolation method, where the image produced is comparable to the original and the computational processor needs are not excessive.

REFERENCES

[1] Gonzalez, R. C., Woods, R. E. and Eddins, S. L. [2009]. Digital Image Processing using MATLAB, 2nd ed., Gatesmark Publishing, www.gatesmark.com.

[2] Kuo, Sen M., Lee, Bob H. & Tian, Wenshun. [2006]. Real-Time Digital Signal Processing: Implementations and Applications, Second Edition, JohnWiley & Sons Ltd.

[3] Qureshi, Shehrzad. [2005]. Embedded Image Processing on the TMS320C6000™ DSP: Examples in Code Composer Studio™ and MATLAB, Springer Scienece+Business Media, Inc.

[4] Ho, Charles Hung Viet. [2007]. Bayer Color interpolation Algorithms in Digital Camera, California State University, Northridge.

[5] Jean, Remi. [Nov. 2011]. Demosaicing with The Bayer Pattern. Department of Computer Science, University of North Carolina.

[6] Gookin, Dan. [2004]. C: All-In-One Desk Reference For Dummies, Wiley Publishing, Inc.

APPENDIX A: MATLAB CODE

```
%parent node

clear all; close all;

image='turtles.jpg';
RGB_original=uint8(imread(image));
[init,red,grn,blu]=missing_data(image);

size(red)

R=cat(1,uint8(red));
G=cat(1,uint8(grn));
B=cat(1,uint8(blu));
dlmwrite('RED.txt',R);
dlmwrite('GRN.txt',G);
dlmwrite('BLU.txt',B);

RGB=cat(3,uint8(red),uint8(grn),uint8(blu));

figure(1);imshow(image); %title('Figure 1. turtles.jpg');
figure(2);imshow(RGB); %title('Figure2. Bayer Pattern')

level=7;

figure(3);imshow(RGB); %title('Figure 3. Bayer Pattern Zoom')
zoom(level);
```

```
%missing_data.m [4]

function[RGB,red,grn,blu]=missing_data(image)

%eliminate color data to replicate Bayer pattern.

RGB=uint16(imread(image));

r=uint16(RGB(:,:,1));
g=uint16(RGB(:,:,2));
b=uint16(RGB(:,:,3));

[Row,Col]=size(r);

data=0;

%no red/blu data at grn pixel; odd row/col;

for j=1:2:Row,
    for k=1:2:Col,
        r(j,k)=data;
        b(j,k)=data;
    end
end

%no red/blu data at grn pixel; even row/col;

for j=2:2:Row,
    for k=2:2:Col,
        r(j,k)=data;
        b(j,k)=data;
    end
end

%no grn/blu data at red pixel; odd row, even col;

for j=1:2:Row,
    for k=2:2:Col,
        g(j,k)=data;
        b(j,k)=data;
    end
end

%no grn/red data at blu pixel; even row, odd col;

for j=2:2:Row,
```

```
    for k=1:2:Col,
        g(j,k)=data;
        r(j,k)=data;
    end
end

red=r;
grn=g;
blu=b;
```

```
%fromC.m

copyfile('c:/Users/Aaron/Documents/ECE696/simple_22/Debug/Rd_22out.txt','c:/Users/
Aaron/Documents/ECE696/MatLab/Rd_22out.txt');
copyfile('c:/Users/Aaron/Documents/ECE696/simple_22/Debug/Gr_22out.txt','c:/Users/
Aaron/Documents/ECE696/MatLab/Gr_22out.txt');
copyfile('c:/Users/Aaron/Documents/ECE696/simple_22/Debug/Bl_22out.txt','c:/Users/A
aron/Documents/ECE696/MatLab/Bl_22out.txt');

copyfile('c:/Users/Aaron/Documents/ECE696/nn/Debug/Rd_NNout.txt','c:/Users/Aaron/
Documents/ECE696/MatLab/Rd_NNout.txt');
copyfile('c:/Users/Aaron/Documents/ECE696/nn/Debug/Gr_NNout.txt','c:/Users/Aaron/
Documents/ECE696/MatLab/Gr_NNout.txt');
copyfile('c:/Users/Aaron/Documents/ECE696/nn/Debug/Bl_NNout.txt','c:/Users/Aaron/D
ocuments/ECE696/MatLab/Bl_NNout.txt');

copyfile('c:/Users/Aaron/Documents/ECE696/nn2/Debug/Rd_NN2out.txt','c:/Users/Aaro
n/Documents/ECE696/MatLab/Rd_NN2out.txt');
copyfile('c:/Users/Aaron/Documents/ECE696/nn2/Debug/Gr_NN2out.txt','c:/Users/Aaro
n/Documents/ECE696/MatLab/Gr_NN2out.txt');
copyfile('c:/Users/Aaron/Documents/ECE696/nn2/Debug/Bl_NN2out.txt','c:/Users/Aaron
/Documents/ECE696/MatLab/Bl_NN2out.txt');

copyfile('c:/Users/Aaron/Documents/ECE696/bilinear/Debug/Rd_BLout.txt','c:/Users/Aa
ron/Documents/ECE696/MatLab/Rd_BLout.txt');
copyfile('c:/Users/Aaron/Documents/ECE696/bilinear/Debug/Gr_BLout.txt','c:/Users/Aar
on/Documents/ECE696/MatLab/Gr_BLout.txt');
copyfile('c:/Users/Aaron/Documents/ECE696/bilinear/Debug/Bl_BLout.txt','c:/Users/Aar
on/Documents/ECE696/MatLab/Bl_BLout.txt');

copyfile('c:/Users/Aaron/Documents/ECE696/color_balance/Debug/Rd_CBout.txt','c:/Us
ers/Aaron/Documents/ECE696/MatLab/Rd_CBout.txt');
copyfile('c:/Users/Aaron/Documents/ECE696/color_balance/Debug/Gr_CBout.txt','c:/Us
ers/Aaron/Documents/ECE696/MatLab/Gr_CBout.txt');
copyfile('c:/Users/Aaron/Documents/ECE696/color_balance/Debug/Bl_CBout.txt','c:/Use
rs/Aaron/Documents/ECE696/MatLab/Bl_CBout.txt');

Rd_22fromC=dlmread('Rd_22out.txt');
Bl_22fromC=dlmread('Bl_22out.txt');
Gr_22fromC=dlmread('Gr_22out.txt');
simple_22_fromC=cat(3,uint8(Rd_22fromC),uint8(Gr_22fromC),uint8(Bl_22fromC));

Rd_NNfromC=dlmread('Rd_NNout.txt');
Bl_NNfromC=dlmread('Bl_NNout.txt');
Gr_NNfromC=dlmread('Gr_NNout.txt');
nn_fromC=cat(3,uint8(Rd_NNfromC),uint8(Gr_NNfromC),uint8(Bl_NNfromC));
```

```
Rd_NN2fromC=dlmread('Rd_NN2out.txt');
Bl_NN2fromC=dlmread('Bl_NN2out.txt');
Gr_NN2fromC=dlmread('Gr_NN2out.txt');
nn2_fromC=cat(3,uint8(Rd_NN2fromC),uint8(Gr_NN2fromC),uint8(Bl_NN2fromC));

Rd_BLfromC=dlmread('Rd_BLout.txt');
Bl_BLfromC=dlmread('Bl_BLout.txt');
Gr_BLfromC=dlmread('Gr_BLout.txt');
bilinear_fromC=cat(3,uint8(Rd_BLfromC),uint8(Gr_BLfromC),uint8(Bl_BLfromC));

Rd_CBfromC=dlmread('Rd_CBout.txt');
Bl_CBfromC=dlmread('Bl_CBout.txt');
Gr_CBfromC=dlmread('Gr_CBout.txt');
color_balance_fromC=cat(3,uint8(Rd_CBfromC),uint8(Gr_CBfromC),uint8(Bl_CBfrom
C));

figure(15);imshow(simple_22_fromC);%title('Figure 15. Simple 2x2 from C');
figure(16);imshow(nn_fromC);%title('Figure 16. Nearest Neighbor from C');
figure(17);imshow(nn2_fromC);%title('Figure 17. Nearest Neighbor Modified from C');
figure(18);imshow(bilinear_fromC);%title('Figure 18. Bilinear from C');
figure(19);imshow(color_balance_fromC);%title('Figure 19. Color Balance from C');
```

```
%error_check

R_source=RGB_original(:,:,1);
R_simple=simple_22_fromC(:,:,1);
R_nn=nn_fromC(:,:,1);
R_nn2=nn2_fromC(:,:,1);
R_bilinear=bilinear_fromC(:,:,1);
R_colorBal=color_balance_fromC(:,:,1);

[M,N]=size(R_source);

%initialize
Rtot_simple=int32(0);
Rtot_nn=int32(0);
Rtot_nn2=int32(0);
Rtot_bilinear=int32(0);
Rtot_colorBal=int32(0);

%Simple_2x2 Sum of Squared Error
for j=1:1:M
   for k=1:1:N
      RSE_simple=(int32(R_source(j,k)-R_simple(j,k)))*(int32(R_source(j,k)-
R_simple(j,k)));
      Rtot_simple=Rtot_simple+RSE_simple;
   end
end

%Nearest Neighbor Sum of Squared Error
for j=1:1:M
   for k=1:1:N
      RSE_nn=(int32(R_source(j,k)-R_nn(j,k)))*(int32(R_source(j,k)-R_nn(j,k)));
      Rtot_nn=Rtot_nn+RSE_nn;
   end
end

%Nearest Neighbor modified Sum of Squared Error
for j=1:1:M
   for k=1:1:N
      RSE_nn2=(int32(R_source(j,k)-R_nn2(j,k)))*(int32(R_source(j,k)-R_nn2(j,k)));
      Rtot_nn2=Rtot_nn2+RSE_nn2;
   end
end

%Bilinear Sum of Squared Error
for j=1:1:M
   for k=1:1:N
```

```
        RSE_bilinear=(int32(R_source(j,k)-R_bilinear(j,k)))*(int32(R_source(j,k)-
R_bilinear(j,k)));
        Rtot_bilinear=Rtot_bilinear+RSE_bilinear;
    end
end

%Color Balance Sum of Squared Error
for j=1:1:M
    for k=1:1:N
        RSE_colorBal=(int32(R_source(j,k)-R_colorBal(j,k)))*(int32(R_source(j,k)-
R_colorBal(j,k)));
        Rtot_colorBal=Rtot_colorBal+RSE_colorBal;
    end
end

 %RMSE

RMSE_simple = sqrt(40450271 / (M*N))
RMSE_nn = sqrt(40450271 / (M*N))
RMSE_nn2 = sqrt(40450271 / (M*N))
RMSE_bilinear = sqrt(38734221 / (M*N))
RMSE_colorBal = sqrt(361768741 / (M*N))

%PSNR

PSNR_simple = 20*log10(255/RMSE_simple)
PSNR_nn = 20*log10(255/RMSE_nn)
PSNR_nn2 = 20*log10(255/RMSE_nn2)
PSNR_bilinear = 20*log10(255/RMSE_bilinear)
PSNR_colorBal = 20*log10(255/RMSE_colorBal)
```

APPENDIX B: Code Composer Studio Code

```
//Simple 2x2 interpolation
#include <stdio.h>
#include <math.h>
#include "Rd.h"
#include "Gr.h"
#include "Bl.h"

main()
{
	int i,j,k;
	int Row = 324;
	int Col = 432;
	FILE *Rd_out;
	FILE *Bl_out;
	FILE *Gr_out;




//interpolate red/blu at grn pixel; odd row/col;

	for (j=0; j<Row; j+=2)
	{
			for (k=0; k<Col; k+=2)
				{
					Rd[j][k]=Rd[j][k+1];
					Bl[j][k]=Bl[j+1][k];
				}
	}

//interpolate red/blu at grn pixel; even rol/col;

	for (j=1; j<Row; j+=2)
	{
			for (k=1; k<Col; k+=2)
				{
					Rd[j][k]=Rd[j-1][k];
					Bl[j][k]=Bl[j][k-1];
				}
	}

//interpolate grn/blu at red pixel; odd row, even col;

	for (j=0; j<Row; j+=2)
```

```
	{
			for (k=1; k<Col; k+=2)
				{
					Gr[j][k]=(Gr[j][k-1] + Gr[j+1][k])/2;
					Bl[j][k]=Bl[j+1][k-1];
				}
	}

//interpolate grn/red at blu pixel; even row, odd col;

	for (j=1; j<Row; j+=2)
	{
			for (k=0; k<Col; k+=2)
				{
					Gr[j][k]=(Gr[j-1][k] + Gr[j][k+1])/2;
					Rd[j][k]=Rd[j-1][k+1];
				}
	}

//output file
	printf("Writing Rd_22out.txt...\n");
	Rd_out=fopen("Rd_22out.txt","w");
	if(Rd_out==NULL)
	{
		puts("Error creating file");
		return(1);
	}

	for (i=0; i<Row; i++)
	{
		for (j=0; j<Col; j++)
		{
			fprintf(Rd_out,"%d",Rd[i][j]);
				if(j!=(Col-1))
				{
					fprintf(Rd_out,",");
				}
						else
							{
								fprintf(Rd_out,"\n");

							}
		}
	}
		fclose(Rd_out);
	printf("Rd_22out.txt generated\n\n");
```

```
printf("Writing Gr_22out.txt...\n");
Gr_out=fopen("Gr_22out.txt","w");
if(Gr_out==NULL)
{
        puts("Error creating file");
        return(1);
}

for (i=0; i<Row; i++)
{
        for (j=0; j<Col; j++)
        {
                fprintf(Gr_out,"%d",Gr[i][j]);
                        if(j!=(Col-1))
                        {
                                fprintf(Gr_out,",");
                        }
                                        else
                                        {
                                                fprintf(Gr_out,"\n");

                                        }
        }
}
        fclose(Gr_out);
printf("Gr_22out.txt generated\n\n");

printf("Writing Bl_22out.txt...\n");
Bl_out=fopen("Bl_22out.txt","w");
if(Bl_out==NULL)
{
        puts("Error creating file");
        return(1);
}

for (i=0; i<Row; i++)
{
        for (j=0; j<Col; j++)
        {
                fprintf(Bl_out,"%d",Bl[i][j]);
                        if(j!=(Col-1))
                        {
                                fprintf(Bl_out,",");
                        }
                                        else
```

```
                                                            {
                                                                    fprintf(Bl_out,"\n");

                                                            }
                        }
                }
                        fclose(Bl_out);
                        printf("Bl_22out.txt generated");
                        return(0);

/*              for (i=0; i<Row; i++)
                {
                        for (j=0; j<Col; j++)
                        {
                                printf("%d ", Rd[i][j]);
                        }
                        printf("\n");
                }
*/
}
```

```
//Nearest neighbor interpolation
#include <stdio.h>
#include <math.h>
#include "Rd.h"
#include "Gr.h"
#include "Bl.h"

main()
{
        int i,j,k;
        int Row = 324;
        int Col = 432;
        FILE *Rd_out;
        FILE *Bl_out;
        FILE *Gr_out;



//interpolate red/blu at grn pixel; odd row/col;

        for (j=0; j<Row; j+=2)
                {
                        for (k=0; k<Col; k+=2)
                                {
                                        Rd[j][k]=Rd[j][k+1];
                                        Bl[j][k]=Bl[j+1][k];
                                }
                }

//interpolate red/blu at grn pixel; even row/col;

        for (j=1; j<Row; j+=2)
                {
                        for (k=1; k<Col; k+=2)
                                {
                                        Rd[j][k]=Rd[j-1][k];
                                        Bl[j][k]=Bl[j][k-1];
                                }
                }

//interpolate grn/blu at red pixel; odd row, even col;

        for (j=0; j<Row; j+=2)
                {
                        for (k=1; k<Col; k+=2)
                                {
```

```
                    Gr[j][k]=Gr[j][k-1];
                    Bl[j][k]=Bl[j+1][k-1];
                }
        }

//interpolate grn/red at blu pixel; even row, odd col;

    for (j=1; j<Row; j+=2)
        {
            for (k=0; k<Col; k+=2)
                {
                    Gr[j][k]=Gr[j-1][k];
                    Rd[j][k]=Rd[j-1][k+1];
                }
        }

//output file
    printf("Writing Rd_NNout.txt...\n");
    Rd_out=fopen("Rd_NNout.txt","w");
    if(Rd_out==NULL)
        {
            puts("Error creating file");
            return(1);
        }

    for (i=0; i<Row; i++)
        {
            for (j=0; j<Col; j++)
            {
                fprintf(Rd_out,"%d",Rd[i][j]);
                    if(j!=(Col-1))
                    {
                        fprintf(Rd_out,",");
                    }
                        else
                            {
                                fprintf(Rd_out,"\n");

                            }
            }
        }
    fclose(Rd_out);
    printf("Rd_NNout.txt generated\n\n");

    printf("Writing Gr_NNout.txt...\n");
    Gr_out=fopen("Gr_NNout.txt","w");
```

```
if(Gr_out==NULL)
	{
		puts("Error creating file");
		return(1);
	}

for (i=0; i<Row; i++)
	{
		for (j=0; j<Col; j++)
		{
			fprintf(Gr_out,"%d",Gr[i][j]);
				if(j!=(Col-1))
				{
					fprintf(Gr_out,",");
				}
					else
						{
							fprintf(Gr_out,"\n");

						}
		}
	}
fclose(Gr_out);
printf("Gr_NNout.txt generated\n\n");

printf("Writing Bl_NNout.txt...\n");
Bl_out=fopen("Bl_NNout.txt","w");
if(Bl_out==NULL)
	{
		puts("Error creating file");
		return(1);
	}

for (i=0; i<Row; i++)
	{
		for (j=0; j<Col; j++)
		{
			fprintf(Bl_out,"%d",Bl[i][j]);
				if(j!=(Col-1))
				{
					fprintf(Bl_out,",");
				}
					else
						{
							fprintf(Bl_out,"\n");
```

```
                        }
                }
            }
        fclose(Bl_out);
        printf("Bl_NNout.txt generated");
        return(0);
}
```

```
//Nearest neighbor interpolation, modified.
#include <stdio.h>
#include <math.h>
#include "Rd.h"
#include "Gr.h"
#include "Bl.h"

main()
{
	int i,j,k;
	int Row = 324;
	int Col = 432;
	FILE *Rd_out;
	FILE *Bl_out;
	FILE *Gr_out;



//interpolate red/blu at grn pixel; odd row/col;

	for (j=0; j<Row; j+=2)
		{
			for (k=0; k<Col; k+=2)
				{
					Rd[j][k]=Rd[j][k+1];
					Bl[j][k]=Bl[j+1][k];
				}
		}

//inerpolate red/blu at grn pixel; even row/col;

	for (j=1; j<Row; j+=2)
		{
			for (k=1; k<Col; k+=2)
				{
					Rd[j][k]=Rd[j-1][k];
					Bl[j][k]=Bl[j][k-1];
				}
		}

//interpolate grn/blu at red pixel; odd row, even col;

	for (j=0; j<Row; j+=2)
		{
			for (k=1; k<Col; k+=2)
				{
```

```
                    Gr[j][k]=Gr[j+1][k];
                    Bl[j][k]=Bl[j+1][k-1];
                }
        }

//interpolate grn/red at blu pixel; even row, odd col;

    for (j=1; j<Row; j+=2)
        {
            for (k=0; k<Col; k+=2)
                {
                    Gr[j][k]=Gr[j][k+1];
                    Rd[j][k]=Rd[j-1][k+1];
                }
        }

//output file
    printf("Writing Rd_NN2out.txt...\n");
    Rd_out=fopen("Rd_NN2out.txt","w");
    if(Rd_out==NULL)
        {
            puts("Error creating file");
            return(1);
        }

    for (i=0; i<Row; i++)
        {
            for (j=0; j<Col; j++)
            {
                fprintf(Rd_out,"%d",Rd[i][j]);
                    if(j!=(Col-1))
                    {
                        fprintf(Rd_out,",");
                    }
                        else
                            {
                                fprintf(Rd_out,"\n");

                            }
            }
        }
    fclose(Rd_out);
    printf("Rd_NN2out.txt generated\n\n");

    printf("Writing Gr_NN2out.txt...\n");
    Gr_out=fopen("Gr_NN2out.txt","w");
```

```
	if(Gr_out==NULL)
		{
			puts("Error creating file");
			return(1);
		}

for (i=0; i<Row; i++)
	{
		for (j=0; j<Col; j++)
		{
			fprintf(Gr_out,"%d",Gr[i][j]);
				if(j!=(Col-1))
				{
					fprintf(Gr_out,",");
				}
					else
						{
							fprintf(Gr_out,"\n");

						}
		}
	}
fclose(Gr_out);
printf("Gr_NN2out.txt generated\n\n");

printf("Writing Bl_NN2out.txt...\n");

Bl_out=fopen("Bl_NN2out.txt","w");
	if(Bl_out==NULL)
		{
			puts("Error creating file");
			return(1);
		}

for (i=0; i<Row; i++)
	{
		for (j=0; j<Col; j++)
		{
			fprintf(Bl_out,"%d",Bl[i][j]);
				if(j!=(Col-1))
				{
					fprintf(Bl_out,",");
				}
					else
						{
							fprintf(Bl_out,"\n");
```

```
                                        }
                        }
                }
        fclose(Bl_out);
        printf("Bl_NN2out.txt generated");
        return(0);
}
```

```
//Bilinear interpolation
#include <stdio.h>
#include <math.h>
#include "Rd.h"
#include "Gr.h"
#include "Bl.h"

main()
{
        int i,j,k;
        int Row = 323;
        int Col = 431;
        FILE *Rd_out;
        FILE *Bl_out;
        FILE *Gr_out;



//interpolate red/blu at grn pixel
//row 1

        Rd[0][0]=Rd[0][1];
        Bl[0][0]=Bl[1][0];

                for (k=2; k<Col; k+=2)
                        {
                                Rd[0][k]=(Rd[0][k-1] + Rd[0][k+1])/2;
                                Bl[0][k]=Bl[1][k];
                        }

                if ((round(Col/2)<=Col/2 + 1)&&(round(Col/2)>=Col/2 - 1))
                        {
                                Rd[0][Col]=Rd[0][Col-1];
                                Bl[0][Col]=Bl[1][Col];
                        }

//odd rows

        for (j=2; j<Row; j+=2)
                {
                        Rd[j][0]=Rd[j][1];
                        Bl[j][0]=(Bl[j-1][0] + Bl[j+1][0])/2;
                                for (k=2; k<Col; k+=2)
                                        {
                                                Rd[j][k]=(Rd[j][k-1] + Rd[j][k+1])/2;
                                                Bl[j][k]=(Bl[j][k-1] + Bl[j][k+1])/2;
```

```
                    }
                        if ((round(Col/2)<=Col/2 +
1)&&(round(Col/2)>=Col/2 - 1))
                            {
                                Rd[j][Col]=Rd[j][Col-1];
                                Bl[j][Col]=(Bl[j-1][Col] +
Bl[j+1][Col])/2;
                            }
        }

//last row odd

    if ((round(Row/2)<=Row/2 + 1)&&(round(Row/2)>=Row/2 - 1))
        {
            Rd[Row][0]=Rd[Row][1];
            Bl[Row][0]=Bl[Row-1][0];
                for (k=2; k<Col; k+=2)
                    {
                        Rd[Row][k]=(Rd[Row][k-1] +
Rd[Row][k+1])/2;
                        Bl[Row][k]=Bl[Row-1][k];
                    }
                        if ((round(Col/2)<=Col/2 +
1)&&(round(Col/2)>=Col/2 - 1))
                            {

    Rd[Row][Col]=Rd[Row][Col-1];
                                Bl[Row][Col]=Bl[Row-
1][Col];
                            }
        }

//even rows

    for (j=1; j<Row; j+=2)
        {
            for (k=1; k<Col; k+=2)
                {
                    Rd[j][k]=(Rd[j-1][k] + Rd[j+1][k])/2;
                    Bl[j][k]=(Bl[j][k-1] + Bl[j][k+1])/2;
                }
                    if (round(Col/2)==Col/2)
                        {
                            Rd[j][Col]=(Rd[j-
1][Col]+Rd[j+1][Col])/2;
                            Bl[j][Col]=Bl[j][Col-1];
```

```
                                        }
                }

//last row even

        if (round(Row/2)==Row/2)
                {
                        for (k=1; k<Col; k+=2)
                                {
                                        Rd[Row][k]=Rd[Row-1][k];
                                        Bl[Row][k]=(Bl[Row][k-1]+Bl[Row][k+1])/2;
                                }
                                        if (round(Col/2)==Col/2)
                                                {
                                                        Rd[Row][Col]=Rd[Row-1][Col];
                                                        Bl[Row][Col]=Bl[Row-1][Col-1];
                                                }
                }

//interpolate grn/blu at red pixel; odd row, even column
//row 1

        for (k=1; k<Col; k+=2)
                {
                        Gr[0][k]=(Gr[0][k-1] + Gr[0][k+1] + Gr[1][k])/3;
                        Bl[0][k]=(Bl[1][k-1] + Bl[1][k+1])/2;
                }

//other pixels

        for (j=2; j<Row; j+=2)
                {
                        for (k=1; k<Col; k+=2)
                                {
                                        Gr[j][k]=(Gr[j-1][k] + Gr[j+1][k] + Gr[j][k-1] +
Gr[j][k+1])/2;
                                        Bl[j][k]=(Bl[j-1][k-1] + Bl[j-1][k+1] + Bl[j+1][k-1]
+ Bl[j+1][k+1])/4;
                                }
                }

//last row/column
//red pixel last column if col even

        if (round(Col/2)==Col/2)
                {
```

```
                    Gr[0][Col]=(Gr[0][Col-1] + Gr[1][Col])/2;
                    Bl[0][Col]=Bl[1][Col-1];
                              if ((round(Row/2)<=Row/2 +
1)&&(round(Row/2)>=Row/2 - 1))
                                        {
                                                  Gr[Row][Col]=(Gr[Row][Col-1] + Gr[Row-
1][Col])/2;
                                                  Bl[Row][Col]=Bl[Row-1][Col-1];
                                        }
          }

//red pixel last row if row odd

     if ((round(Row/2)<=Row/2 + 1)&&(round(Row/2)>=Row/2 - 1))
          {
                    for (k=1; k<Col; k+=2)
                              {
                                        Gr[Row][k]=(Gr[Row][k-1] + Gr[Row-1][k] +
Gr[Row][k+1])/2;
                                        Bl[Row][k]=(Bl[Row-1][k-1]+Bl[Row][k+1])/2;
                              }
          }

//interpolate grn/red at blu pixel; even row, odd col
//row 1
     for (j=1; j<Row; j+=2)
          {
                    Gr[j][0]=(Gr[j-1][0] + Gr[j][1] + Gr[j+1][0])/3;
                    Rd[j][0]=(Rd[j-1][1] + Rd[j+1][1])/2;
          }

//other pixels
     for (j=1; j<Row; j+=2)
          {
                    for (k=2; k<Col; k+=2)
                              {
                                        Gr[j][k]=(Gr[j-1][k] + Gr[j][k-1] + Gr[j][k+1] +
Gr[j+1][k])/4;
                                        Rd[j][k]=(Rd[j-1][k-1] + Rd[j-1][k+1] + Rd[j+1][k-
1] + Rd[j+1][k+1])/4;
                              }
          }

//last row/column
//blu pixel last column if column odd
```

```
	if ((round(Col/2)<=Col/2 + 1)&&(round(Col/2)>=Col/2 - 1))
		{
			if (round(Row/2)==Row/2)
				{
					Rd[Row][Col]=Rd[Row-1][Col-1];
					Gr[Row][Col]=(Gr[Row][Col-1] + Gr[Row-1][Col-
1])/2;
				}
					for (k=1; k<Col; k+=2)
						{
							Rd[Row][k]=(Rd[Row-1][k-1] +
Rd[Row-1][k+1])/2;
							Gr[Row][k]=(Gr[Row][k-1] +
Gr[Row-1][k] + Gr[Row][k+1])/3;
						}
		}

//blue pixel last row if row is even

	if (round(Row/2)==Row/2)
		{
			Rd[Row][0] = Rd[Row-1][1];
			Gr[Row][0] = (Gr[Row-1][0] + Gr[Row][1])/2;
				for (k=2; k<Col; k+=2)
					{
						Rd[Row][k]=(Rd[Row-1][k-1] + Rd[Row-
1][k+1])/2;
						Gr[Row][k]=(Gr[Row][k-1] +Gr[Row-1][k]
+ Gr[Row][k+1])/3;
					}
		}

//output file
	printf("Writing Rd_BLout.txt...\n");
	Rd_out=fopen("Rd_BLout.txt","w");
	if(Rd_out==NULL)
	{
		puts("Error creating file");
		return(1);
	}

	for (i=0; i<=Row; i++)
	{
		for (j=0; j<=Col; j++)
		{
			fprintf(Rd_out,"%d",Rd[i][j]);
```

```
                    if(j!=(Col-1))
                    {
                              fprintf(Rd_out,",");
                    }
                                        else
                                                  {
                                                            fprintf(Rd_out,"\n");

                                                  }
          }
}
          fclose(Rd_out);
printf("Rd_BLout.txt generated\n\n");

printf("Writing Gr_BLout.txt...\n");
Gr_out=fopen("Gr_BLout.txt","w");
if(Gr_out==NULL)
{
          puts("Error creating file");
          return(1);
}

for (i=0; i<=Row; i++)
{
          for (j=0; j<=Col; j++)
          {
                    fprintf(Gr_out,"%d",Gr[i][j]);
                              if(j!=(Col-1))
                              {
                                        fprintf(Gr_out,",");
                              }
                                                  else
                                                            {
                                                                      fprintf(Gr_out,"\n");

                                                            }
          }
}
          fclose(Gr_out);
printf("Gr_BLout.txt generated\n\n");

printf("Writing Bl_BLout.txt...\n");
Bl_out=fopen("Bl_BLout.txt","w");
if(Bl_out==NULL)
{
          puts("Error creating file");
```

```
        return(1);
    }

    for (i=0; i<=Row; i++)
    {
        for (j=0; j<=Col; j++)
        {
            if (Bl[i][j]>255)
                {
                    Bl[i][j]=0;
                }
            fprintf(Bl_out,"%d",Bl[i][j]);
                if(j!=(Col-1))
                {
                    fprintf(Bl_out,",");
                }
                        else
                            {
                                fprintf(Bl_out,"\n");

                            }
        }
    }
        fclose(Bl_out);
        printf("Bl_BLout.txt generated");
        return(0);
}
```

```
//Color balance interpolation
#include <stdio.h>
#include <math.h>
#include "Rd.h"
#include "Gr.h"
#include "Bl.h"

main()
{
        int i,j,k;
        int Row = 324;
        int Col = 432;
        FILE *Rd_out;
        FILE *Bl_out;
        FILE *Gr_out;


//interpolate red/blu at grn pixel
//row 1

        Rd[0][0]=Rd[0][1] - Gr[0][0];
        Bl[0][0]=Bl[1][0] - Gr[0][0];
        Gr[0][0]=(Gr[1][1] - Gr[0][0]) + Gr[0][0]/2;

                for (k=2; k<Col-1; k+=2)
                        {
                                Rd[0][k]=(Rd[0][k-1] + Rd[0][k+1] - Gr[1][k])/2;
                                Bl[0][k]=Bl[1][k] - Gr[0][k];
                                Gr[0][k]= Gr[1][k-1] + Gr[0][k+1] - Gr[0][k] +
(2*Gr[0][k])/4;
                        }

                if ((round(Col/2)<=Col/2 + 1)&&(round(Col/2)>=Col/2 - 1))
                        {
                                Rd[0][Col]=Rd[0][Col-1] - Gr[0][Col];
                                Bl[0][Col]=Bl[1][Col] - Gr[0][Col];
                                Gr[0][Col]=(Gr[1][Col] - Gr[0][Col]) + (Gr[0][Col])/2;
                        }

//odd rows

        for (j=2; j<Row-1; j+=2)
                {
                        Rd[j][0]=Rd[j][1] - Gr[j][1];
                        Bl[j][0]=(Bl[j-1][0] + Bl[j+1][0] - Gr[j][0])/2;
                        Gr[j][0]=(Gr[j-1][1] + Gr[j+1][1] - Gr[j][0] + (2*Gr[j][0]))/4;
```

```
                                for (k=2; k<Col-1; k+=2)
                                        {
                                                Rd[j][k]=(Rd[j][k-1] + Rd[j][k+1] -
Gr[j][k])/2;
                                                Bl[j][k]=(Bl[j][k-1] + Bl[j][k+1] -
Gr[j][k])/2;
                                                Gr[j][k] =(Gr[j-1][k-1] + Gr[j-1][k+1] +
Gr[j+1][k-1] + Gr[j+1][k+1] - Gr[j][k] + (4*Gr[j][k]))/8;

                                        }
                                                if ((round(Col/2)<=Col/2 +
1)&&(round(Col/2)>=Col/2 - 1))
                                                        {
                                                                Rd[j][Col]=Rd[j][Col-1] -
Gr[j][Col];
                                                                Bl[j][Col]=(Bl[j-1][Col] +
Bl[j+1][Col]) - Gr[j][Col-1]/2;
                                                                Gr[j][Col]=(Gr[j-1][Col-1] +
Gr[j-1][Col+1] - Gr[j][Col] + (2*Gr[j][Col]))/4;
                                                        }
                }

//last row odd

        if ((round(Row/2)<=Row/2 + 1)&&(round(Row/2)>=Row/2 - 1))
        {
                Rd[Row][0]=Rd[Row][1] - Gr[Row][0];
                Bl[Row][0]=Bl[Row-1][0] - Gr[Row][0];
                Gr[Row][0]=Gr[Row-1][1];
                for (k=2; k<Col-1; k+=2)
                {
                        Rd[Row][k]=(Rd[Row][k-1] + Rd[Row][k+1] - Gr[Row][k])/2;
                Bl[Row][k]=Bl[Row][k] - Gr[Row-1][k];
                        Gr[Row][k]=(Gr[Row-1][k-1] + Gr[Row-1][k+1] - Gr[Row][k]
+(2*Gr[Row][k]))/4;

                }
                if ((round(Col/2)<=Col/2 + 1)&&(round(Col/2)>=Col/2 - 1))
                {
                        Rd[Row][Col]=Rd[Row][Col-1] - Gr[Row][Col];
                        Bl[Row][Col]=Bl[Row-1][Col] - Gr[Row][Col];
                        Gr[Row][Col]=(Gr[Row-1][Col-1] - Gr[Row-1][k+1] -
Gr[Row][Col] + (2*Gr[Row][Col]))/4;
                }
        }
```

```
//even rows

	for (j=1; j<Row-1; j+=2)
		{
			for (k=1; k<Col-1; k+=2)
				{
					Rd[j][k]=(Rd[j-1][k] + Rd[j+1][k] - Gr[j][k])/2;
					Bl[j][k]=(Bl[j][k-1] + Bl[j][k+1] - Gr[j][k])/2;
					Gr[j][k]=(Gr[j-1][k-1] + Gr[j-1][k+1] + Gr[j+1][k-
1] + Gr[j+1][k+1] - Gr[j][k] + (4*Gr[j][k]))/8;
				}
					if (round(Col/2)==Col/2)
						{
							Rd[j][Col]=(Rd[j-
1][Col]+Rd[j+1][Col]) - Gr[j][Col]/2;
							Bl[j][Col]=Bl[j][Col] - Gr[j][Col];
							Gr[j][Col]=(Gr[j-1][Col-1] + Gr[j-
1][Col] + Gr[j+1][Col-1] - Gr[j+1][Col] - Gr[j][Col] + (4*Gr[j][Col]))/8;
						}
		}

//last row even

	if (round(Row/2)==Row/2)
		{
			for (k=1; k<Col-1; k+=2)
				{
					Rd[Row][k]=Rd[Row-1][k] - Gr[Row][Col];
					Bl[Row][k]=Bl[Row][k];
					Gr[Row][k]=(Gr[Row-1][k-1] + Gr[Row-1][k+1] -
Gr[Row][Col] + (2*Gr[Row][k]))/4;
				}
					if (round(Col/2)==Col/2)
						{
							Rd[Row][Col]=Rd[Row-1][Col] -
Gr[Row][Col];
							Bl[Row][Col]=Bl[Row-1][Col-1] -
Gr[Row][Col];
							Gr[Row][Col]=(Gr[Row-1][Col-1] +
Gr[Row-1][Col] - Gr[Row][Col] + (2*Gr[Row][Col]))/4;
						}
		}

//interpolate grn/blu at red pixel; odd row, even column
//row 1
```

```
	for (k=1; k<Col-1; k+=2)
		{
			Gr[0][k]=((Gr[0][k-1] + Gr[0][k+1] - Rd[0][k]) + (Gr[1][k] -
Rd[0][k]))/3;
			Bl[0][k]=((Bl[1][k-1] - Rd[0][k]) + (Bl[1][k+1] - Rd[0][k]))/2;
		}

//other pixels

	for (j=2; j<Row; j+=2)
		{
			for (k=1; k<Col-1; k+=2)
				{
					Gr[j][k]=((Gr[j-1][k] + Gr[j+1][k] - Rd[j][k]) +
(Gr[j][k-1] + Gr[j][k+1] - Rd[j][k]))/4;
					Bl[j][k]=(Bl[j-1][k-1] + Bl[j-1][k+1] + Bl[j+1][k-1]
+ Bl[j+1][k+1] - Rd[j][k])/4;
				}
		}

//last row/column
//red pixel last column if col even

	if (round(Col/2)==Col/2)
		{
			Gr[0][Col]=(Gr[0][Col-1] - Rd[0][Col] + Gr[1][Col]-
Rd[0][Col])/2;
			Bl[0][Col]=(Bl[1][Col-1] - Rd[0][Col]);
				if ((round(Row/2)<=Row/2 +
1)&&(round(Row/2)>=Row/2 - 1))
					{
						Gr[Row][Col]=(Gr[Row][Col-1] -
Rd[Row][Col]) + (Gr[Row-1][Col] - Rd[Row][Col])/2;
						Bl[Row][Col]=Bl[Row-1][Col-1] -
Rd[Row][Col];
					}
		}

//red pixel last row if row odd

	if ((round(Row/2)<=Row/2 + 1)&&(round(Row/2)>=Row/2 - 1))
		{
			for (k=1; k<Col-1; k+=2)
				{
					Gr[Row][k]=((Gr[Row][k-1] + Gr[Row-1][k] -
Rd[Row][k]) + (Gr[Row][k+1] - Rd[Row][k]))/3;
```

```
                                        Bl[Row][k]=Bl[Row][k];
                                }
                        }

//interpolate grn/red at blu pixel; even row, odd col
//row 1
        for (j=1; j<Row-1; j+=2)
                {
                        Gr[j][0]=(Gr[j-1][0] + Gr[j][1] + Gr[j+1][0])/3;
                        Rd[j][0]=(Rd[j-1][1] + Rd[j+1][1])/2;
                }

//other pixels
        for (j=1; j<Row-1; j+=2)
                {
                        for (k=2; k<Col; k+=2)
                                {
                                        Gr[j][k]=(Gr[j-1][k] + Gr[j][k-1] + Gr[j][k+1] +
Gr[j+1][k])/4;
                                        Rd[j][k]=(Rd[j-1][k-1] + Rd[j-1][k+1] + Rd[j+1][k-
1] + Rd[j+1][k+1])/4;
                                }
                }

//last row/column
//blu pixel last column if column odd

        if ((round(Col/2)<=Col/2 + 1)&&(round(Col/2)>=Col/2 - 1))
                {
                        if (round(Row/2)==Row/2)
                                {
                                        Rd[Row][Col]=Rd[Row-1][Col-1];
                                        Gr[Row][Col]=(Gr[Row][Col-1] + Gr[Row-1][Col-
1])/2;
                                }
                                        for (k=1; k<Col; k+=2)
                                                {
                                                        Rd[Row][k]=(Rd[Row-1][k-1] +
Rd[Row-1][k+1])/2;
                                                        Gr[Row][k]=(Gr[Row][k-1] +
Gr[Row-1][k] + Gr[Row][k+1])/3;
                                                }
                }

//blue pixel last row if row is even
```

```
	if (round(Row/2)==Row/2)
		{
			Rd[Row][0] = Rd[Row-1][1];
			Gr[Row][0] = (Gr[Row-1][0] + Gr[Row][1])/2;
				for (k=2; k<Col; k+=2)
					{
						Rd[Row][k]=(Rd[Row-1][k-1] + Rd[Row-
1][k+1])/2;

						Gr[Row][k]=(Gr[Row][k-1] +Gr[Row-1][k]
+ Gr[Row][k+1])/3;
					}
		}

//output file
	printf("Writing Rd_CBout.txt...\n");
	Rd_out=fopen("Rd_CBout.txt","w");
	if(Rd_out==NULL)
	{
		puts("Error creating file");
		return(1);
	}

	for (i=0; i<Row; i++)
	{
		for (j=0; j<Col; j++)
		{
				if (Rd[i][j]>255)
				{
					Rd[i][j]=0;
				}
			fprintf(Rd_out,"%d",Rd[i][j]);
				if(j!=(Col-1))
				{
					fprintf(Rd_out,",");
				}
						else
							{
								fprintf(Rd_out,"\n");

							}
		}
	}
	fclose(Rd_out);
	printf("Rd_CB.txt generated\n\n");
```

```
printf("Writing Gr_CBout.txt...\n");
        Gr_out=fopen("Gr_CBout.txt","w");
        if(Gr_out==NULL)
        {
                puts("Error creating file");
                return(1);
        }

        for (i=0; i<Row; i++)
        {
                for (j=0; j<Col; j++)
                {
                                if (Gr[i][j]>255)
                                {
                                        Gr[i][j]=0;
                                }
                        fprintf(Gr_out,"%d",Gr[i][j]);
                                if(j!=(Col-1))
                                {
                                        fprintf(Gr_out,",");
                                }
                                                else
                                                        {
                                                                fprintf(Gr_out,"\n");

                                                        }
                }
        }
                fclose(Gr_out);
                printf("Gr_CB.txt generated\n\n");

                printf("Writing Bl_CBout.txt...\n");
        Bl_out=fopen("Bl_CBout.txt","w");
        if(Bl_out==NULL)
        {
                puts("Error creating file");
                return(1);
        }

        for (i=0; i<Row; i++)
        {
                for (j=0; j<Col; j++)
                {
                                if (Bl[i][j]>255)
                                {
                                        Bl[i][j]=0;
```

```
                    }
                fprintf(Bl_out,"%d",Bl[i][j]);
                    if(j!=(Col-1))
                    {
                        fprintf(Bl_out,",");
                    }
                            else
                            {
                                fprintf(Bl_out,"\n");

                            }
            }
        }
            fclose(Bl_out);
            printf("Bl_CB.txt generated");
            return(0);
}
```

```
//format2Darray

#include <stdio.h>

int main()
{
	FILE * inFile1;
	FILE * inFile2;
	FILE * inFile3;
	FILE * outFile1;
	FILE * outFile2;
	FILE * outFile3;

	inFile1 = fopen("c:/Users/Aaron/Documents/ECE696/MatLab/RED.txt","r");
	outFile1 = fopen("Rd.h","w");

	inFile2 = fopen("c:/Users/Aaron/Documents/ECE696/MatLab/GRN.txt","r");
	outFile2 = fopen("Gr.h","w");

	inFile3 = fopen("c:/Users/Aaron/Documents/ECE696/MatLab/BLU.txt","r");
	outFile3 = fopen("Bl.h","w");

	if(inFile1!=NULL)
	{
		char  c;
		fprintf(outFile1,"unsigned short Rd[324][432]={{"); //one to open entire
array, other for first array
		do{
			c = fgetc(inFile1);
			if(c=='\n'||c=='\r')
			{
				fprintf(outFile1,"},{");
			}
			else fputc(c,outFile1);

		}while(c!=EOF);
		fprintf(outFile1,"}}"); //one to close entire array, other for end of last array

	}
	else
	{
		perror ("Error opening file1");
	}

	fclose(inFile1);
	fclose(outFile1);
```

```
//--------------------------------------------------------------
	if(inFile2!=NULL)
	{
		char  c2;
		fprintf(outFile2,"unsigned short Gr[324]432]={{"); //one to open entire
array, other for first array
		do{
			c2 = fgetc(inFile2);
			if(c2=='\n'||c2=='\r')
			{
				fprintf(outFile2,"},{");
			}
			else fputc(c2,outFile2);

		}while(c2!=EOF);
		fprintf(outFile2,"}}"); //one to close entire array, other for end of last array

	}
	else
	{
		perror ("Error opening file2");
	}

	fclose(inFile2);
	fclose(outFile2);
//------------------------------------------------------------
	if(inFile3!=NULL)
	{
		char  c3;
		fprintf(outFile3,"unsigned short Bl[324][432]={{"); //one to open entire
array, other for first array
		do{
			c3 = fgetc(inFile3);
			if(c3=='\n'||c3=='\r')
			{
				fprintf(outFile3,"},{");
			}
			else fputc(c3,outFile3);

		}while(c3!=EOF);
		fprintf(outFile3,"}}"); //one to close entire array, other for end of last array

	}
	else
	{
		perror ("Error opening file3");
```

```
        }

        fclose(inFile3);
        fclose(outFile3);

        return 0;
}
```