MIDDLE EAST TECHNICAL UNIVERSITY

DEPARTMENT OF COMPUTER ENGINEERING

CENG 566 – Fall 2004

IMAGE PROCESSING

FINAL PROJECT

MORPHOLOGICAL OPERATIONS ON COLOR IMAGES

Instructor: Associate Prof. Dr. Sibel TARI

Prepared By: Sezen ERDEM
TABLE OF CONTENTS

TABLE OF CONTENTS .......................................................................................................2

A. MATHEMATICAL MORPHOLOGY .............................................................................3

B. COLOR MORPHOLOGY ..............................................................................................3

  B.1. Mathematical Morphology on RGB Components ............................................4

  B.2. Mathematical Morphology on RGB Components as Vector ............................5

  B.3. Mathematical Morphology on L*m*p* Components .......................................5

C. CONCLUSION .............................................................................................................8

D. REFERENCES ..............................................................................................................9

APPENDIX I ....................................................................................................................10

APPENDIX II ..................................................................................................................14
A. MATHEMATICAL MORPHOLOGY

Mathematical morphology is a powerful methodology for the processing and analyzing geometric structure in signals and images. In image processing, it is important to be able to extract features, describe shapes and recognize patterns. Such tasks refer to geometrical concepts such as size, shape, and orientation. Mathematical morphology uses concepts from set theory, geometry and topology to analyze geometrical structures in an image. In mathematical morphology the geometrical structure of an image is examined by using small patterns called structuring element. These operations lead to non linear image operators which are used for exploring geometrical structures in the image. These non linear operators are applied to images successively in order to make certain features apparent, distinguishing meaningful information from irrelevant distortions. The operations are in fact non linear transformations that modify the geometrical features of an image. Some of these transformations that mathematical morphology concerns are erosion, dilation, opening, closing, hit or miss transformations, skeletonization and watershed transformations.

B. COLOR MORPHOLOGY

Mathematical Morphology is defined in terms of complete lattices. So it generally supports the images which have lattice structure. For an image having lattice structure means having apparent partial ordering characteristics. Mathematical morphology can easily be applied on binary images or single valued intensity images. Because we can define an ordering between the pixel values of the image. Also the objects in a binary image can be viewed as sets. In binary images we have a background and objects. So there are two different sets. When we are given a binary image I, and 2-D structuring element SE, the dilated image is the union of all the pixels that fall under structuring element SE when we move the center of SE into each pixel of I. Similarly the eroded image is the intersection of pixels that fall under structuring element SE. As it can be observed, applying morphological operations on binary images is very easy.

For gray scale images, the situation is similar. We only deal with a single intensity value. Morphological Operations on gray scale images are an extension of the operations on binary images. Grayscale morphological operations are an extension of binary morphological operations to grayscale images. In grey level the image can be viewed as a function f(x,y) = c, where x, y is index values for pixel location and c is the intensity value of the pixel. Erosion
operation is simply assigning the minimum c value found on structuring element neighborhood. Similarly dilation operation is assigning the maximum c value in the structuring element neighborhood. But in gray scale case structuring element plays a bit different role. The values in the structuring element are subtracted from pixel value of the input image to find the minimum in erosion operation.

For color images, partial ordering is not as easy as it is for binary or grey scale images. Because there is no single value that we can build a partial ordering of the values. Also we can not easy partition image into sets. So the morphological operations can not be directly applied on color images. However there are some techniques which can be extended for color images to apply morphological operations. In my Project I will especially deal with three different techniques implementing Mathematical Morphology on color images.

**B.1. Mathematical Morphology on RGB Components**

In this approach, morphological operators are applied on each color value of the RGB image. Namely, we apply operators separately on R, G, B components of the color image. This is the most easiest way to apply morphological operators on color image. First I want to give what erosion and dilation mean when applied to each color component of an image.

**Erosion:** Applying erosion to a color image means taking the minimum color value in the structuring element neighborhood. As an example R – SE means taking the minimum R value in the structuring element neighborhood. To decide the result value of a pixel after erosion, this operation is repeated for Green, Blue components of the color value.

**Dilation:** Applying dilation to a color image means taking the maximum color value in the structuring element neighborhood. As an example R + SE means taking the maximum R value in the structuring element neighborhood. To decide the result value of a pixel after erosion, this operation is repeated for Green, Blue components of the color value.

I implemented erosion, dilation, opening and closing operations using this algorithm. The implementation of this algorithm can be found in Appendix-II and the result images of this algorithm can be found in Appendix-I part of this document.

However there are important problems with these techniques. By taking the maximum or minimum values, we may construct a pixel value which is not in the structuring element neighborhood of the focused pixel. For example consider the pixels with R, G, B values with [100,50,50], [50,100,50] and [50,50,100]. By applying this algorithm we obtain a pixel with
components. However this pixel may not be in the neighborhood of the focused pixel. This leads to violation of some of the characteristics of the morphological operations. As an example for an image $X - SE < X$. We can violate this characteristic of the erosion operator.

### B.2. Mathematical Morphology on RGB Components as Vector

In this approach, we consider the color values of the pixels as vector. R, G, B values constitute the dimensions of the vector. By using their values as our base, we can make vector comparison between the pixels. The definitions of the basic operations are as follows:

**Erosion:** For applying erosion, we take the pixel color value with the minimum R value as the result. If there is equality in R values, we compare G values. If there is also an equality of G values we consider B values.

**Dilation:** For applying dilation, we take the pixel color value with the maximum R value as the result. If there is equality in R values, we compare G values. If there is also an equality of G values we consider B values.

The implementation of this approach can be found in Appendix II. I implemented the basic operations erosion, dilation, opening and closing using this approach. The results of this approach can be found in Appendix I.

However there is a severe problem in this approach. We favor a color component. For example, in my basic operations definitions, first R values then G values (in case of equality) and finally B values are compared. More importance is given to the R component. This is a big disadvantage of the method.

### B.3. Mathematical Morphology on L*m*p* Components

Another approach in color morphology which considers images as vector valued images and applying ordering according to the vector field is L*m*p* approach. The approach that I mentioned above takes favor to a color component. However by using this method, we overcome this problem.

The limitations of ordering and the problems in RGB morphology can be overcome by using the angular hue component of the color value. So we can make use of HLS color space to apply morphological operators to color images. However there are some problems of the HLS color space. First of all it is device dependent. Characteristics of the display device affect the
color coordinates. Another disadvantage is when the image is converted from a rectangular coordinate system to HLS color system; it leads to uneven distribution of hue values. To resolve these problems in the HLS color space, CIE (Commission Internationale de l’Eclairage), in 1976, introduced two device-independent spaces, the L\*m*p* and L\*u*v* spaces. We will make use of L\*m*p* color space in the investigation of third approach for color morphology. Now I will give a brief description of the L\*m*p* color space.

In the L\*m*p* color space:

- L* represents the lightness (luminance).
- m* encodes the red-green sensation
  
  (Positive m* indicates a red color, negative m* indicates green color)
- p* encodes the yellow-blue sensation
  
  (Positive p* indicating yellow and negative p* indicating blue)

More information about the L\*m*p* color space can be found in [4].

At this point, we know that we can apply morphological operations on color images in L\*m*p* color space. But the images are generally in RGB format. So we should convert RGB color space to L\*m*p* color space. The details of the conversion process are as follows:

1. Normalize Image: The R,G,B values of the image are divided to 255 to normalize image.

2. Transform normalized R,G,B values to X,Y,Z values by using

\[
\begin{bmatrix}
X \\
Y \\
Z
\end{bmatrix} = \begin{bmatrix}
0.412453 & 0.357580 & 0.180423 \\
0.212671 & 0.715160 & 0.072169 \\
0.019334 & 0.119193 & 0.950227
\end{bmatrix} \begin{bmatrix}
R \\
G \\
B
\end{bmatrix}
\]

3. Compute

\[
\begin{bmatrix}
X_n \\
Y_n \\
Z_n
\end{bmatrix} = \begin{bmatrix}
0.412453 & 0.357580 & 0.180423 \\
0.212671 & 0.715160 & 0.072169 \\
0.019334 & 0.119193 & 0.950227
\end{bmatrix} \begin{bmatrix}
1 \\
1 \\
1
\end{bmatrix}
\]

4. Compute

\[
L* = 116 \left( \frac{Y}{Y_n} \right)^{1/3} - 16
\]
\[ u^* = 13 \ L^* \ (u' - u'_n) \]
\[ v^* = 13 \ L^* \ (v' - v'_n) \]

Where

\[ u' = \frac{4X}{X + 15Y + 3Z} \]
\[ v' = \frac{9Y}{X + 15Y + 3Z} \]
\[ u'_n = \frac{4Xn}{Xn + 15Yn + 3Zn} \]
\[ v'_n = \frac{9Yn}{Xn + 15Yn + 3Zn} \]

5. Compute

\[ m^* = \sqrt{(u^*)+ (v^*)} \]
\[ p^* = \tan^{-1}\left(\frac{v^*}{u^*}\right) \]

After completing the five steps above we accomplish to convert RGB color space to L*m*p* color space.

Now we can order the values of the pixel values. In fact there are different ways to order the values. In my implementation I followed hue p*, saturation m*, and luminance L* order. Now I want to explain what erosion and dilation mean in this system.

**Erosion:** Erosion takes the color vector with minimum p* value in structuring element neighborhood. If there is equality, saturation values (m*) are compared. If again there is equality luminance values (L*) are compared.

**Dilation:** Dilation takes the color vector with maximum p* value in structuring element neighborhood. If there is equality, saturation values (m*) are compared. If again there is equality luminance values (L*) are compared.

After completing morphological operations the result in L*m*p* color space is retransformed into RGB color space by applying the inverse operations of the five steps mentioned earlier in this document.
I implemented the basic morphological operations erosion, dilation, opening and closing using this approach. You can find the implementation in Appendix II and the output result images in Appendix I.

**C. CONCLUSION**

It is obvious that applying morphological operations to color images is not straightforward as it is for binary and gray scale images. We cannot directly apply the methodologies used in binary and gray scale cases on color images. To implement the morphological operations, we should define comparison logic and ordering between the color values of the pixels. In this project, I try to analyze three different approaches developed for applying morphological operations on color images. First two approaches implement the operations by directly using with R, G, B values of the pixels. However they have some deficiencies so their application may result incorrect results. The third application converts the color values into another format (L*m*p*) which is suitable for partial ordering. As final comparison, the third method is more powerful than the other two because it really does a real comparison between the color values.
D. REFERENCES

1. MATHEMATICAL MORPHOLOGY FOR ANGLE-VALUED IMAGES
   Richard Alan Peters II
   Department of Electrical and Computer Engineering
   Vanderbilt University School of Engineering
   www.vuse.vanderbilt.edu/~rap2/papers/comorph.pdf

2. MORPHOLOGICAL OPERATIONS FOR COLOR IMAGE PROCESSING
   Mary L. Comer and Edward J. Delp
   Computer Vision and Image Processing Laboratory
   School of Electrical Engineering
   Purdue University
   West Lafayette, Indiana

3. MATHEMATICAL MORPHOLOGY IN THE HLS COLOUR SPACE
   Allan Hanbury and Jean Serra

4. MATHEMATICAL MORPHOLOGY IN THE L*A*B* COLOUR SPACE
   Allan Hanbury and Jean Serra
   http://www.prip.tuwien.ac.at/~hanbury/MM_Lab_space.pdf

5. IMAGE PROCESSING, ANALYSIS, and MACHINE VISION
   Milan SONKA, Vaclav Hlavac, Roger BOYLE

6. COLOR IMAGE PROCESSING and APPLICATIONS
   K.N. PLATANIOTIS and A.N. VENETSANOPoulos
   University of Toronto
   Department of Electrical & Computer Engineering
APPENDIX I

Following images are the results of my application. The order of them is erosion, dilation, opening, closing.

Results of first approach

Results of second approach

Results of third approach
Results of first approach

Results of second approach

Results of third approach
Results of first approach

Results of second approach

Results of third approach
Results of first approach

Results of second approach

Results of third approach
APPENDIX II

Source code of the project is as follows:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

typedef struct
{
    int width;
    int height;
    int maxColorValue;
    unsigned char ***pixels;
} imageType;

typedef struct
{
    int width;
    int height;
    float ***values;
} normalizedImage;

int **se;

void prepareStructuringElement(int ***strEl,char *fileName,int *addXVal,int *addYVal)
{
    int x,y;
    int i,j;
    int tmp;

    FILE *fp = fopen(fileName,"rb");
    fscanf(fp,"%d %d",&x,&y);

    (*strEl) = (int **) malloc(sizeof(int *) * x);
    for(i=0;i<x;i++)
    {
        (*strEl)[i] = (int *) malloc(sizeof(int) * y);
    }

    for(i=0;i<x;i++)
    {
        for(j=0;j<y;j++)
        {
            fscanf(fp,"%d",&tmp);
            (*strEl)[i][j] = tmp;
        }
    }

    (*addXVal) = x;
```
void allocImage(imageType *u, int h, int w, int cmax)
{
    int i, j;

    u->width = w;
    u->height = h;
    u->maxColorValue = cmax;

    u->pixels = (unsigned char **) malloc(sizeof(unsigned char **) * h);
    for(i=0; i<h; i++)
    {
        u->pixels[i] = (unsigned char **) malloc(sizeof(unsigned char *) * w);
        for(j=0; j<w; j++)
        {
            u->pixels[i][j] = (unsigned char *) malloc(sizeof(unsigned char) * 3);
        }
    }
}

void allocNormalizedImage(normalizedImage *u, int h, int w)
{
    int i, j;

    u->width = w;
    u->height = h;

    u->values = (float **) malloc(sizeof(float **) * h);
    for(i=0; i<h; i++)
    {
        u->values[i] = (float **) malloc(sizeof(float *) * w);
        for(j=0; j<w; j++)
        {
            u->values[i][j] = (float *) malloc(sizeof(float) * 3);
        }
    }
}

void copyNormalImage(imageType *u, imageType *v)
{
    int i, j;

for(i=0;i<u->heigth;i++)
{
    for(j=0;j<u->width;j++)
    {
        v->pixels[i][j][0] = u->pixels[i][j][0];
        v->pixels[i][j][1] = u->pixels[i][j][1];
        v->pixels[i][j][2] = u->pixels[i][j][2];
    }
}

void copyNormalizedImage(normalizedImage *u,normalizedImage *v)
{
    int i, j;
    for(i=0;i<u->heigth;i++)
    {
        for(j=0;j<u->width;j++)
        {
            v->values[i][j][0] = u->values[i][j][0];
            v->values[i][j][1] = u->values[i][j][1];
            v->values[i][j][2] = u->values[i][j][2];
        }
    }
}

void imdiff(imageType *u,imageType *v,imageType *diff)
{
    int i, j;
    for(i=0;i<u->heigth;i++)
    {
        for(j=0;j<u->width;j++)
        {
            diff->pixels[i][j][0] = u->pixels[i][j][0] - v->pixels[i][j][0];
            diff->pixels[i][j][1] = u->pixels[i][j][1] - v->pixels[i][j][1];
            diff->pixels[i][j][2] = u->pixels[i][j][2] - v->pixels[i][j][2];
        }
    }
}

void imGrayLeveldiff(imageType *u,imageType *v,imageType *diff)
{
    int i, j;
    float grayValue1,grayValue2,grayValue3;
    for(i=0;i<u->heigth;i++)
    {
        grayValue1 = (float)u->pixels[i][0][0];
        grayValue2 = (float)u->pixels[i][1][0];
        grayValue3 = (float)u->pixels[i][2][0];
        diff->pixels[i][0][0] = grayValue1 - v->pixels[i][0][0];
        diff->pixels[i][1][0] = grayValue2 - v->pixels[i][1][0];
        diff->pixels[i][2][0] = grayValue3 - v->pixels[i][2][0];
    }
}
for(j=0;j<u->width;j++)
{
    grayValue1 = (u->pixels[i][j][0] + u->pixels[i][j][1] + u->pixels[i][j][2]) / 3;
    grayValue2 = (v->pixels[i][j][0] + v->pixels[i][j][1] + v->pixels[i][j][2]) / 3;
    grayValue3 = grayValue1 - grayValue2;
    diff->pixels[i][j][0] = diff->pixels[i][j][1] = diff->pixels[i][j][2] = grayValue3;
}

void imGrayScale(imageType *u)
{
    int i,j;
    unsigned char tmp;
    for(i=0;i<u->heigth;i++)
    {
        for(j=0;j<u->width;j++)
        {
            tmp = (u->pixels[i][j][0] + u->pixels[i][j][1] + u->pixels[i][j][2]) / 3;
            u->pixels[i][j][0] = tmp;
            u->pixels[i][j][1] = tmp;
            u->pixels[i][j][2] = tmp;
        }
    }
}

void dummies(imageType *u)
{
    int i;
    int w,h;
    w = u->width;
    h = u->heigth;
    for(i=2;i<h-2;i++)
    {
        u->pixels[i][0][0] = u->pixels[i][3][0];
        u->pixels[i][0][1] = u->pixels[i][3][1];
        u->pixels[i][0][2] = u->pixels[i][3][2];
        u->pixels[i][1][0] = u->pixels[i][2][0];
        u->pixels[i][1][1] = u->pixels[i][2][1];
        u->pixels[i][1][2] = u->pixels[i][2][2];
        u->pixels[i][w-1][0] = u->pixels[i][w-4][0];
        u->pixels[i][w-1][1] = u->pixels[i][w-4][1];
u->pixels[i][w-1][2] = u->pixels[i][w-4][2];
    u->pixels[i][w-2][0] = u->pixels[i][w-3][0];
    u->pixels[i][w-2][1] = u->pixels[i][w-3][1];
    u->pixels[i][w-2][2] = u->pixels[i][w-3][2];
}

for(i=0;i<w;i++)
{
    u->pixels[0][i][0] = u->pixels[3][i][0];
    u->pixels[0][i][1] = u->pixels[3][i][1];
    u->pixels[0][i][2] = u->pixels[3][i][2];
    u->pixels[1][i][0] = u->pixels[2][i][0];
    u->pixels[1][i][1] = u->pixels[2][i][1];
    u->pixels[1][i][2] = u->pixels[2][i][2];
    u->pixels[h-1][i][0] = u->pixels[h-4][i][0];
    u->pixels[h-1][i][1] = u->pixels[h-4][i][1];
    u->pixels[h-1][i][2] = u->pixels[h-4][i][2];
    u->pixels[h-2][i][0] = u->pixels[h-3][i][0];
    u->pixels[h-2][i][1] = u->pixels[h-3][i][1];
    u->pixels[h-2][i][2] = u->pixels[h-3][i][2];
}
}

****************************************************************
/*************************READS IMAGE FILE******************************/
/*********************************************************/
void readImage(imageType *image,char *fileName)
{
FILE *fp = fopen(fileName,"rb");

unsigned char fileFormat[10];
unsigned char temp;
int widthValue,heigthValue,colorRangeValue;
int i=0,j=0;
unsigned char redValue,greenValue,blueValue;

if(fp == NULL)
    {
        printf("ERROR : IMAGE FILE CAN NOT BE OPENED\n");
        exit(1);
    }

fscanf(fp,"%s",fileFormat);
/*gets the new line character after file format value*/
fgetc(fp);
/*
if(strcmp(fileFormat,"P6") != 0 && strcmp(fileFormat,"P3") != 0)
    {
        printf("ERROR : FILE FORMAT IS INCORRECT. TRY A .PPM FILE AS INPUT\n");
        exit(1);
    }
*/
temp = fgetc(fp);
/* reads comment lines*/
while(temp == '#')
{
    while((temp = fgetc(fp)) != '
')
    {
    }
}
/* meet with a number value*/
if(temp < '9' && temp > '0')
{
    widthValue = temp - 48;
    while((temp = fgetc(fp)) != ' ')
    {
        widthValue = widthValue * 10 + (temp - 48);
    }
    fscanf(fp, "%d %d", &heigthValue, &colorRangeValue);
    image->width = widthValue + 4;
    image->heigth = heigthValue + 4;
    image->maxColorValue = colorRangeValue;
}
else
{
    fscanf(fp, "%d %d %d", &widthValue, &heigthValue, &colorRangeValue);
    image->width = widthValue + 4;
    image->heigth = heigthValue + 4;
    image->maxColorValue = colorRangeValue;
}
/*Getting newline character after max color value*/
fgetc(fp);
allocImage(image, image->heigth, image->width, image->maxColorValue);
for(i=2; i<image->heigth-2; i++)
{
    for(j=2; j<image->width-2; j++)
    {
        fscanf(fp, "%c%c%c", &redValue, &greenValue, &blueValue);
        image->pixels[i][j][0] = redValue;
        image->pixels[i][j][1] = greenValue;
        image->pixels[i][j][2] = blueValue;
    }
}
fclose(fp);
dummies(image);

//********************************************************************************
**********
*************************WRITES IMAGE FILE
********************************************************************************/
/***************************************************************************
***********/

void writeImage(imageType *image, char *fileName)
{
    FILE *fp = fopen(fileName, "wb");
    int i=0, j=0;

    fprintf(fp, "P6\n");
    fprintf(fp, "#Sezen ERDEM\n");
    fprintf(fp, "%d %d\n%d\n", image->width-4, image->height-4, image->maxColorValue);

    for(i=2; i<image->height-2; i++)
    {
        for(j=2; j<image->width-2; j++)
        {
            fputc(image->pixels[i][j][0], fp);
            fputc(image->pixels[i][j][1], fp);
            fputc(image->pixels[i][j][2], fp);
        }
    }

    fclose(fp);
}

/***************************************************************************
***********/

void normalizeImage(imageType *u, normalizedImage *result)
{
    int i, j;

    for(i=0; i<u->height; i++)
    {
        for(j=0; j<u->width; j++)
        {
            result->values[i][j][0] = (float) u->pixels[i][j][0] / 255.0;
            result->values[i][j][1] = (float) u->pixels[i][j][1] / 255.0;
            result->values[i][j][2] = (float) u->pixels[i][j][2] / 255.0;
        }
    }

    //printf( "%f %f %f\n", result->values[i][j][0], result->values[i][j][1], result->values[i][j][2]);
    //printf("\n");
    //printf("\n");

    return;
}
void denormalizeImage(normalizedImage *u, imageType *result)
{
    int i, j;
    float midval = 0.0;

    for (i = 0; i < u->height; i++)
    {
        for (j = 0; j < u->width; j++)
        {
            midval = u->values[i][j][0] * 255.0;
            if (midval > 255.0) midval = 255.0;
            else if (midval < 0.0) midval = 0.0;

            result->pixels[i][j][0] = midval;

            midval = u->values[i][j][1] * 255.0;
            if (midval > 255.0) midval = 255.0;
            else if (midval < 0.0) midval = 0.0;

            result->pixels[i][j][1] = midval;

            midval = u->values[i][j][2] * 255.0;
            if (midval > 255.0) midval = 255.0;
            else if (midval < 0.0) midval = 0.0;

            result->pixels[i][j][2] = midval;
        }
    }

    return;
}

void rgb2lmp(normalizedImage *u, normalizedImage *result)
{
    int i, j;
    float xn, yn, zn;
    float up, vp, unp, vnp;
    float L, us, vs;
    float ms, ps;
    float X, Y, Z;
    float mycuberoot;

    float converter[3][3];
converter[0][0] = 0.412453; converter[0][1] = 0.357580;
converter[0][2] = 0.180423;
converter[1][0] = 0.212671; converter[1][1] = 0.715160;
converter[1][2] = 0.072169;
converter[2][0] = 0.019334; converter[2][1] = 0.119193;
converter[2][2] = 0.950227;

xn = converter[0][0] + converter[0][1] + converter[0][2];
yn = converter[1][0] + converter[1][1] + converter[1][2];
zn = converter[2][0] + converter[2][1] + converter[2][2];

for(i=0;i<u->height;i++)
{
    for(j=0;j<u->width;j++)
    {
        // X Value
        X = converter[0][0] * u->values[i][j][0] + converter[0][1] * u->values[i][j][1] + converter[0][2] * u->values[i][j][2];
        //
        // Y Value
        Y = converter[1][0] * u->values[i][j][0] + converter[1][1] * u->values[i][j][1] + converter[1][2] * u->values[i][j][2];
        //result->values[i][j][1] =
        // Z Value
        Z = converter[2][0] * u->values[i][j][0] + converter[2][1] * u->values[i][j][1] + converter[2][2] * u->values[i][j][2];
        //result->values[i][j][2] =
        //printf("%f %f %f
",X,Y,Z);
        //printf("%f\n",cbrt(Y/yn));
        if( X == 0 && Y == 0 && Z == 0)
        {
            up = 0.0;
            vp = 0.0;
        }
        else
        {
            up = ((4*X) / (X+(15*Y)+(3*Z)));
            vp = ((9*Y) / (X+(15*Y)+(3*Z)));
        }
        unp = ((4*xn) / (xn+(15*yn)+(3*zn)));
        vnp = ((9*yn) / (xn+(15*yn)+(3*zn)));
        mycuberoot = (float) pow((float)(Y/yn),(float)(1.0/3.0));
        L = (116 * mycuberoot ) - 16;
        us = 13 * L * (up - unp);
        vs = 13 * L * (vp - vnp);
        //ms = sqrt ((us*us) + (vs*vs));
        //ps = atan2(vs,us);
        //printf("%f %f %f %f %f %f
",X,Y,Z,u->values[i][j][0],u->values[i][j][1],u->values[i][j][2]);
        result->values[i][j][0] = L;
result->values[i][j][1] = us;
result->values[i][j][2] = vs;
}

//printf("\n");
}

printf("--------------------------------
");
}

/*******************************************************************************
**********
*************************LMP 2 RGB
IMAGE*******************************************/
*******************************************************************************
**********

void lmp2rgb(normalizedImage *u, normalizedImage *result)
{
    int i, j;
    float us, vs, up, vp;
    float unp, vnp;
    float X, Y, Z;
    float L, m, p;
    float xn, yn, zn;
    float R, G, B;

    float converter[3][3];
    float invconverter[3][3];

    converter[0][0] = 0.412453;   converter[0][1] = 0.357580;
    converter[0][2] = 0.180423;
    converter[1][0] = 0.212671;   converter[1][1] = 0.715160;
    converter[1][2] = 0.072169;
    converter[2][0] = 0.019334;   converter[2][1] = 0.119193;
    converter[2][2] = 0.950227;

    invconverter[0][0] = 3.240481;   invconverter[0][1] = -1.537151;
    invconverter[0][2] = -0.498536;
    invconverter[1][0] = -0.969254;   invconverter[1][1] = 1.875990;
    invconverter[1][2] = 0.041555;
    invconverter[2][0] = 0.055646;   invconverter[2][1] = -0.204041;
    invconverter[2][2] = 1.057311;

    xn = converter[0][0] + converter[0][1]  + converter[0][2];
    yn = converter[1][0] + converter[1][1]  + converter[1][2];
    zn = converter[2][0] + converter[2][1]  + converter[2][2];

    for(i=0; i< u->height; i++)
    {
        for(j=0; j< u->width; j++)
        {
            L = u->values[i][j][0];
            us = u->values[i][j][1];
            vs = u->values[i][j][2];

            unp = ((4*xn) / (xn+(15*yn) + (3*zn)));
            vnp = ((9*yn) / (xn+(15*yn) + (3*zn)));

            unp = converter[0][0]*unp + converter[0][1]*vs + converter[0][2]*us;
            vnp = converter[1][0]*unp + converter[1][1]*vs + converter[1][2]*us;
            R = converter[2][0]*unp + converter[2][1]*vs + converter[2][2]*us;
            G = converter[2][0]*unp + converter[2][1]*vs + converter[2][2]*us;
            B = converter[2][0]*unp + converter[2][1]*vs + converter[2][2]*us;

            result->values[i][j][0] = (R > 255) ? 255 : R;
            result->values[i][j][1] = (G > 255) ? 255 : G;
            result->values[i][j][2] = (B > 255) ? 255 : B;
        }
    }
}
\begin{verbatim}
{
    // Structuring Element
    minp = atan2(u->values[i][j][2], u->values[i][j][1]);
    minL = u->values[i][j][0];
    minm = sqrt((u->values[i][j][2] * u->values[i][j][2]) +
    (u->values[i][j][1] * u->values[i][j][1]));
    ru = u->values[i][j][1];
    rv = u->values[i][j][2];
    // printf("%f %d %d
", minp, minL, minm);
    for(k=-2; k<3; k++)
    {
        for(t=-2; t<3; t++)
        {
            if(se[k+2][t+2] == 1)
            {
                curpixelp = atan2(u->values[i+k][j+t][2], u->values[i+k][j+t][1]);
                curpixelm = sqrt((u->values[i+k][j+t][2] * u->values[i+k][j+t][2]) +
                (u->values[i+k][j+t][1] * u->values[i+k][j+t][1]));
                // printf("%f %d %d
", curpixelp, i+k, j+t);
                if(curpixelp < minp)
                {
                    // if(curpixelp == minp) {count ++;
                    printf("EQUAL = %d %f %f\n", count, curpixelp, minp);
                    minp = curpixelp;
                    minL = u->values[i+k][j+t][0];
                    minm = curpixelm;
                    ru = u->values[i+k][j+t][1];
                    rv = u->values[i+k][j+t][2];
                }
                else if(curpixelp == minp)
                {
                    if( curpixelm < minm)
                    {
                        minp = curpixelp;
                        minL = u->values[i+k][j+t][0];
                        minm = curpixelm;
                        ru = u->values[i+k][j+t][1];
                        rv = u->values[i+k][j+t][2];
                    }
                    else if( curpixelm == minm)
                    {
                        curpixelL = u->values[i+k][j+t][0];
                        if(curpixelL < minL)
                        {
                            minp = curpixelp;
                            minL = u->values[i+k][j+t][0];
                            minm = curpixelm;
                            ru = u->values[i+k][j+t][1];
                            rv = u->values[i+k][j+t][2];
                        }
                        else if( curpixelL == minL)
                        {
                            curpixelL = u->values[i+k][j+t][0];
                            if(curpixelL < minL)
                            {
                                minp = curpixelp;
                                minL = u->values[i+k][j+t][0];
                                minm = curpixelm;
                            }
                        }
                    }
                }
            }
        }
    }
\end{verbatim}
void colorDilation(normalizedImage *u, normalizedImage *result)
{
    int i, j, k, t;
    int w, h;

    float maxp, curpixelp, curpixelL, curpixelm;
    float maxL, maxm;
    //float se[3][3];
    int count = 0;
    float ru, rv;

    /*
     * se[0][0] = 1.0;   se[0][1] = 1.0;   se[0][2] = 1.0;
     * se[1][0] = 1.0;   se[1][1] = 1.0;   se[1][2] = 1.0;
     * se[2][0] = 1.0;   se[2][1] = 1.0;   se[2][2] = 1.0;
     */
    w = u->width;
    h = u->height;

    for (i = 2; i < h - 2; i++)
    {
        for (j = 2; j < w - 2; j++)
        {
            //Structuring Element
            maxp = atan2(u->values[i][j][2], u->values[i][j][1]);
            maxL = u->values[i][j][0];
            maxm = sqrt((u->values[i][j][2] * u->values[i][j][2]) +
                        (u->values[i][j][1] * u->values[i][j][1]));
            ru = u->values[i][j][1];
            rv = u->values[i][j][2];

            result->values[i][j][0] = minL;
            result->values[i][j][1] = ru;
            result->values[i][j][2] = rv;
        }
    }
}
//printf("%f %f %f\n", minp, minL, minm);
for(k=-2;k<3;k++)
{
    for(t=-2;t<3;t++)
    {
        if(se[k+2][t+2] == 1.0)
        {
            curpixelp = atan2(u->values[i+k][j+t][2], u->values[i+k][j+t][1]);
            curpixelm = sqrt((u->values[i+k][j+t][2] * u->values[i+k][j+t][2]) + (u->values[i+k][j+t][1] * u->values[i+k][j+t][1]));
            //printf("%f %d %d\n", curpixelp, i+k, j+t);
            if(curpixelp > maxp)
            {
                if(curpixel == minp) {count ++;
                    //if(curpixel == minp) (count ++;
                    printf("EQUAL = %d %f %f\n", count, curpixel, minp);)
                maxp = curpixelp;
                maxL = u->values[i+k][j+t][0];
                maxm = curpixelm;
                ru = u->values[i+k][j+t][1];
                rv = u->values[i+k][j+t][2];
            }
            else if(curpixelp == maxp)
            {
                curpixelm = u->values[i+k][j+t][1];
                if( curpixelm > maxm)
                {
                    maxp = curpixelp;
                    maxL = u->values[i+k][j+t][0];
                    maxm = curpixelm;
                    ru = u->values[i+k][j+t][1];
                    rv = u->values[i+k][j+t][2];
                }
            }
        }
    }
    else if( curpixelm == maxm)
    {
        curpixelL = u->values[i+k][j+t][1];
        if(curpixelL > maxL)
        {
            maxp = curpixelp;
            maxL = u->values[i+k][j+t][0];
            maxm = curpixelm;
            ru = u->values[i+k][j+t][1];
            rv = u->values[i+k][j+t][2];
        }
    }
    else if( curpixelm == maxm)
    {
        curpixelm = u->values[i+k][j+t][1];
        if(curpixelm > maxm)
        {
            maxp = curpixelp;
            maxL = u->values[i+k][j+t][0];
            maxm = curpixelm;
            ru = u->values[i+k][j+t][1];
            rv = u->values[i+k][j+t][2];
        }
    }
}
result->values[i][j][0] = maxL;
result->values[i][j][1] = ru;
result->values[i][j][2] = rv;
}
}

/******************************************************************************
**********/
*************************OPENNING
***************************************************/

void colorOpen(normalizedImage *u, normalizedImage *result)
{
    int i, j;
    normalizedImage tmp;

    allocNormalizedImage(&tmp,u->heigth,u->width);
    colorErosion(u,&tmp);
    colorDilation(&tmp,result);
}

/******************************************************************************
**********/
*************************CLOSING
***************************************************/

void colorClose(normalizedImage *u, normalizedImage *result)
{
    int i, j;
    normalizedImage tmp;

    allocNormalizedImage(&tmp,u->heigth,u->width);
    colorDilation(u,&tmp);
    colorErosion(&tmp,result);
}

/******************************************************************************
**********/
*************************RGB
DILATION********************************************/

void RGBDilation(imageType *u,imageType *result)
{
    int i, j, k, t;

unsigned char maxR,maxG,maxB;
unsigned char curR,curG,curB;
/*
int se[3][3];
se[0][0] = 0;   se[0][1] = 1;   se[0][2] = 0;
se[0][1] = 1;   se[1][1] = 1;   se[1][2] = 1;
se[0][2] = 0;   se[2][1] = 1;   se[2][2] = 0;
*/
for(i=2;i<u->heigth-2;i++)
{
    for(j=2;j<u->width-2;j++)
    {
        maxR = u->pixels[i][j][0];
        maxG = u->pixels[i][j][1];
        maxB = u->pixels[i][j][2];

        for(k=-2;k<3;k++)
        {
            for(t=-2;t<3;t++)
            {
                if(se[k+2][t+2] == 1)
                {
                    curR = u->pixels[i+k][j+t][0];
                    curG = u->pixels[i+k][j+t][1];
                    curB = u->pixels[i+k][j+t][2];

                    if(curR > maxR)
                    {
                        maxR = curR;
                    }

                    if(curG > maxG)
                    {
                        maxG = curG;
                    }

                    if(curB > maxB)
                    {
                        maxB = curB;
                    }
                }
            }
        }
        result->pixels[i][j][0] = maxR;
        result->pixels[i][j][1] = maxG;
        result->pixels[i][j][2] = maxB;
    }
}
void RGBErosion(imageType *u, imageType *result)
{
    int i, j, k, t;
    unsigned char minR, minG, minB;
    unsigned char curR, curG, curB;
    /*
    int se[3][3];
    se[0][0] = 1; se[0][1] = 1; se[0][2] = 1;
    se[0][1] = 1; se[1][1] = 1; se[1][2] = 1;
    se[0][2] = 1; se[2][1] = 1; se[2][2] = 1;
    */
    for (i = 2; i < u->height - 2; i++)
    {
        for (j = 2; j < u->width - 2; j++)
        {
            minR = u->pixels[i][j][0];
            minG = u->pixels[i][j][1];
            minB = u->pixels[i][j][2];
            for (k = -2; k < 3; k++)
            {
                for (t = -2; t < 3; t++)
                {
                    if (se[k + 2][t + 2] == 1)
                    {
                        curR = u->pixels[i + k][j + t][0];
                        curG = u->pixels[i + k][j + t][1];
                        curB = u->pixels[i + k][j + t][2];
                        if (curR < minR)
                        {
                            minR = curR;
                        }
                        if (curG < minG)
                        {
                            minG = curG;
                        }
                        if (curB < minB)
                        {
                            minB = curB;
                        }
                    }
                }
            }
            result->pixels[i][j][0] = minR;
            result->pixels[i][j][1] = minG;
            result->pixels[i][j][2] = minB;
        }
    }
}

/***************************************************************************
***********/
/**************************RGB
OPEN******************************************************************************/
******************************************************************************
**********
void RGBOpen(imageType *u,imageType *result)
{
    imageType tmp;
    allocImage(&tmp,u->height,u->width,u->maxColorValue);
    RGBErosion(u,&tmp);
    RGBDilation(&tmp,result);
}
******************************************************************************
**********
/****************************RGB VECTOR EROSION*******************************/
*******************************************************************************/
void RGBVectorErosion(imageType *u,imageType *result)
{
    int i,j,k,t;
    float minR,minG,minB;
    float curR,curG,curB;
    /*
    int se[3][3];
    se[0][0] = 1; se[0][1] = 1; se[0][2] = 1;
    se[0][1] = 1; se[1][1] = 1; se[1][2] = 1;
    se[0][2] = 1; se[2][1] = 1; se[2][2] = 1;
    */
    for(i=2;i<u->height-2;i++)
    {
        for(j=2;j<u->width-2;j++)
        {
            minR = u->pixels[i][j][0];
            minG = u->pixels[i][j][1];
            minB = u->pixels[i][j][2];
            for(k=-2;k<3;k++)
for(t=-2; t<3; t++)
{
    if(se[k+2][t+2] == 1)
    {
        curR = u->pixels[i+k][j+t][0];
        curG = u->pixels[i+k][j+t][1];
        curB = u->pixels[i+k][j+t][2];

        if(curR < minR)
        {
            minR = curR;
            minG = curG;
            minB = curB;
        }
        else if(curR == minR)
        {
            if(curG < minG)
            {
                minR = curR;
                minG = curG;
                minB = curB;
            }
            else if(curG == minG)
            {
                if(curB < minB)
                {
                    minR = curR;
                    minG = curG;
                    minB = curB;
                }
            }
        }
    }
}
result->pixels[i][j][0] = minR;
result->pixels[i][j][1] = minG;
result->pixels[i][j][2] = minB;

/***************************************************************************/
**********
*************************RGB VECTOR
DILATION******************************************
***************************************************************************/

void RGBVectorDilation(imageType *u, imageType *result)
{
    int i, j, k, t;
    float maxR, maxG, maxB;
    float curR, curG, curB;

    /*
    int se[3][3];
    se[0][0] = 1;  se[0][1] = 1;  se[0][2] = 1;
    */
se[0][1] = 1; se[1][1] = 1; se[1][2] = 1;
se[0][2] = 1; se[2][1] = 1; se[2][2] = 1;

*/

for(i=2;i<u->heigth-2;i++)
{
    for(j=2;j<u->width-2;j++)
    {
        maxR = u->pixels[i][j][0];
        maxG = u->pixels[i][j][1];
        maxB = u->pixels[i][j][2];
        for(k=-2;k<3;k++)
        {
            for(t=-2;t<3;t++)
            {
                if(se[k+2][t+2] == 1)
                {
                    curR = u->pixels[i+k][j+t][0];
                    curG = u->pixels[i+k][j+t][1];
                    curB = u->pixels[i+k][j+t][2];
                    if(curR > maxR)
                    {
                        maxR = curR;
                        maxG = curG;
                        maxB = curB;
                    }
                    else if(curR == maxR)
                    {
                        if(curG > maxG)
                        {
                            maxR = curR;
                            maxG = curG;
                            maxB = curB;
                        }
                        else if(curG == maxG)
                        {
                            if(curB > maxB)
                            {
                                maxR = curR;
                                maxG = curG;
                                maxB = curB;
                            }
                        }
                    }
                }
            }
        }
    }
}
result->pixels[i][j][0] = maxR;
result->pixels[i][j][1] = maxG;
result->pixels[i][j][2] = maxB;

/***************************************************************************/

*******/
void RGBVectorOpen(imageType *u, imageType *result)
{
    imageType tmp;
    allocImage(&tmp, u->heigth, u->width, u->maxColorValue);
    RGBVectorErosion(u, &tmp);
    RGBVectorDilation(&tmp, result);
}

void RGBVectorClose(imageType *u, imageType *result)
{
    imageType tmp;
    allocImage(&tmp, u->heigth, u->width, u->maxColorValue);
    RGBVectorDilation(u, &tmp);
    RGBVectorErosion(&tmp, result);
}

int main(int argc, char *argv[])
{
    int i, j, kmax;
    int globalHeigth, globalWidth;
    int operation;
    char inputFileName[25];
    char SEFileName[25];
    int addX, addY;

    imageType inputImage;
    imageType outputImage;
    imageType tmpImage;

    normalizedImage normImage;
    normalizedImage lmpImage;
    normalizedImage erodedImage;

    //kmax = atoi(argv[3]);

    printf("ENTER OPERATION TYPE:
");
    printf("1 - HSV Erosion\n");
printf("2 - HSV Dilation\n");
printf("3 - HSV Open\n");
printf("4 - HSV Close\n");
printf("5 - RGB Erosion\n");
printf("6 - RGB Dilation\n");
printf("7 - RGB Open\n");
printf("8 - RGB Close\n");
printf("9 - RGB Vector Erosion\n");
printf("10 - RGB Vector Dilation\n");
printf("11 - RGB Vector Open\n");
printf("12 - RGB Vector Close\n");

scanf("\nOPERATION NUMBER = ",&operation);
scanf("\nINPUT IMAGE = ",inputFileName);
scanf("\nSTRUCTURING ELEMENT FILE = ",SEFileName);
scanf("\nITERATION NUMBER= ",&kmax);

printf("\nSTARTING............\n");
readImage(&inputImage,inputFileName);
prepareStructuringElement(&se,SEFileName,&addX,&addY);
printf("IMAGE READ............\n");

globalHeigth = inputImage.heigth;
globalWidth = inputImage.width;
allocImage(&outputImage,globalHeigth,globalWidth,inputImage.maxColorValue);
allocImage(&tmpImage,globalHeigth,globalWidth,inputImage.maxColorValue);

switch(operation)
{
  case 1: // HSV EROSION
    allocNormalizedImage(&normImage,globalHeigth,globalWidth);
    allocNormalizedImage(&lmpImage,globalHeigth,globalWidth);
    allocNormalizedImage(&erodedImage,globalHeigth,globalWidth);

    normalizeImage(&inputImage,&normImage);
    rgb2lmp(&normImage,&lmpImage);

    for(i=0;i<kmax;i++)
    {
      colorErosion(&lmpImage,&erodedImage);
      copyNormalizedImage(&erodedImage,&lmpImage);
    }

    lmp2rgb(&lmpImage,&lmpImage);
denormalizeImage(&lmpImage,&outputImage);

printf("HSV COLOR EROSION DONE............\n");

imdiff(&inputImage,&outputImage,&tmpImage);
writeImage(&outputImage,"result1.ppm");
writeImage(&tmpImage,"diff1.ppm");

break;
}

case 2: // HSV DILTION
{
    allocNormalizedImage(&normImage,globalHeight,globalWidth);
    allocNormalizedImage(&lmpImage,globalHeight,globalWidth);
    allocNormalizedImage(&erodedImage,globalHeight,globalWidth);

    normalizeImage(&inputImage,&normImage);
    rgb2lmp(&normImage,&lmpImage);
    for(i=0;i<kmax;i++)
    {
        colorDilation(&lmpImage,&erodedImage);
        copyNormalizedImage(&erodedImage,&lmpImage);
    }
    lmp2rgb(&lmpImage,&lmpImage);
    denormalizeImage(&lmpImage,&outputImage);

    printf("HSV COLOR DILATION DONE............\n");
    imdiff(&inputImage,&outputImage,&tmpImage);
    writeImage(&outputImage,"result2.ppm");
    writeImage(&tmpImage,"diff2.ppm");

    break;
}

case 3: // HSV OPEN
{
    allocNormalizedImage(&normImage,globalHeight,globalWidth);
    allocNormalizedImage(&lmpImage,globalHeight,globalWidth);
    allocNormalizedImage(&erodedImage,globalHeight,globalWidth);

    normalizeImage(&inputImage,&normImage);
    rgb2lmp(&normImage,&lmpImage);
    for(i=0;i<kmax;i++)
    {
        colorOpen(&lmpImage,&erodedImage);
        copyNormalizedImage(&erodedImage,&lmpImage);
    }
    lmp2rgb(&lmpImage,&lmpImage);
    denormalizeImage(&lmpImage,&outputImage);

    printf("HSV COLOR OPEN DONE............\n");
    imdiff(&inputImage,&outputImage,&tmpImage);
writeImage(&outputImage,"result3.ppm");
writeImage(&tmpImage,"diff3.ppm");
break;
}

case 4: // HSV CLOSE
{
allocNormalizedImage(&normImage,globalHeigth,globalWidth);
allocNormalizedImage(&lmpImage,globalHeigth,globalWidth);
allocNormalizedImage(&erodedImage,globalHeigth,globalWidth);

normalizeImage(&inputImage,&normImage);
rgb2lmp(&normImage,&lmpImage);
for(i=0;i<kmax;i++)
{
    colorClose(&lmpImage,&erodedImage);
    copyNormalizedImage(&erodedImage,&lmpImage);
}
lmp2rgb(&lmpImage,&lmpImage);
denormalizeImage(&lmpImage,&outputImage);
printf("HSV COLOR CLOSE DONE............\n");
imdiff(&inputImage,&outputImage,&tmpImage);
writeImage(&outputImage,"result4.ppm");
writeImage(&tmpImage,"diff4.ppm");
break;
}

case 5: // RGB EROSION
{
    allocImage(&tmpImage,globalHeigth,globalWidth,inputImage.maxColorValue);
    copyNormalImage(&inputImage,&tmpImage);
    for(i=0;i<kmax;i++)
    {
        RGBErosion(&tmpImage,&outputImage);
        copyNormalImage(&outputImage,&tmpImage);
    }
    printf("RGB COLOR EROSION DONE............\n");
imdiff(&inputImage,&outputImage,&tmpImage);
writeImage(&outputImage,"result5.ppm");
writeImage(&tmpImage,"diff5.ppm");
break;
}

case 6: // RGB DILATION
{
copyNormalImage(&inputImage,&tmpImage);
for(i=0;i<kmax;i++)
{
}
RGBDilation(&tmpImage,&outputImage);
copyNormalImage(&outputImage,&tmpImage);
}
printf("RGB COLOR DILATION DONE............\n");

imdiff(&inputImage,&outputImage,&tmpImage);
writeImage(&outputImage,"result6.ppm");
writeImage(&tmpImage,"diff6.ppm");

break;
}
case 7: // RGB OPEN
{
RGBOpen(&inputImage,&outputImage);
printf("RGB COLOR OPEN DONE............\n");
imdiff(&inputImage,&outputImage,&tmpImage);
writeImage(&outputImage,"result7.ppm");
writeImage(&tmpImage,"diff7.ppm");

break;
}
case 8: // RGB CLOSE
{
RGBClose(&inputImage,&outputImage);
printf("RGB COLOR CLOSE DONE............\n");
imdiff(&inputImage,&outputImage,&tmpImage);
writeImage(&outputImage,"result8.ppm");
writeImage(&tmpImage,"diff8.ppm");

break;
}
case 9: // RGB VECTOR EROSION
{
copyNormalImage(&inputImage,&tmpImage);

for(i=0;i<kmax;i++)
{
    RGBVectorErosion(&tmpImage,&outputImage);
copyNormalImage(&outputImage,&tmpImage);
}
printf("RGB VECTOR COLOR EROSION DONE............\n");
imdiff(&inputImage,&outputImage,&tmpImage);
writeImage(&outputImage,"result9.ppm");
writeImage(&tmpImage,"diff9.ppm");

break;
}
case 10: // RGB VECTOR DILATION
{
copyNormalImage(&inputImage,&tmpImage);

for(i=0;i<kmax;i++)
{
    RGBVectorDilation(&tmpImage,&outputImage);
copyNormalImage(&outputImage,&tmpImage);

```
printf("RGB VECTOR COLOR DILATION DONE............\n");
imdiff(&inputImage,&outputImage,&tmpImage);
writeImage(&outputImage,"result10.ppm");
writeImage(&tmpImage,"diff10.ppm");
break;
}
case 11: // RGB VECTOR OPEN
{
    RGBVectorOpen(&inputImage,&outputImage);
    printf("RGB VECTOR COLOR OPEN DONE............\n");
imdiff(&inputImage,&outputImage,&tmpImage);
writeImage(&outputImage,"result11.ppm");
writeImage(&tmpImage,"diff11.ppm");
break;
}
case 12: // RGB VECTOR CLOSE
{
    RGBVectorClose(&inputImage,&outputImage);
    printf("RGB VECTOR COLOR CLOSE DONE............\n");
imdiff(&inputImage,&outputImage,&tmpImage);
writeImage(&outputImage,"result12.ppm");
writeImage(&tmpImage,"diff12.ppm");
break;
}
}
printf("IMAGE WRITTEN............\n");
return 0;