**Pattern Recognition**

**for shapes**

An Extra Project for CS464 practice

By

Olina ZHANG

Prepared Under the direction of Professor Henry Chang

School of Engineering

Northwestern Polytechnic University

117 Fourier Avenue, Fremont, CA 94539

March 2013

# ACKNOWLEGEMENTS

I would also like to thank Professor Henry Chang for sharing his time, knowledge and passion to guide and help me during this project.

I also would like to thank Professor YaLee Tsai for her lectures of CS464, which made a fundamental base for me to complete this project.

Last but not least, I would like to thank my classmates, Ocean and Mary and Cao and so on, from the discussion with whom I gained many great ideas.

**CONTENT**

ACKNOWLEGEMENTS ii

ABSTRACT 4

1.0 INTRODUCTION 5

2.0 Software DESIGN 5

2.1 Convert the input grey-level image into binary data 5

2.2 Connectivity analysis to recognize number of objects 6

2.3 Shape Analysis on each objects 6

3.0 IMPLEMENTATION AND RESULTS 8

3.1 Convert the input grey-level image into binary data 8

3.2 Connectivity analysis to recognize number of objects 10

3.3 Shape Analysis on each object 14

4.0 ENHANCEMENT IDEAS 16

5.0 CONCLUSION 16

6.0 BIBLIOGRAPHY 17

7.0 APPENDIX 18

1) pattern.h 18

2) pattern.c 19

3) readPGM.c 33

4) p1\_test.c 36

# ABSTRACT

This technical report will cover topics on the several important and basic steps of pattern recognition, based on C language as the implementation in this project.

# 1.0 INTRODUCTION

This project is based on some basic idea about pattern recognition, aiming to achieve a very simple goal – that is, to recognize from an input .pgm file which could be easily converted from .jpg via some tools 3 basic shapes inside the image, whether square or rectangle or a circle.

# 2.0 Software DESIGN

This project could be organized to several phases (or work packages) as below:

## 2.1 Convert the input grey-level image into binary data

This section will do the following 4 sub-tasks by order:

1. Read the input image file into an array
2. Build up the histogram of each color index for the array from step 1)
3. Calculate the threshold value of the entire image based on the histogram data output from step 2)
4. Convert the input image array into another array containing all binary (0/1) data based on the threshold value from step 3)

An efficient way to calculate the histogram data is to use the elements of the input pixels array as the index of the histogram data array, then simply increase the corresponding histogram data for each pixel.

The key-task in this phase is step 3) – threshold calculation. One possible solution for the threshold calculation is Ostu’s method [[2]](#_6.0_BIBLIOGRAPHY_3).

## 2.2 Connectivity analysis to recognize number of objects

This section will figure out all the isolated objects in the binary image (output from previous step) and assign one unique label for each object.

There are 2 kinds of connectivity, 4-connectivity and 8-connectivity. This project will implement the more complicated one, 8-connectivity, following the traditional algorithm [[3]](#_6.0_BIBLIOGRAPHY_4) to analysis each pixel on its connectivity.

The most challenging part in the connectivity analysis phase is to solve the equivalences between labels. One creative idea, introduced by Luigi Di Stefano and Andrea Bulgarelli [[4]](#_6.0_BIBLIOGRAPHY_1), is to use an 1-dimension array to find out and store the equivalence information during the first scan, after which the equivalences could easily be resolved in the second scan.

## 2.3 Shape Analysis on each objects

This section will perform following tasks by order:

1. Count out the number of objects in one labeled image.
2. Calculate the area of each object by counting the number of pixels of each.
3. Build out the perimeter of each object by border tracing algorithm
4. Determine and tell the shape pattern according to the area and perimeter of each object

The basic idea to determine the shape pattern among square, rectangle and circle is to calculate the rate of each object with the formula: rate = 4 \* PAI \* area / (perimeter \* perimeter) and then check the value of rate:

For square, the rate should be exactly PAI/4

For rectangle, the rate should be less than PAI/4

For circle, the rate should be exactly 1

Above is very ideal, however in the real case, it is unlikely to get that exactly 100% precise data. Hence some balance has to be made in the implementation to get the most possible correct output. Among the 3 shapes which we are targeting to distinguish, it is easy to get an almost 100% precise perimeter of a square or a rectangle, so we can simply distinguish these 2 shapes and leave a relatively wide range for the circle.

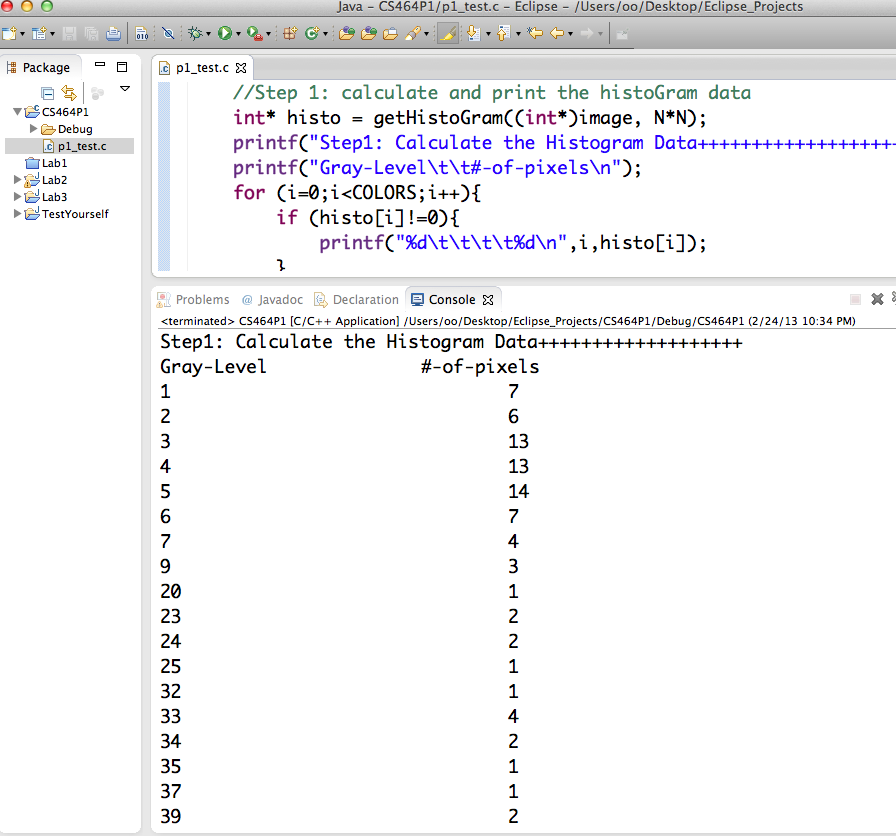
No doubt the most difficult part of this phase is to build out the perimeter as precise as possible especially for the circle shape. Firstly we need to find out the border pixels for each object, which could be done following the traditional algorithm of border tracing [[6]](#_6.0_BIBLIOGRAPHY_2). When finding each border pixel, we add the perimeter accordingly, by increasing one for most of the normal cases. Special cases, such as the corner pixels , should be taken into consideration to get out a more precise perimeter value.

# 3.0 IMPLEMENTATION AND RESULTS

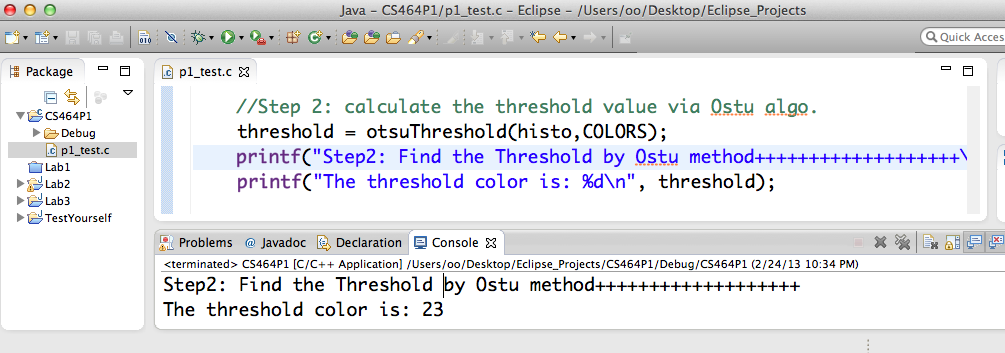
This project is implemented in C language under Eclipse CDT environment.

## 3.1 Convert the input grey-level image into binary data

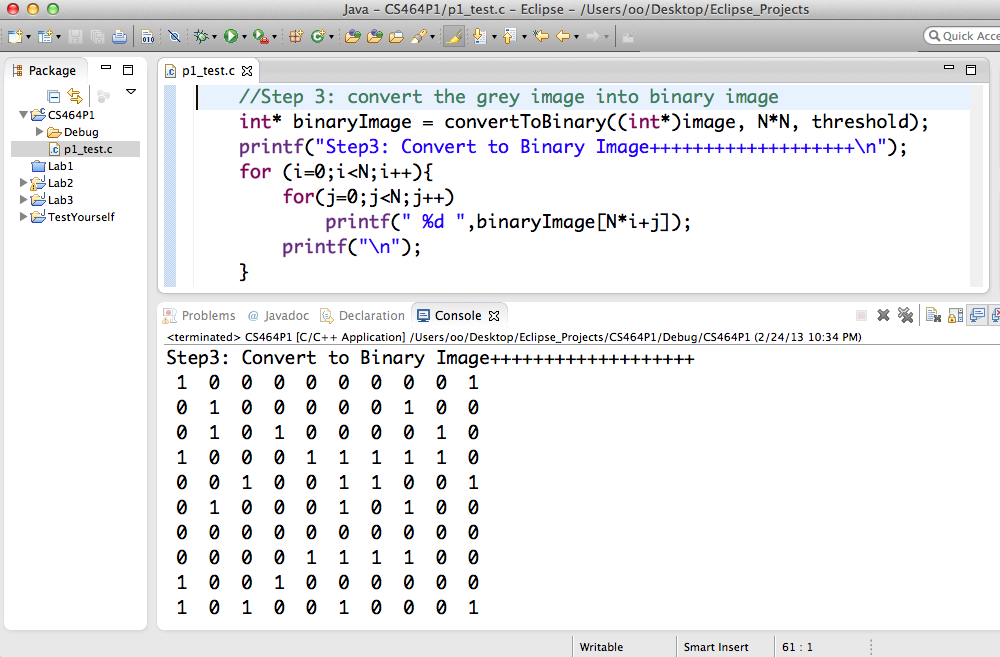
1) Build out the histogram data



2) Calculate the threshold using Ostu algorithm [[2]](#_6.0_BIBLIOGRAPHY_5)



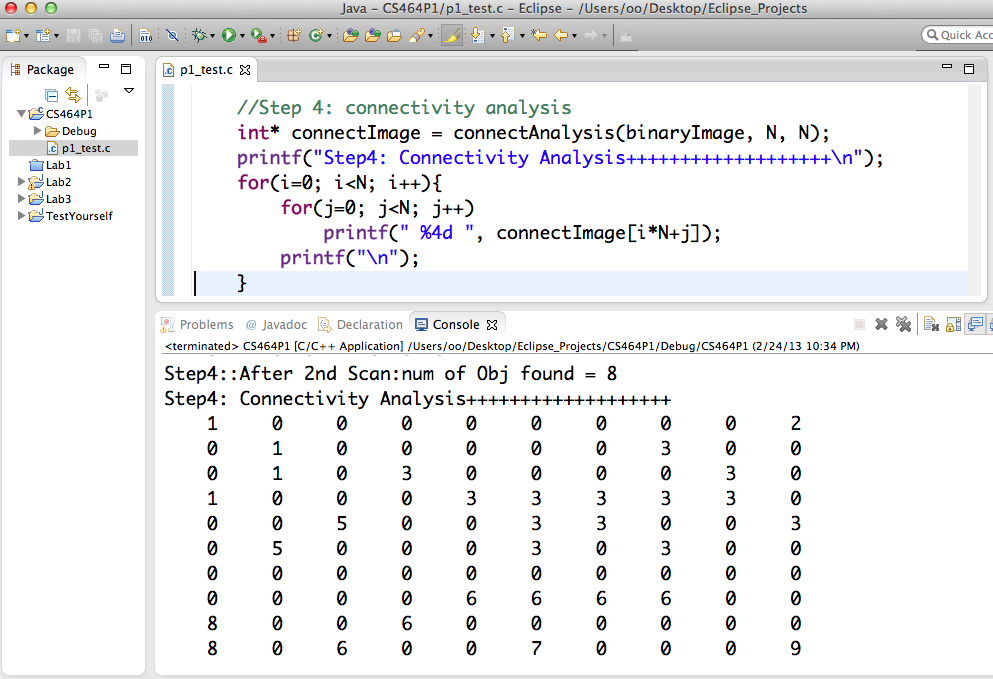
3) Convert the grey-level image into binary data



## 3.2 Connectivity analysis to recognize number of objects

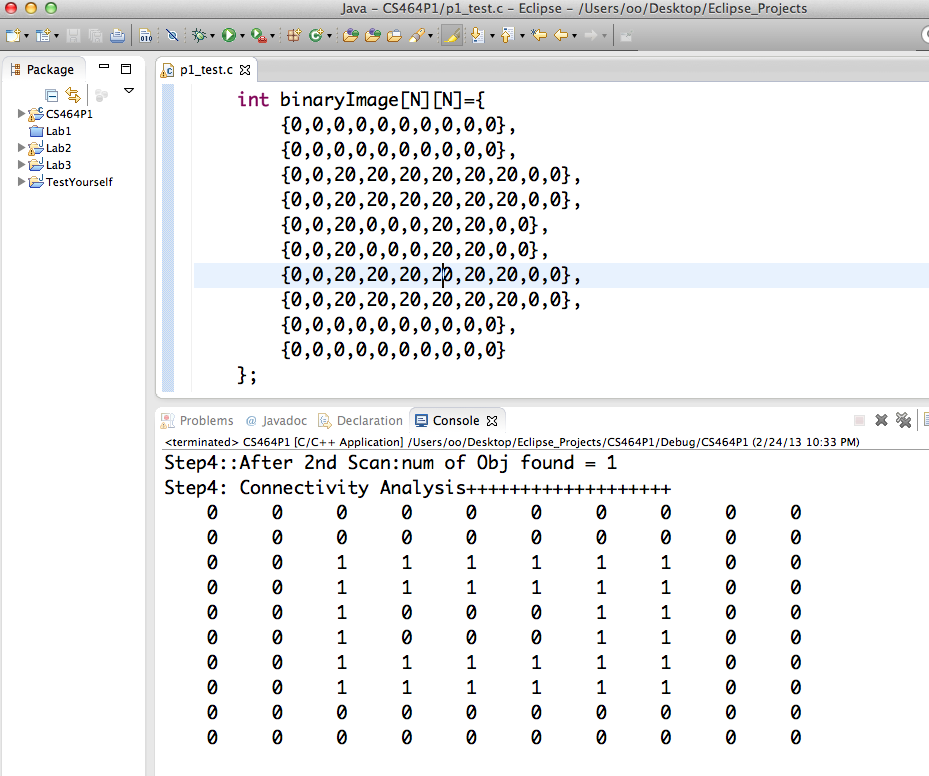
Implement the two-scan algorithm for labeling connected components, introduced by Luigi Di Stefano and Andrea Bulgarelli [[4]](#_6.0_BIBLIOGRAPHY_6). Consider the special pixels on the boundary.

One of the test-run outputs as below:

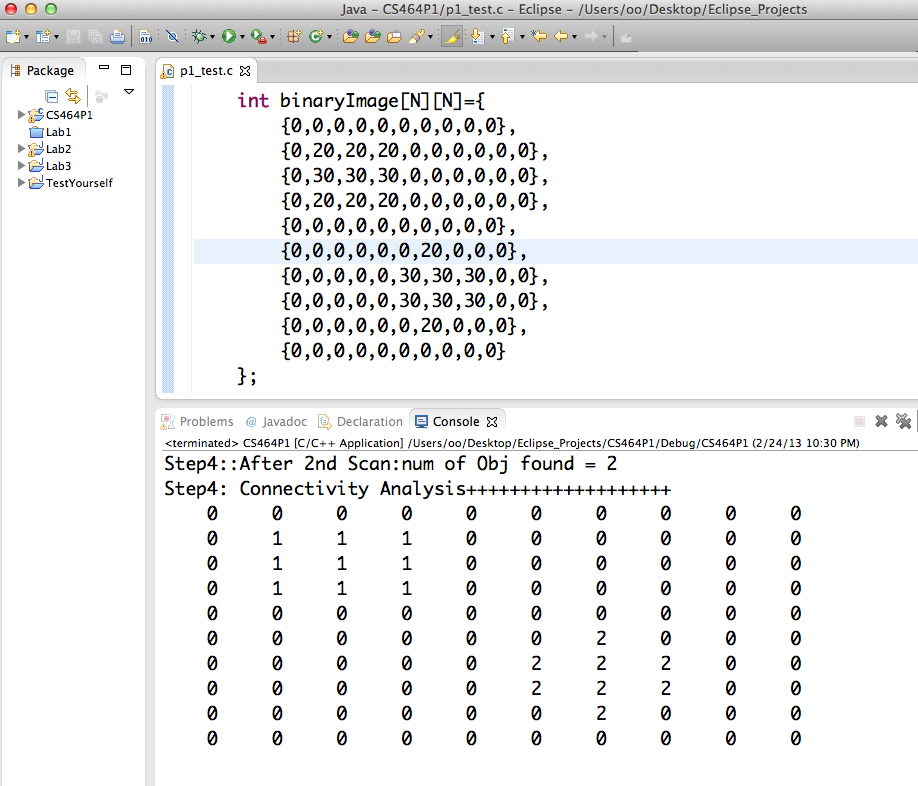


At this step, 3 specific connectivity cases are tested (The hard-coded input data and the output data in a screenshot for each case as following):

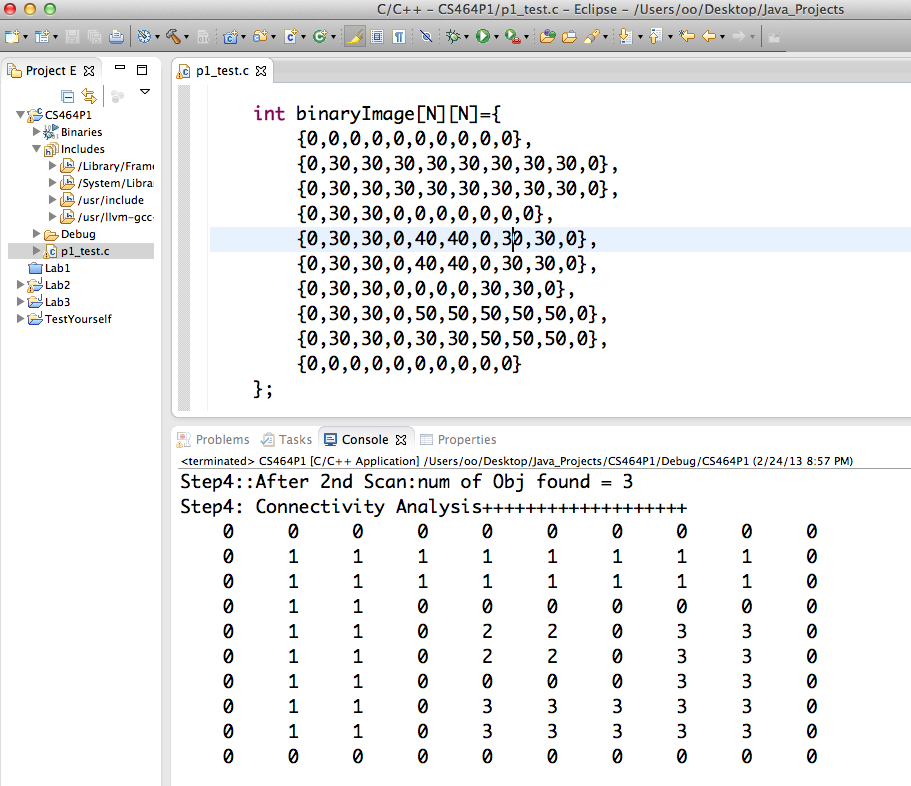
1. Input 1 object (with hole)



1. Input 2 objects



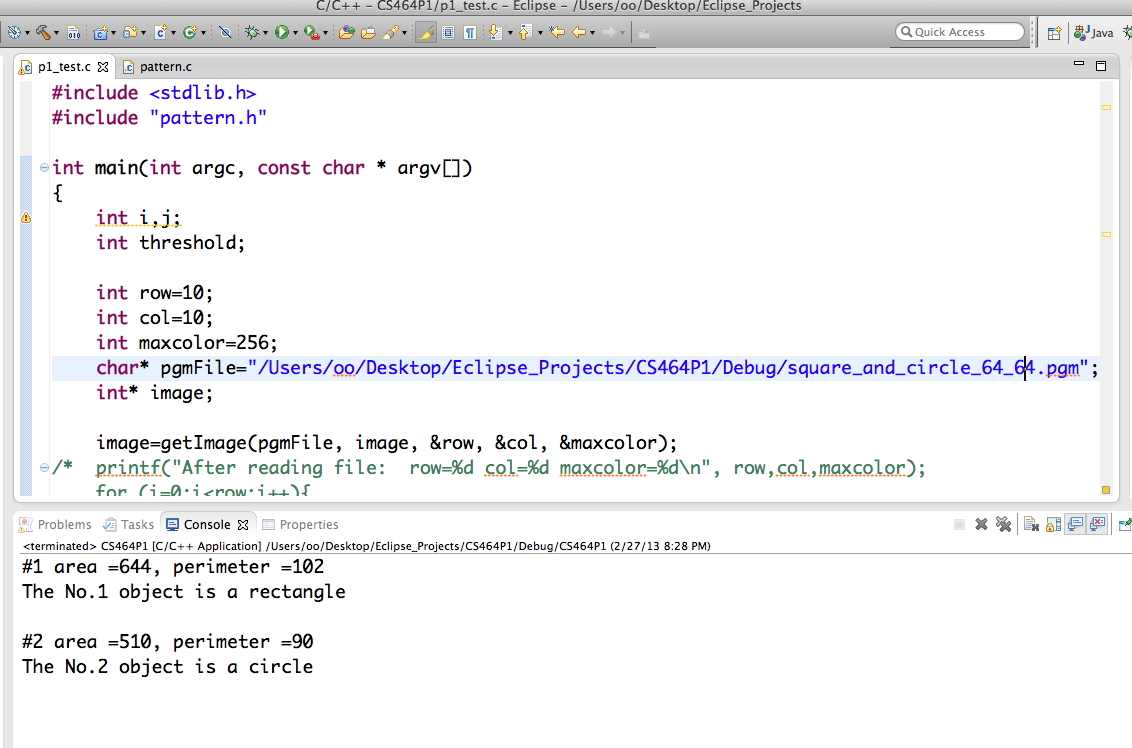
1. Input 3 objects



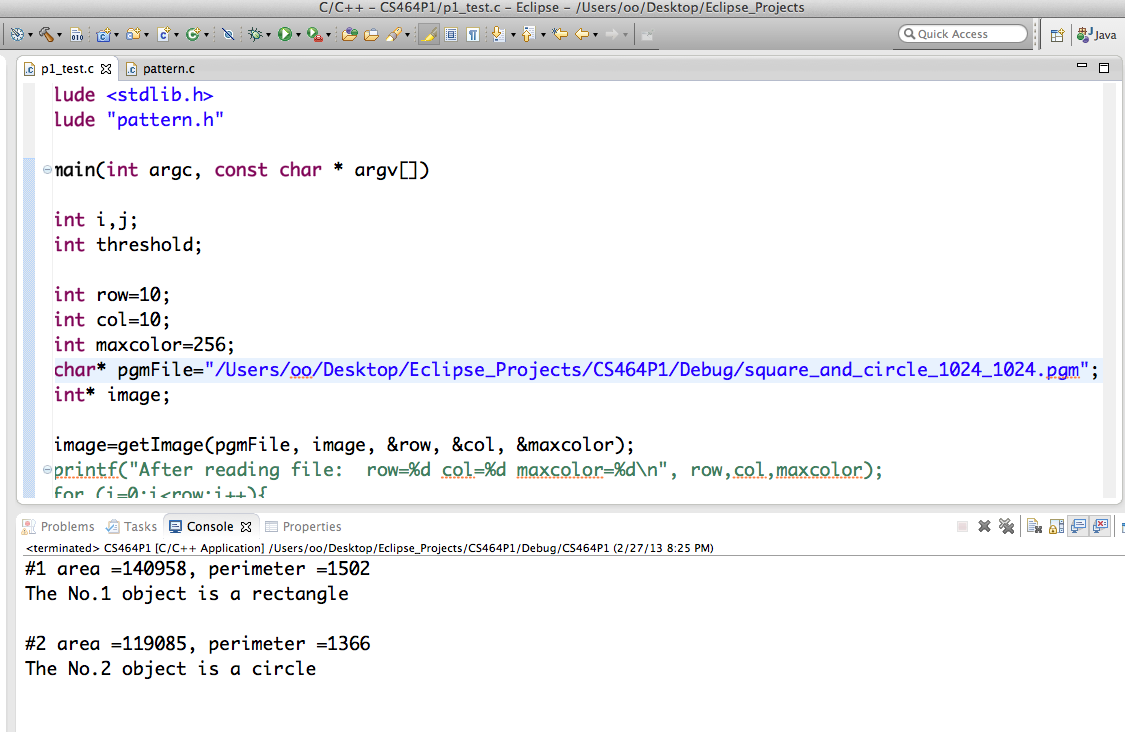
## 3.3 Shape Analysis on each object

Several tests are done on different input image files, showing below:

1. Take the square\_and\_circle\_64\_64.pgm file as input



1. Take the file square\_and\_circle\_1024\_1024.pgm as input



# 4.0 ENHANCEMENT IDEAS

1) The algorithm for the perimeter calculation could be improved by further investigation to achieve better precision, mainly on the circle shapes.

2) More study could be done to extend the capability of this program of recognizing more patterns.

# 5.0 CONCLUSION

Use the elements of another array as the index of array is an easy way to build out histogram data.

Ostu’s method is a very straightforward and efficient way to calculate the threshold value for one grey-level image based on its histogram data.

The new two-scan algorithm for labeling connected components, introduced by Luigi Di Stefano and Andrea Bulgarelli, is really simple and high efficient handling of equivalences. Special boundary check is needed for the connectivity analysis to make sure no overflow run-time error.

It’s not very difficult to follow the traditional method for the border tracing, the most challenging part in this project is to find a good algorithm which is generically suitable for counting the perimeter of all kinds of shape objects. Same as connectivity analysis, the boundary check is needed in this step.

# 6.0 BIBLIOGRAPHY

**[1] The histogram of image intensity levels**

[**http://ftp.utcluj.ro/pub/users/nedevschi/IP/IP\_Labs\_2011/ipl\_03e.pdf**](http://ftp.utcluj.ro/pub/users/nedevschi/IP/IP_Labs_2011/ipl_03e.pdf)

**[2] Ostu’s method for threshold calculation**

[**http://en.wikipedia.org/wiki/Otsu's\_method**](http://en.wikipedia.org/wiki/Otsu's_method)

**[3] Binary objects labeling algorithm**

[**http://ftp.utcluj.ro/pub/users/nedevschi/IP/IP\_Labs\_2011/ipl\_05e.pdf**](http://ftp.utcluj.ro/pub/users/nedevschi/IP/IP_Labs_2011/ipl_05e.pdf)

**[4] A simple and efficient Connected Components Labeling Algorithm**

[**http://octomize.com/~education/classes/image/pattern\_recog/slide/index\_slide.html**](http://octomize.com/~education/classes/image/pattern_recog/slide/index_slide.html)

**[5] Edge based segmentation**

[**http://www.engineering.uiowa.edu/~dip/LECTURE/Segmentation2.html**](http://www.engineering.uiowa.edu/~dip/LECTURE/Segmentation2.html)

**[6] Border Tracing Algorithm**

[**http://ftp.utcluj.ro/pub/users/nedevschi/IP/IP\_Labs\_2011/ipl\_06e.pdf**](http://ftp.utcluj.ro/pub/users/nedevschi/IP/IP_Labs_2011/ipl_06e.pdf)

**[7] All-in-one information about this project**

[**http://octomize.com/~education/classes/image/pattern\_recog/slide/exercise\_pattern\_recog.html**](http://octomize.com/~education/classes/image/pattern_recog/slide/exercise_pattern_recog.html)

# 7.0 APPENDIX

Source code for this project as below:

|  |  |
| --- | --- |
| **File Name** | **Description** |
| pattern.h | The header file with all function prototypes |
| pattern.c | The source file with all the functions written by self |
| readPGM.c | Function copied online: int\* getImage(char \* szFileName, int \*image, int \*row,int \*col ,int \* maxcolor); |
| p1\_test.c | The main() function for testing |

## 1) pattern.h

#define COLORS 256

#define FG 1

#define BG 0

#define LABELS 200

#define PAI 3.14

int\* getImage(char \* szFileName, int \*row,int \*col ,int \* maxcolor);

void patternAnalysis(int\* binaryImage, int row, int col);

int calcPerimeter(int\* binaryImage, int row, int col, int label);

int\* connectAnalysis(int\* binaryImage, int row, int col);

void newPair(int\* label, int label1, int label2, int size);

int\* convertToBinary(int\* image, int size, int threshold);

int otsuThreshold(int\* histoAry,int size);

int\* getHistoGram(int\* ary, int size);

## 2) pattern.c

#include <stdio.h>

#include <stdlib.h>

#include "pattern.h"

void patternAnalysis(int\* binaryImage, int row, int col){

int i,j,k;

int count=0;

int\* label;

int\* perimeter;

int\* area;

double\* rate;

//count the number of objects and record the label value for each obj.

label = (int\*) malloc(LABELS\*sizeof(int));

for(i=0;i<row;i++){

for(j=0;j<col;j++){

if(binaryImage[col\*i+j] != BG){

for(k=0;k<count;k++){

if(binaryImage[col\*i+j] == label[k]){

break;

}

}

if(k==count){

label[count++] = binaryImage[col\*i+j];

}

}

}

}

/\* printf("after label of pattern analysis, count= %d\n", count);

for(k=0;k<count;k++){

printf("No.%d label= %d\t",k+1, label[k]);

}

\*/

//count the area for each obj.

area = (int\*) malloc(LABELS\*sizeof(int));

for(k=0;k<count;k++){

area[k] = 0;

for(i=0;i<row;i++){

for(j=0;j<col;j++){

if(binaryImage[col\*i+j] == label[k]){

area[k]++;

}

}

}

}

/\* printf("\nafter area of pattern analysis, count= %d\n", count);

for(k=0;k<count;k++){

printf("No.%d area= %d\n",k+1, area[k]);

}

\*/

//count the perimeter for each obj.

perimeter = (int\*) malloc(LABELS\*sizeof(int));

for(k=0;k<count;k++){

perimeter[k] = calcPerimeter(binaryImage,row, col,label[k]);

//printf("#%d perimeter= %d\n",k+1, perimeter[k]);

}

/\* printf("\nafter perimeter of pattern analysis\n");

for(k=0;k<count;k++){

printf("#%d perimeter= %d\n",k+1, perimeter[k]);

}

\*/

//check the pattern of each obj.

rate = (double\*) malloc(LABELS\*sizeof(double));

for(k=0;k<count;k++){

printf("#%d area =%d, perimeter =%d\n", k+1,area[k],perimeter[k]);

rate[k] = 4 \* PAI \* area[k] / (perimeter[k] \* perimeter[k]);

printf("rate No. %d is: %f\n", k+1, rate[k]);

if(rate[k] == PAI/4){

printf("The No.%d object is a square\n\n", k+1);

}

else if(rate[k] < PAI/4){

printf("The No.%d object is a rectangle\n\n", k+1);

}

else {

printf("The No.%d object is a circle\n\n", k+1);

}

}

free(label);

free(perimeter);

free(area);

free(rate);

}

int calcPerimeter(int\* binaryImage, int row, int col, int label){

int i, count=0;

int perimeter=0;

int direction=7, prevDir=7;

int index;

int position[2\*(row+col)];

int directionArr[2\*(row+col)];

int curPosition=-1;

int size=row\*col;

//find the first border pixel

for(i=0;i<row\*col;i++){

if(binaryImage[i] == label){

position[count++] = i;

index = i;

//printf("\nfirst elements found at pos. =%d\n", index);

break;

}

}

//trace the border

while(curPosition != position[0]){

if(direction%2==1){

direction = (direction +6) %8;

}

else{

direction = (direction +7) %8;

}

switch(direction){

case 1:

//printf("dir 1\n");

if(index-col+1 < size && binaryImage[index-col+1] == label){

index=index-col+1;

direction=1;

break;

}

if(index-col < size && binaryImage[index-col] == label){

index=index-col;

direction=2;

break;

}

if(index-col-1 < size && binaryImage[index-col-1] == label){

index=index-col-1;

direction=3;

break;

}

if(index -1 < size && binaryImage[index-1] == label){

index=index-1;

direction=4;

break;

}

if(index+col -1 < size && binaryImage[index+col-1] == label){

index=index+col-1;

direction=5;

break;

}

if(index+col < size && binaryImage[index+col] == label){

index=index+col;

direction=6;

break;

}

if(index+col +1 < size && binaryImage[index+col+1] == label){

index=index+col+1;

direction=7;

break;

}

if(index+1 < size && binaryImage[index+1] == label){

index=index+1;

direction=0;

break;

}

break;

case 3:

//printf("dir 3\n");

if(index-col -1 < size && binaryImage[index-col-1] == label){

index=index-col-1;

direction=3;

break;

}

if(index-1 < size && binaryImage[index-1] == label){

index=index-1;

direction=4;

break;

}

if(index+col -1 < size && binaryImage[index+col-1] == label){

index=index+col-1;

direction=5;

break;

}

if(index+col < size && binaryImage[index+col] == label){

index=index+col;

direction=6;

break;

}

if(index+col +1 < size && binaryImage[index+col+1] == label){

index=index+col+1;

direction=7;

break;

}

if(index+1 < size && binaryImage[index+1] == label){

index=index+1;

direction=0;

break;

}

if(index-col +1 < size && binaryImage[index-col+1] == label){

index=index-col+1;

direction=1;

break;

}

if(index-col < size && binaryImage[index-col] == label){

index=index-col;

direction=2;

break;

}

break;

case 5:

//printf("dir 5\n");

if(index+col -1 < size && binaryImage[index+col-1] == label){

index=index+col-1;

direction=5;

break;

}

if(index+col < size && binaryImage[index+col] == label){

index=index+col;

direction=6;

break;

}

if(index+col +1 < size && binaryImage[index+col+1] == label){

index=index+col+1;

direction=7;

break;

}

if(index+1 < size && binaryImage[index+1] == label){

index=index+1;

direction=0;

break;

}

if(index-col +1 < size && binaryImage[index-col+1] == label){

index=index-col+1;

direction=1;

break;

}

if(index-col < size && binaryImage[index-col] == label){

index=index-col;

direction=2;

break;

}

if(index-col -1 < size && binaryImage[index-col-1] == label){

index=index-col-1;

direction=3;

break;

}

if(index -1 < size && binaryImage[index-1] == label){

index=index-1;

direction=4;

break;

}

break;

case 7:

//printf("dir 7\n");

if(index+col +1 < size && binaryImage[index+col+1] == label){

index=index+col+1;

direction=7;

break;

}

if(index+1 < size && binaryImage[index+1] == label){

index=index+1;

direction=0;

break;

}

if(index-col +1 < size && binaryImage[index-col+1] == label){

index=index-col+1;

direction=1;

break;

}

if(index-col < size && binaryImage[index-col] == label){

index=index-col;

direction=2;

break;

}

if(index-col -1 < size && binaryImage[index-col-1] == label){

index=index-col-1;

direction=3;

break;

}

if(index-1 < size &&binaryImage[index-1] == label){

index=index-1;

direction=4;

break;

}

if(index+col -1 < size && binaryImage[index+col-1] == label){

index=index+col-1;

direction=5;

break;

}

if(index+col < size && binaryImage[index+col] == label){

index=index+col;

direction=6;

break;

}

break;

}

//increase the perimeter

if( direction%2==1){

perimeter = perimeter + 0.41;

}

else if(direction != prevDir && prevDir%2 ==0){

perimeter = perimeter +1;

}

else if(index == position[0] + col ){

//printf("1st point is a corner");

perimeter = perimeter +1;

}

perimeter++;

/\*if(direction == directionArr[count-1] && direction%2==1){

//printf("contingours 2 bi-direction\n");

perimeter = perimeter - 1;

}\*/

curPosition = index;

position[count] = index;

directionArr[count++] = direction;

prevDir = direction;

}

return perimeter;

}

int\* connectAnalysis(int\* binaryImage, int row, int col){

int i,j;

int\* connectImage = (int\*) malloc(row\*col\*sizeof(int));

int west, north, northwest, northeast, current;

int numObjects=0;

int newLabel=1;

//the array to resolve labels' equivalence

int\* label = (int\*)malloc(row\*sizeof(int));

for (i=0;i<row;i++)

label[i] = i;

//1st scan to seperate labels

//1st scan: 1)handle 1st row

if (binaryImage[0] !=BG){

connectImage[0] = newLabel++;

numObjects++;

}

else {

connectImage[0] = binaryImage[0];

}

for (j=1;j<col;j++){

if(binaryImage[j] ==BG){

connectImage[j] = binaryImage[j];

}

else{

west = connectImage[j-1];

if(west !=BG){

current = west;

}

else{

current = newLabel++;

numObjects++;

}

connectImage[j] = current;

}

}

//1st scan: 2)handle the body

for(i=1; i<row; i++)

for(j=1; j<col-1; j++){

if(binaryImage[i\*col+j] ==BG){

connectImage[i\*col+j] = binaryImage[i\*col+j];

}

else{

north = connectImage[(i-1)\*col+j];

west = connectImage[i\*col+j-1];

northwest = connectImage[(i-1)\*col+j-1];

northeast = connectImage[(i-1)\*col+j+1];

if(northwest !=BG){

if(north ==BG && northeast !=BG){

current = northwest;

if(label[northwest] != label[northeast]){

newPair((int\*)label, northeast, northwest, newLabel);

numObjects--;

}

}

else{

current = northwest;

}

}

else if(west !=BG){

if(north !=BG){

current = north;

if (label[west] != label[north]){

newPair((int\*)label, west, north, newLabel);

numObjects--;

}

}

else if(northeast !=BG){

current = northeast;

if (label[west] != label[northeast]){

newPair((int\*)label, west, northeast, newLabel);

numObjects--;

}

}

else

current = west;

}

else if(north !=BG){

current = north;

}

else if(northeast !=BG){

current = northeast;

}

else {

current = newLabel++;

numObjects++;

}

connectImage[i\*col+j] = current;

}

}

//1st scan: 3)handle the 1st column & last column

for(i=1; i<row; i++){

if(binaryImage[i\*col] ==BG){

connectImage[i\*col] = binaryImage[i\*col];

}

else{

north = connectImage[(i-1)\*col];

northeast = connectImage[(i-1)\*col+1];

if(north !=BG){

current = north;

}

else if (northeast !=BG){

current = northeast;

}

else{

current = newLabel++;

numObjects++;

}

connectImage[i\*col] = current;

}

if(binaryImage[(i+1)\*col-1] ==BG){

connectImage[(i+1)\*col-1] = binaryImage[(i+1)\*col-1];

}

else{

north = connectImage[i\*col-1];

northwest = connectImage[i\*col-2];

if(northwest !=BG){

current = northwest;

}

else if (north !=BG){

current = north;

}

else{

current = newLabel++;

numObjects++;

}

connectImage[(i+1)\*col-1] = current;

}

}

//2nd scan to merge labels

for(i=0; i<row; i++){

for(j=0; j<col; j++){

if(connectImage[i\*row+j]!= BG)

connectImage[i\*row+j] = label[connectImage[i\*row+j]];

}

}

//printf("Step4::After 2nd Scan:num of Obj found = %d\n", numObjects);

free(label);

return connectImage;

}

void newPair(int\* label, int label1, int label2, int size){

int k;

for (k=1;k<=size;k++){

if (label[k]==label[label1])

label[k] = label[label2];

}

}

int\* convertToBinary(int\* image, int size, int threshold){

int i;

int\* binaryImage = (int\*) malloc(size\*sizeof(int));

for (i=0;i<size;i++){

if (image[i] <= threshold)

binaryImage[i] = 0;

else

binaryImage[i] = 1;

}

return binaryImage;

}

int otsuThreshold(int\* histoAry,int size)

{

int threshold;

double sumColors=0, sumPixels=0;

double sumLowColors=0,sumLowPixels=0, avgColors;

double percentLow, avgColorLow;

double diff,maxDiff=0;

int i,j;

//calculate the avg grey-level

for (i=0;i<size;i++){

sumColors += i\*histoAry[i];

sumPixels += histoAry[i];

}

avgColors = sumColors / sumPixels;

for (i=0;i<size;i++){

if(histoAry[i] == 0)

continue;

//reset the sum for low colors

sumLowColors = 0;

sumLowPixels = 0;

for(j=0;j<=i;j++){

sumLowColors += j\*histoAry[j];

sumLowPixels += histoAry[j];

}

percentLow = sumLowPixels / sumPixels;

avgColorLow = sumLowColors / sumLowPixels;

diff = percentLow \* (avgColorLow - avgColors) \* (avgColorLow - avgColors) / (1 - percentLow);

//calculate the maxDiff for each grey-level

if (maxDiff < diff){

maxDiff = diff;

threshold = i;

}

}

return threshold;

}

int\* getHistoGram(int\* ary, int size)

{

int i;

int\* histo = (int\*) calloc(COLORS,sizeof(int));

for (i=0;i<size;i++){

histo[ary[i]]++;

}

return histo;

}

## 3) readPGM.c

/\*

\* readPGM.c

\*

\* Created on: Feb 26, 2013

\* Author: oo

\*/

#include <stdio.h>

#include <stdlib.h>

#include "pattern.h"

/\*

char \* szFileName : name of image file, [must not null value]

int \*\*image : one dimension array for image information,

[memory all by this function]

int \* row : number of row, [must not null value]

int \* col : number of column, [must not null value]

int \* maxcolor : maximum value of color number, such as 256

means 0-255, [must not null value]

return : 0-success, 1 : failed

\*/

int\* getImage(char\* szFileName, int\* row, int\* col ,int\* maxcolor)

{

\*row = 0;

\*col = 0;

int i;

int j;

//int lineFlag = 0; //0-Empty line

char ch;

//char pre=' ';

FILE \* fpText;

int\* image;

if ((NULL == szFileName)

|| (NULL == row)

|| (NULL == col)

|| (NULL == maxcolor))

{

printf("\aERROR error parameter. \a\n");

return NULL;

}

fpText = fopen(szFileName, "rb");

if( fpText == NULL )

{

printf("\aERROR can not open image file. \a\n");

return NULL;

}

ch = fgetc(fpText); //skip 'p'

if ((ch != 'p') && (ch != 'P'))

{

printf("\aERROR invalid image file(%s). \a\n", szFileName);

return NULL;

}

ch = fgetc(fpText); //skip '5/6'

ch = fgetc(fpText); //skip 'space'

//get width

fscanf(fpText,"%d",col);

ch = fgetc(fpText); //skip 'enter'

fscanf(fpText,"%d",row);

ch = fgetc(fpText); //skip 'enter'

fscanf(fpText,"%d",maxcolor);

ch = fgetc(fpText); //skip 'enter'

(\*maxcolor)++;

if ((\*row <= 0) || (\*col <= 0) || (\*maxcolor <= 0))

{

printf("\aERROR invalid image file(%s). \a\n", szFileName);

return NULL;

}

image = (int\*)malloc(sizeof( int) \* (\*col) \* (\*row) );

if( (image) == NULL )

{

printf("\aERROR alloc memory error. \a\n");

return NULL;

}

for (i = 0; i < \*row; i++)

{

for (j = 0; j < \*col; j++)

{

((image)[i\*(\*col) + j] = fgetc(fpText));

//printf("%d ", ((image)[i\*(\*col) + j] ));

}

}

if ( fclose( fpText ) == EOF )

{

printf( "\aERROR closing file (%s) \n",szFileName) ;

free(image);

return NULL;

} // fclose( fpText );

return image;

}

## 4) p1\_test.c

#include <stdio.h>

#include <stdlib.h>

#include "pattern.h"

int main(int argc, const char \* argv[])

{

int threshold;

int row,col,maxcolor;

char\* pgmFile="/Users/oo/Desktop/Eclipse\_Projects/CS464P1/Debug/square\_and\_circle\_1024\_1024.pgm";

int\* image;

//read image data into one array from pgm file

image = getImage(pgmFile, &row, &col, &maxcolor);

//Step 1: calculate and print the histoGram data

int\* histo = getHistoGram((int\*)image, row\*col);

//Step 2: calculate the threshold value via Ostu algo.

threshold = otsuThreshold(histo,maxcolor);

//Step 3: convert the grey image into binary image

int\* binaryImage = convertToBinary((int\*)image, row\*col, threshold);

//Step 4: connectivity analysis

int\* connectImage = connectAnalysis(binaryImage, row,col);

//Step5: pattern analysis, can differentiate circle and square only

patternAnalysis(connectImage, row, col);

//release the dynamic memory allocated

free(histo);

free(binaryImage);

free(connectImage);

free(image);

return 0;

}