**Hands-on Lab**

**Image Processing**

A digital image is a collection of pixel data. A pixel has a location and value. The location is the row and column in the image whereas the value represents the color at that location. This lab introduces the reading, processing, and writing of digital image using ANSI C (GCC) and RAW image file standard. By using ANSI C, the underlying algorithms should be platform independent. RAW images are standard uncompressed binary files. The net effect is a brief but focused introduction to image processing which is the pre-cursor to robotic computer vision.

**Preliminary:** Code Blocks, IrfanView, and Pixel Former

Before doing this lab, tutorials on Code Blocks, IrfanView and Pixelformer should be completed: These software allows one to: write C programs for image processing (Code Blocks); create bitmap images (Pixelformer); and create RAW files and view results (IrfanView).

Blah: Lecture notes on row-column vector

**Concept 1:** Thresholding images (Read and Write RAW files) **threshold1\_0a.c**

Thresholding reads an image and writes a black-and-white output image. This is important because working with a binary image (i.e. a pixel is either black or white) often simplifies image understanding (e.g. detecting edges, calculating area and centroids, and object counting). Thresholding is thus a “Hello World” example for image processing. Figure 1A lists main for **threshold1\_0a.c** threshold a 256-by-256 grayscale image.

Unlike JPEG, PNG and other image files, RAW files are uncompressed and have no headers. One simply reads the binary file one byte at a time using a loop.

// FILE: threshold1\_0a.c - Works!

// DATE: 02/21/20 08:34

// AUTH: P.Oh

// DESC: Output is threshold of Input image

#include<stdlib.h>

#include<stdio.h>

#include<memory.h>

struct Image {

int Rows, Cols; // image's number of rows and columns

unsigned char \*Data; // pointer to image data

}; // end of struct Image

int main() {

FILE \*ofile;

struct Image In, Out; // Declare input and output images

// Assumes RAW image is 256-by-256 bytes and allocate memory for images

In.Rows = Out.Rows = In.Cols = Out.Cols = 256;

In.Data = (unsigned char \*)calloc(In.Rows, In.Cols);

Out.Data = (unsigned char \*)calloc(Out.Rows, Out.Cols);

Img\_in(&In);

Img\_threshold(&In, &Out);

Img\_out(&Out);

} // end of main

**Figure 1A:** main for **threshold1\_0a.c**

The .h files are standard include files. The structure Image lying outside main is a global variable. Image holds the sizes of the rows (Rows) and columns (Cols) and pixel data (\*Data)

In main, the yellow-highlight shows that data being allocated. It’s assumed that the RAW image will have 256 rows and 256 columns of 1-byte pixels. Next, main calls 3 functions to respectively read an input image, process it, and write the output image.

**Step 1:** Reading a RAW image

void Img\_in(struct Image \*Img) {

FILE \*ifile;

int i;

// NB: Assumes RAW image file 256 x 256 size

// Open file for binary reading

// Assumes RAW file in same directory as this C-program

ifile = fopen("cameraMan.raw", "rb"); // read binary file

// Read directly into the image array

for(i=0; i < Img->Rows; ++i)

fread(Img->Data + i\*Img->Cols, Img->Cols, 1, ifile);

fclose(ifile);

} // end Img\_in

**Figure 1B:** **Img\_in** function

The function Img\_in is used to read a RAW file (**Figure 1B**). As input, it takes a pointer to an Image structure. This function begins with fopen to open the desired input RAW image file (cameraMan.raw in this case). The row-column format is used to store pixel data as a vector. This is implemented by a single for loop an moves the pointer through the image file. The function ends by closing the file. Recall the structure variable Image is a global one, so other functions will be able to access this variable.

**Step 2:** Processing the RAW image

void Img\_threshold(struct Image \*In, struct Image \*Out) {

long i, j;

int val, thresholdValue;

unsigned char \*tmp;

thresholdValue = 50;

for(i=0; i<In->Rows; ++i) {

for(j=0; j<In->Cols; ++j) {

val = \*(In->Data + i\*In->Rows + j);

if(val < thresholdValue) {

val = 0;

}

else {

val = 255;

}

tmp = Out->Data + i\*Out->Rows + j;

\*tmp = (unsigned char)val;

};

};

} // end Img\_threshold

**Figure 1C:** **Img\_threshold** function

The function Img\_threshold is used to implement thresholding (**Figure 1C**). As inputs, this function takes pointers to the Image structures (input and output Image structures.

The threshold value is set (50 in this case) in the variable thresholdValue. The nested for-loops then reads each pixel of the input image data and stores the pixel value in the variable val and compared to thresholdValue.

Recall that 8-bit pixel data ranges in values from 0 (black) to 255 (white). Setting thresholdValue closer to 0 means that the darkest pixels are set black, while all other pixels are set white. The value of resulting threshold is then set to the pointer tmp which stores the data in the global structure variable Out.

**Step 3:** Write the RAW image

void Img\_out(struct Image \*Out) {

FILE \*ofile;

int i;

// Open (or create) binary file for writing

ofile = fopen("thresholdOutput.raw", "wb");

// Output the image by rows

for(i=0; i < Out->Rows; ++i)

fwrite(Out->Data + i\*Out->Cols, Out->Cols, 1, ofile);

fclose(ofile);

} // end Img\_out

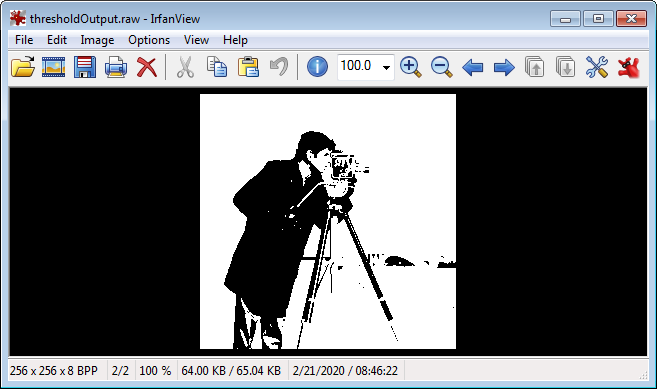
**Figure 1D:** **Img\_out** function

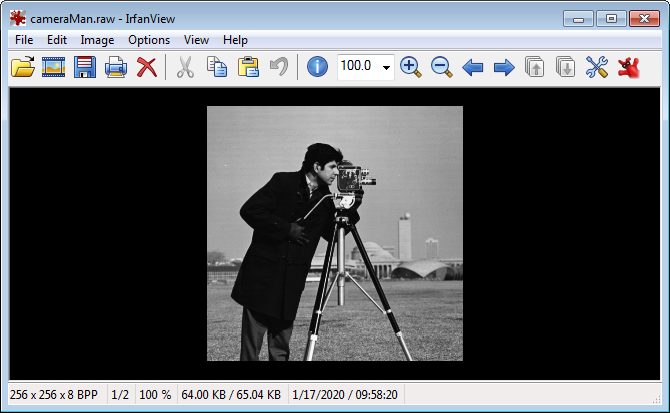
The function Img\_out takes an Image structure (**Figure 1D**). It opens a file (thresholdOutput.raw in this case) and proceeds with a for-loop and fwrite to write the data to the file. The function ends by closing the file.

**Step 4:** Threshold a RAW image file

Combine **Figures 1A** thru **1D** into a single file named **threshold1\_0a.c**. Make sure that the input image file (cameraMan.raw) is in the same folder as threshold1\_0a.c. Compile and execute to generate the output file. View thresholdOutput.raw with IrfanView (**Figure 1E**).

**Figure 1E:** Original cameraMan.raw file (left) processed with threshold1\_0a.c with thresholdValue set to 50. Note that for that value, only the darkest pixels remain black.





Exercises

* 1. Write a program to threshold a RAW grayscale image (e.g. cameraMan.raw) so that only the whitest pixels remain white.

* 1. Write a program to reads a RAW mage file (e.g. cameraMan.raw) and outputs the inverse (i.e. a negative).

**Concept 2:** Areas and Centroids - **areaCentroid1\_0.c**

From lecture, the area is defined as the number of pixels (of a specific value) in the image. The centroid of an image is calculated as and where and are the centroid coordinates, and are the th pixel’s coordinates, and is the area of the object.

**Step 1:** Write the function to calculate the image’s area

**Figure 2A** is a 16x16 RAW test image (16x16-ballRaw.raw).

**Figure 2A:** Pixelformer was used to create a 16x16 image with white (value = 255) background and black (value = 0) pixels to create the ball. Counting the number of black pixels, the area of this image should be 24.



float area(struct Image \*In, int x1, int y1,

int x2, int y2, unsigned char ObjVal) {

// returns calculated area of a RAW image

long i, j;

float areaValue = 0.0; // although this is an int, will use for float division

for(i=x1; i <= x2; ++i)

for(j=y1; j <= y2; ++j) {

if(pix(In, i, j)==ObjVal)

areaValue = areaValue + 1.0;

}

return(areaValue);

} // end function area

**Figure 2B:** Function to calculate the image’s area

The listing in **Figure 2B** takes an Image structure, the starting row and column of the image, the ending row and column of the image, and the desired pixel value (ObjVal). The nested for-loop compares the Image’s pixel value to ObjVal. If it matches, then areaValue is incremented. Technically, areaValue is an integer (i.e. whole number of pixels). However, for calculating the centroid later, is areaValue declared as a float.

To increase the code’s readability, pix(In, i, j)is used to denote the (i,j) pixel of the input Image. This variable is #defined as a global variable.

**Step 2:** Write the function to calculate the image’s centroid

struct coord centroid(struct Image \*In, int x1,

int y1, int x2, int y2,

unsigned char ObjVal) {

// returns calculated centroid (as struct) of RAW image

long i, j;

float calculatedArea;

int xSum, ySum;

struct coord calculatedCentroid;

calculatedArea = area(In, x1, y1, x2, y2, ObjVal);

if(calculatedArea == 0) {

calculatedCentroid.x = -1; calculatedCentroid.y = -1;

return(calculatedCentroid);

};

xSum = ySum = 0;

for(i=x1; i<=x2; ++i)

for(j=y1; j<=y2; ++j) {

if(pix(In, i, j) == ObjVal) {

xSum += j;

ySum += i;

}

}

calculatedCentroid.x = xSum/calculatedArea;

calculatedCentroid.y = ySum/calculatedArea;

return(calculatedCentroid);

} // end function centroid

**Figure 2C:** Listing for function centroid

The centroid function (**Figure 2C**) takes on the same parameters as the area function. It returns a structure coord which is declared as a global variable. This structure will contain the x and y location of the calculated centroid.

The nested for-loop compares the image’s pixel to the desired pixel value (ObjVal). When equal, the column and row values of that pixel are accumulated in xSum and ySum respectively. The centroid is then calculated by dividing those accumulated sums by the image’s area (calculatedArea) and returned.

**Step 3:** Write main program to call area and centroid functions and print results

**Figure 2D** shows the full listing of areaCentroid1\_0.c. The yellow highlights show the #defined variable pix(In, i, j) and global structure variable coord. Much like Concept 1’s threshold1\_0a.c, the functions Img\_in (**Figure 1B**) and struct Image are used.

// FILE: areaCentroid1\_0.c - Works!

// DATE: 02/26/20 09:44

// AUTH: P.Oh

// DESC: Report area and centroid of RAW image

// REFS: areaCentroid0\_1b.c

#include<stdlib.h>

#include<stdio.h>

#include<memory.h>

#include<math.h>

#define pix(Im, x, y) \*(Im->Data + (x)\*Im->Cols + (y))

#define WHITE 255

#define BLACK 0

struct Image {

int Rows, Cols; // image's number of rows and columns

unsigned char \*Data; // pointer to image data

}; // end of struct Image

struct coord {

float x, y; // result's row and column coordinates

};

void Img\_in(struct Image \*Img) {

FILE \*ifile;

int i;

// NB: Assumes RAW image file 256 x 256 size

// Open file for binary reading

// Assumes RAW file in same directory as this C-program

ifile = fopen("16x16-ballRaw.raw", "rb"); // read binary file

// Read directly into the image array

for(i=0; i < Img->Rows; ++i)

fread(Img->Data + i\*Img->Cols, Img->Cols, 1, ifile);

fclose(ifile);

} // end Img\_in

void Img\_out(struct Image \*Out) {

FILE \*ofile;

int i;

// Open (or create) binary file for writing

ofile = fopen("thresholdOutput.raw", "wb");

// Output the image by rows

for(i=0; i < Out->Rows; ++i)

fwrite(Out->Data + i\*Out->Cols, Out->Cols, 1, ofile);

fclose(ofile);

} // end Img\_out

float area(struct Image \*In, int x1, int y1,

int x2, int y2, unsigned char ObjVal) {

// returns calculated area of a RAW image

long i, j;

float areaValue = 0.0; // although this is an int, will use for float division

for(i=x1; i <= x2; ++i)

for(j=y1; j <= y2; ++j) {

if(pix(In, i, j)==ObjVal)

areaValue = areaValue + 1.0;

}

return(areaValue);

} // end function area

**Figure 2D:** Full listing of areaCentroid1\_0.c

struct coord centroid(struct Image \*In, int x1,

int y1, int x2, int y2,

unsigned char ObjVal) {

// returns calculated centroid (as struct) of RAW image

long i, j;

float calculatedArea;

int xSum, ySum;

struct coord calculatedCentroid;

calculatedArea = area(In, x1, y1, x2, y2, ObjVal);

if(calculatedArea == 0) {

calculatedCentroid.x = -1; calculatedCentroid.y = -1;

return(calculatedCentroid);

};

xSum = ySum = 0;

for(i=x1; i<=x2; ++i)

for(j=y1; j<=y2; ++j) {

if(pix(In, i, j) == ObjVal) {

xSum += j;

ySum += i;

}

}

calculatedCentroid.x = xSum/calculatedArea;

calculatedCentroid.y = ySum/calculatedArea;

return(calculatedCentroid);

} // end function centroid

int main() {

struct Image In; // Declare input and output images

struct coord centroidCoordinates;

int areaImage;

// Assumes RAW image is 16-by-16 bytes and allocate memory

In.Rows = 16;

In.Cols = 16;

In.Data = (unsigned char \*)calloc(In.Rows, In.Cols);

Img\_in(&In);

areaImage = area(&In, 0, 0, (In.Rows-1), (In.Cols-1), BLACK);

printf("Area of 16x16 image is: %d\n", areaImage);

centroidCoordinates = centroid(&In, 0, 0, (In.Rows-1), (In.Cols-1), BLACK);

printf("Centroid is (x,y) = (%3.3f, %3.3f)\n", centroidCoordinates.x, centroidCoordinates.y);

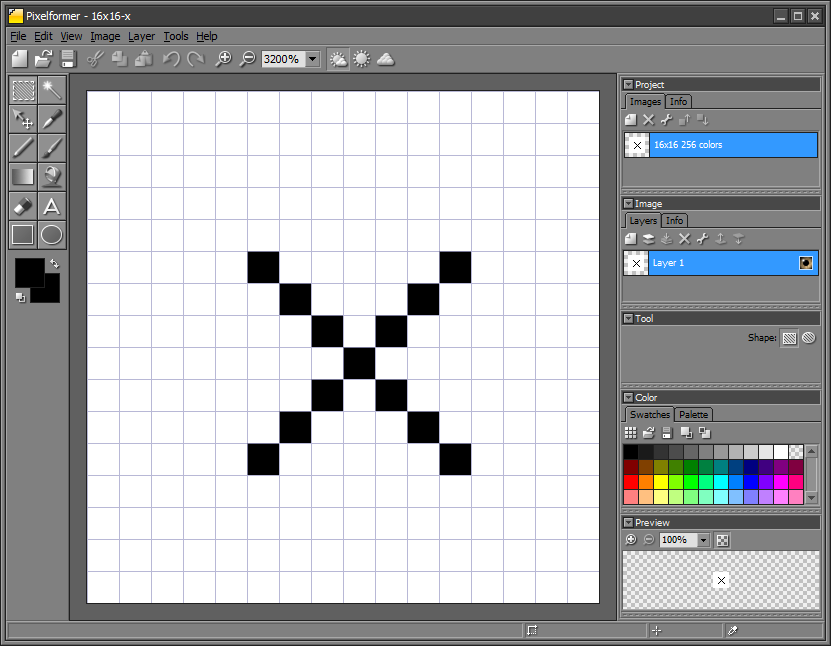
} // end of main

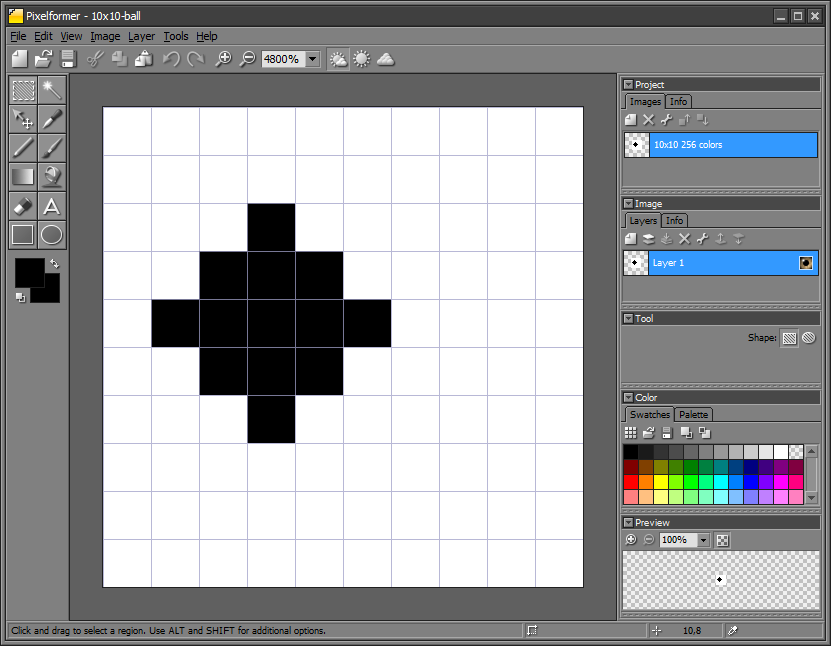
**Figure 2D continued:** Full listing of areaCentroid1\_0.c

Exercises

* 1. Modify areaCentroid1\_0.c to read a 16x16 RAW image 16x16-x-Raw.raw. What are the values of the area and centroid?
  2. Use Pixelformer to create a 16x16 image of a white ball on black background and use IrfanView to create an equivalent RAW image. Hand-calculate the area and centroid. Write a C program to report the area and centroid. Compare with your hand-calculations.
  3. Modify areaCentroid1\_0.c to read a 10x10 RAW image 10x10-ballRaw.raw. What are the values of the area and centroid?

16x16-x-Raw.raw (left) and 10x10-ballRaw.raw (right)





**Concept 3:** Drawing white box

Exercises

* 1. Set numberOfWayPoints to 7 in **xl320-line1\_0.nxc** and the Wait statement and velocities in **xl320-twoLinkFunctions1\_0a.h** using the table below.. Compile, execute, and fill your observations

|  |  |  |  |
| --- | --- | --- | --- |
| numberOfWayPoints | Wait | Velocity  [counts/min] | What path does the end-effector make? How well does it “stay” on the desired line? |
| 15 | 50 | 200 |  |
| 15 | 50 | 400 |  |
| 15 | 100 | 200 |  |
| 15 | 100 | 400 |  |
| 5 | 50 | 200 |  |
| 5 | 50 | 400 |  |
| 5 | 100 | 200 |  |
| 5 | 100 | 400 |  |

* 1. to use the negative square roots of (1). Calculate by hand, what the resulting xCalibrate and yCalibrate points should be, Affix green-colored 1-stud bricks at these points. Confirm your program indeed commands the 2-link planar manipulator’s end-effector to hover over those points.
  2. Unscrew and reverse the beams such that Link 1 is a Beam 7 and Link 2 is a Beam 9 and appropriately change in **xl320-ik-1\_0.nxc**. Affix green-colored 1-stud bricks at these points. Confirm your program indeed commands the 2-link planar manipulator’s end-effector to hover over those points.

**Summary Conclusions:**

1. Using way points to characterize points on a line is a simply and intuitive way to move an end-effector on a path
2. While intuitive, the way point approach is naïve; it’s a position-control approach which yields a go-stop-go motion
3. Velocity-controlled approaches leverage a motor’s “on-the-fly” motion profile. This can yield a smooth motion rather than a go-stop-go one.

There are many methods to implement the last point. The most popular ones define motor velocities based on a polynomial. Examples include cubic and quintic polynomials and B-splines. These methods still use way points, but adjust the velocities as the end-effector passes over them.