**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

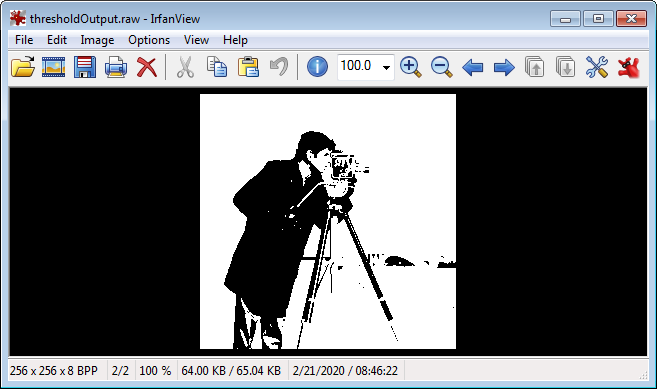
**Figure 1D:** **Img\_out** function

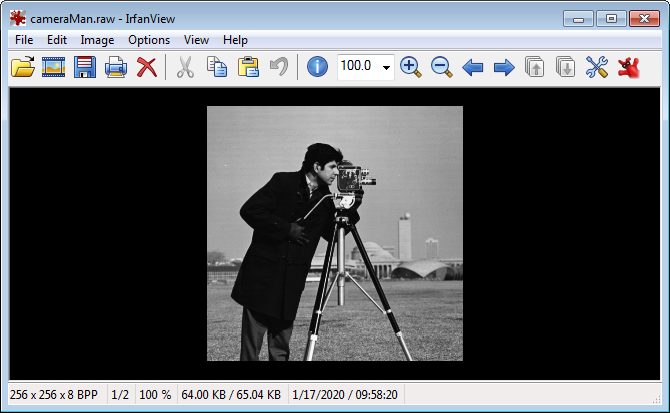
The function Img\_out takes an Image structure (**Figure 1D**). It opens a file (thresholdOutput.raw in this case) and proceeds with a for-loop and fwrite to write the data to the file. The function ends by closing the file.

**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

One key observation from Exercise 1.1 is that the end-effector’s motion is not smooth. Rather it’s a go-stop-go motion. The core reason stems from position, rather than velocity, commands. The program **xl320-line1\_0.nxc** forces the end-effector to visit each way point. This leads to the discrete, instead of a continuous, motion profile. Close examination of the rotateMotorAbsolutely function in **xl320-twoLinkFunctions1\_0a.h** shows:

XL320\_servo(ID\_MOTOR01, theta01InCounts, 200); // motor position at speed 200

// Wait(1500); // Uncomment and change value e.g. 1500 ms if troubleshooting

XL320\_servo(ID\_MOTOR02, theta02InCounts, 200); // motor position at speed 200

Wait(1000); // Uncomment to see impact on communications

PlayTone(TONE\_B3,50);

The Wait(1000)statement creates a 1 second delay. At 57,600 baud, bits are transmitted to through to a XL-320 servo at 17.36 microseconds (or 0.139 milliseconds per byte). There function doesn’t perform error checking or confirms successful transmission of the byte packet. Thus the Wait statement gives some time margin for the XL-320 servos to receive, process, and execute the commanded motion. This time margin can be changed to reduce the go-stop-go phenomena.

Exercises

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

|  |  |  |
| --- | --- | --- |
| numberOfWayPoints | Wait | What path does the end-effector make? How well does it “stay” on the desired line? |
| 15 | 50 |  |
| 7 | 50 |  |
| 5 | 50 |  |
| 1 | 50 |  |
| 15 | 200 |  |
| 7 | 200 |  |
| 5 | 200 |  |
| 1 | 200 |  |

* 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.

**Concept 3:** Velocities versus Positions – On-the-Fly Velocity Commands

Many smart servos have the ability to change velocities “on-the-fly”. This means that the servo’s velocity changes, regardless if it reached the commanded position or not, as soon as it receives a command. One may have observed that in Exercise 2.1. For example, if the Wait time is short (e.g. 50 milliseconds) and the number of way points was small (e.g. 5), the end-effector did not visit all the way points. Rather, the 2-link manipulator traced a curvilinear path. This is because the servos processed the motion command quickly and before it could reach that way point, received the command to the next way point.

Again, close examination of the rotateMotorAbsolutely function in **xl320-twoLinkFunctions1\_0a.h** shows:

XL320\_servo(ID\_MOTOR01, theta01InCounts, 200); // motor position at speed 200

// Wait(1500); // Uncomment and change value e.g. 1500 ms if troubleshooting

XL320\_servo(ID\_MOTOR02, theta02InCounts, 200); // motor position at speed 200

Wait(1000); // Uncomment to see impact on communications

PlayTone(TONE\_B3,50);

The yellow highlight commands each XL-320 at 200 counts/minute. The XL-320 e-manual says that 0.111 RPM per counts/minute. Hence 200 counts/minute is 22.2 RPM. So, in addition to adjusting the Wait statement, one can assign different velocity values in XL320\_servo.

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.