

## Project 03 : Image processing in F#

(doc v1.0)

**Assignment:** F# function library to perform image operations  
**Evaluation:** Gradescope followed by manual execution & review  
**Policy:** Individual work only  
**Complete By:** Thursday, March 30<sup>th</sup> @ 11:59pm CDT

**Late submissions:** 10% penalty for code that is submitted up to 24 hours late (Friday, March 31<sup>st</sup>)  
30% penalty for code that is submitted up to 48 hours late (Saturday, April 1<sup>st</sup>)  
48 hours after due date, code cannot be submitted for credit

### Background

You are going to write a program to perform various operations on images stored in PPM format, such as this lovely image of a piece of cake:



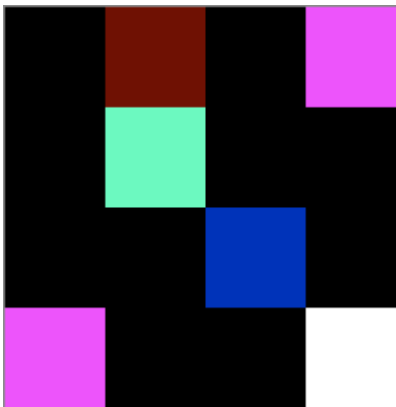
There are many image formats you are no doubt familiar with: JPG, PNG, etc. The advantage of PPM is that the file format is human-readable, so you can open PPM files in a text editor. This makes it easier to write programs that manipulate the images, and it also makes it easier to debug your output — you can simply open the image file in a text editor and “see” what’s wrong. First some background on PPM files, and then the details of the assignment...

## PPM Image Format

The PPM (or Portable Pix Map) image format is encoded in human-readable ASCII text. For those of you who enjoy reading documentation, the formal image specification can be found [here](http://netpbm.sourceforge.net/doc/ppm.html)<sup>1</sup>. Here is a sample ppm file, representing a very small 4x4 image:

```
P3
4 4
255
0 0 0 100 0 0      0 0 0 255 0 255
0 0 0 0 255 175    0 0 0 0 0 0
0 0 0 0 0 0        0 15 175 0 0 0
255 0 255 0 0 0    0 0 0 255 255 255
```

Here is what this image looks like, **magnified 5,000%**. Notice it consists of 16 **pixels**, laid out in 4 rows with 4 pixels per row:



You can think of an image as having two parts, a **header** and a **body**. The **header** consists of information about the image such as width and height, GPS location, time, date, etc.. For PPM images, the header is very simple, and has only 4 entries:

```
P3
4 4
255
```

**P3** is a "magic number". It indicates what type of PPM image this is (full color, ASCII encoding). For this assignment it will always be the string "P3". The next two values, **4 4**, represent the *width* and *height* of the image — or more accurately from a programming perspective, the number of **pixels** in one row is the width and the number of **rows** in the image is the height. The final value, **255**, is the **maximum color depth** for the image. For images in "P3" format, the depth can be any value in the range 0..255, inclusive.

---

<sup>1</sup> <http://netpbm.sourceforge.net/doc/ppm.html>

The **image body** contains the *pixel* data — i.e. the color of each *pixel* in the image. For the image shown above, which is a 4x4 image, we have 4 rows of pixel data:

```
0 0 0 100 0 0      0 0 0 255 0 255
0 0 0 0 255 175    0 0 0 0 0 0
0 0 0 0 0 0        0 15 175 0 0 0
255 0 255 0 0 0    0 0 0 255 255 255
```

Look at this data closely... First, notice the values range from 0 .. maximum color depth (in this case 255). Second, notice that each row contains exactly 12 values, with at least one space between each value. Why 12? Because each row contains 4 **pixels**, but each pixel in PPM format consists of 3 values: the amount of RED, the amount of GREEN, and the amount of BLUE. This is more commonly known as the pixel's **RGB value**. *Black*, the absence of color, has an RGB value of 0 0 0 — the minimum amount of each color. *White*, the presence of all colors, has an RGB value of depth depth depth — the maximum amount of each color. As shown above, notice the first pixel in the 4x4 image is black, and the last pixel is white.

In general a pixel's RGB value is the mix of red, green, and blue needed to make that color. For example, here are some common RGB values, assuming a maximum color depth of 255:

```
Yellow:    255 255 0
Maroon:    128 0 0
Navy Blue: 0 0 128
Purple:    128 0 128
```

You can read more about RGB on the web<sup>2</sup>. We will provide you with 5 PPM images to work with. The image shown above is "tiny4by4.ppm". The images will be made available via replit.com, but there are also available for separate download from [dropbox](#):

- blocks.ppm
- cake.ppm
- square.ppm
- tiny4by4.ppm
- tinyred4by4.ppm

## Viewing PPM Images

PPM files are an uncommon file format, so if you double-click on a ".ppm" image file you will be unable to view it. Here are some options for viewing PPM files, which you'll need to do for testing purposes...

Option #1 is to download a simple JavaScript program for viewing images on your local computer: [ppmReader.zip](#). Download, double-click to open, and extract the file **ppmReader.html** --- save this anywhere on your local computer. When you want to view a PPM image file, download the PPM file to your local computer, double-click on **ppmReader.html**, and select the PPM image file for viewing. This tool is also

---

<sup>2</sup> [http://www.rapidtables.com/web/color/RGB\\_Color.htm](http://www.rapidtables.com/web/color/RGB_Color.htm)

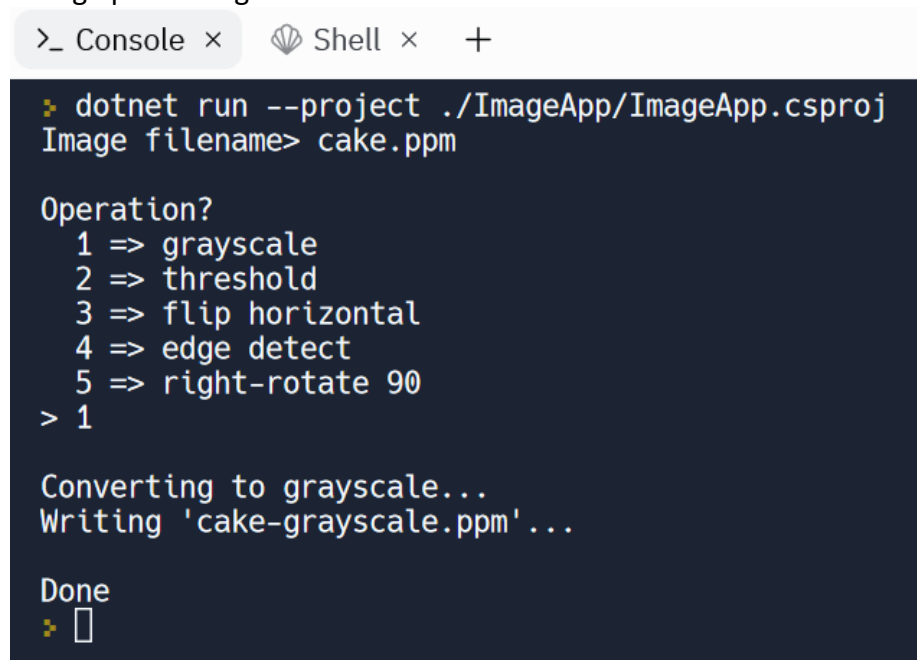
available [online](#).

Option #2: on Windows you can view PPM images using [Irfanview](#)<sup>3</sup>, a free image utility. If the installation fails, note that I had to download the installer and *run as administrator* for it to install properly on Windows: right-click on the installer program and select “run as administrator”.

Option #3: you can “view” PPM files in your local text editor: File menu, Open command, and then browse to the file and open it — you will see lots of integers :-). If you don’t see the PPM files listed in the Open File dialog, try selecting “All files” from the drop-down.

## Assignment

A replit.com project “**Project 03**” has been created with an initial program template (written in C# and F#), along with PPM image files for testing. The provided program implements a simple console-based interface for performing one of five image processing functions:



```
>_ Console x Shell x +
dotnet run --project ./ImageApp/ImageApp.csproj
Image filename> cake.ppm

Operation?
1 => grayscale
2 => threshold
3 => flip horizontal
4 => edge detect
5 => right-rotate 90
> 1

Converting to grayscale...
Writing 'cake-grayscale.ppm'...

Done
> 
```

In the screenshot shown above, the user has entered the name of the cake PPM image file “cake.ppm”, and has selected operation #1 to perform a grayscale conversion. The resulting image is written to the file “cake-grayscale.ppm”.

The console-based interface is written in C#, and can be found in the file “Program.cs”; there is no reason to modify this file (except for debugging purposes). Your assignment is to implement the five image processing functions found in the F# file “Library.fs”: **Grayscale**, **Threshold**, **FlipHorizontal**, **EdgeDetect**, and **RightRotate90**. Here’s the contents of that file. Do not change the API in any way: do not add parameters, do not change their types, etc. We will be grading your F# library against our own test suite, and so your API must match what is given on the next page:

<sup>3</sup> <http://www.irfanview.com/>

```

//
// F# image processing functions.
//
// More details?
//
// Name? School? Date?
//

namespace ImageLibrary

module Operations =
    //
    // all functions must be indented
    //

    //
    // Grayscale:
    //
    // Converts the image into grayscale and returns the
    // resulting image as a list of lists. Pixels in grayscale
    // have the same value for each of the Red, Green and Blue
    // values in the RGB value. Conversion to grayscale is done
    // by using a WEIGHTED AVERAGE calculation. A normal average
    // (adding the three values and dividing by 3) is NOT the best,
    // since the human eye does not perceive the brightness of
    // red, green and blue the same. The human eye perceives
    // green as brighter than red and it perceived red as brighter
    // than blue. Research has shown that the following weighted
    // values should be used when calculating grayscale.
    // - the green value should account for 58.7% of the grayscale amount.
    // - the red value should account for 29.9% of the grayscale amount.
    // - the blue value should account for 11.4% of the grayscale amount.
    //
    // So if the RGB values were (25, 75, 250), the grayscale amount
    // would be 80,  $(25 * 0.299 + 75 * 0.587 + 250 * 0.114 \Rightarrow 80)$ 
    // and then all three RGB values would become 80 or (80, 80, 80).
    // We will use truncation to cast from the floating point result
    // to the integer grayscale value.
    //
    // Returns: updated image.
    //
    let rec Grayscale (width:int)
                      (height:int)
                      (depth:int)
                      (image:(int*int*int) list list) =
        // for now, just return the image back, i.e. do nothing:
        image

    //
    // Threshold
    //
    // Thresholding increases image separation --- dark values
    // become darker and light values become lighter. Given a
    // threshold value in the range  $0 < \text{threshold} < \text{color depth}$ ,
    // each RGB value is compared to see if it's  $> \text{threshold}$ .
    // If so, that RGB value is replaced by the color depth;
    // if not, that RGB value is replaced with 0.

```

```

//
// Example: if threshold is 100 and depth is 255, then given
// a pixel (80, 120, 160), the new pixel is (0, 255, 255).
//
// Returns: updated image.
//
let rec Threshold (width:int)
                  (height:int)
                  (depth:int)
                  (image:(int*int*int) list list)
                  (threshold:int) =
    // for now, just return the image back, i.e. do nothing:
    image

//
// FlipHorizontal:
//
// Flips an image so that what's on the left is now on
// the right, and what's on the right is now on the left.
// That is, the pixel that is on the far left end of the
// row ends up on the far right of the row, and the pixel
// on the far right ends up on the far left. This is
// repeated as you move inwards toward the row's center.
//
// Returns: updated image.
//
let rec FlipHorizontal (width:int)
                      (height:int)
                      (depth:int)
                      (image:(int*int*int) list list) =
    // for now, just return the image back, i.e. do nothing:
    image

//
//
// Edge Detection:
//
// Edge detection is an algorithm used in computer vision to help
// distinguish different objects in a picture or to distinguish an
// object in the foreground of the picture from the background.
//
// Edge Detection replaces each pixel in the original image with
// a black pixel, (0, 0, 0), if the original pixel contains an
// "edge" in the original image. If the original pixel does not
// contain an edge, the pixel is replaced with a white pixel
// (255, 255, 255).
//
// An edge occurs when the color of pixel is "significantly different"
// when compared to the color of two of its neighboring pixels.
// We only compares each pixel in the image with the
// pixel immediately to the right of it and with the pixel
// immediately below it. If either pixel has a color difference
// greater than a given threshold, then it is "significantly
// different" and an edge occurs. Note that the right-most column
// of pixels and the bottom-most column of pixels can not perform
// this calculation so the final image contain one less column

```

```

// and one less row than the original image.
//
// To calculate the "color difference" between two pixels, we
// treat the each pixel as a point on a 3-dimensional grid and
// we calculate the distance between the two points using the
// 3-dimensional extension to the Pythagorean Theorem.
// Distance between (x1, y1, z1) and (x2, y2, z2) is
// sqrt ( (x1-x2)^2 + (y1-y2)^2 + (z1-z2)^2 )
//
// The threshold amount will need to be given, which is an
// integer 0 < threshold < 255. If the color distance between
// the original pixel either of the two neighboring pixels
// is greater than the threshold amount, an edge occurs and
// a black pixel is put in the resulting image at the location
// of the original pixel.
//
// Returns: updated image.
//
let rec EdgeDetect (width:int)
    (height:int)
    (depth:int)
    (image:(int*int*int) list list)
    (threshold:int) =
    // for now, just return the image back, i.e. do nothing:
    image

//
// RotateRight90:
//
// Rotates the image to the right 90 degrees.
//
// Returns: updated image.
//
let rec RotateRight90 (width:int)
    (height:int)
    (depth:int)
    (image:(int*int*int) list list) =
    // for now, just return the image back, i.e. do nothing:
    image

```

## Assignment details

The parameters to the F# functions should be self-explanatory. For example, the **RightRotate90** function takes the image *width*, *height*, *depth*, and the *image data*. The image data is the interesting one... The format is a list of lists of tuples, where each tuple denotes the RGB values for that pixel. For example, on page 2 we presented the file format for the “tiny4by4.ppm” image:

```

P3
4 4
255
0 0 0 100 0 0 0 0 0 255 0 255
0 0 0 0 255 175 0 0 0 0 0 0
0 0 0 0 0 0 0 15 175 0 0 0

```

255 0 255 0 0 0 0 0 0 255 255 255

The image data passed to the F# image functions in this case is a list of 4 lists, one sub-list per row. Each row is a sub-list of tuples (R, G, B), where R, G, and B are color values 0..depth (in this case 0..255):

```
[ [ (0,0,0); (100,0,0); (0,0,0); (255,0,255) ] ;  
  [ (0,0,0); (0,255,175); (0,0,0); (0,0,0) ] ;  
  [ (0,0,0); (0,0,0); (0,15,175); (0,0,0) ] ;  
  [ (255,0,255); (0,0,0); (0,0,0); (255,255,255) ] ]
```

Note each sub-list contains the same number of pixels — 4 in this case. You must work with this format for communication between the console-based front-end and the F# back-end; you cannot change the data structure.

Here is more information about the functions you need to write. Use whatever features you want in F#, except functions that perform direct image manipulations. Strive for efficient solutions using the techniques we have discussed in class; any function that takes over a minute to execute is considered unacceptable and thus wrong. And of course, do *not* use imperative style programming: no mutable variables, no arrays, and no loops.

1. **Grayscale** (width:int) (height:int) (depth:int) (image:(int\*int\*int) list list):

This function converts the image into grayscale and returns the resulting image as a list of lists. Conversion to grayscale is done by calculating a weighted average the RGB values for a pixel, and then replacing them all by that average. The weights used for this calculation are 29.9% for red, 58.7% for green and 11.4% for blue. This is because the human eye perceived green as brighter than red and it perceived red as brighter than blue. So if the RGB values were (25, 75, 250), the average would be 80. Since  $(25 * 0.299) + (75 * 0.587) + (250 * 0.114)$  is 80. Then all three RGB values would become 80 — i.e. (80, 80, 80). Here's the cake in gray:

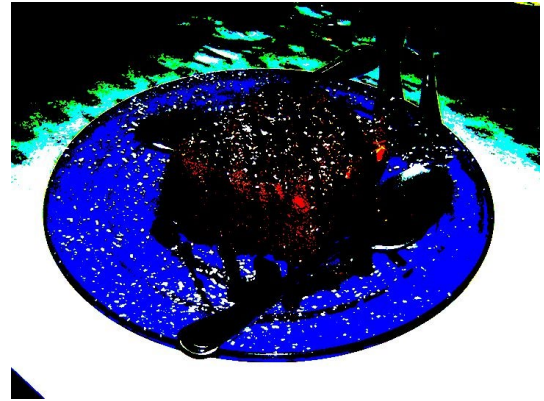


2. **Threshold** (width:int) (height:int) (depth:int) (image:(int\*int\*int) list list) (threshold:int):

Thresholding increases *image separation* --- dark values become darker and light values become lighter. Given a **threshold** value in the range  $0 < \text{threshold} < \text{color depth}$ , any RGB value  $> \text{threshold}$  is set to the color depth (e.g. 255), while any RGB value  $\leq \text{threshold}$  is set to 0. Example: assuming a threshold of 100 and a depth of 255, the pixel (80, 120, 160) is transformed to the pixel (0, 255, 255). Given a grayscale image, after thresholding the image becomes black and white. Another example: given the cake image, the left is the result with a threshold value of 50, and the right with a threshold



value of 200:



3. **FlipHorizontal** (width:int) (height:int) (depth:int) (image:(int\*int\*int) list list):

This function flips an image so that what's on the left is now on the right, and what's on the right is now on the left. That is, the pixel that is on the far left end of the row ends up on the far right of the row, and the pixel on the far right ends up on the far left. This is repeated as you move inwards toward the center of the row; remember to preserve RGB order for each pixel as you flip — you flip pixels, not individual RGB colors. Here's the cake flipped horizontally:



4. **EdgeDetect** (width:int) (height:int) (depth:int) (image:(int\*int\*int) list list) (threshold:int):

Edge detection is an algorithm used in computer vision to help distinguish different objects in a picture or to distinguish an object in the foreground of the picture from the background. Edge Detection replaces each pixel in the original image with a black pixel, (0, 0, 0), if the original pixel contains an "edge" in the original image. If the original pixel does not contain an edge, the pixel is replaced with a white pixel (255, 255, 255).

An edge occurs when the color of pixel is "significantly different" when compared to the color of two of its neighboring pixels. We only compare each pixel in the image with the pixel immediately to the right of it and with the pixel immediately below it. If either pixel has a color difference greater than a given threshold, then it is "significantly different" and an edge occurs. Note that the right-most column of pixels and the bottom-most column of pixels can not perform this calculation so the final image contain one less column and one less row than the original image.

To calculate the "color difference" between two pixels, we treat each pixel as a point on a 3-dimensional grid and we calculate the distance between the two points using the 3-dimensional extension to the Pythagorean Theorem. The distance between  $(x_1, y_1, z_1)$  and  $(x_2, y_2, z_2)$  is:

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

The threshold amount will need to be given, which is an integer  $0 < \text{threshold} < 255$ . If the color distance between the original pixel either of the two neighboring pixels is greater than the threshold amount, an edge occurs and a black pixel is put in the resulting image at the location of the original pixel. Here's is the cake with edge detection with threshold of 50:



5. **RotateRight90** (width:int) (height:int) (depth:int) (image:(int\*int\*int) list list):

This function rotates the image to the right 90 degrees. The image returned by this function will have different dimensions than the original image passed in. Here's the cake rotated right 90 degrees:



## Hints

- FlipHorizontal is probably the easiest and thus the first function to attempt. The built-in List.rev function that reverses a list makes this one pretty simple. So use that one to understand working with a list of lists. No need to modify the tuple for the color values at each pixel.
- RotateRight90 also does not have you modify the tuple for the color values. A built-in function makes this one easy to complete as well. However, determining your own solution will help for some of the other functions.
- Both Grayscale and Threshold require you to access and change the color values in the tuple for each pixel.
- EdgeDetect is the one that is the hardest. The resulting color values at each pixel relies on accessing the color values at two other pixels.

## Electronic Submission to Gradescope Project 03

Before you submit, update the header comment in “Library.fs” with your name, date, school, and more details about the library itself.

When you are ready to submit your program for grading, login to Gradescope and upload your “Library.fs” file. We will test your library functions against our own version of the console-based front-end. You have unlimited submissions, and Gradescope keeps a complete history of all submissions you have made. By default, Gradescope records the score of your last submission, but if that score is lower, you can click on “Submission history”, select an earlier score, and click “Activate” to select it. The activated submission will be the score that gets recorded, and the submission we grade. If you submit on-time and late, we’ll grade the last submission (the late one) unless you activate an earlier submission.

The grade reported by Gradescope will be a tentative one. After the due date, submissions will be manually reviewed to ensure project requirements have been met. Failure to meet a requirement --- e.g. use of mutable variables or loops --- will generally fail the submission with a project score of 0.

## Policy

Unless stated otherwise, all work submitted for grading *\*must\** be done individually. While we encourage you to talk to your peers and learn from them, this interaction must be superficial with regards to all work submitted for grading. This means you *\*cannot\** work in teams, you cannot work side-by-side, you cannot submit someone else’s work (partial or complete) as your own. The University’s policy is available here:

<https://dos.uic.edu/conductforstudents.shtml> .

In particular, note that you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. Academic dishonesty is unacceptable, and penalties range from a letter grade drop to expulsion from the university; cases are handled via the official student conduct process described at <https://dos.uic.edu/conductforstudents.shtml> .