Homework #6

Complete By: Wednesday, October 14th @ 11:59pm
Assignment: completion of following exercise
Policy: Individual work only, late work *is* accepted
Submission: electronic submission via Blackboard

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…

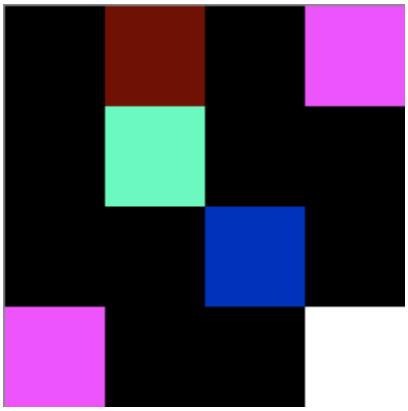
PPM Image Format

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 [1]. Here is a sample ppm file, representing a very small 4x4 image:

[1] http://netpbm.sourceforge.net/doc/ppm.html

```
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 and the number of **rows** in the image, respectively. 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.

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[2]. We will provide you with 5 PPM images to work with. The image shown above is "tiny4by4.ppm". Here are the 5:

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

Viewing PPM Images

There's a simple JavaScript program (.html file) for viewing images on your local computer: under the "Homeworks" link on the course web page, download ppmReader.zip and extract the .html file. Double-click to launch, and select a .PPM file to view. Note that it may chop off the image if it's too big.

On Windows you can view PPM images using Irfanview [3], a free image utility. If the installation fails, note that I had to download the installer and then *run as administrator* for it to install properly on Windows: right-click on setup program and select "run as administrator". Irfanview will also allow you to convert your own images to PPM so you work on your own pictures — keep in mind that you may need to resize your images to be smaller before converting to PPM, otherwise the PPM files become quite large (since they are ASCII, not binary). Save your image in PPM format, and check the option for "ASCII" format;

Another software program is **GIMP**, which runs on a variety of platforms. You can use this program to view PPM images, as well as open images in other formats and save in PPM format (be sure to select ASCII format).

On the Mac, you can download **ToyViewer** from the App store, a free app for image processing, including PPM files. You can use ToyViewer to view the output from your program to see the results. However, I was

[2] http://www.rapidtables.com/web/color/RGB_Color.htm
[3] http://www.irfanview.com/

unable to convert my own images to the proper PPM "P3" format, so if you want to convert your own images, you need to find another utility for the Mac.

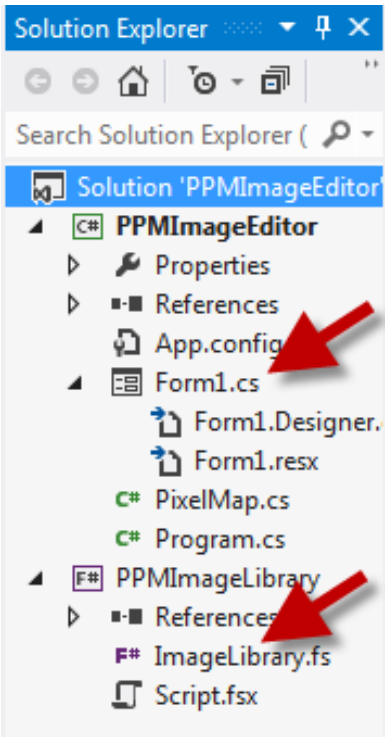
Lastly, keep in mind you can also "view" PPM files in your local text editor: File menu, Open command, and then browse to the file and open it — you will see lots of integers :-) If you don't see the PPM files listed in the Open File dialog, try selecting "All files" from the drop-down.

Getting Started

To make life more interesting (and realistic), we are going to be working with a GUI-based, multi-language application. Here's a snapshot:

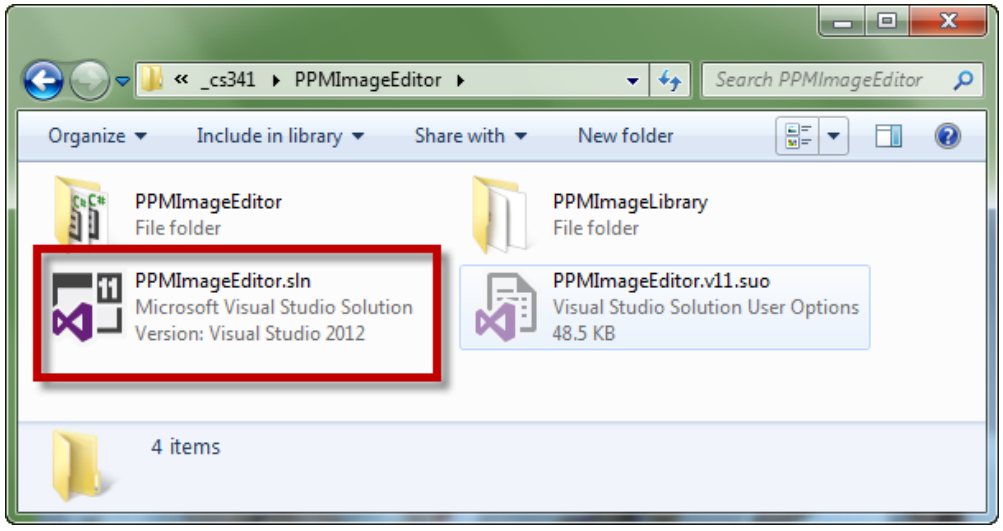
The Visual Studio solution contains 2 parts ("projects"), one representing the GUI front-end written in C#, and the image processing code representing the back-end written in F#:

<< screenshot on next page >>



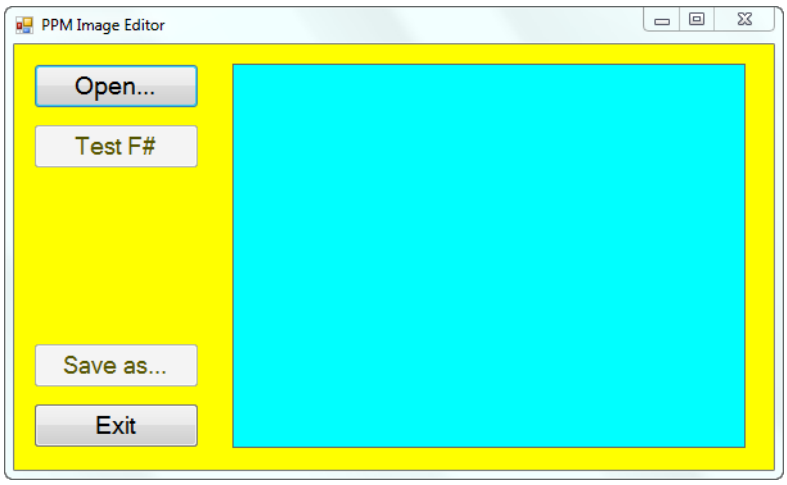
When you want to work on the C# code, you'll open the "Form1.cs" file in the Visual Studio editor. When you want to work on the F# code, you'll open the "ImageLibrary.fs" file.

To get started, browse to the course web page, open the "Homeworks" folder, and drill down into "hw6-files". Download "PPMImageEditor.zip", which is a .zip file of the PPM Image Editor application. After the download, double-click on the .zip file to open, and *extract* the folder by dragging to your desktop. Close and discard the .zip file. Open the folder you extracted, which represents the entire Visual Studio solution to the program. You should see this:

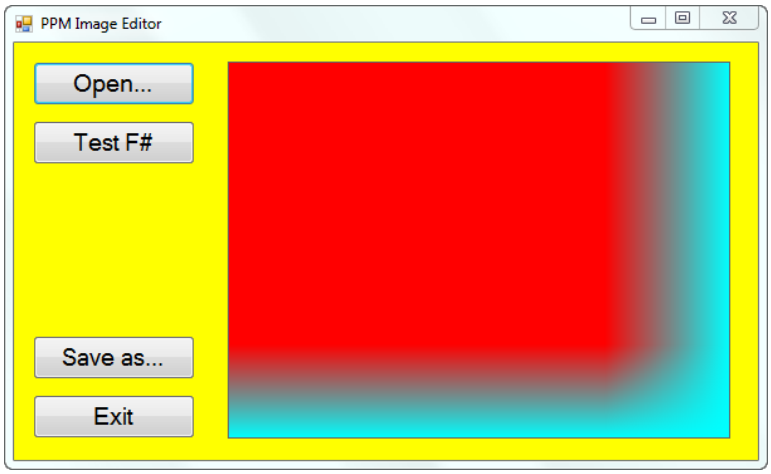


Now open the program in Visual Studio by double-clicking the **Visual Studio Solution** (.sln) file highlighted above. Once in Visual Studio, run with debugging by pressing F5. The program should compile and run

successfully, and you'll see the GUI:



Click the **Open** button — the sample PPM files are already installed in the bin\Debug sub-folder of the GUI project (PPMImageEditor). Selected the simplest file, "tinyred4by4.ppm". This is a tiny image, but the GUI stretches the image to fill the image area, so you'll see something like this:



Finally, click the "Test F#" button, and if you look carefully, you'll notice that the first line of the image is changed to be all white. Open other images such as "cake.ppm", and test F#...

Your Assignment

In Visual Studio, open the "ImageLibrary.fs" file, and you'll see the following code:

```
module PPMImageLibrary

#light

//
// DebugOutput:
```

```
//
// Outputs to console, which appears in the "Output" window pane of
// Visual Studio when you run with debugging (F5).
//
let rec private OutputImage(image:int list list) =
  if image = [] then
    printfn "**END**"
  else
    printfn "%A" image.Head
    OutputImage(image.Tail)

let DebugOutput(width:int, height:int, depth:int, image:int list list) =
  printfn "**HEADER**"
  printfn "W=%A, H=%A, D=%A" width height depth
  printfn "**IMAGE**"
  OutputImage(image)

//
// FirstRowWhite:
//
// An example transformation: replaces the first row of the given image
// with a row of all white pixels.
//
let rec BuildRowOfWhite cols white =
  if cols = 0 then
    []
  else
    // 1 pixel, i.e. RGB value, followed by remaining pixels:
    white :: white :: white :: BuildRowOfWhite (cols-1) white

let FirstRowWhite(width:int, height:int, depth:int, image:int list list) =
  // first row all white :: followed by rest of original image
  (BuildRowOfWhite width depth) :: image.Tail

//
// WriteP3Image:
//
// Writes the given image out to a text file, in "P3" format.  Returns true if
// successful, false if not.
//
let rec WriteP3Image(filepath:string, width:int, height:int, depth:int, image:int list list) =
  //
  // Here's one possible strategy: build a list of strings, then WriteAllLines.
  // Each string appears on a separate line.
  //
  let L = ["Hello"; "World"; "1 2 3"; "10 20 30"]
  System.IO.File.WriteAllLines(filepath, L)
  //
  true  // success
```

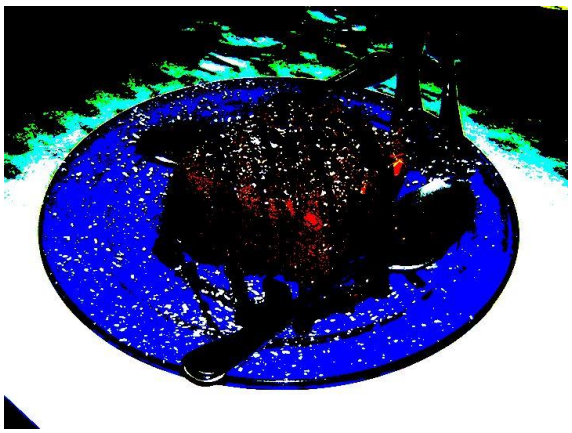
Your assignment is to modify and extend this code by implementing the following *six* functions. Use whatever features you want in F#, except functions that perform direct image manipulations. Strive for

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 for loops. Here are the six functions to implement:

1. **WriteP3Image(…)**: as documented above. Test by making sure you can open and display the new file.

2. **Grayscale**(`width:int, height:int, depth:int,` image:int list list): converts the image into grayscale and returns the resulting image as a list of lists. Conversion to grayscale is done by averaging the RGB values for a pixel, and then replacing them all by that average. So if the RGB values were 25 75 250, the average would be 116, and then all three RGB values would become 116 — i.e. 116 116 116. Here's the cake in gray:

3. **Threshold**(`width:int, height:int, depth:int,` image: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 < threshold < MaxColorDepth, all RGB values > threshold become the max color depth (e.g. white). And all RGB values <= threshold become 0 (e.g. black). Given a grayscale image, after thresholding the image becomes black and white. For 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:



4. **FlipHorizontal**(`width:int, height:int, depth:int,` image:int list list): 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:

5. **Zoom**(`width:int, height:int, depth:int,` image:int list list, factor:int): zooms the image by the given zoom **factor**, which you may assume is an integer 0 < factor < 5. [*Given the file format of PPM images, they grow in size very quickly, so large zoom factors are impractical here.*] There are many zoom algorithms, use the **nearest neighbor** approach where each pixel P in the original image is replaced by a factor*factor block of P pixels. For example, if the zoom factor is 4, then each pixel is replaced by a 4x4 block of 16 identical pixels. The nearest neighbor algorithm is the simplest zoom algorithm, but results in jagged images… The cake image is 720x540; a zoom factor of 4 yields an image that is 2880x2160, but looks exactly the same (only bigger, and more jagged). When viewed in an image viewer, it will appear 4x bigger. However, when viewed in the provided GUI, it will appear exactly the same since by default the image is "Scaled to fit"; switch to "Actual size" and you'll see the effect of the zoom.

6. **RotateRight90**(`width:int, height:int, depth:int,` image:int list list): rotates the image to the right 90 degrees. The list returned by this function will have different dimensions than the original list passed in. Here's the cake rotated right 90 degrees:

<< screenshot on next page >>

In order to test each function, code one of the buttons to call your function; change the button's Text property so it's clear what function this button is performing. Note that the **Save as…** button is already coded to call your **WriteP3Image** function, so use that as an example of how to code the other buttons. In fact, when you are ready to test one of your F# functions, here are the steps to perform in the GUI code to link a button to your function:

1. Expand the PPMImageEditor project in Visual Studio (solution explorer window, upper-right)
2. Double-click on Form1.cs to view the main window in "Design" mode.
3. The "form" opens, which is the application's main window. Double-click on a button...
4. The click event handler is opened for that button in "coding" mode.
5. There is code already written there, just uncomment all the code and call your method. If your method does not appear in IntelliSense, build the solution first.
6. To enable some primitive debug output of the image before/after, uncomment the code in the event handler that calls PPMImageLibrary.DebugOutput. Then run your program with debugging (F5), and view the contents of the Output Debug window (View menu, Output, select Debug from dropdown).

If you want to add your own buttons, drag-drop a new button from the Toolbox, double-click on the button to establish click handler, then code.

The parameters to the F# functions should be self-explanatory. For example, the **WriteP3Image** function takes a string-based filepath as the filename, along with data you need to write to the file: *width*, *height*,

depth, and the *image data*. The image data is the interesting one… The format is a list of lists, where each element is an integer color value. For example, on pages 1-2 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 Library in this case is a list of 4 lists, one sub-list per row:

```
[ [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 color values — 12 in this case. You must work with this format for communication between the GUI front-end and the F# back-end; you cannot change the data structure.

Electronic Submission

The first step is to create a .zip / compressed folder of your *entire* Visual Studio project folder: this should be the folder that you downloaded initially called "PPMImageEditor". Then, using Blackboard, submit this .zip / compressed file under the assignment "HW6". Your F# library source file should include a header comment at the top along the lines of

```
//
// F#-based PPM image library.
//
// <<YOUR NAME HERE>>
// U. of Illinois, Chicago
// CS341, Fall 2015
// Homework 6
//
```

Your program should be readable with proper indentation and naming conventions; commenting is expected where appropriate. You may submit as many times as you want before the due date, but we grade the last version submitted. This implies that if you submit a version before the due date and then another after the due date, we will grade the version submitted after the due date — we will *not* grade both and then give you the better grade. We grade the last one submitted. In general, do not submit after the due date unless you had a non-working program before the due date.

Policy

Late work *is* accepted. You may submit as late as 24 hours after the deadline for a penalty of 25%. After 24 hours, no submissions will be accepted.

All work is to be done individually — group work is not allowed. While I encourage you to talk to your peers and learn from them (e.g. your "iClicker teammates" or Piazza), 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 described here:

http://www.uic.edu/depts/dos/docs/Student%20Disciplinary%20Policy.pdf .

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. It is also considered academic dishonesty if you click someone else's iClicker with the intent of answering for that student, whether for a quiz, exam, or class participation. Academic dishonesty is unacceptable, and penalties range from failure to expulsion from the university; cases are handled via the official student conduct process described at http://www.uic.edu/depts/dos/studentconductprocess.shtml .