# HW #6

**Complete By:** Monday March 3rd @ 9pm
**Policy:** Individual work only, late work not accepted
**Submission:** electronic via Blackboard (see below)

## 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…

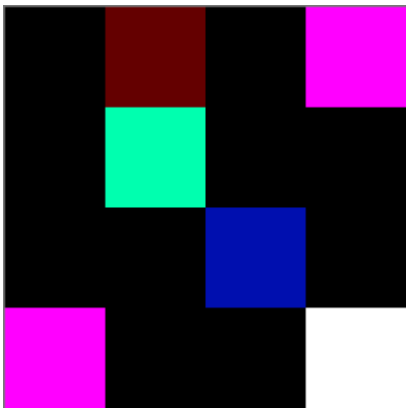
## PPM Image Format

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:

```
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
```

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

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 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** and the number of **rows** in the image, respectively. The final value, **255**, is the **maximum color depth** (value) for the image. For images in "P3" format, this is a 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 columns of **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]. The course web page contains 5 sample PPM images for you to work with. The image shown above is "tiny4by4.ppm":

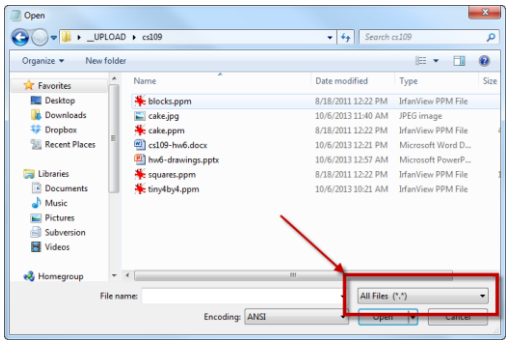
- blocks.ppm
- cake.ppm
- square.ppm
- tiny4by4.ppm
- tinyred4by4.ppm

## Viewing PPM Images

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 7: 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.

On the Mac, I downloaded **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 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.

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 do not see the PPM files listed, in the Open File dialog window change the drop-down list to "view all files" as highlighted below. Then select the PPM file and open it:

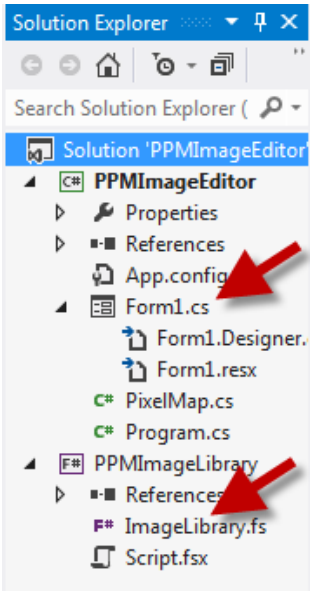



[2] http://www.rapidtables.com/web/color/RGB_Color.htm
[3] http://www.irfanview.com/

## 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:

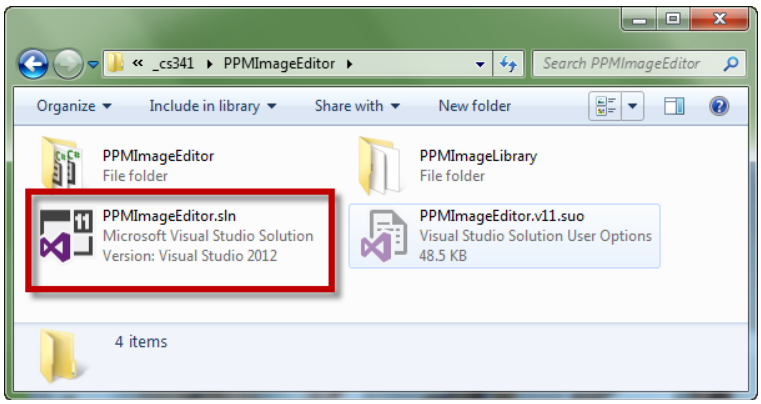

The Visual Studio solution contains 2 parts ("projects"), one representing the GUI front-end written in C#, and the image processing code representing the back-end written in F#:



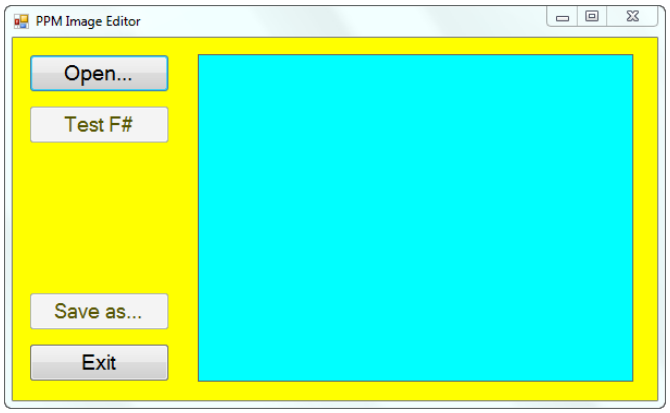

When you want to work on the C# code, you'll open the "Form1.cs" file in the Visual Studio editor. When you want to work on the F# code, you'll open the "ImageLibrary.fs" file.

To get started, browse to the course web page, and download the .zip file of the PPM Image Editor application: "PPMImageEditor.zip". 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

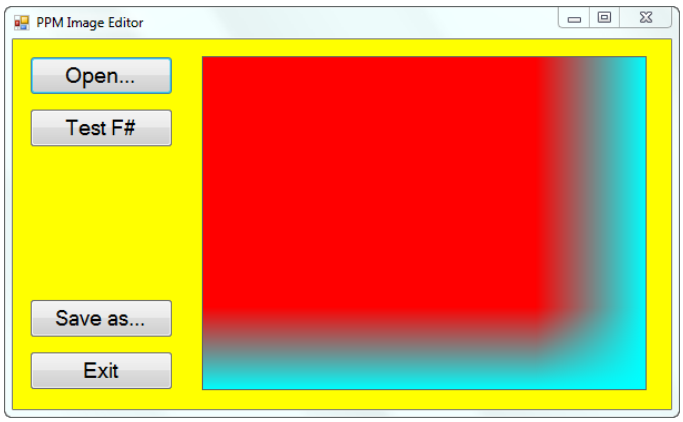
represent the entire Visual Studio solution to the program. You should see this:




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) =
  match image with
  | [ ] -> printfn "**END**"
  |  _  -> 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)

//
// TransformFirstRowWhite:
//
// An example transformation: replaces the first row of the given image
// with a row of all white pixels.
//
let TransformFirstRowWhite(depth:int, image:int list list) =
  let numCols = image.Head.Length  // number of columns in first row
  let AllWhite = [ for i in 1 .. numCols -> depth ]  // white is RGB = depth depth depth
  AllWhite :: image.Tail  // first row all white :: followed by rest of original image

//
// WriteP3Image:
//
// Writes the given image out to a text file, in "P3" format.  Returns true if successful,
// false if not.
//
let WriteP3Image(filepath:string, width:int, height:int, depth:int, image:int list list) =
  //
  //
  // TODO!
  //
  //
  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:

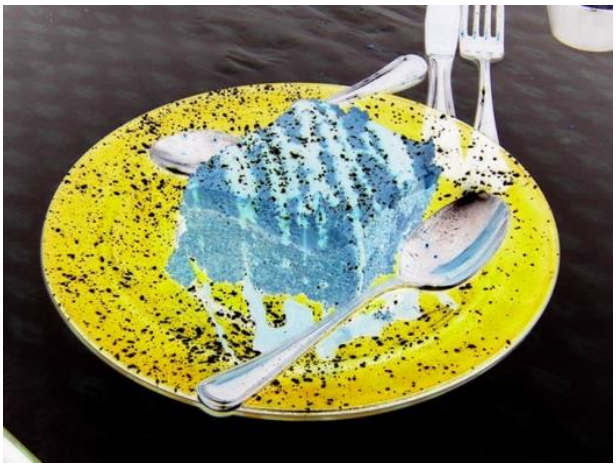
1. **WriteP3Image(…)**: as documented above. Test by making sure you can open and display the new file.

2. **TransformGrayscale(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. **TransformInvert(depth:int, image:int list list)**: To invert a pixel, you invert each of its RGB values. To invert an RGB value, you perform the following computation:

```
newValue = MaxColorDepth – currentValue;
```

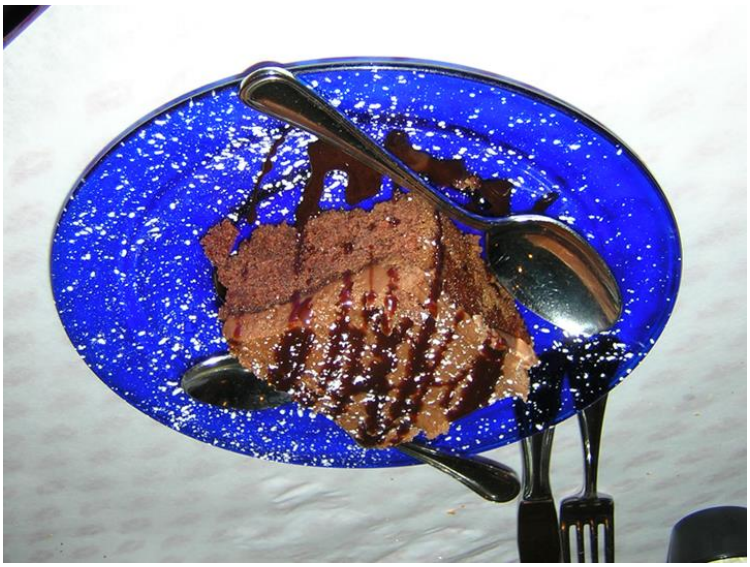
For example, if a pixel has the values 255 128 0 and the image header has a max color depth of 255, then the inverted pixel is 0 127 255. Do not assume the max color depth is 255, use the depth value passed to your function. Here's the cake inverted:



4. **TransformFlipHorizontal(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. **TransformFlipVertical(image:int list list)**: flip the rows at the top of the image with the corresponding rows at the bottom — flip the first row and the last row, then flip the second row with the next to last row, and so on. Here's the cake flipped vertically:



6. Add a **transform function of your choice**. Here are some suggestions if you want a challenge:

   I. Rotate the image 90 degrees.

   II. Encode / decode messages in an image, or obscure the real image. There are a variety of techniques for doing these kinds of things, e.g. http://nifty.stanford.edu/2011/parlante-image-puzzle discusses 3 techniques for obscure images within the RGB values, and

http://nifty.stanford.edu/2009/heeringa-murtagh-secrets-in-images discusses how to encode messages within an existing image. Feel free to add 2 operations if necessary, one to encode/obscure the image, and another to decode the image so you can see if it was encoded/obscured correctly.

III. Look in the menu of your favorite image manipulation program and see what effects are available — blur, soften, etc. Then google and get a feel for how these operations are performed.

In order to test each function, you will need to add a button to the GUI — use the **Test F#** button to help you get started. As you create new buttons, mimic the coding style you see for the Click event handler associated with the Test F# button — i.e. the code for **cmdFS1_Click**. Note that the **Save as…** button is already coded to call your **WriteP3Image** function; that GUI code should be complete and you should be able to use as is.

If you look at the functions provided, the parameters are mostly 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.

## Electronic Submission

The first step is to create a .zip file / 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 file / compressed under the assignment "HW6". We expect your F# code to be commented, including a header comment at the top along the lines of

```
//
// F#-based PPM image library.
//
```

```
// <<YOUR NAME HERE>>
// U. of Illinois, Chicago
// CS341, Spring 2014
// Homework 6
//
```

You may submit as many times as you want before the due date, but we grade the last version submitted.

## Policy

Late work is not accepted. All work is to be done individually — group work is not allowed. Academic dishonesty is unacceptable, and all parties involved will be immediately subject to the official academic integrity review process. The University's policy is quite clear, and can be read here: http://www.uic.edu/depts/dos/studentconduct.html. In particular, note that you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance. 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.