CSE 410
Programming Project 3

Assigned:	Wednesday, October 24, 2001
Due:	Wednesday, October 31, 2001 before class

Introduction

In this project, you will implement a simple image-processing program. The program reads an image file in BMP format, performs a few simple operations on the pixels in the image, and writes the modified image back out to a new file.

All of the main program and supporting procedures have been written and supplied to you. Your task is to write the filter procedures to process the data.

The primary challenge in writing these filters is figuring out how to index through memory in an organized fashion so that you can read an image byte from the correct location, adjust it if necessary, then write it out to another location. The adjustments that you make to the bytes are generally quite simple, you just need to be reading and writing the correct locations.

UNIX note

This project requires the use of the extended version of SPIM that can do file reads and writes. So far as I know, this version has not been rehosted by anyone to UNIX, and so for this project everyone will have to use the PC version.

BMP Format

This format is a commonly used way of storing images on Windows computers. It is relatively simple, and the images we will use have every pixel in the image stored in the file without any compression algorithm, so the pixel data is simple to find.

A pixel is a single dot on the screen. The color value of a pixel is given by three bytes, one for the Blue component of the color, one for the Green component of the color, and one for the Red component of the color. Since these are byte values, this means that the color intensity is specified by three values, each of which can range between 0 and 255_{10}.

Each image file contains an information header and an array of bytes that defines the pixel values. One of the procedures given to you reads the header from the file and stores it in memory. Your filter procedures are given the address of this header as one of their calling arguments.

There are only three values in the header that you will need to use in your code. The offsets and the content of the word at each offset are as follows.

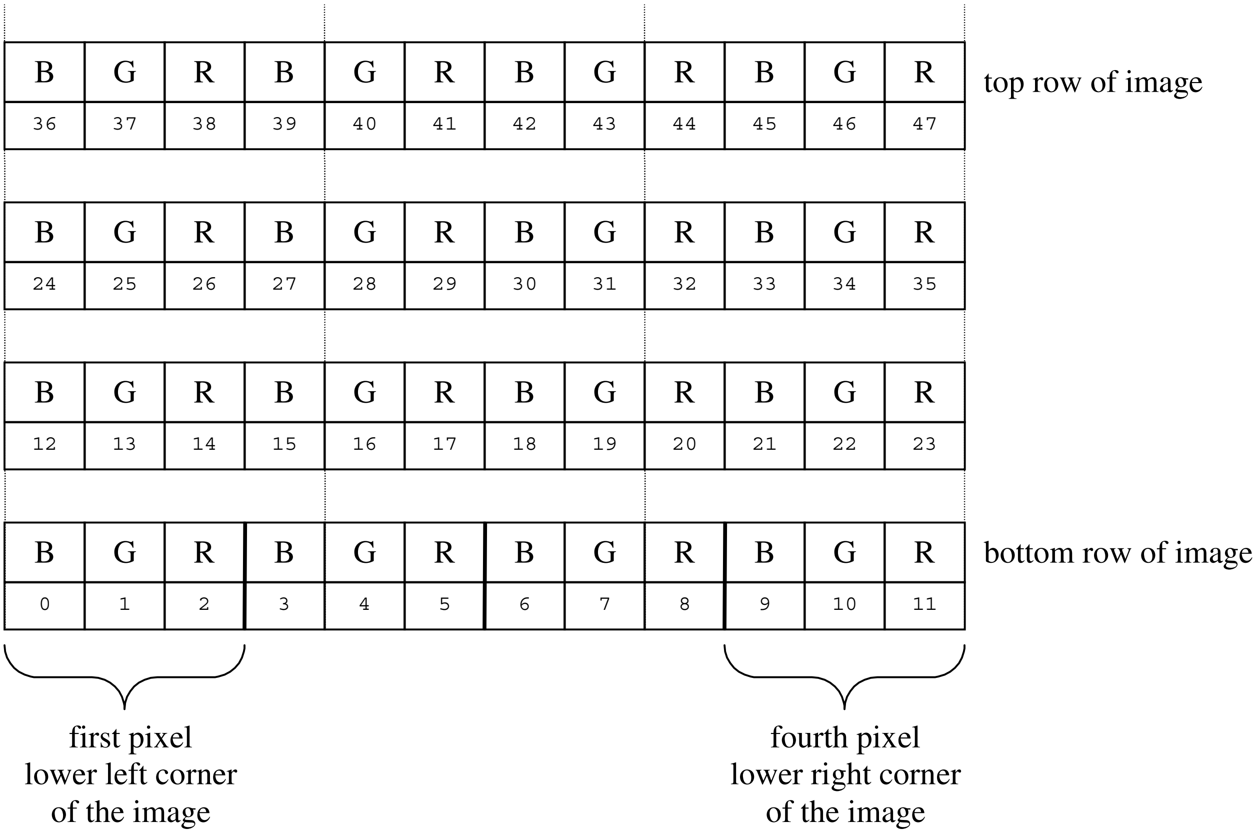
18($a1)	biWidth	number of pixels per row (ie the number columns in the image)
22($a1)	biHeight	height of image (ie the number of rows in the image)
34($a1)	biSizeImage	total number of image bytes (size is >= width*height*3)

From these three values, you can calculate the various offsets and pointers you need in order to step through the values of the image data. Note that ***these offsets are not word aligned***, and so you need to use "ulw" to read them into a register, instead of the more common "lw".

The image data is stored in row order, starting with the bottom row of the image. Three bytes for each pixel are stored in Blue-Green-Red order. If the number of bytes for one row of pixels does not fit exactly into full words, then the row is padded out to end on a word boundary. Thus, the data for every row starts on a word boundary. It is very important that you understand how this affects the storage layout, because this must be correct in your code in order to index through the image data correctly.

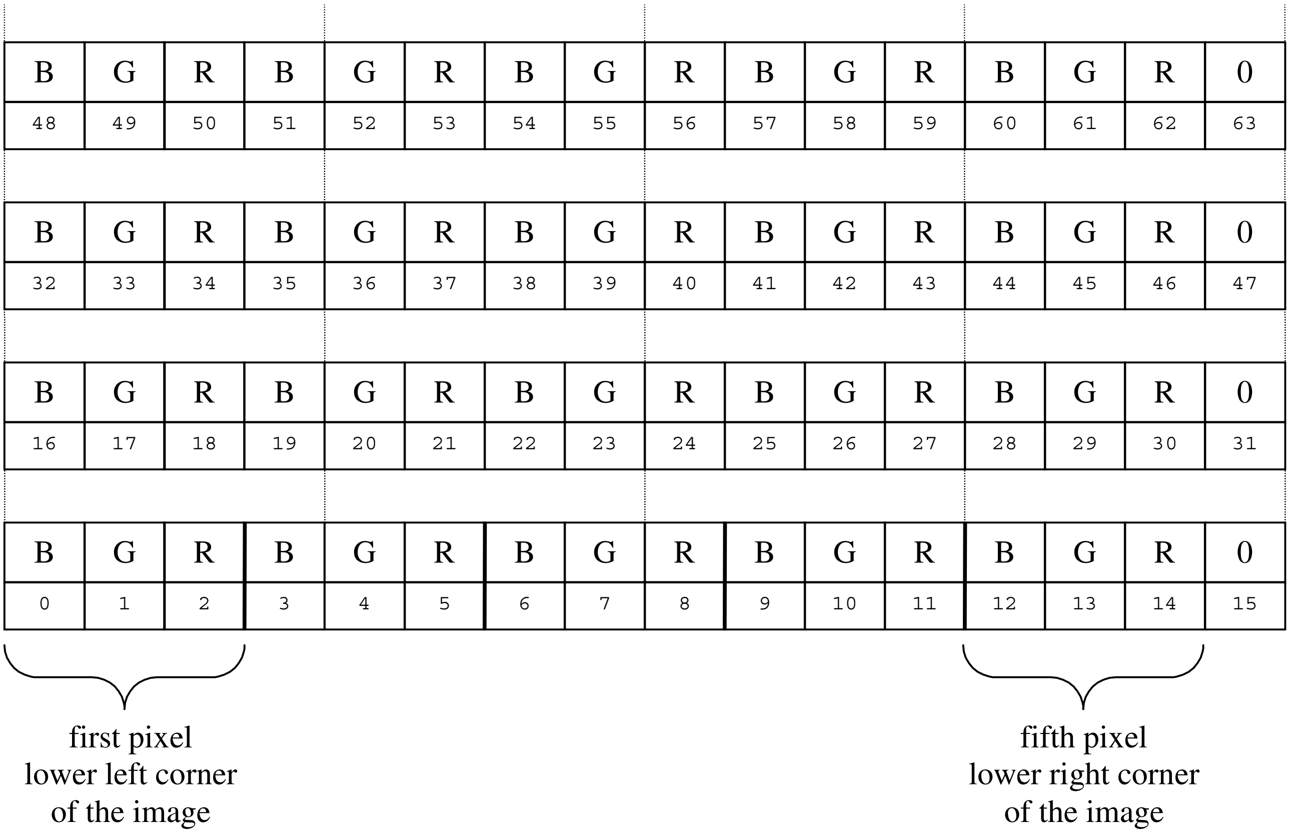
Imagine that we have two very small images: one of them is 4 pixels wide by 4 pixels high, the other is 5 pixels wide by 4 pixels high. Then the data for these images will appear in memory as follows.

Width = 4, Height = 4, SizeImage = 48. The rows of this image fit exactly on word boundaries and so there is no padding at the end of the rows.



B	G	R	B	G	R	B	G	R	B	G	R	top row of image
36	37	38	39	40	41	42	43	44	45	46	47	
B	G	R	B	G	R	B	G	R	B	G	R	
24	25	26	27	28	29	30	31	32	33	34	35	
B	G	R	B	G	R	B	G	R	B	G	R	
12	13	14	15	16	17	18	19	20	21	22	23	
B	G	R	B	G	R	B	G	R	B	G	R	bottom row of image
0	1	2	3	4	5	6	7	8	9	10	11	

Width = 5, Height = 4, SizeImage = 64. The rows of this image do not fit exactly on word boundaries and so there is padding at the end of the rows.



B	G	R	B	G	R	B	G	R	B	G	R	B	G	R	0
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
B	G	R	B	G	R	B	G	R	B	G	R	B	G	R	0
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
B	G	R	B	G	R	B	G	R	B	G	R	B	G	R	0
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
B	G	R	B	G	R	B	G	R	B	G	R	B	G	R	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The Program

The program "filterBMP" reads a 24-bit BMP file in to memory, fiddles with it, and writes it back out in a new file. This program expects to be given at least two arguments on the command line.

> arg 1: input file name root. The filename extension is assumed to be .bmp (and should not be included).
>
> arg 2: filter request number, a positive integer 0..n
>
> arg 3: One or more additional filters can be specified after the first one. They will be applied sequentially.

The output file name is built up from the input file name root, the list of filter numbers and the extension ".bmp". If the file already exists, it will be overwritten.

Typical filters include the following:

filter	function
0	copy from input to output with no change
1	flip image top to bottom
2	mirror image from right to left
3	whatever you define ...

So a simple command line for this program might include the following arguments.

```
test128 1 2
```

Given this command, the program would read file "test128.bmp", call filter procedure 1, then filter procedure 2, and then write the resulting image out to the file "test128-1-1.bmp".

Writing and Installing a Filter Procedure

The copy filter is included in the skeleton. Review it for information about the calling arguments for your filter procedures.

When you add a new filter procedure to the program, you need to update the jump table in procedure dispatchFilterRequest. Add your filter procedure to the list of names, and update the number of values in the table.

```
filterJumpTable:                  # table of filter addresses
     .word  copyFilter,flipFilter,mirrorFilter
filterJumpTableSize:              # number of addresses in above table
     .word 3
```

For this project, you will write some of the other filter procedures as described below. See the grading section for a discussion of how many to write (you don't have to write them all).

- flip

This filter flips the image top to bottom. Look at the drawings on the previous pages. You can see that you have to copy the bottom row of the input image to the top row of the output image, and then proceed row by row, incrementing the source row and decrementing the destination row until you have copied the whole image.

- mirror

This filter reflects the image horizontally. Again refer to the drawings of memory. You can see that you have to copy the first pixel in a row of the input to the last pixel position in the same row of the output, then copy each pixel in turn, incrementing the source position and decrementing the destination position. Do this for every row and you are done.

- select-red, select-green, select-blue

The select filters copy the value for one of the color planes only, and set the byte values for the other two planes to 0 in every pixel. This has the effect of separating out one color of the image, the red, the green, or the blue. In order to implement this, you cycle through all the pixels in all the rows. For each pixel, copy the byte value you want from input to output, and set the other two to 0 in the output.

- brighter

The three numbers that make up each pixel control image brightness. You can increase the brightness by adding 1 or some other value to each existing value from the input image, and storing the new value in the output image. It is important that you "clamp" these values at 255. If the value in the existing image is already 255, then adding 1 to that will make 256. When you store that as a byte value, it will get trimmed to 8 bits, which will be zero. This means that the image gets brighter and brighter until suddenly parts of it go black. Instead, you should check and not increment values that are already at 255.

- darker

The same as brighter, only subtract instead of add. Again, clamp at 0 so that the image doesn't suddenly develop bright spots where it has gone negative and wrapped around.

- threshold

Interesting effects can be gained by setting a threshold for each color, and then pushing values to one extreme or the other depending on that threshold. For example, if the red threshold is 100, then all pixels in the source image that have 100 or less for the red value are set to 0 in the output image. All pixels that have 101 or more for the red value are set to 255. The threshold values can be hard coded in your filter procedure.

- Convolution filters

There are many types of image filtering that can be done, as you know if you've ever used Photoshop, the Gimp, or other image processing programs. You can implement blur filters, sharpen filters, edge enhancement, and so on, all with relatively simple addressing and manipulation of small square areas of pixels. If you are interested in implementing any of these, let me (DWJ) know and we can talk about how to get started.

Grading

The grading for this project is as follows.

You must implement Flip, Mirror, and one other filter (select-red, select-green, select-blue, brighter, darker, or threshold) described above.

Flip	2 points
Mirror	3 points
Another filter	2 points
Questions:	3 points
Total:	10 points * 5 = 50 points for the project

Extra credit: Implement another filter outside of the set you implemented above. In other words, if you implemented select-red, you could implement brighter/darker or threshold for extra credit. However, implementing select-blue would not get extra credit.

add 2 points for select-r/g/b
add 2 points for brighter/darker
add 2 points for threshold
add 6 points for any convolution filter

Up to 10 points extra, making a maximum possible of 60 points.

References

More detailed explanations of the BMP file format. Note that there is a lot of potential complexity that we avoid because we only use uncompressed 24-bit images.

http://www.whisqu.se/per/docs/graphics52.htm
http://web.usxchange.net/elmo/bmp.htm
http://www.daubnet.com/formats/BMP.html