CS(4/5)10 homework 1: Image Editing

Due: 4/13/20

We work with images all the time, but surprisingly few computer scientists know how images really work. Well, we going to fix that. We're learning how to read and manipulate images. How? by making instagram filters. How do you do fellow kids?

Ok, dated jokes aside. If we want to work with images, we need to know how to read them. We're only going to work with bitmaps for this assignment. Trust me, this is enough for now.

Requirements for base assignment:

- Read and write .bmp images by overloading the `>>` and `<<` operators.
- add the cell shaded, gray-scale, pixelate, and blur filters

Requirements for advanced assignment:

- Scale an image by 1/2x or 2x
- rotate an image by 90,180, or 270 degrees
- flip an image vertically, horizontally, or over the two diagonals

There are conceptually two parts to this assignment. First you need to read and write bitmaps. We'll do this by overloading the stream operators `operator>>` and `operator<<`.

There is a comprehensive explanation of the file format here `https://en.wikipedia.org/wiki/BMP_file_format`. Fortunately we don't need all of this complexity. There are several different types of bitmaps determined by the bitmap file header and the compression type. We will only be looking at 2. The first type is a windows bitmap with 24 bit color and no compression and the second type is a windoew bitmap with 32-bit color and no compression. It is possible to store a compressed file, but we won't worry about that for now.

The Header

Now we need the structure of a Bitmap. A bitmap file has two parts. The header, and the data. The header is the most difficult, so we'll start there. The header tells us all of the information we need to read and write bitmaps. The format is given by the following table. There are technically two headers, but we'll combine them for simplicity.

Offset hex	Offset dec	Size	Purpose
00	0	2 bytes	Identifies the type of bitmap.
02	2	4 bytes	the size of the BMP file in bytes
06	6	4 bytes	garbage
0A	10	4 bytes	the offset to the start of the data
0E	14	4 bytes	the size of the second header
12	18	4 bytes	width in pixels (signed)
16	22	4 bytes	height in pixels (signed)
1A	26	2 bytes	number of color planes
1C	28	2 bytes	the color depth of the image
1E	30	4 bytes	the compression method being used
22	34	4 bytes	the size of the raw bitmap data.
26	38	4 bytes	the horizontal resolution of the image
2A	42	4 bytes	the vertical resolution of the image
2E	46	4 bytes	colors in color palate
32	50	4 bytes	the number of important colors used
36	54	4 bytes	Red mask
3A	58	4 bytes	Green mask
3E	62	4 bytes	Blue mask
42	66	4 bytes	Alpha mask
4A	74	68 bytes	Color space information (we can ignore this)

Notes:
The type of bitmap (field 1) will always be "BM" (or 0x424D).
The size of the second header will always be 40 bytes.
The number of color planes must be 1. It's an error if it isn't.
The color depth of the image is either 24 (RGB) or 32 (RGBA).
The compression mode will always either be 0 or 3.
The vertical and horizontal resolutions are used for printers (dots per meter).
They will always be 2835.
We aren't using a color palate, so the number of colors, and number of important colors will both be 0.

The first two boxes are the two headers I talked about. These will be included

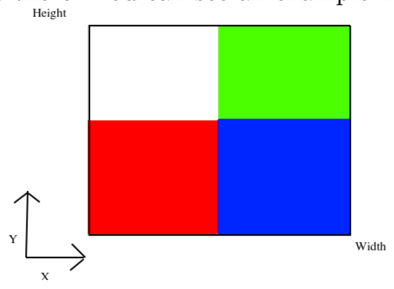
in every bitmap. The third box is only included if the compression mode is 3. If the compression mode is 3, then there will be 4 mask fields. A mask tells us the position of the red green and blue components.
Example:

```
36: 0000 00ff
3A: 0000 ff00
3E: 00ff 0000
42: ff00 0000
```

The masks are always in the order red, green, blue, alpha. So a pixel with the value `ff0f 100c` has a red of `0x0c`, a green of `0x10`, and a blue of `0x0F`, and an alpha of `0xFF`,. The next 68 bytes are not used in our file format, so they can safely be skipped.

The Data

The data is actually pretty straightforward. An image is a two dimensional array of pixels. Each pixel has a red, green, blue and alpha component. We won't be doing anything with the alpha component, but you still need to know it's there. You can see an example in the poorly drawn picture below.



In a 32-bit Image (the most common) Each pixel is represented by a four byte unsigned number. The four bytes represent the alpha, red, green, and blue components of the pixel. The position of the pixel is given by the formula $y \cdot Width + x$.

So the green pixel at position (1,1) in an image of width 2 would be the 4th 4 byte int in the array, and would have the value 0xFF00FF00. The alpha value is always 0xFF.

If we are using 24 bit color, then each row will start on a four byte boundary. Since 24 bits is 3 bytes, then means that if the width of the image isn't a multiple of 4, there will be padding bytes. For example, if the image has a width of 7, then a row will take 21 bytes, so there will be 3 padding bytes on every row.

The Filters

Ok, Now that we can read and write bitmaps we can start working on filters. This part is actually much *much* easier.

The first filter is cell shading. The idea here is really simple. For a 24 or 32 bit image each component can be 0-255. We want to restrict this to 3 possible value (0, 128, 255). Round each component to the nearest value. So a pixel with 23 red, 130 green, and 200 blue will become (0,128,255).

The second filter is gray-scale. We get a gray color when red, green and blue all have the same value. To convert an image to gray-scale we average the red, green and blue component for each pixel.

The third filter is pixelate. Divide your image into 16x16 blocks. For each block compute the average color, and make the entire block that color.

The final filter is Gaussian blurring. This is a little more involved. For each pixel take the 5x5 block surrounding it. This block will contain 25 pixels. For each component color, multiply all of the values by the Gaussian matrix below and add the values together.

$$1/256 \begin{pmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{pmatrix}$$

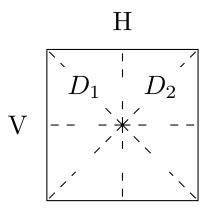
Note: I don't mean multiply the matrices, I mean multiply the top left pixel by 1/256, and the middle pixel by 36/256.
Note 2: make sure the you get out is between 0 and 255.

For a video walking through the structure of a bitmap, along with detailed implementation instructions and code see
`https://www.youtube.com/watch?v=dQw4w9WgXcQ`.

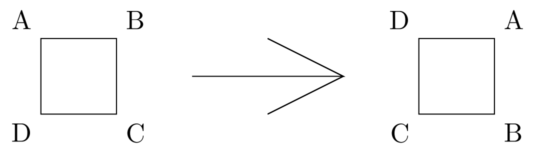
Advanced: image transforms

The final part of this assignment is to transform the images. These transformations aren't any harder, but you will change some of the header data for the image, so you need to keep track of that.

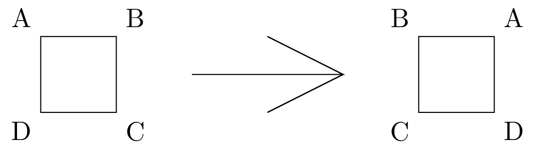
All of these transformations should be pretty self explanatory. To scale an image by 1/2, just remove every other row and column. To scale an image by 2, duplicate every row and column so each pixel is a 2x2 block. We are rotating images by multiples of 90 degrees. The following diagram gives the lines we are flipping over.



Example: Rotate 90 degrees



Example: flip horizontally



Example: flip over D1