PhotoChop

Background

Images

We can think of an image as a grid of squares called pixels. Each pixel has a color value that can be represented as a mix of red, green and blue. If we allow each of these three color "channels" to have a value of 0-255 or 8 bits of storage, the color of one pixel can be represented with 24 total bits (allowing for 2^{24} or over 16.7million different colors).

If all three colors have the same value we get a shade of gray, ranging from 0/0/0 (black to 255/255/255 (white). If the three color values are different, we will get a shade determined by the three values. The table below shows some examples.

	0	1	2	3	4
0	**Red: 255** **Green: 255** **Blue: 255**	**Red: 0** **Green: 255** **Blue: 0**	**Red: 0** **Green: 0** **Blue: 255**	**Red: 255** **Green: 0** **Blue: 0**	**…**
1	**Red: 255** **Green: 255** **Blue: 0**	**Red: 215** **Green: 215** **Blue: 215**	**Red: 200** **Green: 150** **Blue: 255**	**Red: 0** **Green: 0** **Blue: 0**	**…**
2	**…**	**…**	**…**	**…**	**…**

Code

Open the PhotoChop.pro project file in QTCreator to build/run the project.

ImageIOLib.h/.cpp are blackbox code – you do not have to worry about how they work at all

Image.h declares our data types and some constants for working with images

- **byte** is typedefed as a new name for **unsigned char** – the built in type that can store 0-255
  ```
  typedef unsigned char byte;
  ```

- A Pixel is a struct that holds three of those values:
  ```
  struct Pixel {
      byte red;
      byte green;
      byte blue;
  };
  ```

- An Image is a struct that consists of a 2-dimensional array of Pixels.
  ```
  struct Image {
      Pixel data[IMG_HEIGHT][IMG_WIDTH];
  };
  ```

  It uses dimensions based on these constants:
  ```
  const int IMG_HEIGHT = 128;
  const int IMG_WIDTH = 128;
  ```

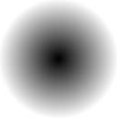
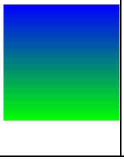
main.cpp has code to create some images, as well as to load an image and modify it. Use the existing code as a model for how to write your own functions that do similar work. Note that you do not have to understand the code behind displayImage( ). Just use it to show the results of any work that you do by passing your image, a string and the **mainLayout** to it.

Assignment

Make **two** or more functions that create or modify an existing image using the PhotoChop code. You must make your functions from different categories listed under Function Ideas sections below (you can't do two "Create New Image" functions). Feel free to invent your own ideas or modify the existing ones.

Function Ideas

Create New Image *– use makeAqua / make Gradient as samples*

Box Write a function that takes in parameters for starting row, col and width and height. It then uses those values to draw a white box (or whatever color you like) on a black background. 	***Circle Gradient*** Calculate each pixels RGB values based on the distance from the center of the image. Set each pixel to a gray value (same RGBs) based on pixel's distance from center. 
Color Gradient Blend from one color to another. 	***Snow*** For each pixel in a new image, pick a random number from 0-255 and use that for all three color values.

Basic Image Modifying *– use RedFilter as a model to get started*

OnlyRed Set the blue and green value for each picture to be 0. 	***Invert*** Set each color to be (255 – the original value).
GrayscaleRed Set the blue and green values for each pixel equal to their red value. *(Or set red and blue = to green OR set all three to the average of their values; each of those recipees produces a different grayscale image.)* 	***Darken*** *or* ***Lighten*** Multiply the red, green and blue values each by a value. A value between 0 and 1 will darken the image; a value > 1 will brighten the image.
BlueScreen Identify any pixel where the blue value is higher than the red value. Turn those pixels to white (255,255,255). You should be able to clear most of the background.	***Noise*** For each color of each pixel, pick a random number from -20 to 20 and add it to the color value.
CropCenter Make a black or dark border around the edge of the image, only leaving the middle visible. 	

Moving Pixels *– use rotate as a model to get started*

Flip vertical or horizontal Flip the image in one dimension or the other. (left-right or up-down).	***Rotate left*** Make a rotation that goes the other direction.

Continues…

Combining Pixels *– use blurFilter as a model to get started*

Sharpen

Multiply the current pixel's RGB values by 5 and neighboring values by -1. Add them all up – this is the new value.

	-1*	
-1*	5*	-1*
	-1*	

Using -2 and 9 makes for a stronger effect. Just make sure the values for the 5 cells add to 1.

Subtracting the neighboring pixels will enhance any color differences, making color transitions look more abrupt and thus "sharper"

FindEdges

If you do a sharpen filter but make the current pixel multiplier value 4 instead of 5, (so that the 5 cells values add to 0) it will do a find edges type filter that shows solid areas of color as black and areas with sharp changes as bright.

8-BitLook

Copy pixels that are on an even row and column into their "odd numbered" neighbors - note odd numbered rows and columns from original are not used.

A	B	C	D
D	E	F	G
H	I	J	K
L	M	N	O

Becomes:

A	A	C	C
A	A	C	C
H	H	J	J
H	H	J	J

Stretch

Make each pixel take up 4x more space. Unlike 8-Bit, this one should not skip rows/columns from the original image. However, because the image must stay the same size, you will only be able to stretch the upper left quadrant of the image.

A	B	C	D
D	E	F	G
H	I	J	K
L	M	N	O

Becomes:

A	A	B	B
A	A	B	B
D	D	E	E
D	D	E	E

Hint: Make sure you don't go too far in the source image or you will end up writing off the edge of the Image's array. That data will "wrap" to the next row and produce weird results.

Multiple Files –

You can open multiple files by making multiple Image structs and doing multiple readImage calls.

Merge

Read in two separate images. Write a function that averages the values of corresponding pixels to create a new image. Or use the top half of one and the bottom of anther. Or alternate lines. Or blend so that you use 100% of one image at the top, 100% of the other at the bottom and gradually shift the weights…

Secret Message

I hid a secret message in crabMessage.bmp. Compare the pixels in it to the pixels in crab.bmp. If corresponding pixels have different green values, turn that pixel white. Otherwise, turn it black.

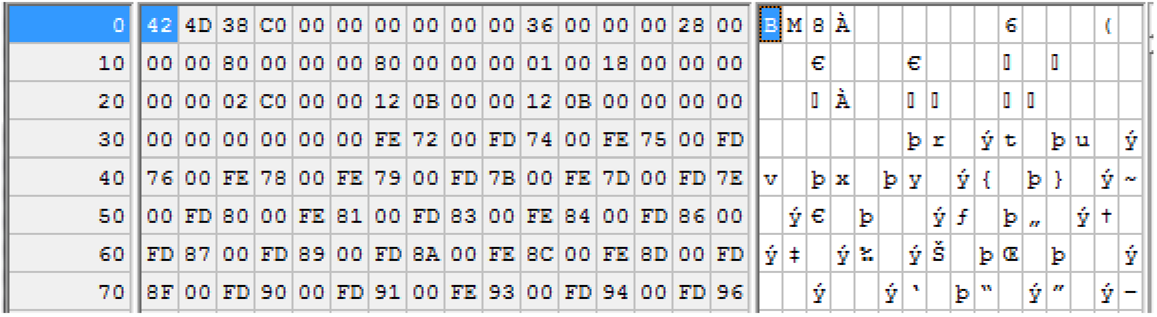
http://en.wikipedia.org/wiki/Steganography

Appendix: (Optional extra info)

Binary Files - Background

Data files are often stored in binary format. It is much more compact to store the number 200000 as a 32-bit integer instead of as 6 characters (each of which take 16-bits to store). It is also faster to read in those raw bits instead of reading strings and converting those to integers.

To read the files humans usually use a hex editor - which shows each byte of the file as a character:



```
 0 | 42 4D 38 C0 00 00 00 00 00 00 36 00 00 00 28 00 | BM8À      6   ( 
10 | 00 00 80 00 00 00 80 00 00 00 01 00 18 00 00 00 |   €   €   ▯ ▯   
20 | 00 00 02 C0 00 00 12 0B 00 00 12 0B 00 00 00 00 |   ▯À  ▯▯  ▯▯    
30 | 00 00 00 00 00 00 FE 72 00 FD 74 00 FE 75 00 FD |       þr ýt þu ý
40 | 76 00 FE 78 00 FE 79 00 FD 7B 00 FE 7D 00 FD 7E | v þx þy ý{ þ} ý~
50 | 00 FD 80 00 FE 81 00 FD 83 00 FE 84 00 FD 86 00 |  ý€ þ  ýƒ þ„ ý† 
60 | FD 87 00 FD 89 00 FD 8A 00 FE 8C 00 FE 8D 00 FD | ý‡ ý‰ ýŠ þŒ þ  ý
70 | 8F 00 FD 90 00 FD 91 00 FE 93 00 FD 94 00 FD 96 |   ý  ý‘ þ“ ý” ý–
```

File formats specify how a particular kind of file is to be written - how to interpret the 1's and 0's. Below is a simplified description of the file format for .bmp (bitmap) image files. *(The full format for .bmp image files can be found at:* *http://en.wikipedia.org/wiki/BMP_file_format* *)*

Byte Address	Description	Sample value from above screen shot
0	Magic Number - verifies really is bmp data - always 66 (42 hex)	42 hex (66 in decimal) - represents character 'B'
1	Magic Number - verifies really is bmp data - always 77 (4d hex)	4d hex (77 in decimal) - represents character 'M'
10	Byte address image data starts	Look in column 10 in the first row. The value is 36hex or 54 in decimal. The image data starts at byte #54.
18	Width of bitmap	Find the 19th byte (address 18) - it is the third one in the second row. The value of 80 means the width is 80 in hex or 128 in decimal.
22	Height of bitmap	The height (also 80 hex/128 decimal) can be seen in byte #22 (halfway across second row).

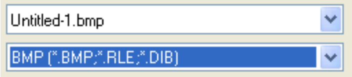
Notes: in the full file format, there are other pieces of information that are stored at other addresses - this is just a sample of what is there. Also, the width and height are stored as a 32-bit integers (not just as one byte).

The ImageIOLib functions do the work of parsing the file and reading the image part of the data into an array.

Your own pictures:

Open the image you want to use in paint / photoshop, etc….

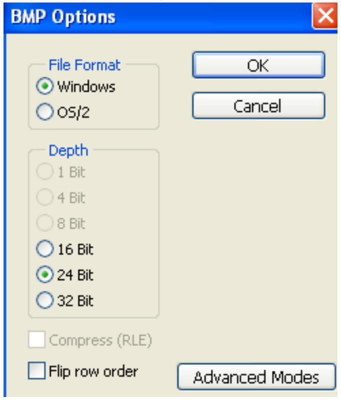
Save as bmp:



Untitled-1.bmp

BMP (*.BMP;*.RLE;*.DIB)

If you have options, use 24 bit color.



BMP Options

File Format
● Windows
○ OS/2

Depth
○ 1 Bit
○ 4 Bit
○ 8 Bit
○ 16 Bit
● 24 Bit
○ 32 Bit

☐ Compress (RLE)
☐ Flip row order

Make sure to modify the constants in ImageIOLib.h to match the new sizes.