Cameron Chalmers

C++ ‘Performance Assignment - Image Processing’ Write-up

The task at hand was to develop a C++ application that processed 12 images. Each image had to be brightened, rotated, greyscaled and downsized. In this document I will provide detailed explanations of some of the main functions and processes of the application, and show what makes the application efficient and fast.

Although this documentation provides some insight, a lot of information is also available in the comments of the code.

Getting and Setting pixel colour values

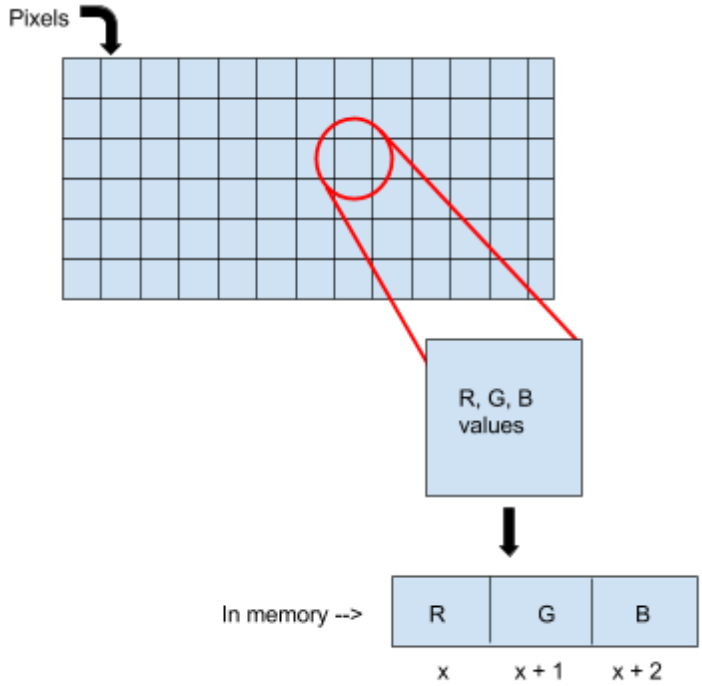
CImage features a native getter and setter function for retrieving individual pixel colour information. However, these functions are incredibly slow, and this slow speed is noticeable when performing the functions on every pixel of an image. To rectify this, I researched a method that greatly increased the speed of retrieval and setting on pixel colour information. The found method uses an algorithm that directly accesses the pixels colour information when it is loaded in the memory. The code can be seen below:

```
R = *(bptr + pitch*y + 3 * x);
G = *(bptr + pitch*y + 3 * x + 1);
B = *(bptr + pitch*y + 3 * x + 2);
```

Each line retrieves the respective red (R), green (G) and blue (B) value of a pixel. It does this by using the following 2 variables:

```
BYTE* bptr = (BYTE*)img->GetBits();
int pitch = img->GetPitch();
```

The ‘bptr’ is a pointer to the first byte of an image that is stored in the memory. The ‘pitch’ indicates whether the BMP data starts from the bottom left or top left of the image. From these 2 values, we can perform some maths to retrieve the exact pixel RGB values of any pixel. In the algorithm, the pitch is multiplied by the y, and this is added to 3 multiplied by the x. This is then added to the pointer of the first byte of memory of the image to get a memory location of the R value of that pixel. To get the G and B values, we simply and 1 and 2 respectively to move along 1 more byte in the memory. This explains why we multiply the x by 3, as every pixel has 3 values stored in it in memory for each R, G and B value. Please find a diagram on the next page to explain:



As well as retrieving the RGB values, we can also set them by giving the pointer locations colour values, as shown in the example below:

```
*(bptr + pitch*y + 3 * x) = R;
*(bptr + pitch*y + 3 * x + 1) = G;
*(bptr + pitch*y + 3 * x + 2) = B;
```

Function Call Hierarchy

When deciding what order to do the operations in, it became apparent that the scaling had to be done first. This significantly reduced the time the operations took to complete. This is because the scaling reduced the workload of each function by half (literally!). By doing the scaling first, the time to completion was reduced from about 6 seconds to around 2.2 seconds. After scaling, the image was then rotated. This is because rotating required a new image to be created, as the dimensions of the image would change and therefore the old image could not be used. Finally, the brighten function was called, then followed by the grayscale function.

Thread Creation / Joining

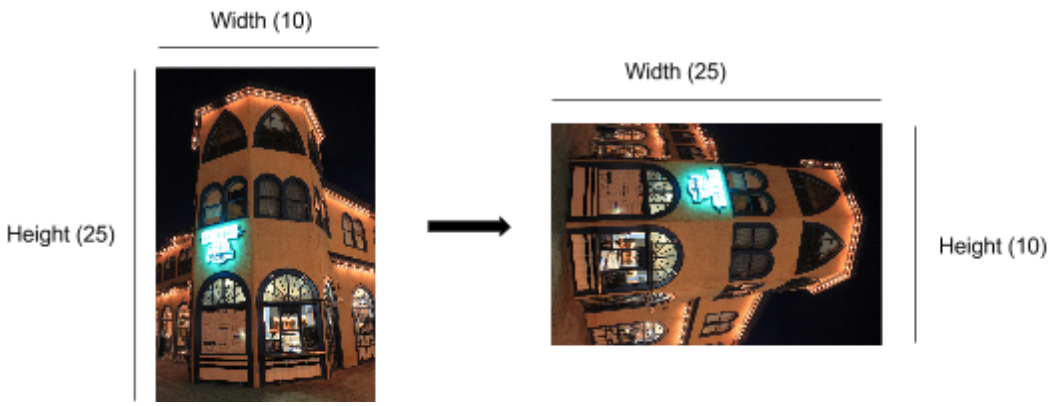
I decided to use multithreading to increase the speed at which the program successfully completed. Before threading, each image function would be called synchronously one after the other, and this took about 10 seconds in total. After introducing multithreading, completion time reduced considerably. I learned about multithreading during the Effective C++ module, although the threading required by this program was not as complicated as some of the methods I discovered whilst on the course. The program did not require any locking, as each thread handled it's own individual data objects and therefore did not

interfere with any other data objects being manipulated by other threads. Therefore, complex solutions such as using mutex were not required.
Thread pooling was also not required in this application, as the applications functions were ‘static’ and not dynamic, as the image file names were hard coded and therefore the application would never need to have idle threads waiting for new instructions. Thread pooling would have been necessary if I had a large number of short tasks, however I instead had a small number of large tasks to complete.
Hardcoding the threads creation and joining was considerably quicker than storing them in a vector. This also removed the need to use dynamically generated file names, and as the application used only the same 12 images the hard coding of file names and threads allowed less time to be spent on dynamically generating new object names. If the application was designed to accept any images, creating a vector of threads would have been vital to the success of the program.

The end of the code starts all 12 threads, followed by their respective join functions, as the application can not stop running until all threads have finished. If the threads were created in a vector, it would have been possible to automatically wait for all threads to have joined before continuing the program. However, the threads were not stored together in an STL container and therefore each thread had to be joined by hard coding their joins. An initial consideration was that as each join was synchronous, this could slow the speed of the program. However, after testing it was shown that allowing the threads to finish synchronously was no faster than allowing each join to complete asynchronously.

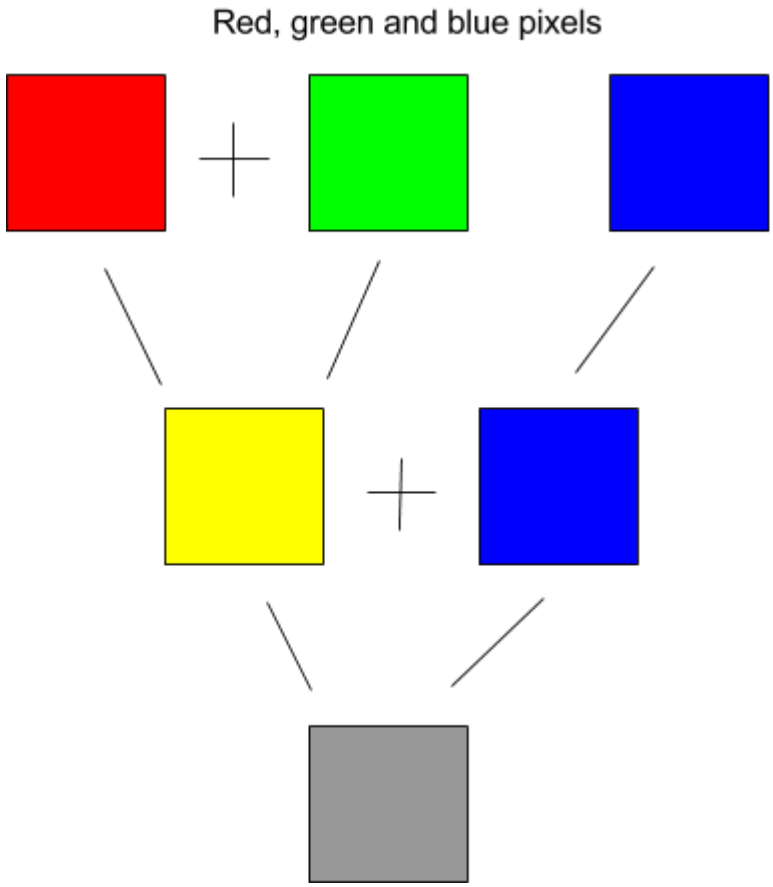
Brighten function
The brighten function, although given as an initial example, was modified significantly to improve it’s speed. To do this, the get and set methods for pixels was removed and replaced with the pointer method detailed at the beginning of this document. The function also changed so that is was not returning a pointer to a new image, and instead directly modified the pixels of the image which is given in the function's arguments.

Rotate function
The rotate function required a new image to be created, as the images dimensions would be changing. As the image was to be rotated by 90 degrees clockwise, the width and the height of the image would be ‘swapped’.



Again, the pixels were modified and retrieved using the direct pointer algorithm. However, this time there was 2 sets of pointers - one to the original image, and one to the new image that was being drawn to.
As 2 different pixel locations had to be kept track of as the original image x and y positions were not going to be the same x and y of the new image, I created 2 new integers to hold the x and y we would be drawing to, called resX and resY. Whilst looping through the original images pixels, resX and resY were also incremented, however they were incremented and reset at a different pace to the x and y being looped through on the original image. You can find the practical example of this in the code.
The function worked because it copied the pixels from the original image and put them in the new x and y location of the new image. The pixels RGB values were never modified.

Greyscale function
The greyscale function, like all the other functions, uses pointers to pixel colour values to modify the colours of the pixels. As the dimensions and orientation of the image do not change, we are able to modify the image directly and therefore are not required to create a new image, which takes time to do.
The greyscale algorithm is one which is used by GIMP, which is a popular image editing program (https://www.gimp.org/). The algorithm in question was found in GIMP's documentation: http://docs.gimp.org/2.6/en/gimp-tool-desaturate.html
GIMP uses 3 different greyscale algorithms and allows users to chose the most relevant one. However, as the task said only to greyscale the image, and did not specify a technique, I chose the 'average' greyscale algorithm that used the least resources and therefore took the least amount of time to compute. To do this, the code averages the R G and B values of each pixel, and sets the average value to each R B and B value respectively.



In the exaggerated example above, you can see how the 3 colours merge together to produce a grey variant. Although the image shows it visually, this technique works the same when using the numeric (0 - 255) rgb values.

```
BYTE gs = (R + G + B) / 3;

*(bptr + pitch*y + 3 * x) = gs;
*(bptr + pitch*y + 3 * x + 1) = gs;
*(bptr + pitch*y + 3 * x + 2) = gs;
```

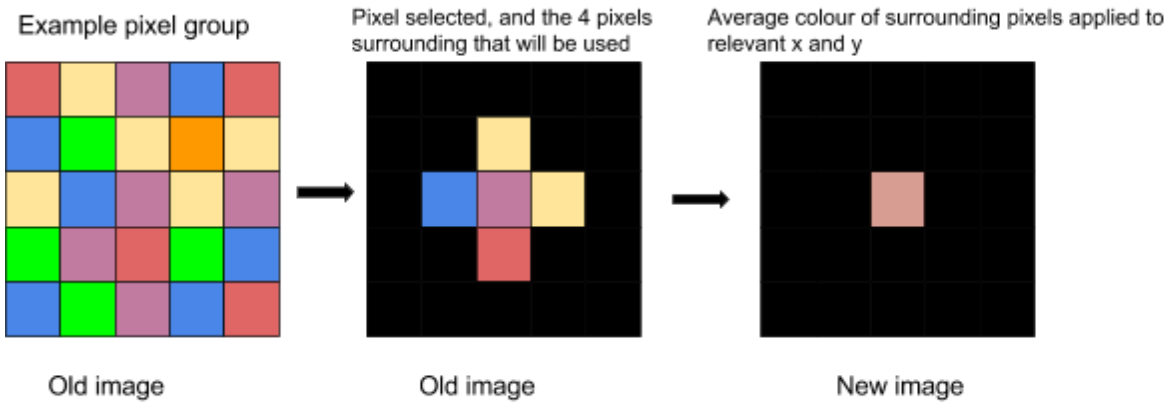
Scale function

The scale function is used to decrease the size of the image by half. As the scale function changed the dimensions of the image, a new image had to be created to be drawn to. Downscaling can produce blurriness in the image, the bilinear filtering algorithm was applied to smooth out the image and increase its quality.

The bilinear filtering algorithm is where a pixel takes its 4 surrounding pixels, averages the colour values and applies this new colour to itself.

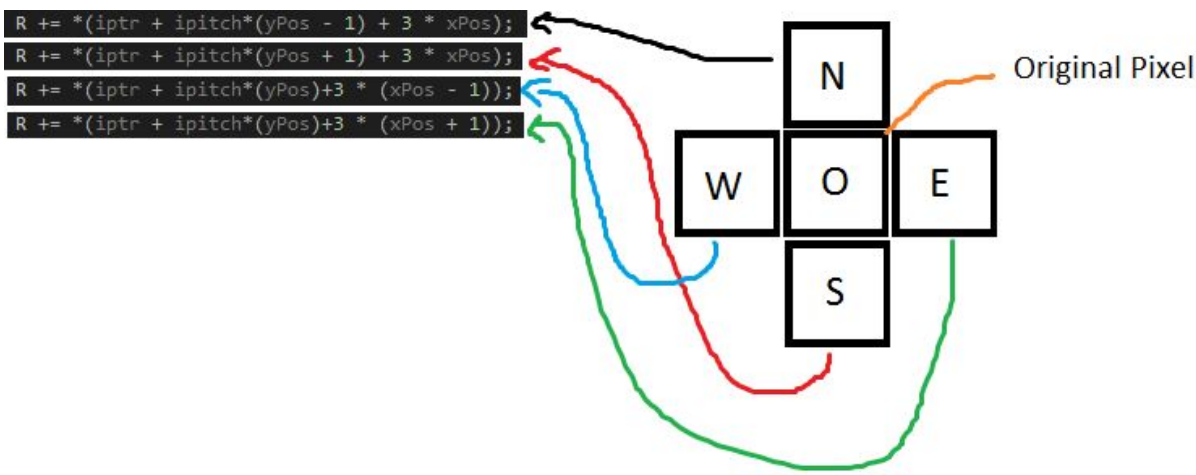
The code works by looping through the original image, and selecting every other pixel in the image, starting at 0 x and y. By skipping every other pixel, the image size was reduced by half.

For every pixel that was selected from the original image, the 4 surrounding pixels colour values were averaged and given to the new pixel drawn on the new image. To do this, a new function was created called **getPixelInterpolated** that returned a pixel object that contained the new RGB value. Please find an example of this below:



As the code looped through each pixel, it also checked to ensure that it was not on the edges of the image. If it was, trying to get a surrounding pixel that went out of bounds would cause a memory error, and crash the program. To ensure that the correct average was always achieved, a counter was created that counted how many pixels were evaluated before averaging out the value, as in some instances only 2 surrounding pixels could be retrieved from the target pixel (eg. if the pixel was on a corner).

Surrounding pixels were found by modifying the x and y values of the targeted pixel. The 4 values changed can be found below:



Notice the xPos and yPos have +1 and -1, changing the location of the pixels.

The scale function also used a separate x and y counter to track the destination for each pixel on the destination image. This is because the original image is being manipulated every second pixel, whereas the destination image is being drawn on **every** pixel.

Copy function

Although originally supplied in the skeleton code, I found that the copy function was slow and obsolete. Although it could have some uses, I found it much more effective to create a new image that was a blank slate and simply apply the original images information over it as necessary. This lack of copying allowed me to directly modify the image memory locations in some situations. For rotating and scaling, copying the image would have been ineffective as the resulting image would be a different orientation and/or different dimensions to the original image.

Run-time issues

There are no runtime issues for this program, however it has the potential for issues if protocol isn't followed. As the image names are hardcoded, removing any of these images from the project folder will cause the program to crash. The program overwrites any images that may be saved in the folder under the same name, so an issue could be that the code will overwrite images any images with the same name with no warning. Aside from that, the program follows the same rules as any other program for runtime errors, such as lack of available memory and basic CPU's that are unable to handle the number of threads.

Third-Party libraries

No third-party libraries were used in this program, therefore there are no external dependencies.

Design considerations

An idea which never came to light was to process 2 images on one thread, by pairing large and small images together. This would allow the program to create less threads, which is an 'expensive' process and time consuming. However, in practice this caused memory issues and the idea was never committed to during development. For future development, I would like to look into this technique and potentially apply it to my program.