PERFORMANCE ASSIGNMENT DOCUMENT

EFFECTIVE C++

Manuel Martinez
Computer Science for Games

1. Code Structure

The application starts reading JPG files from the *input* folder and storing their paths and filenames in a vector. After this, depending on the number of JPG entries, the number of used threads will be decided:

- **If files are more than 8**, the main vector will be splitted into 4 smaller vectors. 3 of them will be sended into different threads and the last one is executed in the main thread.
- **If files are more than 4**, the main vector will be splitted into 2 smaller vectors and will be sended into threads, while the main thread only waits for these threads to end.
- **If files are 4 or less**, the images are processed into the main thread.

The images are loaded using *CImage library,* already provided with the project, and processed one by one.

In order to process images, the following operations are performed:

- **Grayscale**: By getting each pixel of the source image and making the average between its components.

```
//Applying grayscale
//Average value between the 3 channels of each pixel in the source image
uint8_t r = *(current);
uint8_t g = *(current + 1);
uint8_t b = *(current + 2);
uint8_t grayscale = (uint8_t)((r + g + b) / 3);
```

- **Correct Image Rotation:** Images are loaded in memory with the width and height swapped. The application corrects this rotation by returning the image to its original dimensions.

  This is achieved by changing the way the destination image is readed, using height of the source image as width, and the width of the source as height. The *x(width index)* is used to read the current column. The *y(height index)* is used to read the row. Just the opposite as in the source image.

```
/*
Getting the destiny image index to access the right column
Since the image is rotated, we use the height as width
*/
uint32_t dstline_index = (height - y - 1) * channels;
```

```
//Getting the dst pixel
current = (dst + x * dstpitch) + dstline_index;
```

- **Brighten pixels:** Color saturation of pixels is increased by 100 in case this doesn't exceed 255. 2 function pointers are used in order to avoid "if" statements. If the pixel saturation is 155 or less, the add function is called, if not, another one is called which sets the pixel value to 255.

```
/*
Increasing pixel's color saturation with function pointers
Avoiding "if(pixel > 255) pixel = 255"
*/
*(current) = grayscale;
int32_t j = (*current < max);
bright_funcptr[j](current);
```

Since the image is in grayscale, the 3 components of the pixel will have the same value, so we only need to call this function once per pixel.

- **Scale:** The last operation and also the most expensive. A new image is created with twice the dimension of the source image. The new pixels are written in their new destination interpolating their values with nearby pixels.

```
//current pixel
uint8_t c00 = srcb[hyi * srcpitch + wxi * channels];
//east pixel
uint8_t c10 = srcb[hyi * srcpitch + ((wxi + 1) * channels)];
//south pixel
uint8_t c01 = srcb[(hyi + 1) * srcpitch + wxi * channels];
//southeast pixel
uint8_t c11 = srcb[(hyi + 1) * srcpitch + ((wxi + 1) * channels)];

/*
Obtaining a new pixel from the bilinear interpolation of the pixels above
The floating part of the difference between pixel indices will be the curve value
*/
uint8_t result = bPixelLerp(c00, c10, c01, c11, wx - wxi, hy - hyi);
```

One of the main reasons why this function is called later is because interpolating pixels in grayscale is cheaper than doing it with colors, since you need to interpolate RGB values for each pixel.

1.1 Functions

- All of these operations are performed in order on the *ProcessImage* function, that receives a *directory entry*, a JPG file in this case.

```
void ProcessImage(filesystem::directory_entry entry)
```

- Also, the scale operation is encapsulated in another function that returns the scaled image. It receives the source image to scale and the scale values:

```
//Scales an image by scalex and scaley values using bilinear interpolation
CImage* Scale(CImage* src, int scalex, int scaley) {
```

- To increase the saturation of the pixels there are 2 very simple functions.

```
//Function that brights a pixel
void BrightSum(uint8_t* pixel) {
    *pixel += 100;
}
/*
It assigns the maximum possible value to a pixel.
As each pixel component will be made up of unsigned chars, the maximum value will be 255.
Only executed if pixel value is up 155
*/
void BrightMax(uint8_t* pixel) {
  *pixel = 255;
}
```

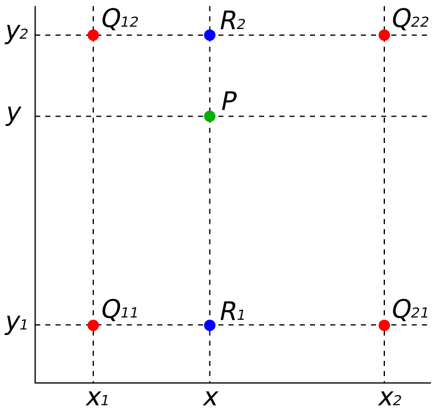
As explained before, these functions are used with a function pointer, calling one or another depending on the pixel overflow.

- Finally, the interpolation functions, linear and bilinear. Bilinear interpolation is used on the scale function, and linear interpolation is only used inside bilinear.

```
//Applies linear interpolation between 2 values(s and e) regarding 't' factor
uint8_t pixelLerp(uint8_t s, uint8_t e, float t) { return s + (e - s) * t; }

//Applies bilinear interpolation between 4 values
//Will be used to scale images without losing quality
uint8_t bPixelLerp(uint8_t c00, uint8_t c10, uint8_t c01, uint8_t c11, float tx, float ty) {
  return pixelLerp(pixelLerp(c00, c10, tx), pixelLerp(c01, c11, tx), ty);
}
```

This algorithm is used so the image doesn't lose detail when scaled. New pixels are obtained finding a color between adjacent pixels.
Bilinear interpolation is performed finding the linear interpolation in two different directions(Q12, Q22 and Q11, Q21), and interpolate again the resulting values(R2, R1), to obtain the final result(P).



2. Decisions taken

Since the application is quite simple, I didn't have the need to split the code in classes to encapsulate methods or data. All the code is written in the main file in about 300 lines of code, including comments.

The use of threads to divide work is a technique that I have taken from the multithreading lessons of this module, taking into account that 2 different threads cannot read or write from the same memory address at the same time.

On the other hand, I had never used function pointers to avoid if statements, but it's quite useful. Also taken from this module, from the optimization lessons.

This assignment reminds me in some ways of another that we had to do last year in Valencia. It consisted of making a simulation of a demo called *Metaballs*, and optimizing it.
https://www.youtube.com/watch?v=5xgNe3Je1kU&ab_channel=jrh2365
Among other things, the value of the pixels had to be added when 2 balls overlapped, taking care that they did not overflow. Also, by treating grayscale pixels greatly increases performance.