Emmanuel Batis September 2017

Sharkmob Blur coding test

Sharkblur – developed in Visual Studio 2015
Works with: Only RLE-compressed or uncompressed true-color **TGA** files (RGB and RGBA)

Notes: Please do run the application in Release mode when trying it, for some reason debug mode takes too much time when decompressing RGBA files only. I have no idea of why this is.

Hello Sharkmob person! First of all, I've got to say that this took me a bit longer than expected. I ran into some problems with TGA RLE decompression and some precision issues when calculating the blur accumulation (type was overflowing), so in the end I spent three days on this instead of the two that I had in mind. But I'm getting ahead of myself. I'm now going to describe how I developed the application.

Preparation (on paper)

When it comes to preparation, I basically broke down the problem into several parts and identified potential classes and structures. I also started thinking about fields and algorithms that were bound to be used, just to make it easier to nail down class names and such. I wrote down a (VERY rudimentary) UML diagram, this always helps me visualize the flow and see if there is anything missing or if I'm using too many objects.

Preparation (code)

After I had everything ready, I started writing down some boilerplate code; class definitions, empty functions, some structures and a general flow. Basically, everything in the `main` function was done, I even wrote some basic error messages that were sure to come up on some scenarios.

Once this was done I decided I was gonna go with TGA files since they were recommended. I searched online for a description of the file format (Wikipedia FTW) and wrote down important fields such as the file header, dimensions, etc.

And… action! (I know, I'm a clown)

The very first thing I wrote down was the code that loaded the TGA file into a `TGAImage` structure. Then I wrote the code to save it to disk. I ran this and realized everything looked horrible. I freaked out and started debugging the application but I couldn't find anything. I then spotted some mysterious byte in the header, ofc I forgot the fact that the image could be encoded in different ways.

It was actually a bit hard to find some useful information regarding RLE encoding and how to parse it correctly. I knew how it worked but I had no idea on how to differ an RLE packet from a pixel. I finally found someone talking about a bit in some forums, but didn't specify which one… I assumed it was the most-significant bit and printed the binary numbers from the image to corroborate. This worked and I was happy.

Blur it

I had several ideas when it came to blurring the image:

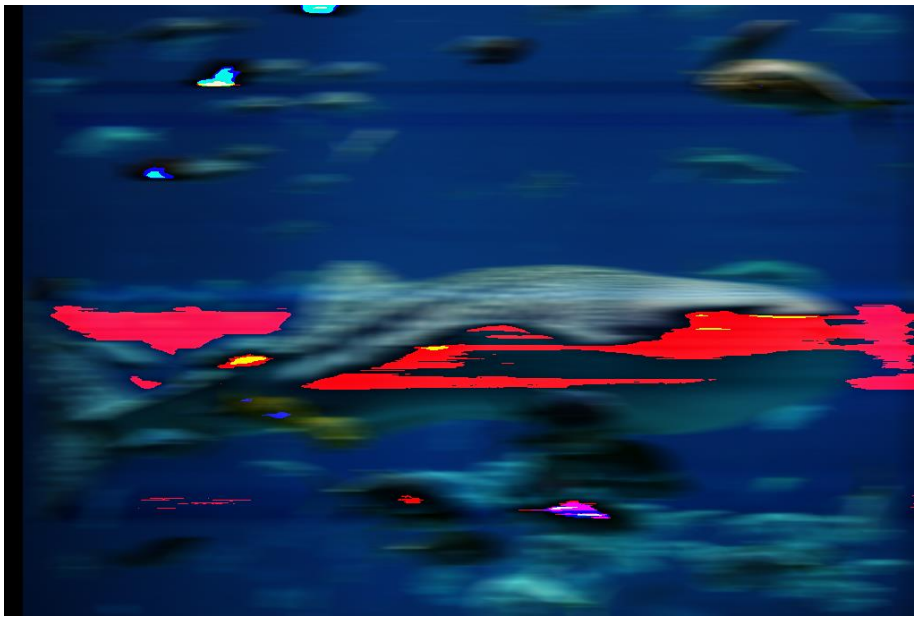
- Translating the whole image and interpolate it with the original
- Scaling down the image and interpolate it with the original
- Naïvely going pixel-by-pixel, sampling its 8 neighbors

Then I turned to them interwebs for some inspiration and read about how you can do it fairly easily in two passes, i.e. horizontal and vertical. That made sense to me so this is the algorithm that I implemented:

```
for each row
    for each pixel-along-width
        pixelCount = 0
        for every neighboring pixel from -radius to radius
            add color to total
            pixelCount++
        blurredPixel = total / pixelCount
```

That is the horizontal pass, for the vertical one you simply flip the implementation.

There is this cool optimization in which instead of recalculating the total color for each pixel, you basically "carry" a total color frame from the first pixel and move it along the row, subtracting the far-left pixel color and adding the new one on the right. Sadly, this didn't work and I wasn't really sure why, I could've spent some time on it but I decided to just go with the normal algorithm since it was already fast on its own (also this needs to be delivered at some point, right?). In the image below you can see how it was producing some weird artifacts.



Incorrect *marching-frame* blur implementation

I just realized I've written too much… I don't really know how detailed this description needs to be, but I'll try to be quick now. In any case I always comment my code and if I miss something here, you should be able to read it there. This section goes through each file in the solution.

Sharkblur VS 2015 Solution

`Sharkblur.h/cpp`

Entry point; this file basically handles the main flow of the application, the only real work it does is validating the command line arguments and storing them in a structure.

`TGA_IO.h/cpp`

I/O manager that is coupled with the TGA description. There was no need to design a fancier, more abstract class due to the size of the program. Basically loads TGA images from disk, decompresses run-length encoded files (if needed), and saves them to disk.

`Blur.h/cpp`

Static class that takes care of the blurring. I find it unnecessary to instantiate a class like this given that it only has one public method that takes a reference to an image. It holds two special structs that are used to accumulate pixel addition information (`BlurBGRPixel` and `BlurBGRAPixel`), adding dozens of bytes cannot really be stored in a single `unsigned char` variable, hence the integers in the structures.

When the `Run()` function is invoked, it detects the pixel depth of the image and runs the appropriate blur function; to keep things simple, only RGB (`BGRBlur`) and RGBA (`BGRABlur`) are supported in this program. I didn't want to write separate functions for this, the problem lies when accessing the image buffer with different pixel depths, using a 24 or 32-bit handle. I actually tried both using templates and through purely virtual inheritance but I always ran into some problems and in the end, I needed to duplicate some code as well.

I could have also written one function and check for an alpha channel just like I did in `TGA::Decompress()`, but I thought that doing this could affect performance since the check would be performed in a nested loop. I could have run some tests but again, time is of the essence and this is supposed to be a simple program.

`TGAImage.h`

Structure that holds TGA image information such as header, dimensions, pixelDepth, and the actual image buffer.

`ErrorConstants.h`

Simple file with a couple of common error messages used for flow control.

`Pixel.h`

Contains an enum for color space type and structures that are used to access TGAImage's image buffer.

That's it! Thanks for reading.