# FLASHPHOTO APP – ITERATION #2

Jonathon Meyer
Steven Frisbie
Jacob Grafenstein

Repository: repo-group-bandcamp

The FlashPhoto App Iteration #2 is complete with all the specifications outlined in the Official Software Requirements. A suite of filters and two additional tools have been developed, and undo/redo and image loading operations have been included.

To compile, go to the iteration2 directory and use the command ‘make’. If there is a problem, it is possible that you may need to reinstall the libraries, in which case call the commands ‘make cleanjpeg’, ‘make cleanpng’, and ‘make rebuild’. This series of commands will reinstall the libraries and rebuild the whole program.

The most important design decision that was made during the development of iteration #2 was the inheritance class design for the various filters. We discovered that the filters affected the canvas in two different ways: 1) by changing pixels throughout the entire PixelBuffer by adjusting the RGBA values of each pixel; and 2) through the use of a kernel and a temporary buffer to store the new RGBA values and then swapping the displayBuffer with the temporary buffer. Therefore, it didn't make sense for the two types of filter to inherit from the same class because it would have to be robust enough to handle both cases. We decided to create two parent filter classes, Filter.h and ConvolutionFilter.h, to handle filter type 1 and 2, respectively. Only two functions are defined in Filter.h: a pure virtual function modifyPixel() (which will be different for every subclass) and a generic applyFilter() function that subclasses have the option of modifying. Filters that inherit from Filter.h will not need to know any of the surrounding pixels in order to render correctly; the modifyPixel() function changes the RGBA values of given pixel. The ConvolutionFilter.h parent class has three functions: applyFilter(), applyKernel(), and resizeKernel(). The applyFilter() function iterates over the entire PixelBuffer and calls applyKernel() on each pixel. The applyKernel() function uses the kernel to garner the ColorData from the surrounding pixels on the PixelBuffer and adjusts the the pixel based on that ColorData. Then it stores the newly adjusted pixel in a temporary PixelBuffer. After the PixelBuffer has been iterated through, the temporary PixelBuffer is set as the displayBuffer. The resizeKernel() function is necessary because some kernels need to be resized depending on user input. However, other kernels never change, so the ConvolutionFilter.h class has an empty resizeKernel() to handle those cases.

We discussed several other ways to achieve this functionality, but ultimately decided on the one detailed above because it allowed the most flexibility in the creation of our filters and removed unnecessary classes. Two alternatives that were discussed are detailed below.

The first alternative was to include a parent Filter.h class, which would have two children inherit from it: ConvolutionFilter.h and NonConvolutionFilter.h. This was thought of because we thought we would want to have an array of Filters similar to the array of Tools. However, when we started working on the project, we soon found that having an array of Filters was unnecessary because the GLUI buttons for each filter called a callback function. So all we had to do was declare each filter in the FlashPhotoApp.cpp file and we could utilize it's applyFilter() function in the callback. Therefore, we did away with the unnecessary parent class and turned NonConvolutionFilter.h into Filter.h. Another reason why we decided against this inheritance structure is because the filters affected the PixelBuffer so differently.

The second alternative we discussed was to implement a kernel class that would encapsulate the function to modify individual pixels. Here, we would implement another function that would set the size and values of the kernel. However, this method soon proved to be more complex than was necessary. Because each individual ConvolutionFilter subclass has a constructor, the size and values of the kernel (a double array of floats) could be set and modified as class member variables. Additionally, most kernels affect the PixelBuffer in the same way, so there is no need to redefine that function in a mask class. We made the modifyPixels() and applyFilter() functions virtual so that children classes can redefine the functions as necessary.

Overall, the design we chose for inheritance structure minimized dependencies and the complexity of the program. It removed unnecessary classes and reduced the amount of duplicated code.

The second most important design decision that we made was how to implement the stamp tool. This was tricky because of our design decisions from the first iteration of the project. All of the masks of our tools were strictly squares, and the function to apply the mask to the canvas took in a maskSize variable. This posed a problem based on the fact that there were bound to be rectangular images that we would be loading into the stamp tool so a simple maskSize wouldn't quite work. This raised a few questions, which in turn led to a discussion of multiple different implementation techniques. We ended up creating a Stamp Class which is a subclass of the Tool class. This was necessary for a few, very important reasons. For one, the GLUI required us to initialize the stamp as a tool in order to select it. We stored all of our tools inside of a Tool array and so therefore, there wasn't really an option to not make the Stamp class a Tool Class. This led to a very easy way to implement when the Stamp tool gets applied, however. We simply had to write that when the current tool is the stamp tool, or the eighth index in our tool array, instead of applying the function paintMask() that all of our other tools use, we apply the stamp image which is essentially transferring PixelBuffer data. We figured the paintMask() function and masks in general weren't really applicable to the very different way that the Stamp tool applied itself to the display PixelBuffer. Beyond the stamp tool needing to be a Tool subclass, the Stamp class doesn't really do anything. Instead, We came up with the idea to just simply make another variable inside of the FlashPhotoApp class. This variable would be a PixelBuffer and we would just decompress the image from the loadStamp function into this new PixelBuffer. All we would need to do from here is iterate through the stamp pixel buffer and transfer each pixel's ColorData onto the display PixelBuffer. This ended up being extremely convenient and easy thanks to the getPixel() and setPixel() functions.

There were many other ways that we could have potentially implemented how the stamp tool would work. One of these possibilities was changing all of the code of our other tool classes to have a width and a height rather than just a single size. As we delved further into this, we saw that a lot of code would need to be rewritten and adapted to fit the width and height structure. Another possibility that we considered was to just make the mask of the stamp tool into a square that was the next highest square that fit around the whole rectangular image. After delving further into this implementation, we came to the conclusion that this would also take a lot of rigorous time and also that the stamp class wouldn't really feel like a tool class due to all of the new functions that we would need to write specifically for it. We also felt that the Stamp tool wasn't something that would really needed to be adaptive in regards to code that will be written in later iterations of the project due to it being a one of a kind type of thing.