Group Design Document, Iteration #2

1. Names of group members.

Cheng Chen
Kaitlyn Christensen
Michael Cooper
Simin Sun

2. Name of github group repository.

11amGroup02
https://github.umn.edu/umn-csci-3081S16/repo-group-11amGroup02

3. We believe that we have completely and correctly and implemented FlashPhoto according to the specifications provided.

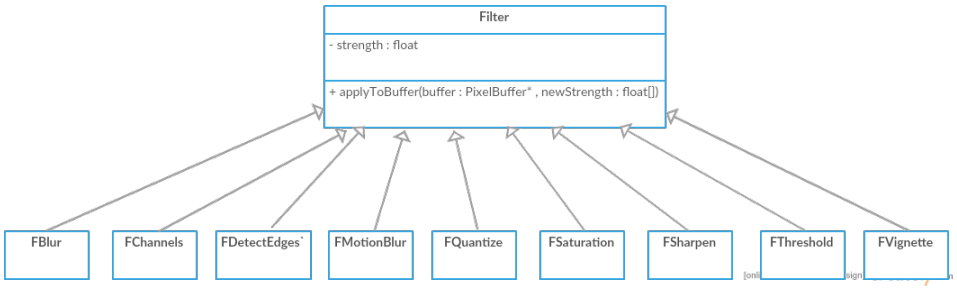
The most important decision:

A very important design decision for this iteration was the way we would implement our filters. We decided to structure our design around a Filter super-class that all of our filters would inherit from. Using a Filter super-class would provide a uniform interface to call any filter and also to allow them to be stored in an array in the main program. This also allows for more filters to be added in the future in an organized and structured way.

The main function to apply a filter to the canvas is called applyToBuffer(), which is a name borrowed from our Tool classes. The reasoning behind using the same name was to imply that variations of the same action are being performed, which is true—tools change the color data of a subset of the canvas's pixels, while filters change the color data of the entire canvas.

Initially, Filter::applyToBuffer accepted a pointer to a PixelBuffer and a single float value as its parameters, which worked well for all of the filters except Channels since Channels accepts multiple parameter values through UI fields. Since most of the other filters had been implemented and were working properly, there was a strong temptation to leave the other filters alone and to pass the three parameters for Channels in a workaround way, such as multiplying the second and third parameters by different large-order values.

However, recognizing the potential for a wide variety of filters in the future, we decided to put in the extra work for the sake of good coding practice making Filter::applyToBuffer() accept an array of float parameters instead of a single one. This was a relatively simple modification that required changing the signatures of this method on each of the filters and using the 0^{th} element of the array instead of the single value. This change also introduced opportunity for error in the number of arguments being passed through the array. For example, passing an empty array or an array with too many arguments might provide unexpected or undefined results. On the other hand, we were only calling the functions in a controlled fashion through the UI, which used parameters from a set number of fields, therefore we made the decision that the overhead of error-checking for these cases was not worth it.
The naming convention also mimicked that of the Tool classes by using an F and then the filter name for each individual filter. Since most programs display filenames in alphabetical order, this convention allowed all the tools to be grouped together and all of the filters to be grouped together.



The second most important decision:

For the second most important consideration, we think is the data structure for our undo/redo operations in our project design. If we want to do the undo and redo operations, a good way is to store every current PixelBuffer into a data structure when we create different PixelBuffer on the canvas. We can use the data structure to store the needed pointers of PixelBuffers in our design as follows:

1. We create two different objects of the data structure named d1 and d2;

2. We use the d1 to store the pointers of PixelBuffers that can be used to do undo operation. We do this by deleting the current PixelBuffer pointer in d1, and set the value of the current PixelBuffer pointer to be the same with the value of the most recent PixelBuffer pointer (which is on the top after deleting);

3. We store the PixelBuffer pointer to the d2, and then we can use d2 to do the redo operation;

4. If we want to redo, we just delete the most recent PixelBuffer pointer in d2 and set its value as the value of the current PixelBuffer pointer;

5. Add the PixelBuffer pointer from the d2 to the d1 again. A single operation needed.

Based on what we descripted above, we need to find a data structure that can delete and add the elements conveniently. And it can retrieve the old element and new element in an easy way. Another thing that we need to consider is that when we add or delete element in the data structure, it should be based on the "LIFO" principle which means "last in first out". According to these considerations, after the discussion, we decide to use the linked list at the beginning.

The linked list satisfies the requirements that I mentioned before, so the Alternative #1 data structure we chose in our project was the linked list. The advantage of the linked list is obvious: we can do the operations like adding, deleting, clearing conveniently by just changing the links between parents and children. But the disadvantage appeared when I try to build a class to create the linked list. Since the linking structure is very complex between the parents and children objects, I have to code very carefully. If something wrong with relinking, then the whole class will not work. I have tried many times to build the class, but it just didn't work because of its large complexity. So I consider to find another data structure that has the same advantages of the linked list but is easy to build. Finally, I find the vector.

Since it is very easy to build and it has more advantages than the linked list, the vector became the Alternative #2 selection and final selection for our project design. The attribute of the vector determines that it's very easy to build a class for it. The vector has lots of useful functions for redoing and undoing which includes adding, deleting, clearing, getting element by index, etc. Also it satisfies the requirements that I mentioned before. Actually we can also use the vector with its attributes in the FlashPhotoApp class directly, but I used a class to build it because I want to make it modular. After building the class successfully, I just completed the functions in the FlashPhotoApp.cpp file according to the instructions that I mentioned in the first paragraph easily. And the design works later on.

The main advantage of our final design as compared to Alternative #1 is that we can build its data structure's class much easily and we also can use it in a more convenient way. Another advantage is that the vector of the final design satisfied almost all of the requirements that we need to implement the undo/redo functions. Based on what I discussed before, we believe that our final design (Alternative #2) is the best selection for building the undo/redo functions in our project design.