repo-group-MidsummerNightsDream

Brent Higgins
Levi Cottington
Andreas Hochrein

Design Decision #1 -Choice of Storage system for Undo/Redo

A very important decision that was made during the development of this iteration of the project was the choice of how to store the PixelBuffer data for the undo/redo system. The decision to store PixelBuffer pointers was very simple due to the previously defined copy method which allows for simple copying of the data. The difficult decision was what would be the best way to store and access these pointers. The end decision was to use a pair of standard library stacks one to store data for the undo and the other to store data for the redo. This was not the only option explored as there were a multitude of other structures to be considered in the decision.

The first major alternative was actually my first thought for how to implement the storage consisting of a version of a doubly linked list that would store both the data for the undo and redo along with a secondary pointer to keep track of the current buffer. This data structure has multiple advantages including ease of limiting memory use because the max length of the list could be defined by a programer along with methods for handling reaching the maximum number of buffers stored. Another advantage it had was that it would be easy for other programmers to use because it would be in its own class with method for adding and getting the correct buffers from the structure. Despite these advantages the drawbacks of the class far were to high. The largest drawback of this structure was its complexity, in implementing this structure the programer is required to keep track of pointers to structs which in turn have pointers to the PixelBuffer objects that is what is actually cared about. This level of complexity can make it extremely difficult for future developers to modify if necessary. In addition in creating the data structure there needs to be carefully crafted methods for deleting the structure otherwise there will be memory leaks.

Another far simpler option that was considered was simply creating arrays to store the PixelBuffer pointers. This is an extremely simple option which eliminates the need to keep track of pointers to pointers but it has its own downfalls that caused us to look to other option. The one of the main drawbacks of using an array is that the program must ether keep track of index of where the last buffer is, or even worse index though every element in the array in order to add and remove buffers. Iterating through is terribly inefficient with O(n) runtime every time it has to find the end while the index can be kept track of other data structures make accessing the end far easier to program and more importantly understand.

In deciding to use stacks the benefits came primarily in the ease of use and the ability for future developers to modify the code. Using the standard stack class gives developers access to the number of built in functions for adding, removing, and other operations on the stack. While many of these can be coded without too much difficulty the extendability of the program is improved by having it already built in. Another benefit of these functions is the simplicity of the function calls in use, any developer who has an understanding of the stack data structure can see the function calls and instantly understand what is going on in the program. Finally by not having to create nodes manually eliminates a level of deletion that is needs to be created further lessening the risk of memory leaks.

Design Decision #2 - Making Filters Their own Classes

Another important design decision that we spent some time contemplating was how we were going to implement all of our filter functions. Originally we thought that the implementation would not be that in depth and that we could probably do the coding within FlashPhotoApp.cpp with little to no issues. But as we delved into the programming of it a little more in depth, we soon realized that this could not happen with ease. For one, FlashPhotoApp is long enough as it is, if we were to add all the coding that we did for filters, I have no doubt it would have been over 1200 lines of code in one file.

With the filters now acting as their own classes, we knew that this would help immensely for future iterations, or more properly for future programmers to use and manipulate. We can now see each one individually, see what each is doing in a relatively small amount of lines, and be able to reuse and call elsewhere if necessary, without having to worry about it affecting other class files. Personally, I prefer shorter bits of code because I hate having to scroll through hundreds of lines and constantly keeping track of what is going on and where each variable is coming from. Alternatively, if we were to put all this code if Flash photo, granted we wouldn't have to worry about all the variable passing and the extra files that occur when splitting it up. But retrospectively, the benefits of separating them far outweighs the benefits of the alternative. Because as I mentioned, the file would be a lot less dense and we wouldn't have to worry about loss of comprehension of the code because it was too long.

We first knew that we would have to have a similar approach as we did with the tool classes in the previous iteration. Therefore we began by making a parent class appropriately named "Filter" to act as a parent class to all the filters much like Tool did for all the tool functions. Using this approach, it was fairly straight forward how we would implement each function within the Flash Photo App.h by first including the header files of each class then creating instances of each as private member variables. Then it was a simple matter of calling each filter's respective class member functions that actually implemented the drawing and passing in the proper input variables.

In conclusion, we are very happy with the decision to split up the filters. We were easily able to split up the work amongst ourselves by each taking a number of filters to work on individually. This also helped for when we came together to discuss our own progress on the design as we could pull up our specific files without having to scroll through some big long file to find our code. It also gave us the ease of debugging by knowing exactly which file that error was occurring in. Although the src directory was pretty full of file names, it was the most logical decision we could have made regarding this part of the program design.