CSCI 3081

Group “App”

Design Document - Iteration 2

Steven Vande Hei

Alex Long

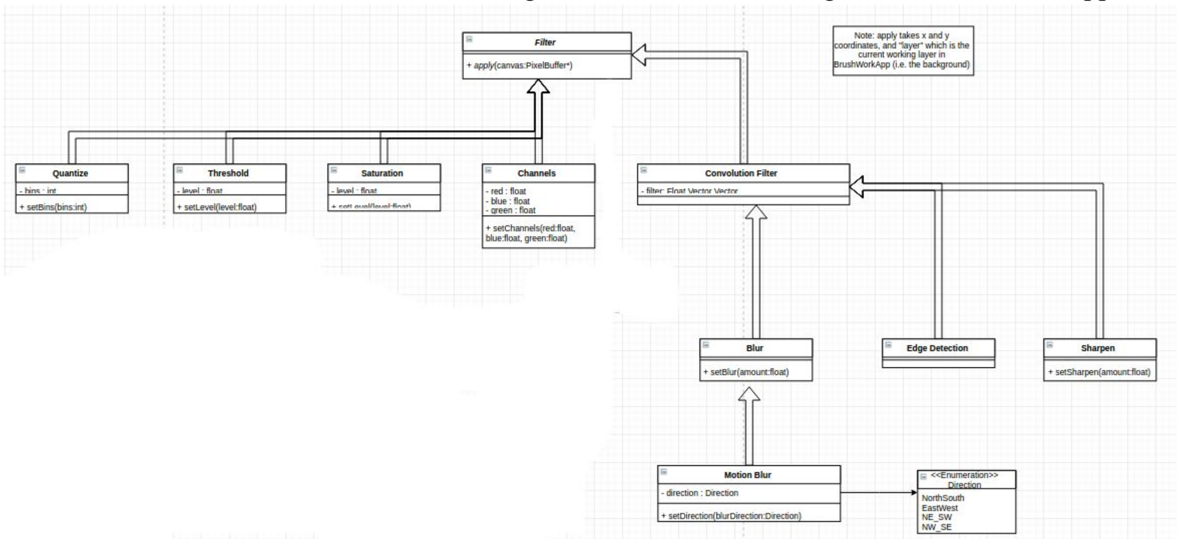
Grant Knott

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

We believe that we have successfully completed the requirements for CSCI 3081 Project Iteration 2: FlashPhoto, according to the software design specifications given to us.

Our first major design decision for iteration 2 was whether or not to follow a factory design method for the creation of filters. Following iteration 1, in which we implemented a number of user-controlled tools, our group discussed the possibility of changing the design of our program for the second iteration. We did not use the factory method for tool creation in iteration 1, and ultimately decided not to use it for filter creation in iteration 2 for reasons that will be discussed in the following paragraphs. We will look at the factory design method as it might be implemented for iteration 2, then discuss the design used by our group to make the filters for FlashPhotoApp.

For the class TA solution to iteration 1, a tool factory was used to handle the creation of new tools. The tool factory creates a tool based on the input from the user. Each specific tool class then calls the correct mask creation function, such as MOval or MLinear. Our group came up with a similar design for filter creation. Under this design, a filter factory would create the correct filter class (e.g. Blur, or Threshold) based on the user input. That filter class would be a subclass of two possible "base" filter classes - ConvolutionFilter or PixelFilter, which differ based on filter application methodology. These calls would then construct the correct kernel based on the chosen filter, where a PixelFilter would have a kernel of height and width one and thus only edit one pixel at a time. Although this design method would certainly work, our group decided not to pursue this design for iteration 2 because of the inherent differences between the filters. Instead, we chose to dismiss the factory and just use the template method. To understand our choice, observe this small segment from our UML diagram for FlashPhoto app:


Filter
+ apply(canvas:PixelBuffer*)
Note: apply takes x and y coordinates, and "layer" which is the current working layer in BrushWorkApp (i.e. the background)
Quantize
- bins : int
+ setBins(bins:int)
Threshold
- level : float
+ setLevel(level:float)
Saturation
- level : float
+ setLevel(level:float)
Channels
- red : float
- blue : float
- green : float
+ setChannels(red:float, blue:float, green:float)
Convolution Filter
- filter: Float Vector Vector
Blur
+ setBlur(amount:float)
Edge Detection
Sharpen
+ setSharpen(amount:float)
Motion Blur
- direction : Direction
+ setDirection(blurDirection:Direction)
<<Enumeration>> Direction
NorthSouth
EastWest
NE_SW
NW_SE

At the top, we have a Filter base class. The four classes on the left inherit from the Filter base class and define their own 'filterfunc' methods. Because these filters only work with individual pixels, not pixel neighborhoods, a common definition for filterfunc isn't shared between them. Therefore, we decided to have these filters inherit directly from the Filter base class. For the kernel filters, we decided to make a ConvolutionFilter subclass of Filter from which the four convolution filters inherit. These four filters each use a kernel, some of variable size, and the method of application to the canvas is more or less the same between them, so it made sense for our group to have "is a" relationships between these filters and a ConvolutionFilter class.

Because of the differences between the individual pixel filters, our group decided not to use the factory method on this iteration, and just stick to the highly structured template method of program design. If there were more similarities between the pixel filters and even the four convolution filters, the factory method probably would have been a stronger design candidate. But the fact is these differences between the filters would nullify the advantage of simplicity that makes the factory method helpful.

For our second important design decision, we will talk about our implementation of the undo/redo features in FlashPhotoApp. This is an interesting design decision because of the variety of structural options available when implementing these features. For example, the features could be successfully implemented using either a doubly linked list, two stacks, or double-ended queues (deques). We will briefly describe two alternative designs using doubly linked lists and stacks, then the design that we implemented for our FlashPhoto app and why our design made sense given the project guidelines and requirements.

Using doubly linked lists, one could simply add nodes to the list as modifications are made to the canvas. To undo, simply move back to the previous node. To redo, get the previous node in the opposite direction. This implementation is perfectly acceptable, though it is perhaps a bit more difficult to conceptualize than using stacks or deques since doubly linked lists would require more handling of pointers. In addition, although not a dealbreaker, keeping track of and reassigning pointers could also be a programming headache. The other two data structures eliminate this issue.

As mentioned above, two stacks could accomplish the same thing with fewer lines of code. Simply create two stacks and pop and push PixelBuffer pointers between the two as needed. The C++ Standard Library stacks come with a variety of useful, predefined functions that would have allowed us to move objects between the stack without difficulty. Our chosen data structure, the deque, includes similar helpful utilities plus a few extras as described below.

The deque implementation is very similar, but there is one important distinction: deques allow insertion and removal of elements from either end of the deque, whereas stacks are first-in-last-out, meaning you can only insert and remove elements on one side. This FILO property of stacks is problematic if you need to restrict the number of PixelBuffer objects a user can store between undo and redo at any given point. As per the project description, our program should allow undos and redos as long as there is memory to support the actions. In reality, we could likely store an unreasonable number of PixelBuffer objects on the two stacks. To be safe, though, we set a limit on the number. This limit simulates any real-world scenario where a product might need to run on a wide variety of machines with large differences in hardware capabilities. Instead of having the hardware set a limit (i.e. amount of memory available) we've set our own limit. When the undo stack reaches capacity, removing the oldest state would require having to pop all of the newer off the undo stack, remove the oldest state, and then push all of the other states back onto the stack in the same order they were in before. The setting of this limit would be possible to implement with stacks, but you would need to use three of them, resulting in some sort of Towers-of-Hanoi-like situation that would be a logistical mess.

So, using the two deques, we can construct the undo and redo functionalities for the user. Each time a modification is made to the canvas, we use a push_back command to add a pointer to the previous PixelBuffer object to one of the deques. Should the user press undo, we push the current state to the back of the redo deque, pop the element at the back of the deque, and set that element to the current state. As mentioned above, there's a size restriction placed on our deques. The standard library includes size and max_size functions that allow us to enforce this restriction. While a deque's size is not max_size, we can add elements to it. Should it reach max_size, we can remove elements from the front of the deque using pop_front.

From the description above, it's clear there are several ways to implement undo/redo functionality in FlashPhotoApp. We have described why our group thought deques would be the most effective data structure for this program given the requirements and ease of maintainability.