CSCI 3081
Group “App”

Design Document

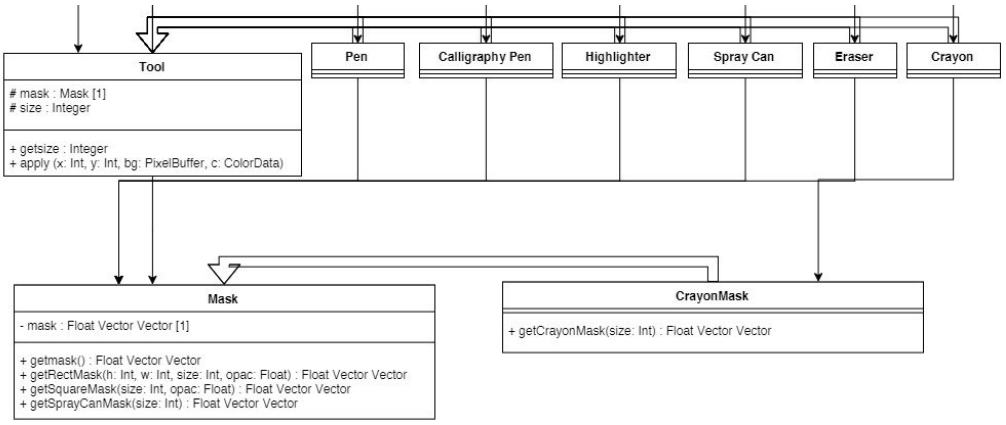
Steven Vande Hei
Alex Long
Grant Knott

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

We believe that we have successfully completed the requirements for CSCI 3081 Project Iteration 1: Brushwork, according to the software design specifications given to us.

One important and deliberate decision which we had to make early on in the design process was how to implement tools. We knew from the project specifications that each tool had to contain a "mask," or a 2D array of values which was pre-computed and copied to the underlying background upon mouse drag. However, the C++ language left open many different options for how to construct these masks, and implement them inside of each tool. In our final project design, we decided to separate tools and masks, having each tool get and maintain a mask object without the mask class being aware of which tool it was generating a mask for.

A simple UML diagram for our final tool and mask class design can be seen below. The image has been cropped from the full project design, where the class "BrushWorkApp" is aware of and includes the Tool class, as well as each individual tool subclass.



We did not come to this design immediately. Before beginning group work, everyone on our team split off and designed the program their own way. When we came back together we noticed that, while each of our designs fit the project specifications, there was one big difference between the three. This difference was the level of complexity within the Tool class. In one design, all of the above classes were combined into one "ToolMask" class, which contained a 2D array and a number of "setInfo" methods. While this system reduced the number of classes necessary, it relied heavily upon BrushWorkApp to instantiate each tool and to set the info for each tool's mask. The main drawback to this system was that it made it difficult to view the system at an abstract level, since the functionality of each tool was hidden inside BrushWorkApp. It would also have been difficult for tools to apply themselves to the canvas, since they existed only in BrushWorkApp.

Another design added the "Tool" class, which contained a 2D mask array and a function to apply mask to an instantiation of PixelBuffer (to be passed in by BrushWorkApp). This design also included a subclass for each tool. However, it did not include a Mask class. Instead, each tool set the info for its own mask. The main drawback to this design was that tools with similar features, such as the highlighter and calligraphy pen, contained much of the same code that would then show up twice in our program.

After our group came back together with these designs, we discussed the pros and cons of each of them and decided to include the functionality of the second design as well as a Mask class which would take in size and mask type specifications from the tool subclass which called it, while maintaining a level of generalization which would make our program more modular and flexible to change. In this final design, tools utilize the factory method, which provides a strong hierarchy within the design and makes the code clean and legible. The use of a mask superclass "institutionalizes" a solution to the problem of creating masks: masks are something every tool needs, which share many common features, but which are slightly different for each instantiation.

The second major design decision we made as a group concerned the application of the various masks to the canvas. As mentioned above, our group made the decision to separate the mask and tool classes to allow greater modularity, keeping in mind the future iterations and potentially changing requirements of the project. Separating the two, however, presented a logistical problem regarding the application of color data and mask data to the canvas. The project requirements state that tools must apply themselves to the canvas, so the decision of where to put the apply function in our program had already been made for the group. This left us with the challenge of somehow combining mask and color data, both stored separately in our program, and applying them correctly. We first thought of a general formula to calculate the correct values for each individual pixel:

$$(1 - \alpha) * (r, g, b) + (\alpha) * (r, g, b)$$

In the above formula, we have individual color values for red, green, and blue (r, g, b) and an alpha value to represent opacity. All four of the values are floats with possible values between 0 and 1.0. The first half of the formula gets the current color value on the canvas and multiplies it by 1 minus the alpha value of a particular index in the mask. The second half takes the current color value selected by the user and multiplies it by that same mask opacity value. This formula correctly blends the colors already on the canvas with those being applied to the canvas.

There is more to the apply function than just setting pixel color values, of course. Given that our masks were 2-dimensional data structures, we were faced with the challenge of centering those structures around the cursor and applying the correct mask and color values in the precise order specified by the current tool's mask. To accomplish this, we included two offset values in our apply function. The function gathers the x and y coordinates and the tool's mask dimensions and sets the offset values to the following:

x_offset = xCoordinate - (toolWidth/2) + iteratorValue
y_offset = yCoordinate - (toolHeight/2) + iteratorValue

The iterator values are added in to account for the function's position in the mask. Using these offsets, our apply function can apply the correct values determined by the mask to the correct pixels around the user's cursor. To iterate through the 2d vector data structure and get the mask value stored at each index, we used nested for loops. The outer loop takes the first row of the mask and the inner loops gets the opacity values stored in the mask for each column of that row. The function then moves on to the second row, retrieves that row's values, and continues this way until it reaches the end of the mask. It would be possible to iterate through the mask in some other way, perhaps using the width for the outer loop instead of the height. While the various iteration techniques would perform at about the same level of efficiency, ours has the advantage of following an intuitive iteration pattern. That is to say, we begin in the upper left corner, iterate to the right, then proceed downward to the next row. This simplifies the code design and index arithmetic. Other methods would work, but they would be less intuitive.

In the above paragraphs, we demonstrated how our function iterates through the mask data structure, correctly positions itself on the cursor, and calculates the correct color values. After these items are complete, the function takes the color and coordinate values and passes them to PixelBuffer, a class provided in the support code. Through the simple formula shown earlier, our apply function seamlessly takes color and opacity data and applies it to the canvas, providing an easy and responsive experience for the user.