CSCI 3081W
Group Design Document

1. Names of group members.

Cheng Chen
Kaitlyn Christensen
Michael Cooper
Simin Sun

2. Name of github group repository.

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

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

The most important thing we discussed is how should we implement the draw function. We decided that all brushes are inherited from the Tool class at the very beginning, however, should we include the draw function in the Tool.cpp file or in each brush's file? The apply function actually draws everything by setting the pixel to a different color. In our final design, except for eraser, every specific tool inherits the apply function from the parent class and only needs to declare the value of maskWidth, maskHeight, centerX, centerY in their own file. This design is the result of several hours of discussion and diagramming.

During the programming process, we also found out that there are two alternative ways to do this. The first one is, we still declare all brushes are inherited from tool, but they have their own masks and apply function. All apply functions can call the getPixel () function to get the current color. For eraser, we can simply just call getBackgroundColor () function and pass it into the function. For the highlighter, we can define it as 40% of the current color and 60% of the canvas color. In this way, all brushes have their own way to deal with different color requirements.

However, we figured out that actually most brushes have a similar apply function. Therefore, our team members considered the third way. Instead of defining the apply function in each of the files, we define it in the Tool.cpp. In this case, as all brushes are inherited from the Tool class, they will automatically get the apply function. For the eraser, we actually add one more "if" statement to distinguish it with other tools that have been used. Then, we will be fine when we use eraser with the same apply function as other brushes.

The main advantage of our final design as compared to the first way is, the apply functions in each tool is actually the same. According to the number of the mask, we can immediately apply the color to that location. Our final design takes advantages of this and defines the apply function in the Tool.cpp file. Therefore, when we declare each specific tool is inherited from the Tool class, it automatically has the apply function and the only thing we need to do is make the mask right. This will significantly reduce our work and make the coding style look better.

Compared with the alternative way 2, our final design works better because all five tools work almost in the same way and the eraser is the only special case. We can use the "if" statement to tell whether it is eraser; however, this also means that we need to check for each tool. This is a waste as it is one of the six tools, and it is better to overwrite this function by defining a new apply function in the Eraser.cpp file. In this case, if we want to add more properties to the brushes, we don't need to think about it again and check it every time before running it.

We believe our final design is the best because it emphasizes the inherent relationships between different brushes and main tool class. It also makes the logic clear how we use the mask by following the same equation: currentColor * mask[i][j] + canvasColor * (1-mask[i][j]). In this way, many repeated and redundant code lines can be avoided. Our design also deals with the special case of eraser by including its own apply function. In this case, the eraser will apply the background color to the selected positions and won't affect the apply functions for the other brushes.

The second most important consideration our group made concerning our project design was how to implement the mask for each brush. The main considerations for our mask implementation were (1) that each type of tool had a mask that should (ideally) be used and implemented in the same way, (2) that each tool may require a different-sized mask, (3) that each tool would only have one mask that it would have for the duration of the tool's life, and finally (4) that the mask, at this point in our project, serves a relatively passive role.

The first consideration led to a discussion of making a Mask class. The benefits to having a Mask class would be that each mask is constructed in the same way and that there is a consistent interface being called for each instantiation. If necessary, inheritance might simplify the mask types so that, for example, we could have SquareMasks and CircleMasks.

The second consideration made us analyze the best way to make different-sized and different-shaped masks. Given an assorted set of "brushes," every single mask could potentially have its own shape, and so trying to categorize them by shape might not be very helpful and may not really save much code. Additionally, when we take into consideration variable size, we introduce even more complexity. In the end we decided that the gains didn't surpass the costs of added complexity and voted in favor of a simpler model based on a square array with unique variations in "influence."

The third consideration was another clue that perhaps a class wasn't the best choice to implement a mask. If a mask is only used a constant number of times when the program is first run, and it's used uniquely each time, then not much code is being saved.

For the fourth consideration, we had to decide whether our concept for a mask really warranted a class. A class by definition is a collection of attributes and methods, and so in order for the class to be warranted, a Mask would need to have useful and fitting actions associated with it. The only actions we could think of for our current design would be initializing itself, which would be a unique process for each tool (since each tool does something inherently different to the canvas), and resizing itself. Since resizing is not a requirement for our project currently, we decided that our decision shouldn't be based solely on that feature, and so we are left with just a collection of attributes and an initialization method that would need a large amount of input data, which is not a strong case for a class.

After all our discussions, we decided that the best way to implement the mask would be a private array inside each type of Tool. This way, each tool could create its own size, shape, and influence pattern, and the overhead associated with another class would be eliminated. Actions on arrays can be very efficient, and since the initialization of the mask (setting each element's influence) could potentially be the most time-consuming action on the mask, it makes sense to just let each class deal with it within themselves. The uniqueness of the mask is more a property of the tool than of the mask itself. Since the mask should not really change for the duration of the program, it will be instantiated once when the tool it belongs to is instantiated.