**Pretty Paint Design Document**

Ian Cowan, Xu Song

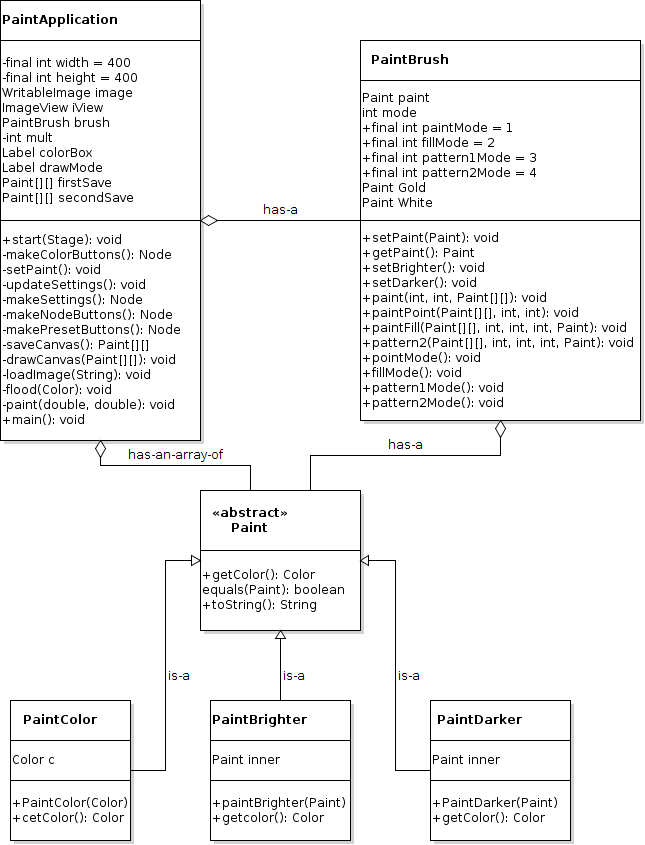
**Purpose**

We have been asked to create a paint program and specifically, our job was to implement the point painting, fill, and the two patterns. The requested patterns have been provided for us and our only job is to create the implementation. This project will help our skills with recursion and give us a chance to turn recursive functions into stacks to simplify, or possibly complicate, the code.

**Specifications**

The Pretty Paint application should allow the user to select one of five colors, black, white, red, green, or blue. The user should also be able to make these colors either brighter or darker depending on their preference. They should also be able to switch colors throughout the use of the application. Along with this, the user should be able to specify the resolution by changing the pixel size using a sliding bar at the top of the application. The user can change the mode that they paint with, those being either “Point”, “Fill”, “Pattern 1”, or “Pattern 2”. Point should just fill a single point that the user clicks. Fill should fill every pixel that has the same color of the clicked pixel and other colors should act as a wall from the color spreading. Pattern 1 should act like fill, but it should fill with a Centre College yellow and white striped pattern instead of the selected color. Finally, Pattern 2 should appear to work the same as Fill, but it should not use recursion. Fill should use some type of recursion, but Pattern 2 should not use recursion, instead, it should use a stack. For now, Pattern 2 only fills the selected color, but this could easily be changed in the future depending on the requested features. Finally, all of the functions should be a reasonable length and not excessively long.

**Design Overview**



**Figure 1. Class Diagram for Pretty Paint**

As stated above, all of the classes were prewritten for us except for PaintBrush. The beginning of the application begins within PaintApplication which has a PaintBrush in order to store the current color and paint the pixels as appropriate. When the program begins, the default color within the PaintBrush is set is black and the display is initialized. When the user wants to brighten or darken the color they are painting with, they can click the “Brighter” or “Darker” button, respectively. This will cause the PaintApplication to call the setBrighter() or setDarker() methods within PaintBrush. Next, when the color is changed by the user’s input, PaintApplication will call setPaint() and pass a Paint object with the new color. We then can handle the storing of the paint color within PaintBrush and allow the PaintApplication to retrieve the paint color with getPaint() which returns a Paint object. When setting the modes for painting, PaintApplication will call one of four methods: pointMode(), fillMode(), pattern1Mode(), or pattern2Mode(). These set the paint brush to do one of the following four actions: fill only the point clicked by the user, fill the entire adjacent area of similarly colored pixels around the point clicked by the user, fill with a white and gold pattern, or finally simply fill like before but with an explicit stack instead of recursion. When one of these is called, the mode attribute is set as appropriate for when the painting is initiated by the user.

Anytime the user clicks a point with the intent of painting, the PaintApplication will call the paint() method. Into this method, PaintApplication will pass an x and y coordinate along with a two dimensional array, mesh, with the colors of each pixel within the application. We then determine which mode the user has set, and follow through the process depending on the mode. If the mode is set to paint only a point, we simply pass the mesh, x, and y to the paintPoint() method and it then changes the color of that one point and we are finished. If the user has set the mode to fill, we pass the mesh, x, y, maxXY, and a Paint object: the color of the clicked point. We then use this information to recursively call paintPoint(), passing the same information as before, until all of the appropriate pixels are filled in with this new color. If we are set to fill with pattern 1, the process is the exact same as fill, but within paintPoint() it checks to see if the mode is set to pattern 1 and if so, it paints the pattern instead of the color chosen by the user. Finally, we have the pattern 2 mode. This fills and appears to act the same way as the regular fill, but it works by using an explicit stack instead of using recursion. This allows us to fill without worrying about a stack overflow, which is likely to happen with the recursion because of the number of times we have to recurse for the fill to work correctly.

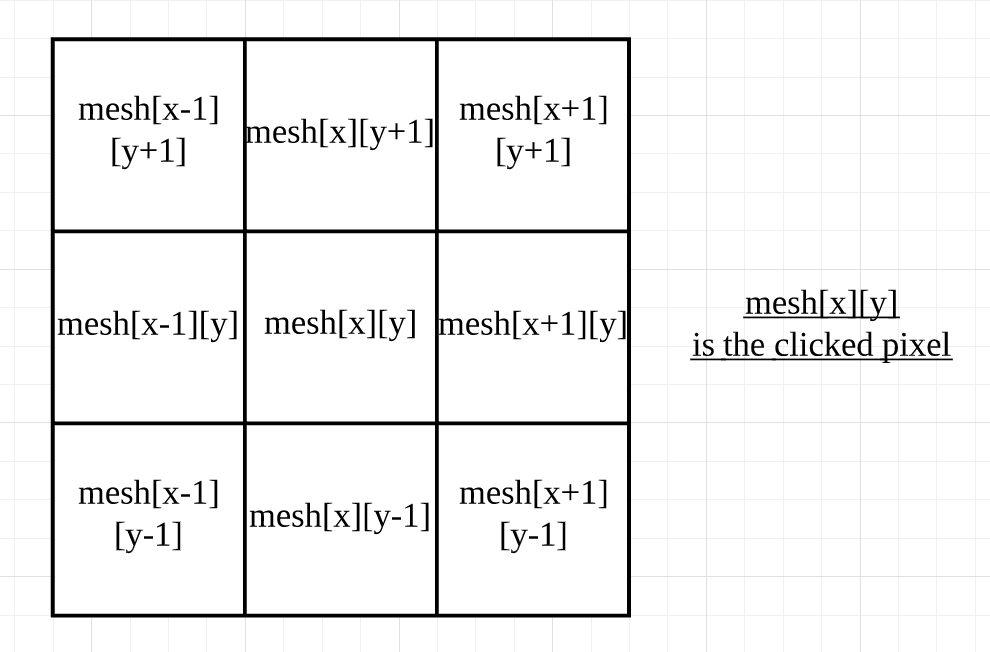
**Detailed Design Overview**

PaintBrush has two attributes, one is a Paint object called paint and the other is an integer called mode. The paint will be set when the user clicks on colors. There are four modes which are 1, 2, 3 and 4 and they respectively represented point paint, paint fill, pattern 1 and pattern 2. PaintApplication will call the paint() method when the user clicks on the canvas, then the paint() method will fill out the canvas properly according to the mode and color selected. PaintApplication gives paint() three parameters, the clicking point’s x,y and a 2 dimensional Paint object array called mesh. The mesh covers the whole canvas, the length of the mesh depends on how big the mult is, the shorter the length of the mesh is, the bigger is one pixel. Since the canvas is a square, the length of every array inside the mesh equals the length of the mesh. Therefore we can set the maxXY as the length of the array minus 1 and later the maxXY can be used as the upper boundary. To fill the canvas, the basic idea is to detect pixels around the starting pixel, if they have the same color then we will use the corresponding method to change the color. Therefore, we create a new paint object called startPaint to store the color of the starting pixel.

When the mode is 1, only the color of the clicked pixel will change. The paint() will call paintPoint(). According to the x and y passed in, the specific paint object stored in mesh[][] will be found and color changed.

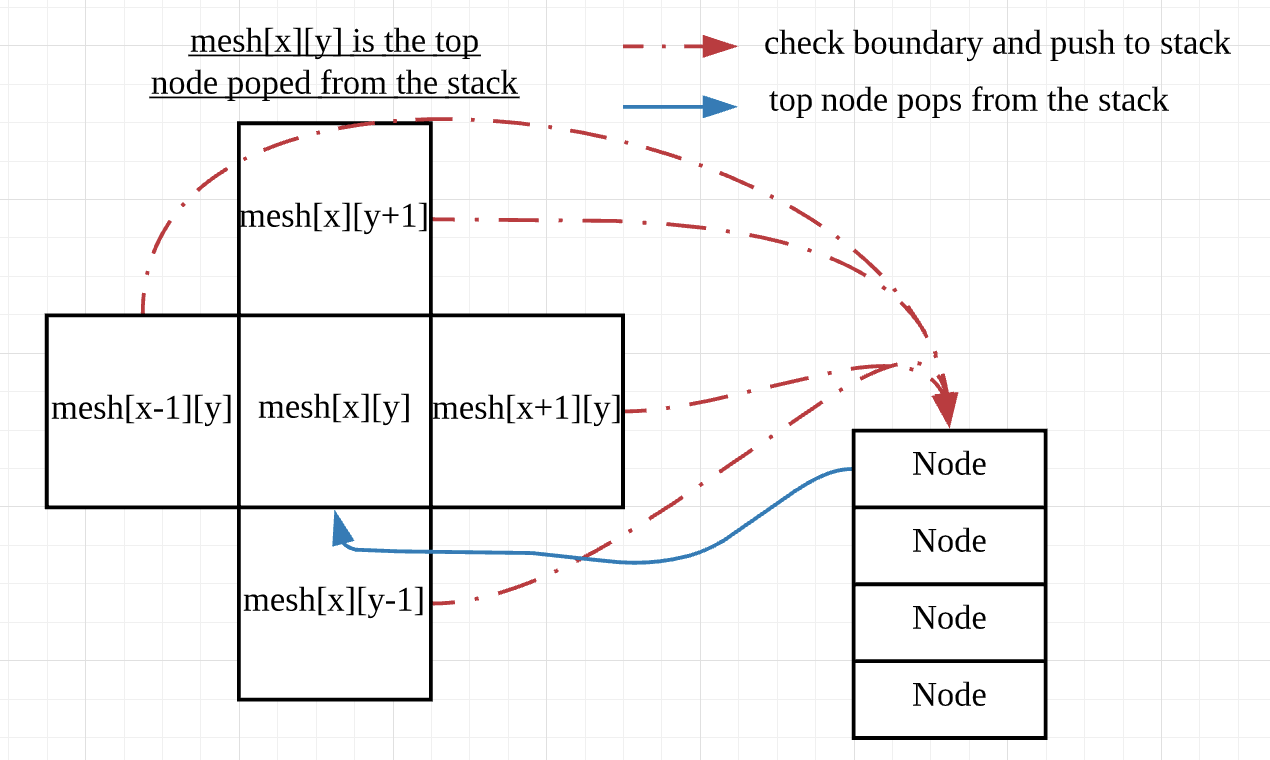
For mode 2, the paint() will call the paintFill() method we created. paintFill() uses recursion to complete the goal. The base case is when x and y smaller than zero or bigger than maxXY, the function returns. Otherwise, paintFill()will then find paint object in mesh[][] and compare the color to the startPaint; if they have the same color, the paintPoint() will be used to fill the object. After that, the paintFill() method will be called for 8 times to check the next 8 pixels around the paint object that just has been changed through change the x and y values.

For mode 3, the paint() will still call the paintFill() method which means the process of finding paint objects in mesh[][] array and changing the color are using the same recursion. The only difference is that when paintFill() call the paintPoint() method. The paintPoint() has an if statement for mode 3. When the mode is 3, it will detect if the x is an odd number or an even number. If it is even, the object will be painted as gold; otherwise, it will turn to white.



**Figure 2. Diagram for paintFill()**

Mode 4 is doing the same job as mode 2. However, instead of using recursion, we create a stack to implement the purpose. The stack contains Node object, each Node object three attributes: int x, int y and a paint object that stored in the mesh that can be found by x and y. When a pixel on the canvas is clicked, a new Node object will be created and pushed into the stack. While the stack is not empty, the top node will be popped out of the stack and following four if statements, each checks one pixel’s boundary conditions around the paint object the top node contains, will create Node objects if the pixels need to be changed and push them into the stack. At the end of the while loop, the paint object contained by the Node object popped from the stack will be changed by calling the paintPoint() method. The while loop won’t stop until the stack is empty, which means there is no paint object in the mesh that needs to be changed.



**Figure 3. Diagram for mode 4**

**Analysis**

Every method we created for filling ended up building on each other, so this reduced the amount of code we had to write immensely. We made a choice to begin with a method to paint a point, then use that method to allow our fill to work, and then use that fill method to allow the pattern to paint, and then finally we duplicated our fill method with another method that used an explicit stack instead of recursion, but still uses the paint point method. We could have written each of these separately and it still would have worked, but we would have had to write much more code.

Some risks of this project is that it is very common to get a stack overflow if the pixel size is set to small and the user uses the recursive fill. This is because the recursion occurs enough times that Java runs out of memory, so Java crashes with a stack overflow instead of crashing with a memory error. The fill with an explicit stack could run into the same issue, but it is much less likely because less is being stored on the stack at one time, because of our implementation. There is still the risk there of a stack overflow or an error caused by running out of memory. Other than those two methods, most of the other methods have little to no risk.

All of the methods within PaintBrush, except for paintFill() and pattern2(), have constant time complexity, . Both paintFill() and pattern2() run in linear time, that is . Both of these methods will run as long as there are still valid points to be painting. The more important aspect of these two methods, as stated above, is their use of memory. Both of these methods have the likelihood to cause a stack overflow error if Java runs out of memory because of the size of their stacks. This could be corrected by increasing the amount of memory Java is allowed to use, but would still be limited to the amount of memory on the computer.

**Conclusion**

In this design document, we lay out the purpose of the project and then describe the overall layout of our UML diagram including relationships of classes and objects and how each method is implemented in detail. We also analyzed the benefits and drawbacks of two different ways to achieve the goal of filling the canvas, one with recursion and the other with the iteration. After the lab, we have a better understanding of how the recursion works, we also solve the problem that the recursion can’t stop even if we already write the base case during the lab.

To sum up, our design can be considered as successful; it passed the unit testing and fulfills all the requirements that are asked.