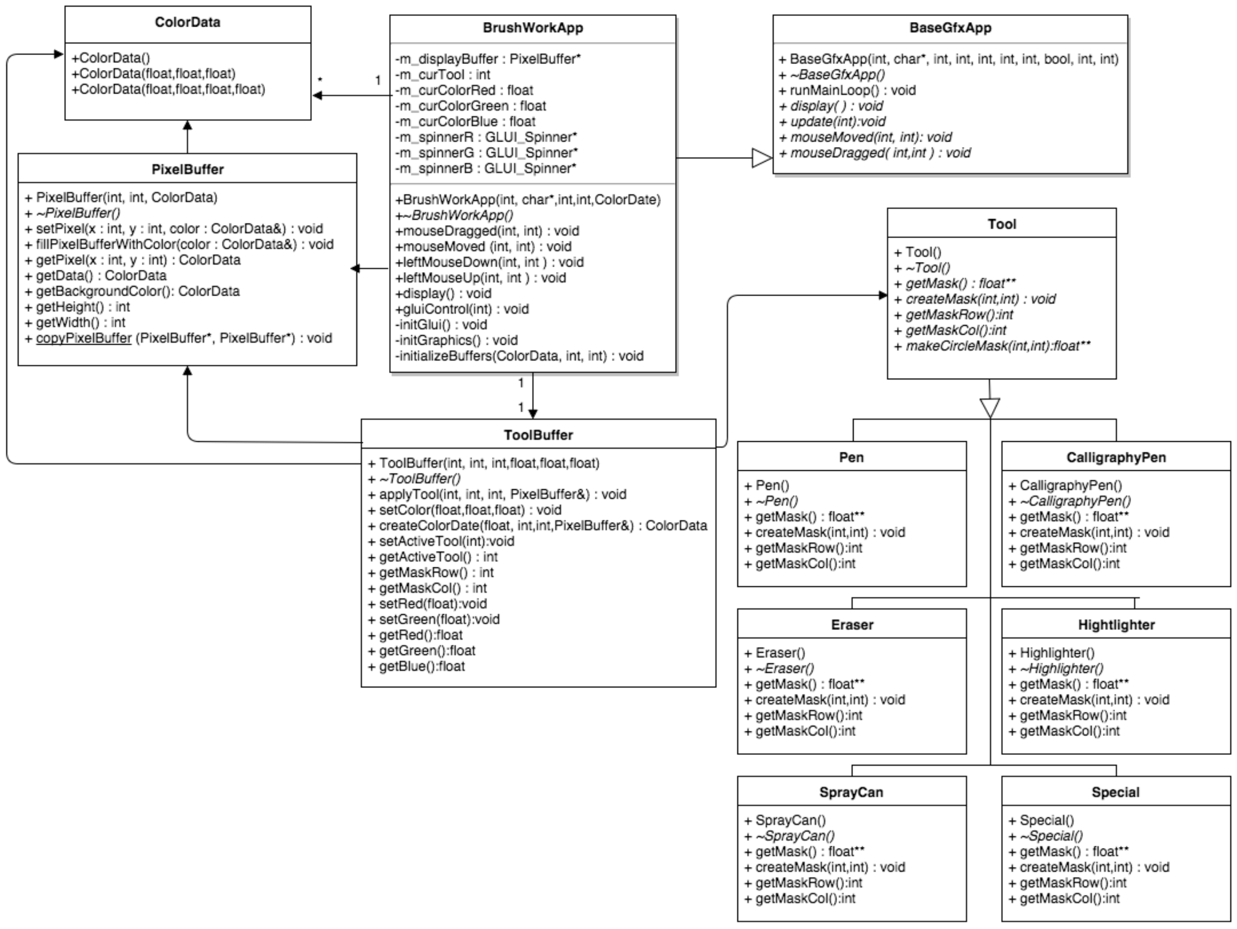
Members:
Peter Nguyen
Thanh-Mai Phan
Phoebe Zhang

Group Repository:
repo-group-11amGroup04

Statement of completeness of the solution:
We have completely and correctly implemented Brushwork according to the specifications provided.


ColorData
+ColorData()
+ColorData(float,float,float)
+ColorData(float,float,float,float)
BrushWorkApp
-m_displayBuffer : PixelBuffer*
-m_curTool : int
-m_curColorRed : float
-m_curColorGreen : float
-m_curColorBlue : float
-m_spinnerR : GLUI_Spinner*
-m_spinnerG : GLUI_Spinner*
-m_spinnerB : GLUI_Spinner*
+BrushWorkApp(int, char*,int,int,ColorDate)
+~BrushWorkApp()
+mouseDragged(int, int) : void
+mouseMoved (int, int) : void
+leftMouseDown(int, int) : void
+leftMouseUp(int, int) : void
+display() : void
+gluiControl(int) : void
-initGlui() : void
-initGraphics() : void
-initializeBuffers(ColorData, int, int) : void
BaseGfxApp
+ BaseGfxApp(int, char*, int, int, int, int, int, bool, int, int)
+ ~BaseGfxApp()
+ runMainLoop() : void
+ display( ) : void
+ update(int):void
+ mouseMoved(int, int): void
+ mouseDragged(int,int) : void
PixelBuffer
+ PixelBuffer(int, int, ColorData)
+ ~PixelBuffer()
+ setPixel(x : int, y : int, color : ColorData&) : void
+ fillPixelBufferWithColor(color : ColorData&) : void
+ getPixel(x : int, y : int) : ColorData
+ getData() : ColorData
+ getBackgroundColor(): ColorData
+ getHeight() : int
+ getWidth() : int
+ copyPixelBuffer (PixelBuffer*, PixelBuffer*) : void
Tool
+ Tool()
+ ~Tool()
+ getMask() : float**
+ createMask(int,int) : void
+ getMaskRow():int
+ getMaskCol():int
+ makeCircleMask(int,int):float**
ToolBuffer
+ ToolBuffer(int, int, int,float,float,float)
+ ~ToolBuffer()
+ applyTool(int, int, int, PixelBuffer&) : void
+ setColor(float,float,float) : void
+ createColorDate(float, int,int,PixelBuffer&) : ColorData
+ setActiveTool(int):void
+ getActiveTool() : int
+ getMaskRow() : int
+ getMaskCol() : int
+ setRed(float):void
+ setGreen(float):void
+ getRed():float
+ getGreen():float
+ getBlue():float
Pen
+ Pen()
+ ~Pen()
+ getMask() : float**
+ createMask(int,int) : void
+ getMaskRow():int
+ getMaskCol():int
CalligraphyPen
+ CalligraphyPen()
+ ~CalligraphyPen()
+ getMask() : float**
+ createMask(int,int) : void
+ getMaskRow():int
+ getMaskCol():int
Eraser
+ Eraser()
+ ~Eraser()
+ getMask() : float**
+ createMask(int,int) : void
+ getMaskRow():int
+ getMaskCol():int
Hightlighter
+ Highlighter()
+ ~Highlighter()
+ getMask() : float**
+ createMask(int,int) : void
+ getMaskRow():int
+ getMaskCol():int
SprayCan
+ SprayCan()
+ ~SprayCan()
+ getMask() : float**
+ createMask(int,int) : void
+ getMaskRow():int
+ getMaskCol():int
Special
+ Special()
+ ~Special()
+ getMask() : float**
+ createMask(int,int) : void
+ getMaskRow():int
+ getMaskCol():int
*
1
1
1

Design Choice #1

The most important design decision we made was creating a “tool buffer” class. The “tool buffer” (TB) is a vector that holds each of our brushes. These brushes are objects that inherit from a common “tool” class and are differentiated by a unique mask. The TB also has an important function called “applyTool” which effectively connects the different components of our application.

Implementation of TB
With TB, we can easily connect our brushes to the BrushWorkApp class. We create a TB object upon instantiation of our program that contains a vector of tools. We used a vector so we don't have to hard code the number of brushes we use, since vectors can be dynamically allocated. Each brush in our TB vector corresponds to the index of a GUI button and has a precomputed mask that is ready for use. Because they're directly related, we're able to easily keep track of the active tool.

With this variable, we use TB's “applyTool” to update PixelBuffer when the mouse is clicked and dragged. This function takes the current color data, location of PixelBuffer, and current coordinate provided by BrushWorkApp. Using the active tool's mask, we compute the new pixel color and directly update the canvas.

Alternatives We had two alternatives idea within our team of how we should go about the final design.

Computing the mask within TB
For this iteration, the only difference between each brush was its mask. Because of this, one of our group members reasoned that we wouldn't need to create a separate object for each mask. Instead, the masks could be precomputed within TB and placed into the vector.

While this would work for this iteration, we ultimately decided against this because it doesn't take into account future changes to BrushWork and creates dependency within the TB class. By using inheritance, we're able to make changes to each brush without having to change much code within TB.

Brushes exist as variables within BrushWorkApp
Initially, it seemed to make sense that brush objects could exist without a mediating class within BrushWorkApp. However, this would entail adding a significant amount of code to the main class and creating dependency. Using a vector also provides a clear connection to the GUI.

Why TB? Our design decision has multiple benefits: clarity, efficiency, and scalability.

Clarity
We believe this design decision has optimal clarity because the TB reduces dependency between BrushWorkApp and each of the brushes. It provides a means to uniformly update the canvas with data from BrushWorkApp and the precomputed mask.

Efficiency
Because the TB is created immediately, there is no computation involved when the user selects a brush. Its one-to-one correspondence with the GUI ensures efficiency because each brush is readily available.

Scalability
Our design scales! Because TB holds a vector, we're able to quickly add new brushes to our program by adding a couple lines of code to BrushWorkApp. We can also alter our tool class without having to change much code in other components. We also designed our “tool” class to be flexible – each brush is created by specifying the dimensions. This ensures that the pattern of each mask is scalable.

Design Choice #2

The second most important design decision our group made was how to make the brush more fluid as to not show any gaps when the user drags the mouse very fast.

Implementation

We wanted to come up with a solid, robust and adaptive way to implement this feature. The way we decided to accomplish this was to add some member variables to BrushWorkApp.cpp to keep track of the previous coordinates. Once we have that we can calculate the distance from the previous coordinates to the ones that are being fed to the mouse dragged method. Then if the distance is greater than that of the range of the mask divided by two since the coordinates are always at the center of the 2-dimension mask, we will partition the distance into equal sections depending on the size of the range and apply our tool to those coordinates to fill in the empty gaps. This is done within a for loop that will only run if the distance is greater than the range we defined to be equal to the mask divided by two.

Alternatives

Our design was the result of considering a lot of different possibilities and putting into consideration the amount of manpower and time we had to accomplish this task. We had two main alternative to think about.

Our group had to think a lot about whether adding in this extra functionality to our code was worth it or not. We had to weigh the cost of time and effort needed. Considering that is was not part of the requirement, our group came to a conclusion that we should implement this. We decided that having our application be more fluid for the user is worth the amount of effort needed to code this extra feature. Considering that an application for painting and drawing that is not very response to the user or makes it harder for the user to do what they want is practically useless and will not be a product that meets the industry standards.

Another alternative we had to think about was instead of using a for loop to fill in the gaps depending on the distance we could have used a set amount of times to fill the gap since the gap size seems to level off the faster you move the mouse. Meaning the max distance between two points would have hit a limit. So we could have calculated a set distance and used a static variable to determine the amount of times we would need to apply our tool to the gaps.

Advantages

The main advantage of our final design as compared to the first alternative was that the application more reflect that of other competitive applications. Having it be more fluid is definitely something that is worth the time and effort to implement because it makes our application more usable and adds quality to our product. Even if it is not a requirement the advantages of having a more solid application outweighs the cost. We also predict that eventually we would want to produce an application that does meet the standards of other paint applications so adding in this extra feature early would not only help us with future testing of added functionality but also gives us a more realistic view of what our end product should be.

The main advantage of our final design as compared to the second alternative was that our application was more robust when we used a for loop to dynamically fill in the gap based on the distance from the previous coordinates rather than have a set amount of times we would fill in the gap. Even though the max distance hits a limit we can not really predict the user's movement when they are using the application. Doing it this way has a lower chance of introducing new bugs as we add more functionality to our application in the future because it fill in gaps based on the individual distances given to the application in real time.