Michael Adkins, Cuiqing Li, Logan Seeley

GitHub Repository: **repo-group-10amGroup06**

B*rushwork has been correctly and completely implemented according to the specifications provided. The special tool specified is a cloning tool that samples 50 pixels left and down of the cursor's position. Additionally, an attempt to interpolate between mouseDragged events has been implemented. See the final page for a brief UML diagram of major design choices.*

Design choices and justification

Initially, each type of tool used by BrushWorkApp belonged to its own class so that there were 6 different tool classes: Pen, Eraser, Highlighter, CalligraphyPen, Spraycan, and SpecialTool. However, the placement of these tools in separate classes seemed excessive as each class was only instantiated once. Another issue with this implementation is that the application of tools required special handling for each tool class. Furthermore, if new tools were to be created in the future, each one would require a new class which duplicates much of the previous code, as well as the modification of functions across the entire code base. To resolve these redundancies, the tools were generalized to their basic shared feature: the mask.

The mask defines the most important characteristics of the tools: shape, size, and opacity. To simplify the tools to a mask allowed simple instantiation of each tool with the appropriate arguments in BrushWorkApp. The tools defined in the specification sheet were easily generalized to circular and rectangular masks which inherited from the primary Mask class. This inheritance allowed BrushWorkApp to assume that each tool was a square mask designating opacity values while the creation of the specific mask shape was left to the children classes which were handed opacity settings indicating a solid opacity (one value), linear decay of the opacity from the center point, etc. If, in the future, it is decided that a new tool is to be created with a shape different from a rectangle or a circle, a new subclass of Mask could be created to implement the desired shape without changing any other code.

```
void CircularMask::build() {
    // Find the center of the mask
    int center = m_diam/2;
    for (int i = 0; i < m_diam; ++i) {
        for (int j = 0; j < m_diam; ++j) {
            float val, dist;
            // Calculate distance from center
            dist = sqrt((i-center)*(i-center)+(j-center)*(j-center));
            if (dist > m_diam/2.0)
                val = 0.0;
            else
                val = buildpoint(dist);
            m_data[i][j] = val;
        }
    }
}
```

```
// Spray can implementation
tool[2] = dynamic_cast<Mask*> (
    new CircularMask(41,MaskOpacitySettings(LINEARDECAY, 0.2, 0.01))
);
```

```
float Mask::buildpoint(float dist_cent) {
    float val;
    switch (m_settings.type) {
        case SOLID:
            val = m_settings.coeff[0];
            break;
        case LINEARDECAY:
            // y = mx + b
            val = m_settings.coeff[1] * dist_cent * -1 + m_settings.coeff[0];
            break;
```

In this implementation, the application of the tools was left to BrushWorkApp as before, but it now only required a single loop over a square mask rather than a switch over each type of tool each with its own specialized loop. This makes future expansion of the program to include other tools significantly easier than it would have been otherwise, since no modifications need to be made to the way tools are applied in BrushWorkApp.

This technique of implementing tools is an example of good design saving future time while simplifying the current implementation. Rather than coding a few lengthy functions handling many cases, the exceptional classes were turned into children of a superclass that contains many small functions handling most of the cases. The children then only need to override the functions that make them unique. The modification or creation of any mask based tool is now arbitrarily simple as show above in the declaration for the spray can. However, this implementation does have limitations that are addressed in the major next design choice as tools that do not use masks are introduced.

Initially, the array of initialized tools was implemented as an array of Mask pointers. However, with the addition of a special tool, a major design choice had to be made. The previous design choices regarding mask classes made the creation of a novel special tool tricky because all implementations of a special tool that used a mask would require little program design. The departure from using only mask based tools meant that the array of initialized tools would have to be changed. Here, there were several options:

- Have several different arrays of initialized tools based on tool type
- Create another mask child that overrides many of the initial mask functions
- Create a new class of type Tool that functions as a wrapper for several types of tools

The Tool wrapper was chosen to future-proof the code and to keep the minimalistic implementation of tool-handling within the BrushWorkApp class. Many of the tools used in photo-editing or drawing applications exceed the basic design of cursor centered mask-based painting. In preparation for the addition of more realistic tools, a Tool wrapper allows the BrushWorkApp to be ignorant of the type of tool being used--simply sending the relevant information to the Tool class and letting it deal with all individual functionality. This allowed the retention of the simplifications made in BrushWorkApp as well as improvement to the readability and design of tool application.

The application of each tool to the canvas was moved into the Tool class (virtually) and implemented by each child class. This drastically reduced the amount of code present in the mouseDragged handler which in the long-term could make it easier to see what sub-events are called when the handler is called (as more are added). The Tool wrapper also allowed the movement of all the helper functions for tool application such as color blending and bound checks into a relevant container instead of free-floating in the BrushWorkApp. These helper functions are necessary to reduce redundancy across tool application for different tool types and decrease the depth of the code for tool application for readability.

Additionally, the use of a Tool wrapper prevented the Mask class from gaining dependence on other parts of the application such as PixelBuffer and ColorData. Considering the mask is a purely mathematical construct, its implementation should be functional as a stand-alone library. In contrast, the Tool exists to apply ColorData to a PixelBuffer and its dependence on those classes is reasonable.

Unfortunately, it's hard to predict the specific details of future additions which means that the Tool wrapper may not be sufficient. It's largest limitation (as currently implemented) is that the virtual apply function takes a set of arguments defined in the wrapper which may include information that is excessive for the child tool or be missing essential information for application. An example of this is the passing of the foreground color into the SampleBasedTool which goes unused. Additional tools that rely on an initial click and drag to draw a shape such as a rectangle would also exceed the limitations of the apply function as currently described. Such new tools are not the end of the Tool wrapper though, simple separation of apply into new functions such as dragApply, mousedownApply, and mouseupApply with relevant information would allow inheritance by most conceivable tools.

```
// Spray can
tool[2] =
    dynamic_cast<Tool*> (
        new MaskBasedTool (
            dynamic_cast<Mask*> (
                new CircularMask(41,MaskOpacitySettings(LINEARDECAY, 0.2, 0.01))
            ),
            PHOTONFLUXBLEND
        )
    );
```

```
void MaskBasedTool::apply(int x, int y, PixelBuffer* buf, ColorData fg) {
    // Grab important mask information
    float **mask = m_mask->OpacityData();
    int size = m_mask->getSize();
    // Find the upper left corner of area to apply to
    int app_x = x - size/2;
    int app_y = y + size/2;
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            // Find the real coordinates, fixing y inversion
            int x = app_x + i;
            int y = buf->getHeight() - app_y + j;
            // Update pixel if applicable
            if (mask[i][j] > 1.0e-6 && checkBounds(x, y, buf)) {
                ColorData col_new = blendColor(buf->getPixel(x, y), fg, mask[i][j], m_blendmode);
                buf->setPixel(x, y, col_new);
            }
        }
    }
}
```

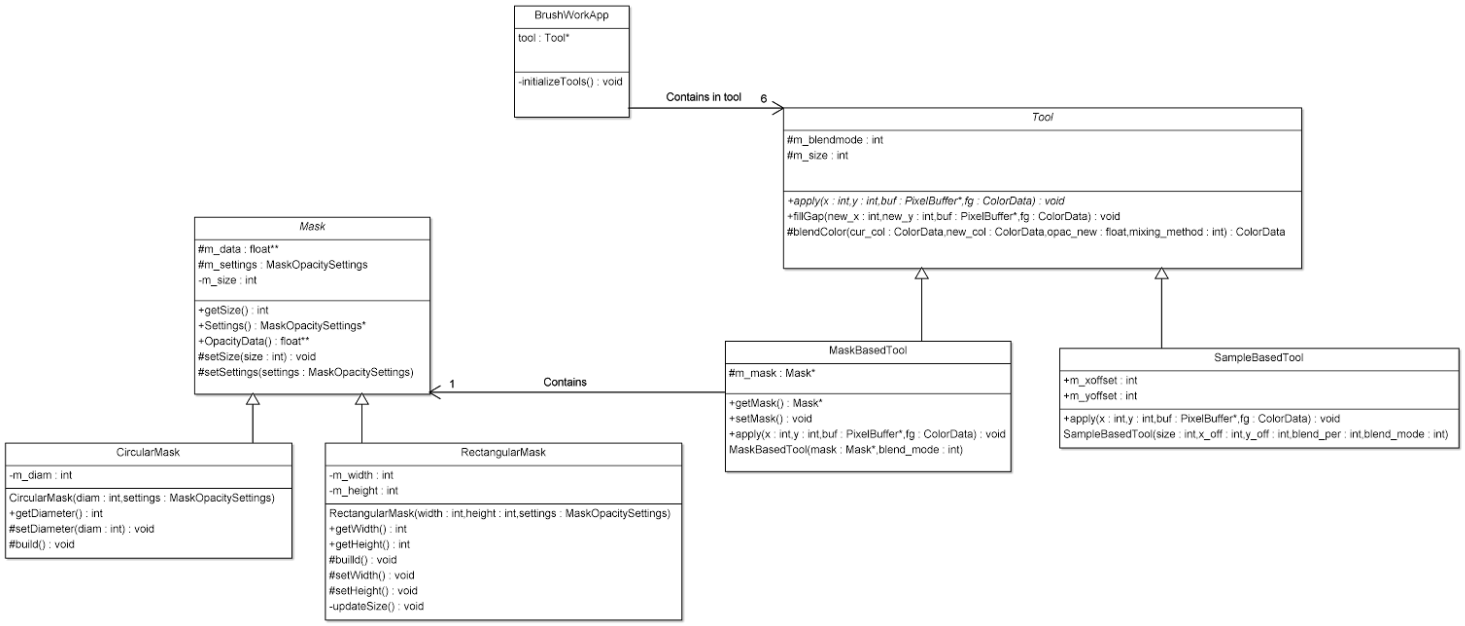

Figure 1: Simplified UML Diagram depicting BrushWorkApp, and the Tool and Mask classes