

Assignment 0: Brush

Algo Due: September 12, 3:00PM in the 123 handin bin on the 2nd floor

Help Session: September 12, 5:00PM (algo handed back at help session)

Due: September 17, 11:59PM

1 Introduction

Computer graphics often deals with digital images, which are two-dimensional arrays of color data (the pixels on the screen). Although they are just data, these arrays can convey knowledge, emotion, or even beauty. There are plenty of ways to create a digital image: scanning a still photograph, capturing a screenshot of video, rendering a three-dimensional scene, drawing a picture using a “paint” or “draw” program, and even algorithmically generating an image from a function. But images are rarely perfect when they first hit silicon (or phosphor...). We often want to make them look better than the original. One common method of doing so is with a photo editing program. You just select the airbrush tool, pick a convincing color, and before you know it, your younger brother looks like Mr. Clean.

In this assignment, you will be writing a simple application that will sport a few different types of airbrushes, as well as have the capability to save the images you alter. Check out the demo in `/course/cs123/demo/brush` to get an idea of how your program should look and behave.

Writing brush will give you a gentle introduction to programming in C++ using CS123 support code. Memory management and object-oriented design will be crucial to your success. Also, you’ll get your feet wet with some graphics programming basics. Blending colors and drawing images will be introduced, and you’ll learn to cache calculations and tighten your loops to squeeze every ounce of performance out of our new PCs.

2 Requirements

If you have played with the demo, you’ll see that it pops up a gui with a canvas and controls in the menu at the top. Don’t worry – all the groundwork of setting up the GUI is taken care of for you. All you have to do is write a program that adds on to the provided GUI and allows the user to airbrush onto the canvas by clicking and dragging the mouse. You must implement three different airbrushes, and in doing so take into account different colors of paint, different paint flow rates, and different radii for your airbrush. Your airbrush must also incorporate the use of a mask (a copy of the airbrush distribution) as a mechanism for reducing the amount of computation you do.

2.1 Airbrush

All the airbrushes you are required to create are (or should be) circular in shape. They will differ from each other in the *distribution* of paint that they place on the canvas. The distribution describes how much paint is placed at a certain point based on its position relative to the point where the user clicks (since the airbrush will affect more than just the pixel the user clicks). Below are details about the distributions for this assignment. The first three of these are required (i.e. you must code them to get full credit). For the ambitious, there are two additional types of brushes which can be implemented and will earn you extra credit if done well. Keep in mind that you **must** implement your airbrush such that it makes use of a mask. That

is, you must store the distribution information of your brush in a buffer (in this case a buffer is an array) so that those values can be quickly accessed from the buffer rather than calculated every time you need to use them. The buffer must only update when a change in brush parameters forces you to do so. This will give you much faster results.

REQUIRED DISTRIBUTIONS

constant — This distribution will place an equal amount of paint at each pixel within the airbrush’s radius.

linear — This distribution will place a linearly decreasing amount of paint at each pixel as you move away from the center point. So at the center point, you will have full intensity of paint, whereas at *radius* pixels away from the center, no paint will be put on the canvas.

quadratic — This distribution will place a quadratically decreasing amount of paint at each pixel as you move away from the center point (i.e., the amount of paint used as you move away from the center decreases with the square of the distance). So, as with the linear distribution, the center point will have full paint intensity, whereas at *radius* pixels away from the center, almost no paint will be put on the canvas. There can be multiple ways to write the quadratic distribution, so your quadratic distribution doesn’t need to look exactly like the demo. (*Note:* Your quadratic function should at least satisfy the property that when the distance x from the center of the circle is zero, the value of your function is "1", when the distance x is halfway to the radius, the value of your function is "1/4", and when the distance x is greater than or equal to the radius of your brush, the value of your function should be zero.)

EXTRA CREDIT DISTRIBUTIONS

gaussian — This distribution will use an airbrush mask that uses a gaussian distribution (i.e., a bell curve). Not all that tricky once you have done the above, but spiffy nonetheless.

special — This distribution will be whatever you want. The demo has a “doughnut” brush as its special brush with some additional effects. For the special brush, you have great freedom regarding what you can do. It doesn’t even have to be an airbrush. It can, for example, *warp* the image rather than put paint on it. You could try writing a brush that twirls or smudges the paint already on the canvas, or some other bizzare modification of the pixels. However, we don’t want to see special brushes which are merely quadratic or linear brushes that didn’t work. We reserve the right to look at the equations for special brushes.

2.2 Paint

Colors on our canvas are represented by three 1-byte unsigned char values, one for each of red, green, and blue (r, g, and b for short). That means the each of r, g, and b can range from 0 to 255, where 0 means there is no contribution from that color, and 255 means there is full contribution from that color. This allows you to use about 16 million possible color combinations. One important question is “How exactly do I choose what colors to color the canvas with?” Well, the things you have to take into account are:

1. The distribution of the airbrush
2. The color of the pixels you are painting on
3. The color of the airbrush

You will want to somehow blend the colors on the canvas with the current paint color of the airbrush. How much of each color you use will depend on how far the pixel you are coloring is from the point where the user clicked, the distribution, the radius of the airbrush, and the paint flow. We are not going to give you many details about how to do this – one of the important things about this assignment (as well as the assignments to come), is that there is often more than one way to go about solving the problem. In any case, you should try to figure out your own way of tackling the problem, while maintaining results consistent with the demo.

3 Support Code

To start work on this assignment, you will need to first copy the support code to your directory (preferably a subdirectory thereof). The support code can be found in `/course/cs123/asgn/brush`. A Makefile is provided, and you should only need to modify it if you add more source objects (e.g extra brushes).

To help you get used to C++ on this first assignment, we have provided a set of skeleton classes to for structuring your program. You will need to make additions to almost all the files, including the header files. Don't expect to see many more skeletons like this in cs123. In the future, you'll be expected to come up with your own design to complete the assignments. You should be able to compile and run the skeleton project without any modifications. The skeleton consists of the following files:

```
main.cpp
BrushApp.[cpp, h]
BrushCanvas.[cpp, h]
Brush.[cpp, h]
ConstantBrush.[cpp, h]
LinearBrush.[cpp, h]
MyBrushCanvas.[cpp, h]
QuadraticBrush.[cpp, h]
moc_BrushApp.cpp
moc_BrushCanvas.cpp
```

3.1 Support Code

The CS123 support code library lives in `/course/cs123/lib/src`. We make it readable for you guys on purpose; take advantage of that fact. Also, you would be wise to browse the code online. Click on "documentation" from any page on the cs123 web site (<http://www.cs.brown.edu/courses/cs123>) to be taken to our online documentation. You can browse the sources and see the relationships between classes graphically.

3.2 main.cpp

This is the mainline of your program. It constructs instances of `BrushApp`, `BrushCanvas`, and starts up the GUI. You needn't worry about the details here, just know that it is necessary in order to create the GUI interface, and to listen for various mouse and keyboard events.

3.3 BrushApp.[cpp, h]

`BrushApp` is a class which constructs the GUI for you and connects the hooks to the rest of the logic. The GUI consists of various pull down menus and sliders to control the behavior of your brush. You do not need to worry about how the GUI is setup. If you are interested, you can look at Qt documentation (<http://doc.trolltech.com/4.3/>).

The user can interactively change various parameters for the airbrush including color, radius, flow, and distribution. All the sliders pass integers in the range of 0 to 255. It's not necessary to query the sliders for their values, because the value will be passed to the `BrushCanvas` everytime it is changed.

When the user selects one of the distribution radio buttons (constant, linear, etc.), one of the following values will be passed to the `newDist` function as defined by this `enum` in `Brush.H`:

```
enum BRUSH_TYPE
{
    CONSTANT,
    LINEAR,
    QUADRATIC,
    GAUSSIAN,
```

```
    SPECIAL
};
```

The “Brush Radius” slider controls how large the brush is. The value of the slider should correspond to how many pixels away from the center the mask extends. Note that the mask should always have an odd width and height; thus, if the radius is 1, the mask width and height should be 3, if the radius is 3, the mask width and height should be 7, et cetera. If the radius is 0, the mask width and height should be 1.

The “Paint Flow” slider controls how much paint is laid down when the brush is applied. For example, if the current brush is a constant distribution, and flow is set to 255, then the color being laid down should completely replace the existing color on the canvas. If flow is set to zero, then the brush should have no effect. Intermediate values should somehow modulate between the existing pixel color and the new pixel color. We are being purposefully vague here. It is up to you to implement a paint combination model that mimics reality. Your results should be consistent with the demo.

The color chooser is fairly self explanatory allowing you to select the color of your Brush. There are also several buttons in the file menu. “Save” and “Load” will prompt for a filename and then call the corresponding method in `BrushApp`. The “Quit” button will cause your program to terminate.

3.4 MyBrushCanvas.[cpp, h]

`MyBrushCanvas` is a subclass of `BrushCanvas`, which in turn is a subclass of `CS123Canvas`. This class represents the 2d array of pixels that compose your image. It has methods for getting and putting pixels, as well as directly accessing the window’s buffer for better performance. The canvas is also responsible for collecting mouse events.

The mouse down, drag, and up events should be defined to match the behavior of the demo. You will need to implement the methods in this class.

The important parts of the the header file for `MyBrushCanvas` is reproduced below:

```
class MyBrushCanvas : public BrushCanvas {

public:
    MyBrushCanvas();
    virtual ~MyBrushCanvas();

    //these functions are called when mouse events are detected
    virtual void mousePressed(int x, int y);
    virtual void mouseReleased(int x, int y);
    virtual void mouseDragged(int x, int y);

    void newRadius(int radius);
    void newFlow(int rate);
    void newBrushType(int brushType);
    void newColor(unsigned char red, unsigned char green, unsigned char blue);

private:
    Brush* m_brush; //pointer to the current brush
};
```

Also important is the `CS123Color` struct as defined in `CS123Common.H`

```
struct CS123Color {
```

```
    unsigned char b, g, r, a;
};
```

`mousePressed`, `mouseReleased`, `mouseDragged` will be invoked automatically when the user interacts with the canvas. These should kick off the painting process and should mimic the behavior of the demo. (`mouseReleased` is not necessary but we've provided the hook if you want to use it for your special brush).

`newRadius`, `newFlow`, `newBrushType` and `newColor` will be invoked automatically from the GUI. Update the settings on your brush as appropriate.

`setPixel` and `getPixel` are methods provided in `CS123Canvas` that allow you to modify the content on the canvas. They are typical accessor and mutator methods that receive and return `CS123Color` structs.

`update` will need to be called in order for any of your changes to display graphically. When you are done painting the canvas, you will need to call `update` to see the new canvas in the gui. You can call `update` with no parameters to update the entire canvas or pass it a specific rectangular sub region to update. The rectangular region is specified by passing the x and y coordinates of the top left corner and the width and height of the rectangle. For speed reasons, it is important to only update the region of the canvas that has been changed. See the CS123 documentation for details.

There is one other method that can be used to modify the contents of the `CS123Canvas`. It is `getData` which will return a pointer to the beginning of the buffer that stores the raster data array. Use of this function is not needed for a functional program, but can yield significant speed-ups. It is explained in greater depth in the **Extra Credit** section.

3.5 Brush.[cpp, h]

The `Brush` class is the superclass for all other brush types. This class is `pure virtual` because it does not have a definition for the method called `makeMask`. The term `pure virtual` in C++ has a similar meaning to the keyword `abstract` in Java. Any subclass of `Brush` must define the `makeMask` method if it is to be instantiated.

The `Brush` class is responsible for storing the various brush parameters such as color, flow and radius. It also stores a pointer to a mask which it will use to apply paint to the canvas. `Brush` does not know how to create these masks, but it assumes the job will get done if it calls the `makeMask` method on itself.

3.6 Brush Subclasses

The sole responsibility of the brush subclasses is to define the `makeMask` method. Once this has been done for each subclass, the correct `makeMask` method will be called through the magic of polymorphism.

Note that you cannot call `makeMask` in the superclass's constructor, because it is declared as pure virtual in the superclass. To generate the mask at the brush's construction, call `makeMask` in the subclass's constructor instead.

4 Warning!

You may have already run into this snafu before in your CS career but if you haven't *beware of integer division!* $255 / 256 = 0$. To avoid this problem, just cast one of them to `float`. Alternatively if one of the numbers is a constant, append ".0" or ".f" to the end of it. For example, "foo / 256" becomes "foo / 256.0" or "myInt / 256.f".

5 Extra Credit

It is one thing to write an airbrush, it is another thing to make it fast. Make yours fast. One of the techniques you can use to speed up the operation of your airbrush is to access the image memory itself, rather than

modifying each pixel individually using functions. To utilize this technique, you will need to set aside the `setPixel` and `getPixel` methods described in the support code section, and instead set your sights on the `data` method that we mentioned only briefly. `getData` will return a pointer to the beginning of the block of memory used to store the canvas. If you modify the memory directly, you can achieve much better results than if you called a function for every pixel. Just think: if you have a large radius, you may have to check and modify as many as 100,000 pixels, which comes out to over 400,000 function calls for just one click! Obviously this isn't very efficient. If you modify the memory directly, not only will you have better results, but you will be using the method most often used for fast raster graphics. (You will almost never see `put` and `get` methods being used for pixels.)

You may notice that `getData()` returns an `CS123Color*`. This is a pointer to the beginning of an array of `CS123Colors`. Each `CS123Color` is a 4-byte struct with a red, green, blue and alpha value ranging from 0 to 255. You can either access the members of each `CS123Color` using the `.r`, `.g`, `.b`, and `.a` fields, or you can *cast* the `CS123Color*` to an `unsigned char*`. The `CS123Colors` are stored in *row-major order*. Each consecutive `CS123Color` is one pixel to the right of the last one, and the wrap around at the border of the image to the left column of the next row. (Just like the characters on this page)

Warning: You should not use `SetData` at the end of your method, this function will replace the entire data on the canvas.

```
CS123Color *mypixeldata = someImage.data();
unsigned char *sameData = (unsigned char*)mypixeldata;

// test the blue value of the first pixel
if (sameData[0]==mypixeldata[0].b) {
    printf("This will always print out\n");
}

// test the blue value of the second pixel
if (sameData[4]==mypixeldata[1].b) {
    printf("This will always print out\n");
}

// test the blue value of the third pixel
if (sameData[8]==mypixeldata[2].b) {
    printf("This will always print out\n");
}

// test the green value of the third pixel
if (sameData[9]==mypixeldata[2].g) {
    printf("This will always print out\n");
}
```

Please note that image data is stored in B-G-R-A.

Another speed up is to use only integer math in the assignment. Attempting this can lead to incorrect rounding hell so *only attempt this if you know what you're doing and want a challenge*.

6 Handing In

To hand in your assignment, type `/course/cs123/bin/cs123_handin` brush at a shell prompt.

| | | |
|---------------------------|------------------------|----------------------|
| Handout: | Thursday, September 10 | |
| Algorithm Handin: | Saturday, September 12 | 3:00PM |
| Help Session: | Saturday, September 12 | 5:00PM, Location TBA |
| Electronic Handin: | Thursday, September 17 | 11:59PM |

In CS123, there will be two handins for each assignment. One of the handins is the typical electronic handin of your program. The other is a written handin due *before* the electronic handin. In fact, if you look at the chart above, you will see that the “Algorithm Handin,” as it is called, is due just 2 days after you get the assignment handout.

In this first handin you must describe the algorithms and design you will use for the actual programming component of the assignment. For each assignment, we will ask some specific questions, and may give you other guidelines about what we expect you to turn in. Immediately after the due date for the algorithm handins, we will make available a handout that answers those questions and generally describes the TAs’ suggestions on how to approach the program. You are free to use the approach that we suggest in the second handout, but we recommend using your own method if you feel at all comfortable with it. Because this second handout is released immediately after the algorithm handin is due, the algorithm handin **will not be accepted late**.

The algorithm handin is worth 10% of the total grade for an assignment. This is to get you thinking about the assignment early on, and to help you clarify your understanding of the important concepts of a program before you start trying to code. It is meant to help you determine how well you understand things, to help you understand them better, and to emphasize the importance of the concepts involved. **Don’t leave it until the last minute.**

Since this is the first assignment of the semester, it’s a good time to get in the habit of starting early. We only give you 7 days to complete this one, so even if it seems easy, it will still be a challenge to finish in time. Remember, the assignments build upon each other, so it’s important to keep up. Good luck.