Project 1 Brush

Introduction to Computer Graphics, Fall 2012

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, or 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 at **/course/cs123/bin/cs123_demo** 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 the CS123 support code. You will be introduced to basic graphics concepts like blending colors and drawing images, and you'll learn to cache calculations and tighten your loops to make your programs much more efficient.

2 Requirements

If you have played with the demo, you'll see that it pops up a GUI with a canvas and controls. 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 left mouse button. 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.

All the airbrushes you are required to create are 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.

2.1 Mask 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.)

2.2 Smudge Brush

In addition to the paint brushes described in the previous section, you are also going to implement a "smudge" brush which will result in a "paint smear" effect when dragged across your canvas. For this brush you will ignore the paint color selected by the user. You can use whatever kind of mask distribution you think works best (there is no correct answer): constant, linear, quadratic, or something else. We're going to give you the pseudocode for the smudge brush; you just need to implement it. The purpose of this exercise is to teach you a little more about memory management in C++.

There are two basic operations you will need to implement for the smudge brush.

1. **Pick up paint**: This operation copies the part of your 2d image that is currently under the mouse cursor. You should allocate a temporary buffer to be used as storage for this image fragment. Do not copy the entire image – just the part that's under the brush mask.

2. **Put down paint**: This operation blends the contents of the buffer you filled when "picking up paint" with the region of your canvas that is under the brush mask centered about the current mouse position.

Here's how the smudge brush works.

1. When the canvas is clicked, *pick up paint*.
2. When the mouse is dragged, *put down paint* and then immediately *pick up paint* again, taking the paint you just put down into account.

2.3 [Optional] Extra credit

We have provided two slots for extra brushes you may wish to create for extra credit. You have great freedom regarding what you can do. The distribution can be whatever you want; it doesn't even have to be an airbrush. In the demo, we have provided a *spiderweb* brush, which performs a line-drawing effect, as well as a *ripple* brush which introduces animation. You can rename the radio buttons in the GUI by editing mainwindow.ui. You can earn up to 10 points of extra credit, but we're going to favor *quality over quantity* here. We'd rather see you make one *really awesome* brush than two run-of-the-mill brushes.

You may notice that when drawing with a semi-transparent brush the results aren't quite correct, because the brush mask overlaps itself. To correct this, you may implement a seperate drawing layer that holds all the user's drawing until the mouse is released (i.e., the alpha blending is deferred until mouse-up). The selected opacity should be applied to the entire temporary layer during drawing.

3 Paint

Colors on our canvas are represented by four 1-byte unsigned char values, one for each of red, green, blue, and alpha (r, g, b, and a for short). 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. Similarly, the alpha value can range from 0 to 255. 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.

4 Getting Started

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 is located inside /course/cs123/src/projects. You will want to copy the entire projects directory to your own course directory to make changes:

```
cp -r /course/cs123/src/projects ~/course/cs123
```

A Qt project is provided. You can generate a Makefile by using qmake, and you can use Qt Creator to edit your code. Use qmake and make to build your project. You may need to make additions to many of the files, including the header files. Code carefully! Poor design and coding decisions now will come back to haunt you later. You will be using the same support code for the rest of the semester, as your projects build upon one another.

5 Support Code

The CS123 support code library is documented using Doxygen comments. You can browse the documentation online. Click on "documentation" from the docs section on the cs123 web site (http://www.cs.brown.edu/courses/cs123/docs.html) to be taken to our online documentation. You can browse the sources and see the relationships between classes graphically. For this assignment, these are the classes you will care most about:

5.1 settings.[cpp, h]

You don't need to edit this file. The static settings variable, declared in settings.cpp and settings.h, allows you to easily retrieve parameters set by the GUI interface. You will need to use the Brush settings in order to correctly implement this assignment. These include:

```
bool brushDockVisible;

int brushType;

int brushRadius;

int brushRed;

int brushGreen;

int brushBlue;

int brushAlpha;
```

The user can interactively change various parameters for the airbrush including color (including alpha), radius, 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 static *settings* object every time it is changed.

When the user selects one of the distribution radio buttons (constant, linear, etc.), one of the following values will be set in the *brushType* variable according to an enum defined in settings.h.

The "Brush Radius" slider, bound to the *brushRadius* setting, 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 "Alpha" slider, bound to the *brushAlpha* setting, 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.

5.2 canvas2d.[cpp, h]

Canvas2D will be your implementation of a 2-dimensional image canvas. You will be using this canvas for all projects in this course which require pixel-level manipulation of images, which are *Brush*, *Filter*, *Intersect*, and *Ray*.

The *Canvas2D* class you are given extends from *SupportCanvas2D*, which provides several key facilities for you:

- You can resize the canvas by calling *resize()*. The canvas is not automatically resized when the GUI resizes; rather, scroll bars allow you to view the entire image.
- Image load and save – *loadImage()* and *saveImage()*
- Marquee selection via right-mouse drag – you can get the two corners of the selection by calling the *marqueeStart()* and *marqueeStop()* functions
- Display buffer to avoid threading issues
- Mouse events – *mouseDown()*, *mouseDragged()*, and *mouseUp()*
- Access to the raw image pixel data with *data()* – keep reading for more on this

5.3 Dealing with raw image pixel data

In this assignment, you are required to operate directly on pixel data, rather than relying on convience methods. There are several reasons for this requirement. One is that keeping track of array indices is a common theme in computer graphics, so becoming comfortable dealing directly with arrays will help greatly in the long run. The other is that speed is paramount in computer graphics, and operating directly with pixel data is efficient. Though speed may not be a factor in Brush, it will be much more important in later assignments. To get the raw image data, you will need to use the *Canvas2D::data()* method. *data()* will return a pointer to the beginning of the block of memory used to store the canvas. In graphics, you will almost never see put and get methods being used for pixels.

You may notice that *data()* returns an BGRA*. This is a pointer to the beginning of an array of BGRAs. Each BGRA is a 4-byte struct with blue, green, red, and alpha values (in that order) ranging from 0 to 255. You can either access the members of each BGRA using the .b .g, .r, .a fields, or you can cast the BGRA* to an unsigned char*. The BGRAs are stored in *row-major order*. Each consecutive BGRA 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)

5.4 Memory management

We've spent a lot of time ensuring that our support code doesn't leak memory, and we expect you'll do the same. Be sure to *delete* any memory you new, and *free* any memory you malloc. Don't mix and match, and don't forget to delete or free! Stack allocation is simpler and more efficient than heap allocation, just keep in mind that stack space is much more limited than heap space. We'll be assessing hefty penalties at grading time if your program leaks memory.

5.5 Other files

There are many other source files that you will be building along with this project. While you don't need to worry about what they do (or don't do), feel free to explore. We've intentionally provided you with all the code you're going to get for this course right off the bat. You might not like something we've done or maybe feel restricted in some way; feel free to hack away to your heart's content! Just be sure to keep the support code intact for your future assignments. Back up your files often, or better yet, use version control.

You might also feel like some things are missing. That's because there *are* things missing! Feel free to extend the support code to provide whatever functionality you need. Since this course will get complex quickly, we recommend that you use the class and inheritance facilities that C++ provides. Good organization now will pay dividends very quickly. In fact, we think it's one of the best investments you can make at this point.

5.6 Warning about numerical precision

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 $foo/256.f$.

6 Handing In

In CS123, there are two separate handins for each assignment. The first is a written handin in which you answer questions about algorithms and design; this is typically referred to as an "algo". The other is the typical electronic copy of your program's source code.

6.1 Algorithm Assignment

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 this 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**.

6.2 Handing in your code

Before you hand in your code, write a brief "readme" file that explains any design decisions you made and any bugs your program might have. We will be more lenient while grading if we find a bug that was mentioned in your readme file. Call your readme file something like README_Brush.txt (you'll be using the same code for all your assignments this semester). When you are ready to turn in your assignment, type ***/course/cs123/bin/cs123_handin brush*** at a shell prompt.

Since this is the first assignment of the semester, it's a good time to get in the habit of starting early. Remember, the assignments build upon each other, so it's important to keep up. Good luck!