Assignment 1: Airbrush

Written Homework Due: Friday, March 20 (9:30 AM)
Programming Project Due: Friday, March 20 (5:00 PM)

1 Introduction

Computer graphics often deals with digital images, which are two-dimensional arrays of colored pixels (the dots on a 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, 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, geometric drawing tools, and have the capability to load and save the images you alter. Check out the demo to get an idea of how your program should look and behave (select "brush" on the `lab` menu).

Writing brush will give you a gentle introduction to programming in C++ using CSC321 support code. Memory management and object-oriented design will be crucial to your success. Also, you'll get your feet wet in graphics programming concepts. Blending colors, drawing images, and 2D graphical objects will be introduced. You'll learn to cache calculations and tighten your loops to squeeze every ounce of performance out of the PCs.

2 Requirements

If you have played with the demo, you'll see that it pops up a canvas as well as a panel chock full of controls. Don't worry, all that 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 to allow the user to interact with the canvas by clicking and dragging the mouse. You must implement three different airbrushes, as well as 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 calculations you do. You will then have the choice of either drawing lines and circles, filled polygons, or image filtering. You will also turn in an image made (or altered) with your airbrush.

In addition to creating an airbrush you must also implement **one** of the following:

- Render lines and circles using the scan-line rendering algorithm.

- Create filled polygons using the scan-line algorithm.
- Implement a general-purpose filter which can be used to blur or sharpen the image, or detect edges.

Implementing more than one of the above will be considered extra credit.

2.1 Airbrush

All of 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). Here 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, no paint will be put on the canvas. Note: There are two perfectly valid ways to make a quadratic brush.

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 (not all that exciting, really). 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 bizarre modification of the pixels. Make sure you document what your brush is suppose to do in the Read Me file.

2.2 Paint

Colors on our canvas are represented by three numbers, one for each of the red, green, and blue channels (r, g, and b for short). Each of r, g, and b can range from 0 to 1, where 0 means there is no contribution from that color, and 1 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 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. I am 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, and in any case, you should try to figure out your own way of tackling the problem, while maintaining results consistent with the demo.

2.3 Lines and circles

Draw lines and circles in the current color with the current line width/brush radius. You must use the scan-line algorithm to determine which pixels to turn on.

Lines: Input is the mouse down and mouse up points. Compute the slope of the line and determine in which direction (left-right or right-left) the line is running. What special case(s) should you check for when calculating the slope? When is the slope not valid? There are four possible variations of the scan-line algorithm (W or NW, W or SW, E or NE, E or SE). If you're clever, you only need to write two of them...

There are several methods for making "fat" lines; use one of the following two. Let r be the desired line width (use the current brush radius as the line width):

- Draw the boundaries of the fat line by shifting the start and end points in the direction perpendicular to the line. Then fill in the center section by scanning lines between the boundaries. This will give you lines with flat ends. This is a simplified version of the polygon drawing code (since the edges are always parallel).
- Similar to above, but round the ends (hint: use your circle drawing code).

Circles: Input is the mouse down (which gives the center of the circle) and the mouse up (which provides the desired radius). The radius is rounded to the nearest integer.

The simplest method for making "fat" circles is to repeat the circle drawing code r times for increasing and decreasing radii. This will most likely give imperfect results however because of small holes between consecutive circles. A better method is to draw a filled circle by connecting vertically and horizontally symmetrical points on the circle, then adjusting your code to leave out the center ring to produce the desired result.

2.4 Filled polygons

A polygon is defined by a list of vertices. The user specifies the vertices by clicking a series of points; the last point is assumed to link back to the first one to provide a closed boundary. In the demo, you place points by clicking with the left-mouse button and place the final point by clicking the right-mouse button.

Use the scan-line algorithm to fill the polygon using the currently selected color. You must correctly handle convex, concave and complex polygons.

2.5 Filtering

This option is somewhat more difficult because we will not be covering the material in class before the assignment is due. However, the concept is a fairly straightforward extension of the mask idea, and a great deal of material can be found on the web.

Here is the basic idea: Define an $n \times n$ mask, where n is odd. Place the mask over a pixel (x, y). Calculate the new pixel value at (x, y) as follows:

$$\sum_{i,j \in [0,n-1]} C_{x+i-n/2, y+j-n/2} M_{i,j}$$

where $M_{i,j}$ is the value of the mask and $C_{x,y}$ is the color at pixel (x, y). Note that the new color at pixel (x, y) will be a *blend* of the colors of the nearby pixels — the bigger n, the more pixels are involved.

Although M can theoretically be anything, there are several M's which are interesting.

- Blur: $0 \leq M_{i,j} \leq 1$, with $\sum_{i,j} M_{i,j} = 1$, with the values of $M_{i,j}$ decreasing towards the boundary (i.e., a "bump"). You can use your linear, quadratic, or Gaussian distributions, provided you *normalize* them first by dividing each element by the sum.
- Edge detection: In this matrix, the middle value is positive, and the outer values are negative and sum to the negative of the middle value. This matrix looks for horizontal and vertical edges; if the -1's were on the diagonals, it would highlight diagonal edges.

$$\frac{1}{5} \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

- Sharpening. This is the result of adding the edges back into the original image — so instead of replacing the color at pixel (x, y), calculate the edge value and add it to the color at (x, y).

User interface: The user selects a rectangular region of the image by clicking on two corners. Your program should filter only the part of the image inside of the rectangle. Be careful about bounds checking.

Implementation issues: Unlike the airbrush, you can't just calculate the value at the current pixel and replace it, since you'll want the *original* pixel value when calculating the neighboring pixel. This means you must create a scratch image space, compute the new colors and put them in the scratch space, then copy the results back when you're done.

3 Support Code

You have already copied and compiled the shell support code. For this assignment you will be filling in the `MyBrush` class. A dialog allowing you to change the current color, the brush radius, and the flow rate already exists in `BrushInterface.fld`. You will need to add code to react to the changing parameters in the dialog.

The image itself is an array of colors stored as unsigned chars. (If the image is n by m, and each pixel is represented by an Red Green Blue triplet, how big is this array?) The `BrushUI` class has methods for getting and setting pixels, which take care of indexing into the unsigned char array

and converting from floats (range 0...1) to unsigned chars (range 0...255). The data can be accessed directly through the `pixelData` member variable.

The get and set pixel methods of the `MyBrush` class represent colors in the range 0..1 ((0,0,0) is black, (1,1,1) is white). The dialogs produce data in the range (0..1).

The mechanisms for reading and writing images (File→load image and file→save) are already taken care of and are available from the File menu.

`MyBrush` has an enumerated constant (BrushType) which represents the brush type:

```
typedef enum {
   BRUSH_CONSTANT = 0,
   BRUSH_LINEAR = 1,
   BRUSH_QUADRATIC = 2,
   BRUSH_GAUSSIAN = 3,
 } BrushType;
```

The GUI: The dialog for the brush assignment (and all the other dialogs) are created using a program called *fluid*, located in the `csc321files` directory. Fluid reads in a `.fld` file and automatically produces `C++` code (`BrushInterface.`). *You should never edit these files directly* — use Fluid. The interaction between the Fluid widget and your code is as follows. The GL window inherits from `MyBrush` — therefore, every widget in the dialog has access to the public methods of `MyBrush` through the member variable `myBrush`. Each menu button, widget, etc., has a callback (under the `C++ tab` in Fluid) which can access `myBrush` — for example, the callback for all of the brush items calls `myBrush.changedBrush()` to let the brush know it needs to update the mask.

In addition to these callbacks, `MyBrush` is passed a pointer to the widget dialog, which is kept in the member variable `brushUI`. Access to the data in the widget is through `brushUI` via the methods that have been added to the widget (using Fluid), such as `getBrushType`.

Whenever the brush type is changed, the `changedBrush()` method is called. The `changedBrush` method can then access the data in the widget through the member variable `brushUI`. The methods are pretty self-explanatory. The `getColor()` method returns a `Color`, the `getRadius` method returns the radius, etc.

All drawing code should be robust with respect to clipping. If the user drags the mouse outside the canvas, your application should not crash. The `getPixel` and `putPixel` methods do not do any bounds checking, other than the out-of-bounds checking provided by `stl::vector`. The questions describe how to write a check function that "disappears" in optimized mode; you must implement and add this check function.

To keep your code fast you will want to limit the amount of checking that *has* to be done, i.e., in your optimized version you don't want to be checking if you're out of bounds for every pixel. Try to restrict the region of pixels you will read from/write to as a pre-process.

One final, important issue is the ability to have the airbrush slowly deposit more paint on the canvas if the mouse button is held down. This is what the `draw_callback` method on `MyBrush` is for. This gets called every so many hundredths of a second.

The mouse interaction is handled for you in the `handle` method. The mouse locations are stored in `mouseDown` (e.g., **the x, y coordinates of the mouse location can be accessed by `mouseDown`[0] and `mouseDown`[1]**) and `mouseDrag`, and the correct `draw` routine called on mouse up. You should use the current brush color and brush radius as the current drawing color and thickness parameter (stored in `brushUI`).

The data for the `drawPolygon` method is stored in the member variable `polygon`. You should assume that the last vertex is connected to the first one in order to always have a closed polygon boundary.

4 Making a nice UI

You'll notice that the demo has a preview of the appropriate primitive as you drag the mouse. This is rendered on top of the image using OpenGL. When the user releases the mouse button, the corresponding call is made to `drawLine, drawCirle,` etc. You must add in the appropriate OpenGL calls to provide feedback; if you don't like the demo's feedback, you're free to try your own, just document it in the Read Me file. The draw routine is in `MyBrush_UI.cpp`, and is the only routine in that file you should have to edit.

The OpenGL commands you need are the following:

- `glColor*` Set the color of the line/points etc. `glColorfv( &Color[0])` .
- `glBegin(...)` `glEnd()` brackets drawing. Inside, `glVertex*` draws a vertex.
- Stippling: You need to enable stippling with `glEnable(GL_LINE_STIPPLE)` and set the stipple mask `glLineStipple( 1, 0xF0F0)` is what is used in the demo.

5 Coding style

- **Memory management:** I highly recommend using STL to do your memory management, both for this assignment and the others. You will lose points for excessive allocation/deletion, for allocating more memory than you need, and not correctly deleting memory when you're done with it.
- **Efficiency:** You do not want to perform bounds checks for every pixel you visit (the debug check doesn't count). It's better to do a bit of work up front and correctly calculate the start/stop values for your `for` loops.

  Once you have the basic routine working, you can try to make it faster. There is a clever way to avoid doing any multiplication to calculate the index in the inner loop; what debug check can you put in to make sure you're still not going out of bounds? Please include a complexity analysis of the draw routine in your Read Me file; it should count the number of adds, multiplies, and assignments.
- **Correctness:** Since this is graphics, we can cheat a bit and just check that the algorithm, e.g., handles the corners of the image correctly by painting them. Similarly for polygons with repeated points, circles that fall off the edge, etc.

6 Extra Credit

Making a Gaussian brush counts as extra credit. Making a spiffy brush counts as extra credit as well. Doing more than one of the three options counts as 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 `putPixel()` methods, and instead set your sights on the raw data in `pixelData`. 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 100000 pixels, which comes out to over 400000 function calls (and

index calculations) for just one click! Obviously this is not 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). The memory used by the canvas has a very simple format: each pixel is represented by three bytes. The first byte is red, the second green, and the third blue. The pixels are found in row-major order; that is, stored moving horizontally across a row, then the second row is stored, the third, and so on. If you need some more clarification, check out the file `Mybrush.h`, and look closely at how the access methods are written. Your speed-up routine should not be a simple flattening-out of your current code — it should take advantage of the linear nature of the data to do clever indexing. You should still perform the numerical calculations in floating point and not cast to unsigned char's until the end.

Programming tip: If you decide to speed up your program in this manner, save your original, working code (as, say `drawBrushSlow`) so you can compare the results against it. Also, I will want to verify that you have the correct checks in the (hopefully) correctly working slow code. You should, again, write check functions.

7 Handing In

Upload your zip file to Moodle as you did in Assignment 0. Don't forget to include the executable, an image, and the ReadMe file. Make sure the executable compiles and runs on a machine in one of the labs. Make sure your handin does not include .o/.obj files.

The ReadMe file should include:

- A description of any additional classes, methods, or files you have added.
- Any known bugs.
- A brief description of your mask implementation. Do not just include the header file — this should be in English with good grammar and complete sentences.
- A brief description of any extra credit. See above.
- Which of the three additional drawing routines you implemented.
- Complexity analysis of your main draw routine
- Names of people with whom you worked and any web sites, books, or other resources that you used (if any).

In CSC 321 there are two handins for each assignment. One of the handins is the typical electronic handin of your program. The other is a written homework due *before* the electronic handin. The homework is due at 9:30 AM on the due date; the answers will be posted and discussed at the beginning of morning class. Because the solutions are released immediately after the written homework is due, the written homework **will not be accepted late**.

In the written homework you must describe the algorithms and design you will use for the actual programming component of the assignment. For each assignment, I will ask some specific questions and may give you other guidelines about what I expect you to turn in. I will make available a handout that answers those questions and generally describes my suggestions on how to approach the program. You are free to use the approach that I suggest in the second handout, but I recommend

using your own method if you feel at all comfortable with it. I also recommend looking at the support code and experimenting with it as soon as possible so there are no surprises when you go to code up your algorithms.