CS 248
Introduction to Computer Graphics
Stanford University

Apostolos (Toli) Lerios
(edited by Marc Levoy)
Spring, 1995, Handout #7

# Paint programs and tinting brushes

This is an outline of suggestions and pointers for programming assignment #1 .

## 1. Picking a color.

There are several ways for the user to select the current color **C** of the brush:

- Move the cursor over the image and click on an image pixel. The color of this pixel becomes the current color.

- Select the color in some color space (RGB or HSV) by typing coordinates or by moving correspondingly labeled sliders.

- Select a color from a standardized palette, either by typing in the color name (e.g. "PANTONE 333 CVC") or by clicking on a color sample.

- Click on some point in the visualization of a color space (RGB or HSV).

For this assignment, you are required to implement sliders to select colors. You may implement additional methods as you like.

## 2. Your main loop.

Your program should be event-driven. For each canvas, you register a function in your code that is called whenever the user is interacting with the image displayed in the canvas window. "xsupport.h" contains all of the messy details on when your routine is called. Once the program flow enters your routine, you should take the following steps:

1) First, since your code is responsible for drawing a cursor on the screen, you want to re-draw the portion of the image that was covered by the cursor *possibly* drawn earlier in step (4) below. You would not have drawn a cursor in step (4) if the last time your callback was called, the mouse was outside the canvas.

2) If the button is down, then you modify each image pixel under the brush on the basis of the current color. This last sentence is quite vague, so let's clarify what we mean:

- A brush is a rectangle whose side lengths are measured in pixels. It is often more convenient for brushes to have odd length sides. The center of such a brush is the rectangle's center.

- A brush is positioned over an image using the mouse. When the mouse is at image coordinates (x,y), we think of the brush as positioned so that its center appears right over the center of the image pixel at (x,y).

- The current color is some color **C** that the user has selected during some previous interaction with the paint program, probably using one of the methods described in the first section.

- There are several ways to modify the image pixels under the brush. Section 3 will survey several of these.

3) If the image was modified in step 2, display the modified portion of the image on the screen.

4) Draw a cursor on the screen at the current mouse position, if it is within canvas bounds.

There is a problem with this approach: you cannot draw directly on the screen using xsupport (our drawing package); instead, you can only draw on a canvas data structure (a rectangular array of pixels), and display arbitrary portions of it on the screen. Hence, in order to display a cursor, you need to draw it right on top of the image in your canvas structure, and then display the modified portion of the canvas. But then, you are destroying the image itself! Well, the answer is to store the image pixels under the cursor in temporary storage prior to step (4), and restore both the canvas and the screen in step (1).

Remember that the above is a brief guide to direct your thinking and possibly the very first stages of your implementation. There are details left for you (e.g. efficiency considerations) to discover and implement.

## 3. Modifying the color of image pixels: basics.

The simplest method to modify the image pixels under the brush is *overpainting*. In this case, the image pixels under the brush are all set to the current color **C**.

*Tinting* is a more interesting method: in this case, we must first represent **C** in some color space. For example, let's represent **C** in RGB: $(C_r, C_g, C_b)$. Also, we must represent the image pixels under the brush in the same color space; so, if **P** is the color of some image pixel, say the pixel at coordinates (x,y) let us use the notation $(P_r, P_g, P_b)$ for the RGB representation of **P**. Tinting is an operation that requires one more parameter in addition to **C** and **P**. This parameter is a user-defined scalar $\alpha$, often called the *weight* or *strength*, ranging between 0 and 1. The tinting operation can now be defined: the new color of the pixel at coordinates (x,y) is

$$(1-\alpha)(C_r, C_g, C_b) + \alpha(P_r, P_g, P_b)$$

For example, if I pick 0.1 for $\alpha$, and the current color is light gray (0.8, 0.8, 0.8) in RGB, then an image pixel positioned under the brush and having a pure red color, represented in RGB as (1, 0, 0), will acquire the new color

$$\begin{aligned}&(1-0.1)\,(0.8, 0.8, 0.8) + 0.1\,(1, 0, 0) =\\&(0.72, 0.72, 0.72) + (0.1, 0, 0) =\\&(0.82, 0.72, 0.72).\end{aligned}$$

A more flexible version of tinting allows the user to modify any subset of the color components of the pixel. Extending our previous example, instead of setting the new pixel color to (0.82, 0.72, 0.72), modifying thus all the RGB components, we could set the new pixel color to (0.82, 0, 0.72), modifying only the R and B components. Of course, the user should control which components are modified.

Tinting can also be done in HSV space, whereby you get very different results. For our previous example, the representation of the current color **C** in HSV space is (showing Hue in degrees, and then Saturation and Brightness in the standard 0 to 1 range) (0, 0, 0.8). **P** in HSV is represented as (0, 1, 1). So the new color for the pixel at image coordinates (x,y) is

$$
\begin{aligned}
&(1-0.1)\ (0, 0, 0.8) + 0.1\ (0, 1, 1) = \\
&(0, 0, 0.72) + (0, 0.1, 0.1) = \\
&(0, 0.1, 0.82).
\end{aligned}
$$

The new color is represented in HSV. The RGB representation of this color is (0.82, 0.74, 0.74) which differs from the previous result. The difference can become much more dramatic as we change the distance of **C** and **P** in each color space, and as the value of $\alpha$ moves closer to 0.5.

Note that the way you pick the current color **C** (see section 1), and the choice of color space for tinting are completely **independent** decisions. In assignment #1, we require that you pick colors using sliders, but we have not specified in which space; that is up to you. You are required, however, to performing tinting in HSV space.

**4. Modifying the color of image pixels: intermediate.**

In order to get more interesting painting effects, we can refine one of our previous steps. Earlier we said that we "modify each image pixel under the brush on the basis of the current color". How about modifying some, but not all the pixels under the brush? We can do this if we allow the user to specify a *mask*. A mask is a square array of bits, just as big as the brush. Hence, using a mask, we can associate with every image pixel p under the brush the number 0 or the number 1: a 0 in the mask means "do not modify the image pixel p" while a 1 in the mask means "modify it". Painting using masks is called *mask-driven* painting.

Since the choice of applying a mask is independent of the method by which we modify image pixels, we can get any combination of these two options: simple overpainting, mask-driven overpainting, simple tinting, or mask-driven tinting.

Masks can be used to extend other steps of the painting process as well. Binary masks, i.e. rectangular arrays of bits, are not very versatile, so most programs use continuous 2-d functions as masks, called *weighted masks*. Such a mask associates with each point of the brush a real number in the range 0 to 1. The function should have high values in the center of the brush and should fall off smoothly to zero on the perimeter of the brush. We encourage you to experiment with different weighting functions, e.g. an exponential, or a pyramid, or a cone with an apex of height 1 over the brush center.

A good (and required) use of a weighted mask is to modulate the $\alpha$ value for tinting. Different image pixels will be under different parts of the brush, which will be associated with different mask values; the mask value defines the weight $\alpha$ used for tinting. Hence the name *weighted* mask.

## 5. Modifying the color of image pixels: advanced.

Filters are very interesting operators used extensively for image processing, as well as in computer-assisted painting in order to get some intriguing painting effects. In the previous section, we used masks to modify some of the parameters we had employed in our initial discussion of painting (section 2): we experimented with modifying the effective area of a brush, and modulating $\alpha$, both via masks. The next step is to modify the current color **C**. As we said earlier, **C** is usually directly specified by the user, and does not vary as we process the image pixels under the brush. However, this does not need to be the case. In *filter painting,* the current color varies between different image pixels under the brush, and is independent of the user's color selection. In particular, if

- the color of the image pixel with coordinates (u,v) is given by p(u,v) in some color space (usually, but not necessarily, RGB),
- the user has specified a 3x3 matrix M; let M(i,j) denote the entry in row i and column j of this matrix,

then the <u>current color</u> **C** for the image pixel with coordinates (x,y) is the sum

$$\begin{array}{llllll} M(1,1)*p(x-1,y-1) & + & M(1,2)*p(x,y-1) & + & M(1,3)*p(x+1,y-1) & + \\ M(2,1)*p(x-1,y) & + & M(2,2)*p(x,y) & + & M(2,3)*p(x+1,y) & + \\ M(3,1)*p(x-1,y+1) & + & M(3,2)*p(x,y+1) & + & M(3,3)*p(x+1,y+1) & \end{array}$$

divided by the sum of M's entries (if non-zero). The formatting of the expression above suggests what we are doing. Essentially, we are positioning the matrix over the image, with the matrix center over pixel (x,y); then, we multiply each matrix entry by the image pixel under it, and finally take the sum of those products. We conclude by dividing by the sum of M's entries, if it is non-zero; if zero, we just skip this last step.

Do not confuse the current color **C** of a pixel with its new color; the current color, now varying over different image pixels, is used to find the new color of each pixel, using tinting or overpainting or any other basic method you can contrive on the basis of sections 3 and 4.

Filter painting as it stands is very restrictive. However, it is not hard to figure out its immediate extension: pick any matrix, square or rectangular! For 3x3 matrices like above, the following are good examples to experiment with:

$$\begin{array}{ccc} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{array} \qquad\qquad \begin{array}{ccc} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{array}$$

Blurring. Vertical edge detection.

While we no not require you to implement filter painting in this assignment, we encourage you to play with it, and you are welcome to use it as one of your fancy brushes for bell/whistle #2.