Problem Set 3
CMSC 427
Assigned October 23, 2007, Due November 6, 2007

Introduction: This problem set will focus on writing a program to morph one image into another. You will implement the approach described in: "Feature-Based Image Metamorphosis," by T. Beier and S. Neely, *Computer Graphics*, 26(2) 25-42 (available on-line at: http://www.cs.princeton.edu/courses/archive/fall00/cs426/papers/beier92.pdf).

You will be given some basic code, as well as an executable for finished code. Some things to notice about this code:

- BMPImage.cpp and .h define an image class for bmp type images. This includes functions to read in an image and to save an image. Note there is no function to copy images, so if you want to call a function using an image as an argument, you should pass an address, so that it doesn't need to be copied.
- texture.cpp and .h contain functions that load an image from a file and turn them into a texture.
- main.cpp contains code that reads in two images, and provides an interactive tool that allows you to specify corresponding line segments in the two images. These will be needed to create correspondences for morphing. To use this interface, try the following operations:
    - Click left on an image to start a line segment. Drag the mouse to specify the end of the segment.
    - 'u' undoes the last line segment on the left image. 'v' does this for the right image.

You will modify this code to use these line segments to morph between the images. In addition to these features, the executable contains a couple of other features:

- 'm' causes the left image to morph into the right, using the line segments that you've specified in each image. Note that the order and direction used to create the line segments matters. The first line in the left image matches the first one on the right, and the first points of each segment match.
- 'f' causes the left image to fade into the right.
- 's' causes all the images in the morph to be saved in a directory called 'Results' (you should create this directory ahead of time.
- 'r' replays the morphing using the images saved in 'Results'. This is useful if you are morphing larger images, which my program will not morph in real time. Replaying gives you a better sense of the morph.

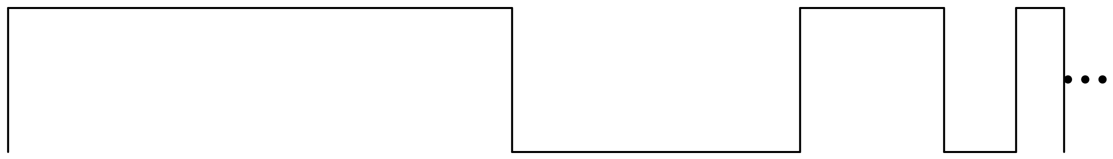
1. **Fade (15 points):** Implement a fade operation. This is begun by typing 'f'. The image on the left of the screen should gradually fade out, being replaced by the image on the right. Let's call the image on the left I, and the image on the right J. If the fade lasts n units of time, then after k units of time pixel (x,y) will have intensities given by (1-k/n)*I(x,y) + (k/n)*J(x,y). You might try, for example, generating a fade that lasts 4-6 seconds, at 30 frames a second.

2. **Morphing (50 points):** In this part, you will implement morphing. This problem will be satisfied by a complete morphing program, but I'm going to suggest that you break this into parts, doing first (a), then modifying this to satisfy (b), and then completing (c). You can receive partial credit for turning in a program that implements (a) or (b).
    a. **Interpolate Lines (10 points):** When the user types 'm' morph lines drawn in the left image into the lines drawn on the right. That is, as a line morphs from the left to the right the positions of its end points are linearly interpolated from their start position to their end position. When you finish this part, the behavior of your program should be the same as in the executable in terms of how the lines move, although the underlying image needn't change.
    b. **Morph the left image (15 points):** Now change the underlying image so that the left image is morphed as the lines move. The interpolated lines from part (a) are used to define coordinates in the new image you generate, and pixel intensities are transferred from the left image to this image. This is like full morphing except that the image you generate isn't a combination of the left and right images, but just the left image. So, for example, with proper placement of lines, you can cause the left image to be compressed, or distorted.
    c. **Morphing (25 points):** Now implement the whole morphing program of Beier and Neely. This means applying your code from part (b) to both the left and right images, and then displaying an interpolation of the two, similar to that produced with fading.
3. **Generate image or video (5 points):** Use your code or mine to generate an interesting or compelling image or video (which can be displayed with 'r'). If you want to use your own code, you should implement 's' and 'r'. A prize will be given to any notable results. In addition to sending this to Carlos, please email to me (djacobs at cs) a zip file containing your start image, end image, and your morphed image or video. **(5 points extra credit for implementing 's' and 'r', up to 5 points extra credit for creative images).**
4. **Extra Credit:** Integrate the results of morphing into your game. For example, create a cube with a texture-mapped image that morphs into something else when you get close to it.
5. **Extra Credit:** In the morphing program, to fill in a pixel in the new image, we compute values (x,y) and use these to get the value of a pixel in the old image. Typically, x and y will not be integers. You can handle this by rounding off x and y. For extra credit, implement subpixel interpolation. So, for example, if we want to get a pixel value at (6.2, 7.3) we should compute this value as a weighted average of the pixel values at (6,7), (6,8), (7,7), (7,8). If you implement this, document clearly where in your code it is done.

A maximum of 20 extra credit points are available.

Paper and Pencil Exercises:

1. **Smoothing (10 points)** Suppose we smooth a function f by averaging nearby points. So, we get a smoothed version, g(x) by taking g(x) = (f(x-.01) + f(x) + f(x+.01))/3. (You can use a calculator or computer for these problems).
   - **a.** Suppose f(x) = cos(x). Calculate the value of f(0) and g(0).
   - **b.** Suppose f(x) = cos(kx). Make a plot of g(x)/f(x) for at least 10 varying values of k.
   - **c.** Suppose f(x) = cos(kx). Prove that g(0) >= g(x) for any values of x and k.
   - **d.** If you add together two cosine waves of the same frequency but different phases you will get a cosine wave with the same frequency but with a different phase and amplitude. For example, cos(x) + cos(1+x) = 1.76*cos(1/2 + x). In general, cos(a + kx) + cos(b + kx) = mcos(c + kx) for some c and m. Supposing that f(x) = cos(kx), use this fact to prove that g(x) = g(0)*cos(kx). That is, you are showing that g(x) is a scaled version of f(x), and that the scale factor is g(0).
2. **Sampling (10 points)**
   - **a.** Suppose f(x) = cos(x). We sample f(x) at the points x = 1 and x = 1.1. If we use linear interpolation between these two points to estimate the value of f(1.025), how much error will there be between this and the true value?
   - **b.** How much error will there be if we use bicubic interpolation?
   - **c.** What would the error in part (a) be if f(x) = cos(5x)?
   - **d.** Suppose f(x) = cos(kx). We sample f(x) at the points x = 1 and x = 1.1. If we use linear interpolation between these two points to estimate the value of f(1.025), give a value of *k* for which the error between the real and interpolated values would be as large as possible.
3. **Anti-aliasing** Consider a 1D image that is like something we would get looking at a checkered floor vanishing into the distance. We will describe the image as f(x) for 0 <= x < 1, with f(x) = 255 for x < 1/2, f(x) = 0 for 1/2 <= x < 3/4, f(x) = 255 for 3/4 <= x < 7//8, …



   - **a.** Suppose we divide the image into pixels of length .2, starting with a pixel going from 0 to .2, then .2 to .4 and so on. To produce an image, we sample each pixel in the middle (.1, .3, …). What will the resulting image be?
   - **b.** Suppose that instead, we supersample each pixel with five uniformly spaced points, so that each pixel is the average value of these samples. What will this image be then?

c. Suppose we smooth the image with an averaging filter that has a width of one pixel, and then sample it as in (a). What will the image be then?
d. Now let's consider an even simpler image, in which f(x) = 255 for 0< x < k and f(x) = 0 for k <= x < 1, for some k. Suppose we use the sampling method in (a), and try to reconstruct f using linear interpolation. What value of k will produce the greatest possible error at some location?
e. What about if we first apply the averaging filter in (c)?

Challenge problem: Suppose that we smooth a signal by averaging. Prove that this smoothing cannot increase the number of points *x,* at which *f(x)* is a local maximum value. Note that in some sense this is a measure of how "wiggly" the curve is, so that this shows that smoothing cannot make a curve more wiggly. **Hint:** Instead of thinking about how smoothing affects *f*, try thinking about how the inverse process would affect *g.*