Project 3
CSCI-4961/6961: 3D Computer Graphics

Due: Tuesday, November 7, 2006, 11:59:59pm

1 Overview

In this project, you are to write an image metamorphosis program. Given an input of two images, your program will allow the user to define a set of feature correspondence pairs between the two images, and produce a morphed image at the desired interpolation level. You will be implementing a popular method that was described by T. Bier and S. Neely in their paper "Feature-Based Image Metamorphosis" published at SIGGRAPH in 1992. You can see the Bier and Neely paper at `http://www.hammerhead.com/thad/morph.html`.

Please see the course web page at `www.cs.rpi.edu/~sakella/graphics/` for project updates and additional information.

2 Project Tasks

2.1 Morphing

We will use the same definition of morphing as in the Bier and Neely paper: "The combination of generalized image warping with a cross-dissolve between image elements. The morph process consists of warping two images so that they have the same "shape", and then cross dissolving the resulting images. Cross-dissolving is simple; the major problem is how to warp an image." For this assignment, you will be implementing the Beier and Neely warp. Their warping algorithm is discussed fully in the paper titled "Feature-Based Image Metamorphosis", available at `http://www.hammerhead.com/thad/morph.html`, and we summarize the key aspects of their algorithm.

To warp an image, the algorithm goes through the destination image pixel by pixel and samples the correct pixel from the source image. We first describe how to identify the pixel coordinate in the source image for each pixel in the destination image. Pairs of line segments (P_iQ_i defined relative to the destination image, $P_i'Q_i'$ defined relative to the source image) define a coordinate mapping from the destination image pixel coordinate X to the source image pixel coordinate X'. This mapping is defined by a weighted transformation function; the pseudocode for this function is reproduced here from their paper:

```
For each pixel X in the destination image
        DSUM = (0,0)
        weightsum = 0
        For each line PiQi
                calculate u,v based on PiQi and X
                calculate X'i based on u,v and Pi'Qi'
                calculate displacement Di = Xi' - Xi for this line
                dist = shortest distance from X to PiQi
                weight = (length^p/(a + dist))^b
                DSUM += Di * weight
                weightsum += weight
        X' = X + DSUM/weightsum
        destinationImage(X) = sourceImage(X')
```

where

$$u = \frac{(X - P) \cdot (Q - P)}{||Q - P||^2} \tag{1}$$

$$v = \frac{(X - P) \cdot Perp(Q - P)}{||Q - P||} \tag{2}$$

$$X' = P' + u \cdot (Q' - P') + \frac{v \cdot Perp(Q' - P')}{||Q' - P'||} \tag{3}$$

and $Perp()$ returns the vector perpendicular to[1], and the same length as, the input vector. The value u is the position along the line, and v is the distance from the line (see figure 1). The value u goes from 0 to 1 as the pixel moves from P to Q, and is less than 0 or greater than 1 outside that range. The value for v is the perpendicular distance in pixels from the line. a, b, and p are constants that can be used to change the relative effect of the lines. If a is barely greater than zero, then if the distance from the line to the pixel is zero, the strength is nearly infinite. With this value for a, the user knows that pixels on the line will go exactly where he wants them. As a increases, the warp will become smoother, but with less precise control. The variable b determines how the relative strength of different lines falls off with distance. If b is large, then every pixel will be affected only by the line nearest it. If b is zero, then each pixel will be affected by all lines equally. Values of b in the range [0.5, 2] are the most useful. The value of p is typically in the range [0, 1]; if p is zero, then all lines have the same weight. If p is one, then longer lines have a greater relative weight than shorter lines.

2.2 Steps

These steps provide a guideline:

1. First, write an OpenGL program to display two images (and optionally any predefined feature correspondence lines specified in the scene file), and allow a user to specify feature

[1] There are two perpendicular vectors; either the left or right one can be used, as long as it is consistently used throughout.

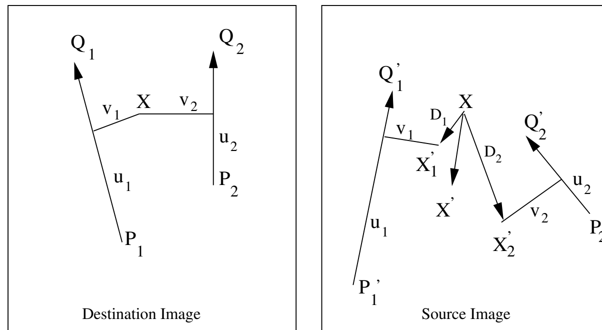

Figure 1: Multiple line pairs. X′ is the location to sample the source image for the pixel at X in the destination image. That location is a weighted average of the two pixel locations X1′ and X2′, computed with respect to the first and second line pair, respectively. Figure based on `http://www.hammerhead.com/thad/morph.html`.

correspondence lines between the images. Code for loading and saving the images in PPM format is available on the project web page. Use `glDrawPixels` to display the images. You may assume the images have the same dimensions.

2. Allow the user to add feature correspondence lines using the `glutMouseFunc` and `glutMotionFunc` callbacks. The following mouse functionality should exist:
   - `button == GLUT_LEFT_BUTTON && state==GLUT_DOWN`. Add the P endpoint of the feature line.
   - `button == GLUT_LEFT_BUTTON && state==GLUT_UP`. Add the Q endpoint of the feature line.
   - `button == GLUT_RIGHT_BUTTON && state==GLUT_DOWN`. Select the nearest endpoint, and allow the user to move the endpoint.
3. Allow the user to save the feature lines by use of a keyboard event. If the user presses the `s` key, save the current set of feature lines in a file called `myfeatures.txt`.
4. Allow the user to begin the metamorphosis by use of a keyboard event. If the user presses the `m` key, call your morphing function with the current set of user specified feature lines. Display the resulting image in a new OpenGL window or write the image to a file depending on the runtime arguments given.
5. Write a `crossDissolve` function to interpolate the two images. This function takes for input two images ($I0$ and $I1$) and the interpolation level ranging between [0.0, 1.0]. The interpolation level t allows for unequal weight in color selection ($I = I0(t) + I1(1 - t)$). A value of 0 implies the color comes only from the first image, a value of 0.5 is an even dissolve, and a value of 1.0 implies the color is only from the second image. The function returns a new image, which is the cross dissolve.
6. Write a `endPointInterpolate` function. This function takes for input two lists of feature lines and the level of interpolation t. It returns a new list of feature line segments. The new

list of feature line segments is created by linearly interpolating the endpoints of each line segment in the first list to the corresponding line segment in the second list by the value of t.

7. Write a function `warp`. This function takes for input two sets of feature pairs and an image (source image), and returns a new image (destination image). For each pixel in the destination image, run the Beier and Neely *Feature-Based Image Metamorphosis* algorithm mentioned above and presented in full detail at `http://www.hammerhead.com/thad/morph.html`.

8. Write the function `morph`. For input it takes the two images to morph, the corresponding feature lines, and an interpolation level. In pseudocode, it will look like this:

```
Image morph(Image I0, Image I1, FeatureLines I0Lines,
            FeatureLines I1Lines, double t){
 List ILines; // The intermediate Feature lines

 Image intermediateImage0, intermediateImage1, returnImage;

 ILines = endPointInterpolate(I0Lines, I1Lines, t);

 intermediateImage0 = warp(I0, I0Lines, ILines);
 intermediateImage1 = warp(I1, I1Lines, ILines);

 returnImage = crossDissolve(intermediateImage0,
                                    intermediateImage1, t);

 return returnImage;
}
```

9. Your `morph` program should use command line arguments:
`morph <scenefile> <image1> <image2> -window`
`morph <scenefile> <image1> <image2> -ppm <imagefile>.ppm`
The first command displays the morphed image in a new openGL window. The second command writes the morphed image to a PPM format image file.

10. Finally, you should create a valid input scene file of your own in the same format, and submit it along with the two ppm images. The name of this file should be `myscene.txt`.

2.3 Input Scene File

The following information is included in the input scene file.
The 3 tuning parameters: a, b, and p
The interpolation amount t
Feature lines (optional)

If $t = 0$, the resulting image is exactly the first image. If $t = 1$, the resulting image is exactly the second image. For values between (0, 1), the resulting image is a linear combination of the two.

The optional list of feature lines is to allow consistent testing of your morphing programs. It allows one to specify a set of feature line correspondences without having to use a user interface.

2.4 Software Design

You should create a class to represent the feature lines. This class will contain the endpoints of the lines, and a color (for visually distinguishing the matching lines between the images). You may also find it useful to create a class for vectors. Create math helper functions to perform operations such as computing vector cross products and dot products, normalizing vectors, etc.

3 Grading

Your project will be graded for a total of 100 points as follows:

- 10 points for displaying the two images.
- 15 points for creating the user interface to specify feature lines.
- 45 points for producing the morphed image.
- 5 points for saving the feature lines.
- 10 points for an example scene file (with specified feature lines).
- 10 points for code structure, clarity, and documentation.
- 5 points for any special features or creative enhancements.

Example enhancements include: antialiasing, calculating a sequence of morphs and displaying a "flip-book" animation, allow morphing between images of different sizes, allowing deletion of feature lines, interpolate the center positions of each feature line, letting the user see the interpolated positions, etc.
Describe any enhancements or special features in your README file, and ensure that they do not interfere with the required project functionality.
Lateness policy: Please read the lateness policy in the course syllabus. Use your late days carefully, if at all.

4 Submission

The code must be submitted no later than **11:59:59 pm on November 7, 2006**. You will follow the same submission procedure as for Project 1. Specific instructions for handing in your code are provided on the course web page. Your source code must be readable and commented. The comments need not be extremely long, just explain clearly the purpose of each block of code.

You must submit a single zip file containing your README file, your `myscene.txt` file, your 2 images, your source code (source and header files), and if necessary, a Makefile to compile it.

Your submission should NOT include any object files or executable files. Your submission should not use `glaux.h` or Windows API files (e.g., `windows.h`). Every file should have your name in a comment line at the top. The README file should contain the following information: your name, email address, instructions on how to compile the code and run it, known bugs or limitations, any extra credit enhancements, and any other relevant information.

Proper submission is entirely **your responsibility**. Contact the TA if you have any doubts whatsoever about your submission. Do **NOT** submit your project via email. Be sure to keep copies of your files and **do not change them after submitting**. After grades are posted, you have exactly one week to resolve all problems. All grades are final two weeks after they are posted.

A project that does not follow the submission guidelines will receive a **10 point deduction**. Please observe the academic integrity guidelines for the course as described in the course syllabus.