Assignment 4: The Grand Metamorphosis

CS 148 Autumn 2012-2013

Due Date: Monday, October 22th, 2012 at 7pm

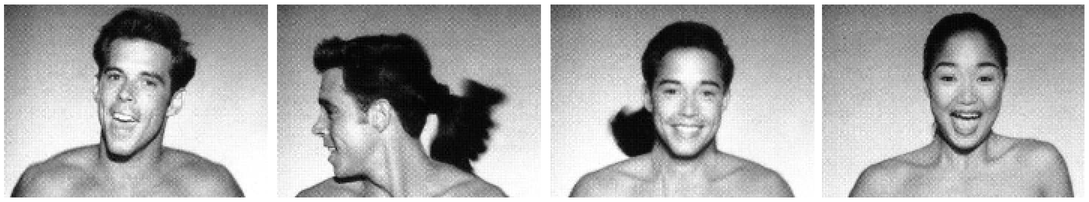


Figure 1: Snapshots of the morphs seen in Michael Jackson's music video, "Black or White"

Introduction The goal of this assignment is to learn the uses and importance of interpolation in computer graphics through the implementation of an image metamorphosis algorithm.

We will be undertaking a one-of-a-kind class project for assignment 4. The goal is to create a video of all the students in the class comprising of an unbroken sequence of morphs from each of you to the next, and you will be responsible for writing a program to generate the set of frames that morphs you into the next student on the list!

The morphing algorithm we will be using is described in the article Feature-Based Image Metamorphosis by Thaddeus Beier and Shawn Neely. It is the same algorithm used to generate the morph seen in Michael Jackson's music video, Black or White. A few frames from the video are shown in Fig. 1, and you can watch it in motion on YouTube here (starting 5m 30s into the video).

Your goal is to generate a morph from yourself to another student in the class (the one below you on the list provided by the course staff on the course website). You can access the picture of the student at `www.stanford.edu/~sunetid/sunetid` where `sunetid` is the SUNetID of the student. In order to compute a morph between the two images, you will need to mark key facial features on each image. The application

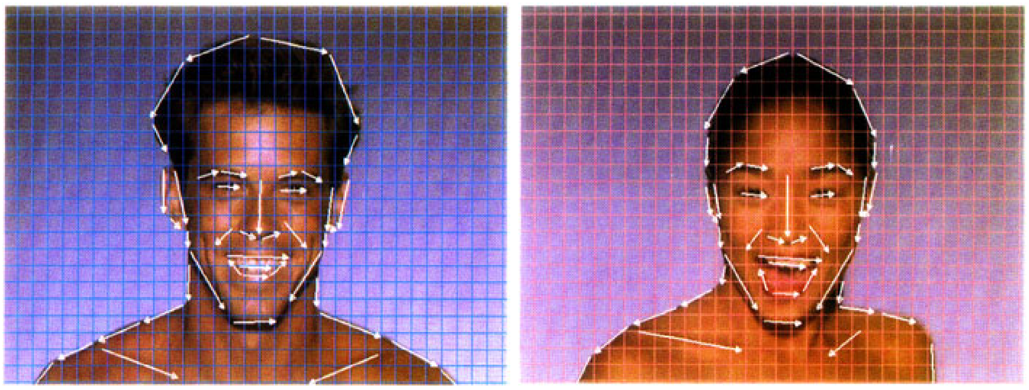


Figure 2: Two images with correspondence lines shown in white

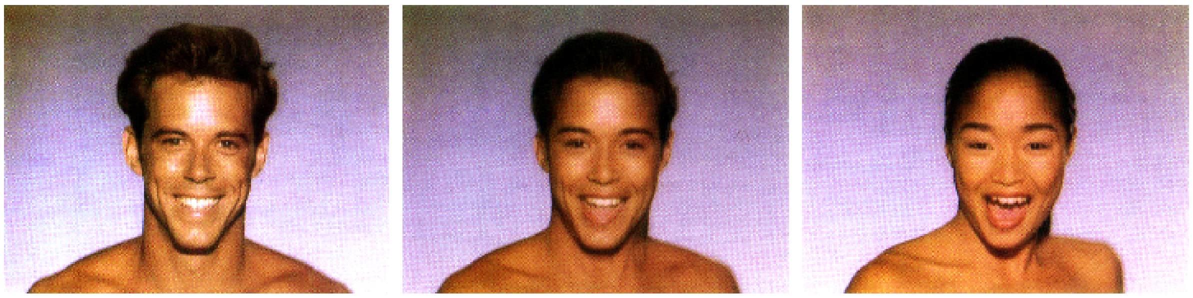


Figure 3: The sequence of frames of the image metamorphosis.

you created in the previous assignment will allow you to do this if you completed each of the extensions. If you did not complete the extensions credit part, don't worry, we provide an editor that has that capability.

Recall that the morphing algorithm uses corresponding directed line segments to mark the features in the images. To set up the morph, configure your correspondence editor to load your image and that of the next student. Draw lines to mark key corresponding facial features on both images like what is shown in Beier and Neely example in Fig. 2. Don't worry if the correspondences are not perfect at this point, as you will likely come back to tweak them later as you see the results of the morph and understand the intricacies of how it works. Learning to design a good morph is part of this week's exercise.

Once you have set up the images and feature correspondences, you will have to write the code to generate your frames that will go into the metamorphosis video. The morphing algorithm uses interpolation in a variety of different ways to achieve the morph, and the primary goal of this assignment is for you to learn the importance and uses of interpolation in computer graphics. Examples of interpolation (or averaging) that appear include

1. Interpolating colors to blend images
2. Interpolating pixels when sampling an image at arbitrary positions
3. Interpolating between corresponding features (lines)
4. Weighted interpolation of displacement vectors to compute a distortion
5. Non-linear interpolation to control the pace of an animation through time

Use the instructions in the next section to guide your implementation of the image metamorphosis algorithm. When you have successfully completed the assignment, you will be able to run your program on your images and the set of feature correspondences you defined to produce a sequence of frames that look like Fig. 3.

Instructions

Step 1: Understanding the Algorithm Download a copy of Beier and Neely's paper, Feature-Based Image Metamorphosis, or HTML Version, more readable (without pictures). We recommend you read the paper in its entirety before starting your implementation. It is a short, easy read by today's standards, and there are many subtleties about coding the algorithm that are discussed throughout. At this time you should ask questions (on Piazza or during office hours) if parts of algorithm are still unclear.

Step 2: Familiarization with the Starter Code The starter code archive contains three source files and the project/make files for building them on a variety of platforms. You can choose the right one for your platform and discard the rest. The files `parseConfig.h` and `parseConfig.cpp` are the same as the previous assignment, and serve to load the configuration and feature sets you created and saved using your MicroUI. You will only need to edit morph.cpp to complete your implementation.

When you run the program, you should see a test image of the Stanford 'S' on a grid background. Despite there being a graphical display, the morphing algorithm can be implemented to run and do its job entirely offline without a UI. The graphical display exists to help you display an intermediate or final result as an image, which can assist greatly with testing or debugging. The remainder of the steps in this assignment will have you implement the functions `BlendImages()`, `FieldMorph()`, `MorphImages()`, and `GenerateMorphFrames()`.

Download and try out the provided correspondence editor if you need it. Instructions on how to use it can be found in the archive. You may also use a friend's correspondence editor, or any other tool that may do the job as long as it can export the features in a format that your morphing program can read.

Step 3: Image Blending by Color Interpolation As a warm-up exercise, implement the blending of two images by performing a linear interpolation between them on each of their color channels. The effect you get should be like that of a video "cross-fade" operation often used to switch between two streams of video. You can look at the `STImage` class to learn how to read and set pixels on an image, and you may also examine `STVector` for an example of linear interpolation in the `Lerp`() function.

Look for the `BlendImages()` function and write your code in there. The parameter t determines the mix between the images: The first (source) image is dominant when t is near 0, and the second (target) image is dominant when t is close to 1.

When you have completed the `BlendImages()` function, you can test it by calling it from `main()` and using the UI to display the result.

Step 4: Interpolating between Features to Compute a Field Morph Recall that a feature in the morphing algorithm is a corresponding pair of directed line segments. The `Feature` struct is defined to represent one of the two line segments. Thus, there will be a feature set for each of the two images. The feature sets should have the same size, and features with the same index are corresponding features across the two images.

For every pixel in the result image, the field morph algorithm uses each line correspondence to determine what position in the source image would be moved there for that particular feature. Transformation of a single line correspondence is described in Section 3.2 of Beier and Neely. The feature line to which the first line must be transformed is again a function of a parameter t, and must be interpolated between the source line and the target line. Each displacement computed for a correspondence is then weighted by a function of line length and distance to the feature according to equation (4), as described in Section 3.3 of the article. The final averaged displacement is applied to the original position to determine the location from which to sample the source image.

The weighting formula requires three parameters (a, b, and p) whose effects are described in the same section of Beier and Neely. The application has them set to $a = 0.5$, $b = 1.0$, and $p = 0.2$ by default, but you may want to tweak them to achieve the best result possible when generating your final morph.

Write your code for this image warping algorithm in the `FieldMorph()` function. At any stage of your implementation where you can stop to test, you may find it useful to call `FieldMorph()` from `main()` to get the UI to display an intermediate result. If you are having difficulties getting the full algorithm to work right away, you may find it helpful to implement the transformation with just a single pair of corresponding lines at first.

Step 5: Sampling an Image using Bilinear Interpolation You may have noticed that the displacement vector from the previous step, and thus the final position from which to sample the source image, does not necessarily have integral coordinates. If you simply truncated or rounded these coordinates to integers in order to retrieve the nearest pixel from the source image, you will notice that the grid lines on the test image appear jagged or aliased. This is a very undesirable effect!

Instead of simply retrieving the nearest pixel for a given position, modify your code to retrieve the four pixels surrounding the point and compute a final color based on a bilinear interpolation of the four colors. You may find it easier to create a new function that does this, rather than placing the bilinear interpolation code directly within `FieldMorph()`.

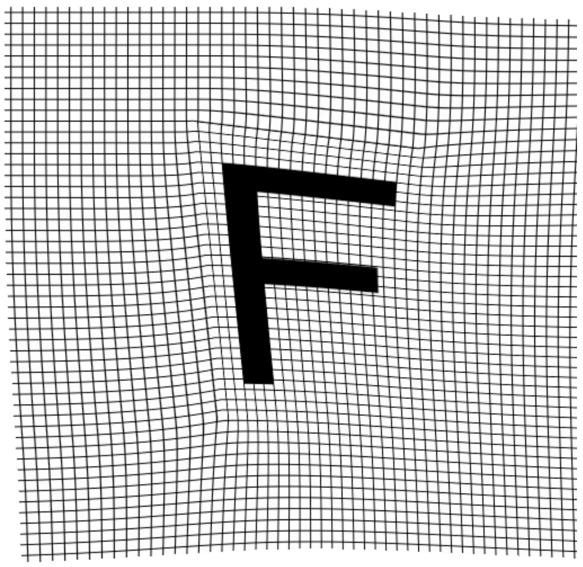


Figure 4: Sample image using interpolation when sampling the source image

Once you start using interpolation when sampling the source image, the transformed grid lines on the test image should look a lot smoother, like those in the sample image in Fig. 4. At this point, save a screen shot (press 's') of your transformed test image to submit later.

Step 6: Controlling the Pace of the Morph through Time After completing the implementation of `FieldMorph()`, you will have all the tools you need to compute an image metamorphosis! Morphing between two images is described in Section 3.4 of Beier and Neely. The idea is to distort the source image toward the target according to the stage of the morph (parameter t), distort the target image toward source, then blend the two for the final result. Implement the `MorphImages()` function to do this, and call the function from `main()` to test it when you are done.

Finally, you will need to implement `GenerateMorphFrames()` to repeatedly call `MorphImages()` to generate your 30 individual frames that will go into the metamorphosis video. The simplest way of doing this is to do a linear interpolation between the start and end values of t (i.e. $t = 0$, $t = 1/30$, $t = 2/30$, ..., $t = 1$) and feed those to the `MorphImages()` function. However, this would most likely result in a jerky or unnatural appearance in the class video when it transitions between people.

The final video will look a lot smoother if the morph starts transforming you into the next student slowly at first, quickens in the middle, and slows down again as the final likeness of the next person is formed. The plot of the parameter t vs. frame number in Fig. 5 may help you visualize this effect. In our application, you can think of it as performing a non-linear interpolation between the start and end frames. Controlling the pace of an animation in such a way is called easing or tweening. Choose an ease function that has this effect and use it to generate values of t to drive your morph over the 30 frames.

When you have completed all of the above, fire up your correspondence editor with your image and that of the student below you on the Participants page, and create your set of feature lines (if you haven't done so already). If the next student does not have a picture available, you can go down the list until you find the first student who does. If you're at the bottom, then start back at the top. Then modify `config.txt` to use your real data rather than the test images, run your program, and watch it morph! The frames may take a while to compute, but that's normal.

Questions

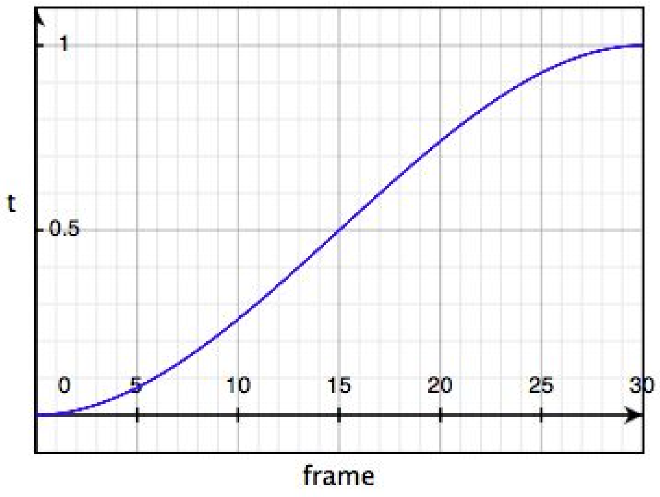


Figure 5: Time vs. frames

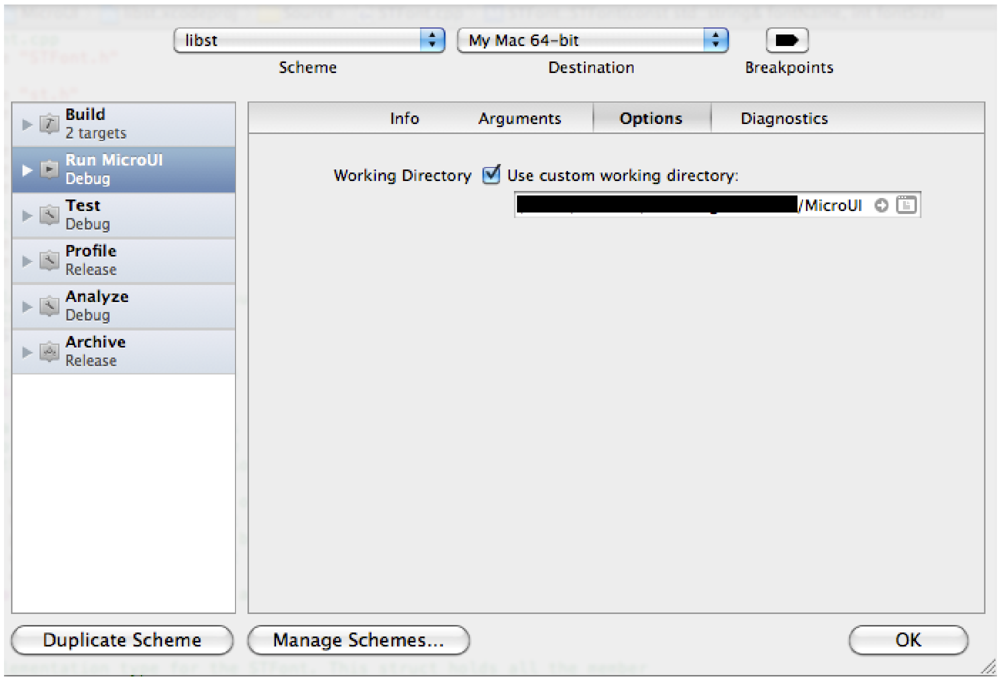
1. When computing the warped image in the field morph function, why would you iterate over all the pixels in the destination image, rather than over those in the source image?
2. Beier and Neely describe two different methods for interpolating line segments (Section 3.4). What are the two different ways, and what are their relative merits?
3. After computing a transformation on a pixel position, can the resulting location you are trying to read from the source image lie outside its bounds? If so, how might you handle that scenario?
4. What is the difference between an interpolation calculated using an ease function, and one calculated using linear interpolation?

Grading: This week's 4 grading points are broken down as follows:

1. Correct answers to assignment questions (1 point)
2. Correct implementation of image blending and correct implementation of the field morph (1 point)
3. Use of bilinear interpolation to sample the image (1 point)
4. Putting the morph together and generating 30 nice frames with an ease function (1 point)

Tips/Troubleshooting

- In order to run the correspondence editor, execute the binary appropriate for your system. We currently do not have a binary available for Mac OS X due to an update to `libpng`. Instead, you should `ssh -Y` into the `myth` cluster and run `editor-myth` to draw the appropriate feature lines.
- Doing all the computation to generate the morphed frames can be quite slow, especially if you have a lot of features defined. After you've debugged your algorithm, you may find that your program will run a lot faster if you build it in release mode. In Xcode or Visual Studio, simple switch the configuration to "Release" on the toolbar. If you're using the Makefile, first do `make clean`, and then type `make release` rather than just `make`.

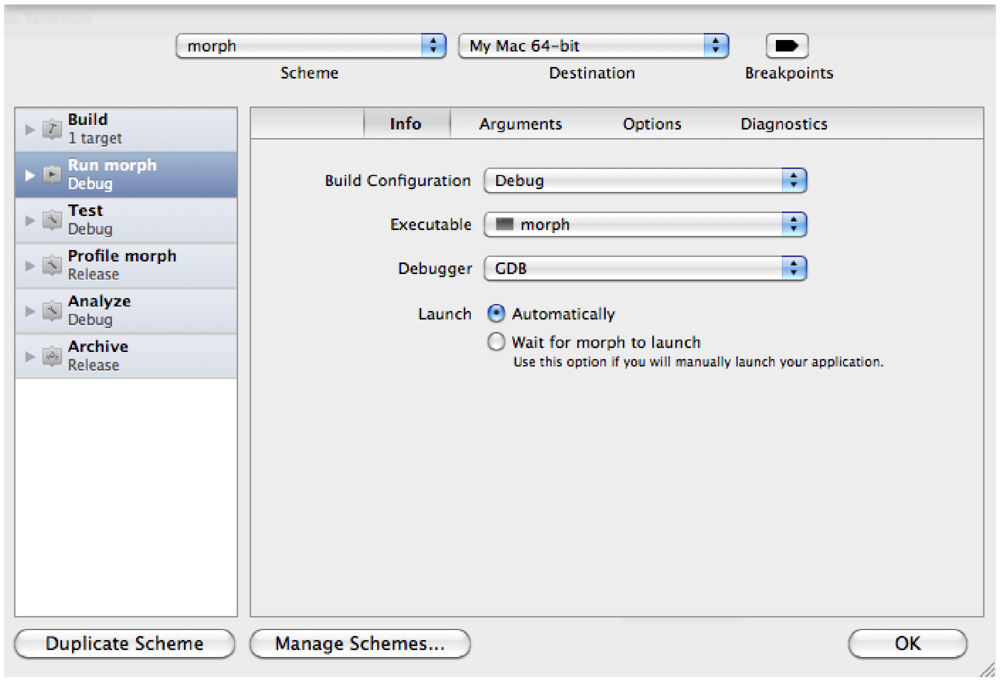


Scheme: libst
Destination: My Mac 64-bit

Working Directory [✓] Use custom working directory:
[redacted]/MicroUI

Figure 6: Xcode 4.x: Use custom working directory

- The starter code assumes that the resource files (input text files) are put in the project directory.
- XCode 4.x: Please set the working directory to Custom Directory. Product → Edit Scheme → Run morph → Options. Set the Working Directory to the one containing your XCode project on your computer (See Fig. 6).
- If for some reason you cannot Run the Project (Run is greyed in your menu), Do: Product → Edit Scheme... → Run morph → Info → Executable → morph (See Fig. 7).
- XCode 3.x: Please set the working directory to Project Directory. To do this in Xcode: Project → Edit Active Executable "morph" → General → Set the working directory to → Choose Project Directory. See Fig. 8.



Scheme: morph
Destination: My Mac 64-bit
Breakpoints

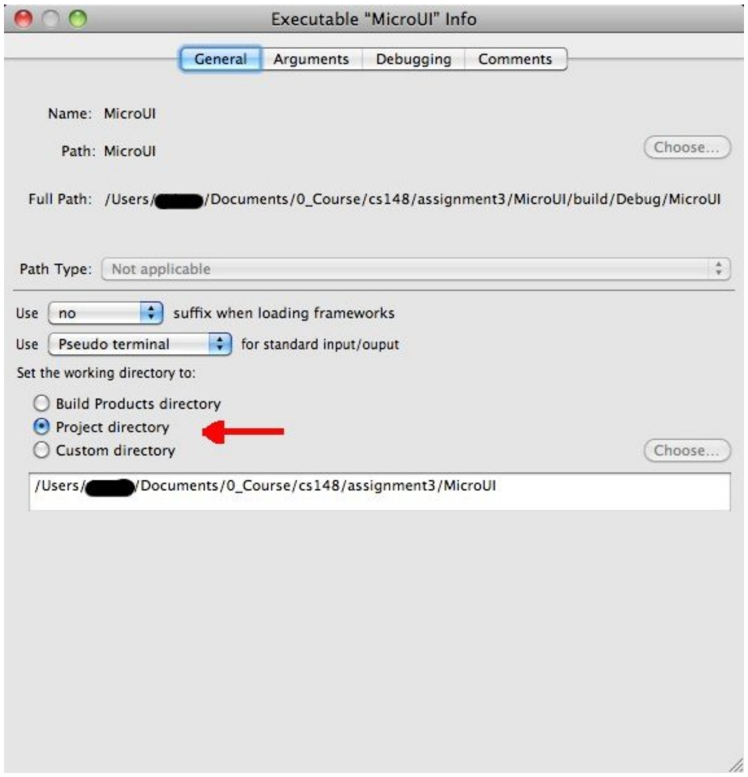
Build — 1 target
Run morph — Debug
Test — Debug
Profile morph — Release
Analyze — Debug
Archive — Release

Info | Arguments | Options | Diagnostics

Build Configuration: Debug
Executable: morph
Debugger: GDB
Launch: [✓ Automatically] Wait for morph to launch
Use this option if you will manually launch your application.

Duplicate Scheme | Manage Schemes... | OK

Figure 7: Set Executable to morph



Executable "MicroUI" Info

General | Arguments | Debugging | Comments

Name: MicroUI
Path: MicroUI
Full Path: /Users/[redacted]/Documents/0_Course/cs148/assignment3/MicroUI/build/Debug/MicroUI
Path Type: Not applicable

Use no suffix when loading frameworks
Use Pseudo terminal for standard input/ouput
Set the working directory to:
Build Products directory [✓ Project directory] Custom directory
/Users/[redacted]/Documents/0_Course/cs148/assignment3/MicroUI

Figure 8: Xcode 3.x: Set working directory to Project directory