MIT EECS 6.815/6.865: Assignment 5:

Resampling, Warping and Morphing

Due Wednesday October 21 at 9pm

1 Summary

This problem set has several questions for extra credit. Feel free to attempt them, but do the main problems first. The maximum extra credit is 10%.

- Scaling using nearest-neighbor and bi-linear reconstruction
- Rotation using linear reconstruction (6.865 only)
- Image warping according to one pair of segments
- Image warping according to two lists of segments, using weighted warping
- Image morphing
- Morph sequence between you and a peer (due on Friday, October 9).

2 Resampling

In this section, we will rescale and rotate images, starting with simple transformations and naïve reconstructions.

Below we show an example of scaling with various methods, together with a small crop of the resulting image to highlight the differences.

Figure 1: Original Image

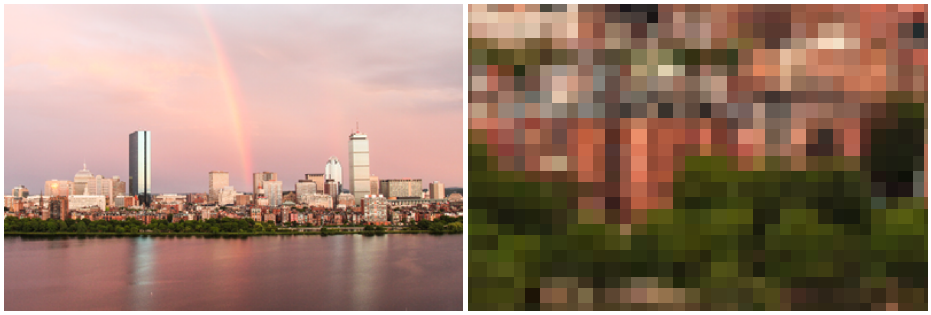

Figure 2: Scaling by 3.5× with nearest neighbor interpolation using `scaleNN`

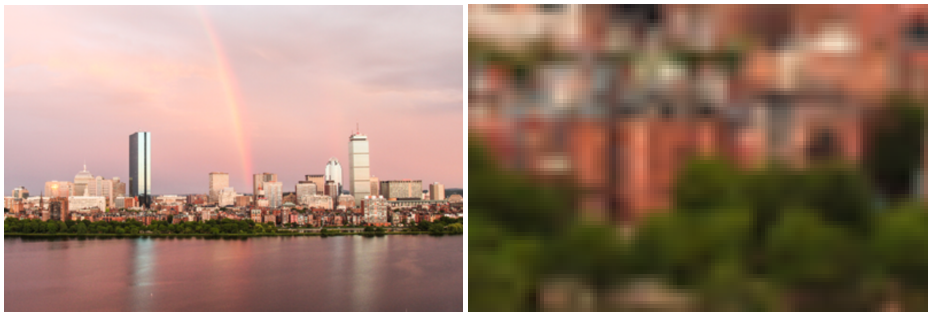

Figure 3: Scaling by 3.5× with bilinear interpolation using `scaleLin`

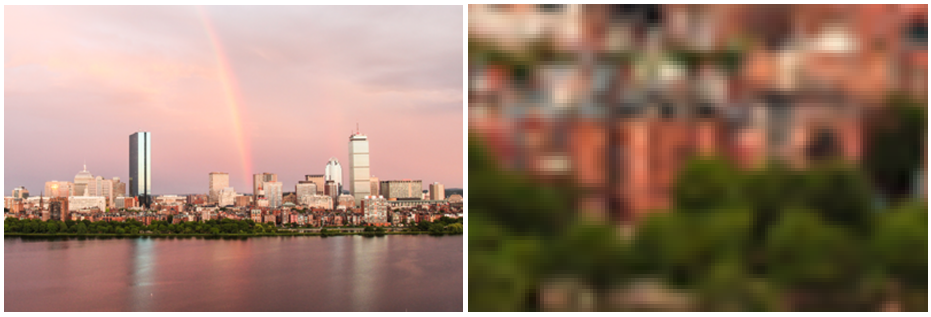

Figure 4: Scaling by 3.5× with bicubic interpolation (not implemented in the pset)

2.1 Basic scaling with nearest-neighbor

The first operation we consider is the re-scaling of an image by a global scale factor s. If $s > 1$, the operation will enlarge the input. If $0 < s < 1$, it will shrink the input.

To implement this operation, we need to create a new `Image` object that is s times larger (resp. $\frac{1}{s}\times$ smaller if $s < 1$) than the original in each dimension. Use `floor()` to get an integral size for the new image.

We now have to fill the pixel values of this new image from those of the input: this is called re-sampling. For each pixel in the new image, we look up the color value in the input image at a location that corresponds to the inverse transform (in this case a scaling by $\frac{1}{s}$). In general this location will not be on grid and we'll have to estimate the color at this location using information from the neighboring pixels.

The simplest technique to sample the new value is called **nearest-neighbor re-sampling**: we round-off the real coordinates to the nearest integers and use the input's color at this new location to be the value of the current output pixel.

1 Implement the `Image scaleNN(const Image &im, float factor)` function in `basicImageManipulation.cpp`. This function should create a new image that is $factor$ times the size of the input using the nearest-neighbor re-sampling method.

2.2 Scaling with bilinear interpolation

Nearest-neighbor re-sampling creates blocky artifacts and pixelated results. We will address this using a better reconstruction based on bilinear interpolation. For this, we consider the four pixels immediately around the computed real coordinates and perform two linear interpolations. We first linearly interpolate along x the colors of the top and bottom pairs of pixels. Then we interpolate these two values along y to get the final sample. The interpolation weight are driven by the distance from the corners. (see Fig. 5).

2.a Implement `float interpolateLin(const Image &im, float x, float y, float z, bool clamp=0)` in `basicImageManipulation.cpp`. This function takes floating point coordinates and performs bilinear reconstruction. Don't forget to use smart accessors to make sure you can handle coordinates outside the image.

2.b Next, write an image scaling function `Image scaleLin(const Image &im, float factor)` in `basicImageManipulation.cpp` that rescale using linear interpolation by calling `interpolateLin` where appropriate.

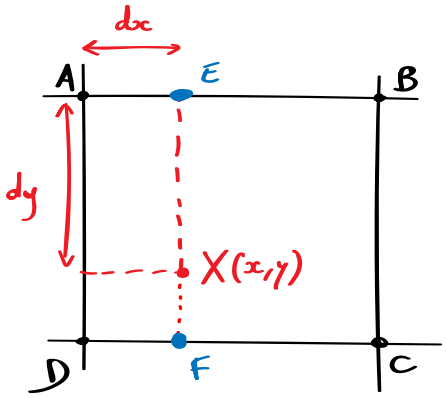

Figure 5: Bilinear interpolation. We sample the value at location X (non integer coordinates) by interpolating the neighboring pixels' values. First, linearly interpolate the values of A and B along x. Do the same for D and C. This gives you pixel values at E and F. Linearly interpolate these two values to obtain X's colors.

2.3 Rotations (required for 6.865, 5% extra credit for 6.815)

3 Implement the function `rotate(const Image &im, float theta)` in `basicImageManipulation.cpp` that rotates an image around its center by θ. **Hint:** use your bilinear interpolation function. Use the center position already present in the starter code as the rotation center.

Figure 6: Example rotation by $\frac{\pi}{4}$ using `rotate`

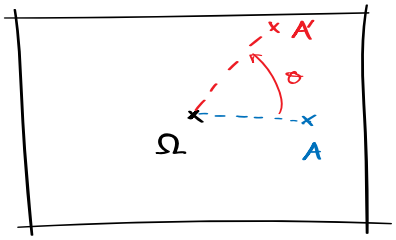

Figure 7: Rotation with an angle $\theta > 0$ with respect to the center of the image. To get the pixel value at A', sample from location A in the input image.

2.4 Extra credit (5%): Bicubic or Lanczos

You can obtain a better interpolation by considering a larger pixel footprint and using smarter weights, such as that given by a bicubic or Lanczos functions. If you choose to implement one of these interpolation schemes, please let us know in the submission form!

3 Warping and morphing

In what follows, you will implement image warping and morphing according to Beier and Neely's method, which was used for special effects such as those of Michael Jackson's *Black or White* music video
https://www.youtube.com/watch?v=F2AitTPI5U0.

We highly recommend that you read the provided original article, which is well written and includes important references such as Ghost Busters.

> Beier, Thaddeus, and Shawn Neely. "Feature-based image metamorphosis." ACM SIGGRAPH Computer Graphics. Vol. 26. No. 2. ACM, 1992.

The full method for warping and morphing includes a number of technical components and it is critical that you debug them as you implement each individual one.

A copy of the paper is included in the handout

3.1 Basic Vector Tools

Warping and morphing geometrically distort an input image. This requires a few vector operation which you'll implement. We provide you a basic `Vec2f` class to represent 2D vectors. Don't forget to test your functions!

4.a In `morphing.cpp`, implement `Vec2f add(const Vec2f & a, const Vec2f & b)` to sum two vectors $\mathbf{a} + \mathbf{b}$.

4.b In `morphing.cpp`, implement `Vec2f subtract(const Vec2f & a, const Vec2f & b)` that returns the difference $\mathbf{a} - \mathbf{b}$.

4.c In `morphing.cpp`, implement `Vec2f scalarMult(const Vec2f & a, float f)` that implements multiplication by a scalar $f \cdot \mathbf{a}$.

4.d In `morphing.cpp`, implement `Vec2f dot(const Vec2f & a, const Vec2f b)` that implements the dot product of two vectors: $\langle \mathbf{a} | \mathbf{b} \rangle$.

4.e In `morphing.cpp`, implement `float dot(const Vec2f & a)` that returns the length of a vector (in the L^2 sense): $||\mathbf{a}|| = \sqrt{a_x^2 + a_y^2}$.

3.2 Segments

Now that we have some basic tools, we will implement the `Segment` class, which is critical to Beier and Neely's warping. Test these methods thoroughly as they will be used in the warping and morphing functions in the later part of the problem set.

A `Segment` represents a directed line segment $\overrightarrow{PQ}$. The class holds a copy of the endpoints P and Q, the length of the segment $||\overrightarrow{PQ}||$, and a local orthonormal frame $(\overrightarrow{e_1}, \overrightarrow{e_2})$. We define:

$$\overrightarrow{e_1} = \frac{\overrightarrow{PQ}}{||\overrightarrow{PQ}||} \tag{1}$$

$\overrightarrow{e_2}$ completes the orthonormal frame.

Both frame vectors and the length of the segment need to be initialized in the class constructor.

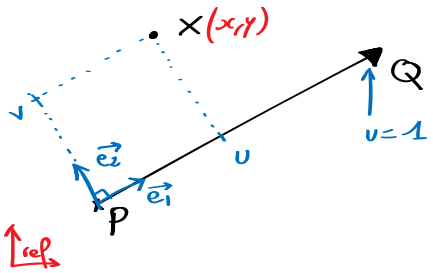

Figure 8: Directed segment and its local coordinate system. Don't forget that u as defined in the paper is not exaclty the coordinate in the local frame.

5 In `morphing.cpp`, complete the constructor `Segment::Segment(Vec2f P_, Vec2f Q_)` that creates a segment from its two endpoints and initializes the data structure properly.

Now that our `Segment` class is usable let's implement methods to convert from the global (x, y) coordinates of a point to the local (u, v) coordinates in the reference frame as in Beier and Neely. **Be careful:** although v exactly corresponds to the second coordinate in the local frame, u is actually rescaled so that the u coordinate of Q is 1 (Equations (1) and (2) in the paper).

6.a Implement `Vec2f Segment::XtoUV(Vec2f &X)` to compute the (u, v) coordinates of a 2D point $X = (x, y)$ with respect to a segment as described in the paper.

6.b Conversly, implement `Vec2f Segment::UVtoX(Vec2f &uv)` to compute the (x, y) global coordinates of a 2D point from its local (u, v) coordinates.

Hint: use simple examples to test your method, as this method will be the core of the rest of the problem.

6.c Implement the point to segment distance function `float Segment::distance(Vec2f X)` as described in the paper.

3.3 UI

We provide you with a rudimentary (to say the least) interface to specify segment location from a web browser. It is based on javascript and the raphael library (http://raphaeljs.com/).

The two images should have the same size.

For the warp part, you can use the same image on both sides. To change the image, open `morph_ui.html` in a text editor and update `path1` and `path2` according to your images.

You must click on the segment endpoints in the same order on the left and on the right. Unfortunately, you cannot edit the locations after you have clicked. Who said rudimentary? Once you are done, simply copy the C++ code below each image into your main function to create the corresponding segments.

3.4 Warping according to one pair of segments

Now that we have a functionnal `Segment` class and an interface to specify line segments, let the fun begin!

The core component to warp images is a method to transform an image according to the displacement of a segment.

```
vector<Segment> segsBefore;
segsBefore.push_back(Segment(56, 111, 69, 90));
segsBefore.push_back(Segment(97, 98, 119, 104));
```

```
vector<Segment> segsAfter;
segsAfter.push_back(Segment(61, 103, 78, 76));
segsAfter.push_back(Segment(93, 93, 114, 101));
```

Figure 9: The UI to specify line segments

7 Once you are convinced that you can transform 2D points according to a pair of before/after segments, implement `Image warpBy1(const Image &im, Segment segBefore, Segment segAfter)`. This is a resampling function that warps an entire image according to one a pair of segments.

The output should be an image of the same size as the input such that the feature under `segBefore` is now at the location of `segAfter`. Use bilinear reconstruction. Again, use simple examples to test this function. Once you are done with this, you have completed the hardest part of the assignment.

Figure 10: `warpBy1(im, Segment(0,0, 10,0), Segment(10, 10, 30, 15))`. You can use the javascript UI to specify the segments, using the same image on both side for reference.

3.5 Warping according to multiple pairs of segments

In this question, you will extend you warp code to perform transformations according to multiple pairs of segments. For each pixel, transform its 2D coordinates according to each pair of segments and take a weighted average according to the length of each segment and the distance of the pixel to the segments. Specifically, the weight is given according to Beier and Neely:

$$weight = \left(\frac{length^p}{a + dist}\right)^b$$

where a, b, p are parameters that control the interpolation. In our test, we have used $b = p = 1$ and $a = 10$ (roughly 5% of the image size).

8.a Implement `float Segment::weight(Vec2f &X, float a, float b, float p)` based on the formula above.

8.b Implement `Image warp(const Image &im, const const vector<Segment> src_segs, const vector<Segment> dst_segs, float a=10.0, float b=1.0, p=1.0)`,
which returns a new warped image according to the list of before and after segments, using the `weight` function.

Use the provided javascript UI to specify segments. The points must be entered in the same order on the left and right image. You can then copy-paste the C++ code generated below the images to create the corresponding segment objects.

3.6 Morphing

Given your warping code, we will write a function that generates a morph sequence between two images. Again, make sure you are familiar with morphing from the article.

You are given the source and target images, and a list of segments for each image (the position in the list defines the corresponding pairs of segments, so the lists should have the same number of elements). You must generate N images morphing from the first input image to the second input image.

For each image, compute its corresponding time fraction t. Then linearly interpolate the position of each segment's endpoints according to t, where $t = 0$ corresponds to the position in the first image and $t = 1$ is the position in the last image. You might want to visualize the results for debugging.

You now need to warp both the first and last image so that their segment locations are moved to the segment location at t, which will align the features of the images. We suggest that you write these two images to disk and verify that the images align and that, as t increases, the images get warped from the configuration of the first image all the way to that of the last one.

Finally, for each t, perform a linear interpolation between the pixel values of the two warped images.

Your function should return a sequence of images. For debugging you can use your main function to write your images to disk using a sequence of names such as `morph_1.png, morph_2.png, ...`

9 Implement `morph(im0, im1, listSegmentsBefore, listSegmentsAfter, N=1, a=10.0, b=1.0, p=1.0)`. It should return a sequence of N images in addition to the two inputs (i.e., when called with the default value of 1, it only generates one new image for $t = 0.5$). The function should check that `im0` and `im1` have the same size.

Figure 11: Morph example

Visualize Results Aside from just looking at the images, you can explore your results at http://tipix.csail.mit.edu if you wish. Click on load and load your morphed images (make sure to respect the required naming). Then explore your results using the pointer, or click "play" from the top-right information panel.

There are several ways to make a video (or a .gif) out of your files (**note** this is not required). You can install ffmpeg `http://ffmpeg.org/` but this involves some number of dependencies. Then use it with, e.g. `ffmpeg -i tes_morph_%02d.png out.gif`.

3.7 Class morph

We'll use the code implemented in this problem set to create a class morph. Details to come soon!

4 Extra credit

Here are ideas for extensions you could attempt, for 5% each. At most, on the entire assignment, you can get 10% of extra credit:

- improve the silly javascript UI.
- extend to movies, where segments are specified at a number of keyframes.
- morphable face models (see http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.9275)

5 Submission

Turn in your files to the online submission system (link is on Stellar) and make sure all your files are in the `asst` directory under the root of the zip file. If your code compiles on the submission system, it is organized correctly. The submission system will run code in your main function, but we will not use this code for grading. The submission system should also show you the image your code writes to the `./Output` directory

In the submission system, there will be a form in which you should answer the following questions:

- How long did the assignment take? (in minutes)
- Potential issues with your solution and explanation of partial completion (for partial credit)
- Any extra credit you may have implemented and their function signatures if applicable
- Collaboration acknowledgment (you must write your own code)
- What was most unclear/difficult?
- What was most exciting?