6.815/6.865: Assignment 2:

Resampling, Warping and Morphing

Due Wednesday February 25 at 9pm

1 Summary

Important: This problem set is longer than the previous two. All of the procedures are described in the lecture slides. Reading the slides carefully can easily save you hours of headaches. Do not worry about gamma encoding in this pset.

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, Feb 27).

2 Accessor

When resampling images or performing neighborhood operations, we might try to access a pixel at `x, y` coordinates that are outside the image. To handle such cases gracefully, it is good to write functions that take `x, y` as input and check them against the bounds of the image before looking up a value.

Multiple options are possible when a pixel is requested outside the bounds, but the two most common are to return a black pixel or the closest available pixel (that of the closest edge). We recommend you implement both. For the latter, just clamp the pixel coordinates to `[0..height-1]` and `[0..width-1]` and perform the lookup there.

Black padding is more appropriate for applications such as scale and rotation whereas edge-pixel padding looks better for warping and for the next assignment on convolution.

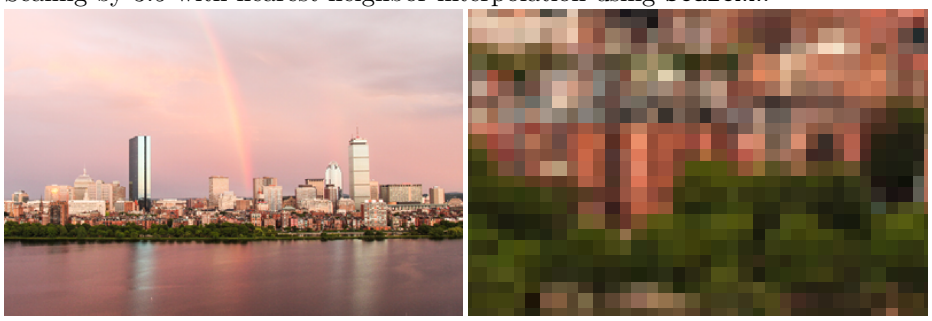
2.0.1. Implement `float Image::smartAccessor(int x, int y, int z, bool clamp=false) const` in `Image.cpp`.

3 Resampling

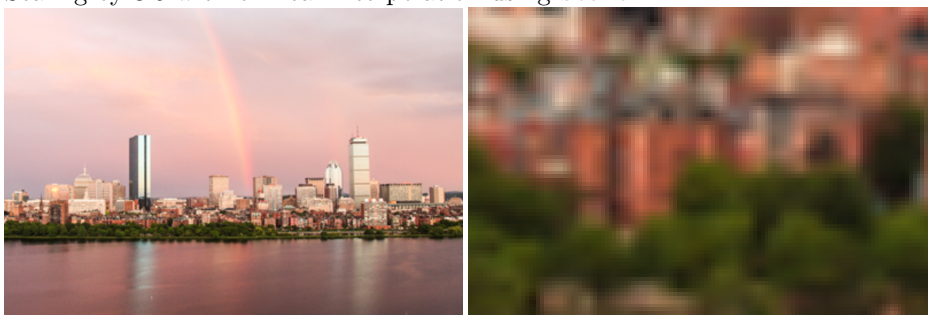
In this section, we will scale and rotate images, starting with simple transformations and naïve reconstruction. We use images of Boston in the summer to remind you that the snow will melt at some point :). Below we show an example of scaling with various methods, and a small crop of the resulting image to highlight the differences.

Original Image:

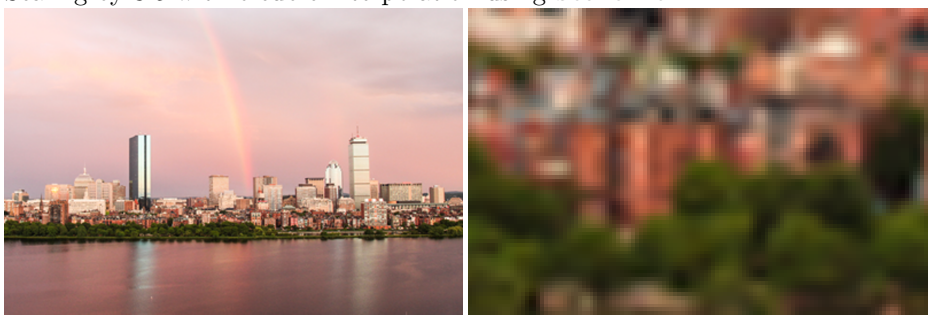
Scaling by 3.5 with nearest neighbor interpolation using `scaleNN`:



Scaling by 3.5 with bilinear interpolation using `scaleLin`:



Scaling by 3.5 with bicubic interpolation using `scaleBic`:



3.1 Basic scaling with nearest-neighbor

We first consider scaling the image by a constant scale factor

3.1.1. Implement the
`Image scaleNN(const Image &im, const float &factor)`
function in `basicImageManipulation.cpp`. This function should create a new image that is *factor* times the size of the input. For this, you need to create a new Image object that is larger than the original by a factor in each dimension (use `floor()` to determine the size of the new image). You then loop over all the pixels of the new image, and assign them a color by looking up the appropriate coordinates in the input image. In this simple function, we will use **nearest-neighbor reconstruction**, which means that all you need to do is round floating point coordinates to the nearest integers.

3.2 Scaling with bilinear interpolation

Nearest-neighbor reconstruction leads to blocky artifacts and pixelated results. You will address this using a better reconstruction based on bilinear interpolation. For this, consider the four pixels around the computed coordinate location and perform bilinear reconstruction by first performing two linear interpolations along x for the top and bottom pairs of pixels, and then another interpolation along y for the two results. In each case, the interpolation weights are given by the fractional x and y coordinates.

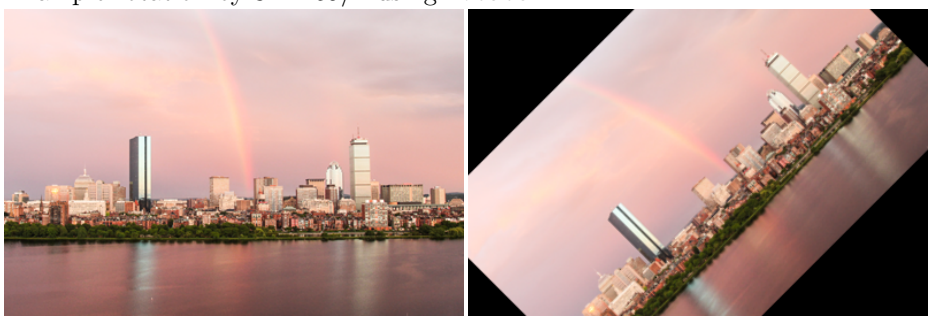
3.2.1. Implement `float interpolateLin(const Image &im, float x, float y, float z, bool clamp=0)` in `basicImageManipulation.cpp`. This function takes floating point coordinates and performs bilinear reconstruction. Don't forget to use smart accessors to make sure you can handle coordinates outside the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

3.2.2. Next, write an image scaling function `Image scaleLin(const Image &im, const float &factor)` in `basicImageManipulation.cpp` that uses linear interpolation by calling `interpolateLin` appropriately.

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

3.3.1. Implement the function `rotate(const Image &im, const 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 in your starter code.

Example rotation by 3.14159/4 using `rotate`:



3.4 Extra credit (5%): Bicubic or Lanczos

You can obtain better reconstruction by considering more pixels and using smarter weights, such as that given by a bicubic or Lanczos functions.

4 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
`http://www.youtube.com/watch?v=bBAiZcNWecw`.

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.

4.1 UI

We provide you with a simple (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/). Copy `click.html` and `raphael-min.js` to a location with your two input images. 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, modify the beginning of `click.htm` and update imagewidth, imageheight, 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.

Yes, this is primitive. Once you are done, simply copy the C++ code below each image into your main function to create the corresponding segments.

```
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));
```

4.2 Segments

We will start by implementing the Segment class, which is critical to warping. Test these methods thoroughly as they will be used in warping and morphing functions below.

4.2.1. In `morphing.cpp`, implement the `Segment` class. Complete the constructor `Segment::Segment(const float &x1, const float &y1, const float &x2, const float &y2)` that takes in four floating point values corresponding to the ends of a segment. In particular, this is the syntax that our (hacky) UI expects. Look in the comments to figure out what you need to implement in the constructor. You may find the functions `Segment::subtract`, `Segment::dot`, and `Segment:scalorMult` useful.

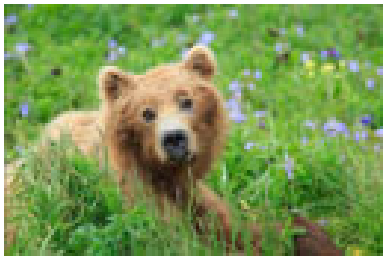
4.2.2. Consider a pair of segments, corresponding to a before and after configuration. Implement the computation of the u and v coordinates of a 2D point with respect to a segment as described in the slides and in the paper, by completing `float Segment::getU(float &x, float &y)` and `float Segment::getV(float &x, float &y)`. Given these coordinates, you can then compute the new x, y position of this point given the location of the other segment in `vector<float> Segment::UVtoX(float &u, float &v)`. Hint: use simple examples to test your method, as this method will be the core of the rest of the question.

4.2.3. Implement the point to segment distance function `float Segment::dist(float &x, float &y)`, remembering the three cases discussed in class.

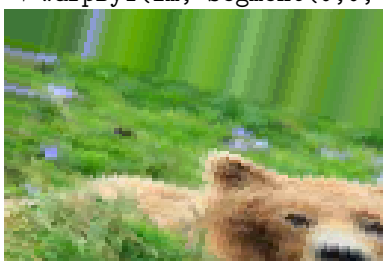
4.3 Warping according to one pair of segments

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

4.3.1. Once you are convinced that you can transform 2D points according to a pair of before/after segments, implement `Image warpBy1(const Image &im, segmentBefore, segmentAfter)`. This is a resampling function that warps an entire image according to such a pair of segments. The last two arguments are a single segment from the `Segment` class above. The output should be a warped image of the same size as the input. 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.



→ `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.

4.4 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 equal to roughly 5% of the image.

4.4.1. Implement `float Segment::weight(float &x, float &y, float &a, float &b, float &p)` based on the formula above.

4.4.2. Implement `Image warp(const Image &im, listSegmentsBefore, listSegmentsAfter, a=10.0, b=1.0, p=1.0)`, which returns a new warped image according to the list of before and after segments, using the `Segment::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.

4.5 Morphing

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

You are given the two images, and a list of corresponding segments for each image. 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 endpoint 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 warp 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, ...`

4.5.1. 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.

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 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 -r 20 -b 1800 -y -i morph%03d.png out.mp4`. A windows-only alternative appears to be `http://home.hccnet.nl/s.vd.palen/index.html` but we haven't tried it. See also: `http://www.videohelp.com/tools/Virtualdub`. MATLAB also has some decent tools for this.

4.6 Class morph (due Friday, Feb. 27th)

Use your morphing code to morph between your face and that of the next student in the list. Turn in 15 PNG frames of size 200x200.

5 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)

6 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.

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

- How long did the assignment take?
- Potential issues with your solution and explanation of partial completion (for partial credit)
- Any extra credit you may have implemented
- Collaboration acknowledgement (but again, you must write your own code)
- What was most unclear/difficult?
- What was most exciting?