MIT EECS 6.815/6.865: Assignment 2:

Resampling, Warping and Morphing

Due Wednesday September 18 at 9pm

1 Summary

- Scaling using nearest-neighbor reconstruction
- Scaling with 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 Wednesday).

2 Basic tools

While not part of the requirements, we highly recommend that you write simple tools to make your life easier. Hint: see the lecture notes.

iterator You are going to iterate a lot across pixels, and we recommend writing an iterator that takes an image as input and generates an iterator that return a series of `y, x` coordinates that span the image width and height, such that you can write `for y, x in imIter(im):`. Use the python `yield` command.

Acessor When resampling images or performing neighborhood operations, we might try to access a pixel at `y, x` coordinates that are outside the image. To handle such cases gracefully, it is good to write functions that take `y, x` 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.

2D Points We highly recommend that you represent 2D points as numpy[1] arrays of size 2. Arithmetic operations on numpy arrays behave as you would expect them to on vectors, while operations on python lists might not (for instance, multiplication of an integer by a python list repeats the elements in that list). Be careful with the order of x and y. 2D coordinates are most often represented in the order x, y (for instance in the segment UI we describe later), but our images follow y, x. Make sure things are ordered correctly.

A word about performance Because Python is interpreted, your code will run slowly. Debugging on small images will help.

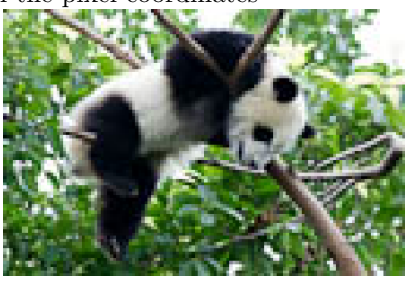
3 Resampling

In this section, we will scale and rotate images, starting with simple transformations and naïve reconstruction.

3.1 Basic scaling with nearest-neighbor

We first consider scaling the image by a constant scale factor `k` with respect to the upper left corner at coordinates $0, 0$. Write a `scaleNN(im, k)` function that creates a new image that is k times bigger than the input. For this, you need to create a new bigger array. You then loop over all its pixels and assign them a color by looking up the input image at the appropriate coordinates. 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.

Hint: be careful how you transform the pixel coordinates

 → `scaleNN(im, 3.5)` → 

3.2 Scaling with bilinear interpolation

Nearest-neighbor reconstruction leads to blocky artifacts and pixelated results. We will address this using a better reconstruction based on bilinear interpolation. For this, consider the 4 pixels around the required reconstruction location and perform bilinear reconstruction by first performing two linear interpolations along x for the top and bottom pair of pixels, and then another one along y

[1] my spell checker likes to replace numpy by bumpy. Can be amusing.

for the two results. In each case, the interpolation weights are given by the fractional x and y coordinates.

Write a function `interpolateLin(im, y, x)` that 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. You might want to implement two separate functions linear interpolation functions for black vs. edge padding or take the accessor as an additional input.

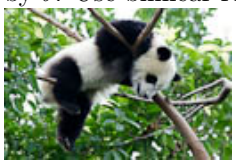
Then write an image scaling function `scaleLin(im, k)` that uses linear interpolation.

Make sure your test your function both with integer and floating point scale factors.

 → `scaleLin(im, 3.5)` → 

3.3 6.865 only: Rotations

Write a function `rotate(im, theta)` that rotates an image around its center by θ. Use bilinear reconstruction.

 → `rotate(im, math.pi/4)` → 

Extra credits: 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.

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, path1and 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 Python code below each image to create the corresponding segments.

segmentsBefore=numpy.array([segment(86, 128, 111, 128), segment(143, 127, 163, 130), segment(101, 202, 136, 199),

segmentsAfter=numpy.array([segment(84, 113, 106, 109), segment(141, 103, 160, 103), segment(104, 171, 146, 171),

Our javascript UI.

4.2 Warping according to one pair of segments

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

Segments We require that you implement a segment class with a constructor that takes 4 floating point values `x1, y1, x2, y2`. In particular, this is the

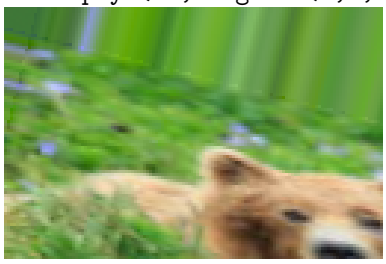
syntax that our (hacky) UI expects. Make sure that you force the stored 2D points to be encoded as floats, since the coordinates returned by the UI are integer, which could affect the precision of later computations. For example, my constructor sets the first 2D point as
`self.P=numpy.array([x1, y1], dtype=numpy.float64)`

Segment transform Consider a pair of segments, corresponding to a before and after configuration. You need to 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. Given these coordinates, you can then compute the new x, y position of this point given the location of the other segment. Use simple examples to test your method

Warping Once you are convinced that you can transform 2D points according to a pair of before/after segments, implement a resampling function that warps an entire image according to such a pair of segments. Again, use simple examples to test this function. Once you are done with this, you have completed the hardest part of the assignment.

The function should be callable according to
`warpBy1(im, segmentBefore, segmentAfter)`
where the last two arguments are a single segment of the class above. The output should be a warped image of the same size as the input. Use bilinear reconstruction.

→ `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.

4.3 Warping according to multiple pairs of segments

Extend the above 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. More 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.

Implement `warp(im, listSegmentsBefore, listSegmentsAfter, a=10, b=1, p=1)`, which returns a new warped image.

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 python code generated below the images to create the corresponding segment objects.

4.4 Morphing

Given your warping code, write a function that generates a morph sequence between two images. 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 and you must generate $N + 1$ images morphing from the first image to the second one, where image 0 is the first input and image $N + 1$ is the second one.

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 put the features of the images into correspondence. 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 image, which is very easy using numpy. No loop over pixels for this step! Your code is probably already quite slow.

Your function should return a sequence of images. For debugging you can usa a separate script to write your images to disk using a sequence of names such as `morph000.png, morph001.png, .../` Hint: `str('%03d' %i)` pads integer i with zeros until it has 3 digits.

Your function should be called
`morph(im1, im2, listSegmentsBefore, listSegmentsAfter, N=1, a=10, b=1, p=1)`
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 $t = 0.5$). `im1` and `im2` should have the same size.

Not required: movie file If you want to compile your individual files into a movie, you can install ffmpeg `http://ffmpeg.org/` but it is not required (and 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.

Also: `http://www.videohelp.com/tools/Virtualdub`

4.5 Class morph (due Slightly later (check back))

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. Make sure that they are indexed with a constant number of digits, using zero-padding.

5 Extra credit

If you love morphing[2], here are ideas for extensions:

- non-linear interpolation
- 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)

[2] Another funny one: the spellchecker wanted me to write: "if you love morphine."

6 Submission

Turn in your files and make sure all your functions can be called from a module `a2.py`. Put everything into a zip file and upload to Stellar.

Include a `README.txt` containing the answer to 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?