MIT EECS 6.815/6.865: Assignment 8:

Blending & Full Panoramas

Due Wednesday April 15 at 9pm

1 Summary

- Linear blending
- Two-scale blending
- Mini Planets
- 6.865: Full Panoramas
- Make your own panorama!
- Halide prep

Note that we wrote the following small functions that might be helpful:

- `Image copychannels(const Image &im, nChannels)` Useful in expanding a single-channel image to three channels.
- `Matrix eye(int n)` Useful for quick initialization of Identity Matrices
- `vector <Matrix> stackHomographies(vector <Matrix> Hs, int refIndex)` Useful for stacking N images properly via homographies.

2 Blending

So far, we have reprojected input images into a common domain without paying much attention to problems at the boundaries. Our goal is now to mask the transition between images.

2.1 Linear blending

We will first use a weighted average of two images to implement a smooth transition between them. For this, the output image will be a weighted average of the reprojected input ones, where the weights go from 1 at the center of an image to 0 at the edges.

Weights are not easy to compute in the output image because of the reprojection, which makes it hard to tell how far a pixel is from an image's boundary. Instead, we will compute weights in the source domain where it is trivial to tell where a pixel is compared to the image boundary.

Modify your automatic stitching code to perform smooth blending:

2.1. Implement `Image blendingweight(int imwidth, int imheight)`, which returns an Image of weights. We will use piecewise linear weights in the source domain. The weights will be given by a separable function, which means that it is the product of a function only in x and another function only in y. Each of these two functions will be piece-wise linear with a value of 1.0 in the center and 0.0 at the edges. For images with even width or height, the center doesn't have to be at a pixel (i.e. if the image has width 2, the center should be at 0.5). If you need a function to compute absolute value, make sure you use `fabs`, which is for floats.

2.2. Implement `void applyhomographyBlend(const Image &source, const Image &weight, Image &out, Matrix &H, bool bilinear)`, which is similar to `applyHomography` (or `applyHomographyFast`). But instead of writing the pixels of the input image to the output image, this function should **add** the pixels of the input image (`source`) times the `weight` to the output image.

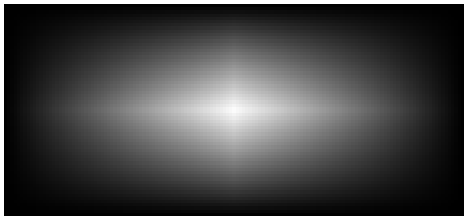
2.3. Implement `Image stitchLinearBlending(const Image &im1, const Image &im2, const Image &we1, const Image &we2, Matrix H)`, which stitches two images using the given weights. Note that stitch should not do any normalization – if the two weights don't add up to 1 at some pixel, that's ok. Use `applyhomographyBlend` to help you with this.

2.4. Implement `stitchBlending(Image &im1, Image &im2, Matrix H, int blend)`, where blend can be 0, 1 or 2. If blend is 0, then this should behave the same as the normal stitching (**using im2 as reference**). If blend is 1, stitch the two images using linear blending. Make sure you figure out some way of keeping track of the sum of the weights at each pixel. If blend is 2, use 2-scale blending (next section).

2.5. Implement `Image autostitch(Image &im1, Image &im2, int blend, float blurDescriptor=0.5, float radiusDescriptor=4)`, which

is the same as your Pset7 autostitch, except it calls the stitch implementation above, with the fourth parameter `blend`. In other words, it automatically finds the homography (via RANSAC) before doing the stitch with blending 0, 1 or 2.

Here is our weight for the poster image, and our applyhomographyBlend applied to the green/poster combo with a constant weight of 0.5 for every pixel of the poster:



2.2 Two-scale blending

The problem with smooth transitions like the ones above is that the resulting image can be blurry at the transition or exhibit ghosts when features are not exactly matched.

To fix this, we will use a two-scale approach that uses a smooth transition for the low frequencies and an abrupt transition for high frequencies.

2.5. Finish implementing `stitchBlending(Image &im1, Image &im2, Matrix H, int blend)` for the case when `blend` is 2.

First, decompose each source image into low frequencies and high frequencies using a Gaussian blur. Use a spatial sigma of 2 pixels.

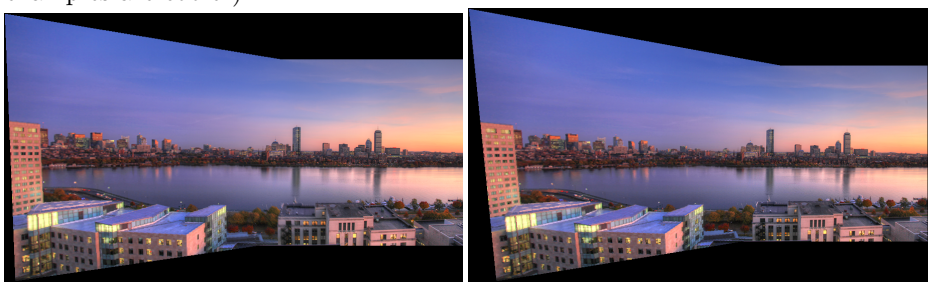
For the low frequencies, use the same transition as above.

For the high frequencies, use an abrupt transition that only keeps the high frequency of the image with the highest weight.

Compute the final image by adding the resulting low and high frequencies.

Our results for blend = 0, 1, and 2 with a pre-determined homography for Stata:

zoom in on the tree on your pdf viewer to see the difference between the second and third image. And the Boston autostitch (since we all know Boston skyline examples are cooler):



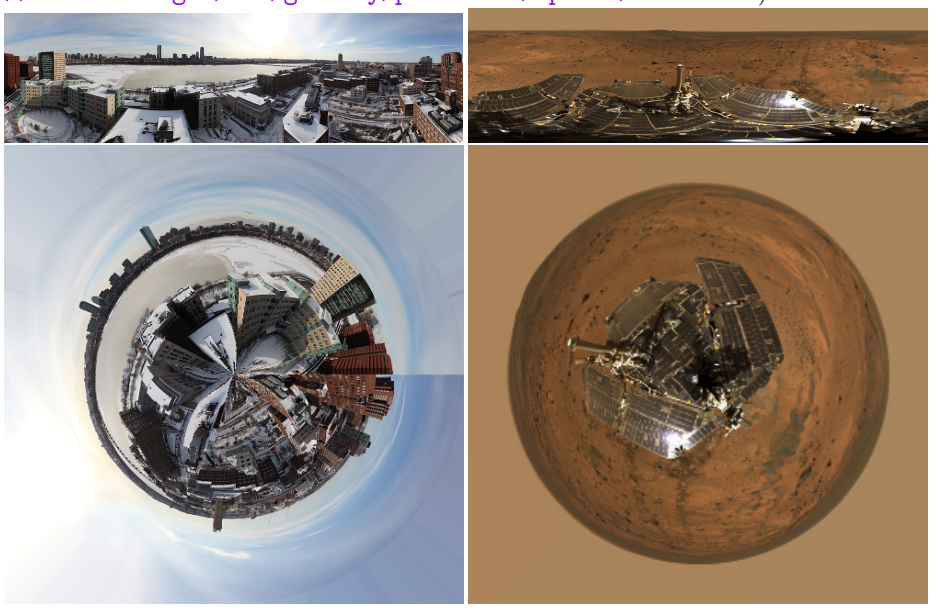
You can see more problems in the linear blending, as expected from lecture notes. Note, interestingly, that the projections are slightly different. Why? We're running RANSAC each time, and we're getting slightly different Homographies!

3 Mini planet

Assume you've been given a panorama image. Use the stereographic projection to yield the popular mini planet view. See e.g. http://en.wikipedia.org/wiki/Stereographic_projection, http://www.miniplanets.co.uk/ .

3.1. Implement `Image pano2planet(const Image &pano, int newImSize, bool clamp=true)`. Make a new image of square size (`newImSize`), and for each pixel `(x, y)` in the new image, compute the polar coordinates `(angle, radius)` assuming that the center is the floating point center as in `blendingweights`. Map the bottom of your input panorama to the center of the the new image and the top to a radius corresponding to the distance between the center and right edge. The left and right sides of the input panorama should be mapped to an angle of 0, along the right horizontal axis in the new image with increasing angle in the output corresponding to sweeping from left to right of the input panorama. Assume standard polar coordinate conventions (angle is 0 along right horizontal axis and $\frac{\pi}{2}$ is along the top vertical axis). Use `interpolateLin` to copy pixels from panorama to planet image. Hint: see C++'s `atan2`.

Here's a Boston winter panorama, and the resulting planet. Note that there is a rough line in the middle of the sky. This is because the panorama is not really 360 degrees. By contrast, the Mars panorama on the right is 360, and we have a much smoother planet result (Mars pano credit: http://mars.nasa.gov/mer/gallery/panoramas/spirit/2005.html):



4 6.865: Stitch N Images (6.815: Extra Credit 10%)

In this section you're going to compose a larger panorama using N images! Finally, some **really** fun stuff!

4.1. Implement `vector<Matrix> sequenceHs(vector<Image> ims, float blurDescriptor=0.5, float radiusDescriptor=4);` which computes a sequence of N-1 homographies for N images. `H[i]` should take `ims[i]` to `ims[i+1]`.

4.2. Implement `vector <Matrix> stackHomographies(vector <Matrix> Hs, int refIndex);`. This takes the N-1 homographies from the previous function, and translates them into N homographies for the N images. `H[i]` takes `ims[i]` to image `ims[refIndex]`. Therefore, `H[refIndex]` should be identity. Note that this requires some chaining of pairwise homographies to get the global homographies. Pay attention: things are different before and after the reference image.

4.3. Implement `vector<float> bboxN(const vector<Matrix> &Hs, const vector<Image> &ims);`, which takes in N-1 homographies and N images, and computes the overall bounding box.

4.4. Implement `Image autostitchN(vector<Image> ims, int refIndex, float blurDescriptor=0.5, float radiusDescriptor=4);`, which computes the sequence of homographies using `sequenceHs`, then propagates those homographies using `stackHomographies`, then computes the overall bounding box, then the translation of the bounding box to (0,0), and finally applies the homographies to all images to get the panorama. **Use linear blending**.

Here's our answers with refIndex=1 for Boston and guedelon:

5 Make your own panorama

Capture your own sequence and run it through your automatic algorithm. Two images for 6.815, at least three images (and use the N stitching) for 6.865.

Make sure you keep the camera horizontal enough because our naive descriptors are not invariant to rotation. Similarly, don't use a lens that is too wide angle (Don't push below a 35mm equivalent of 24mm). Your total panorama shouldn't be too wide angle (don't go too close to 180 degrees yet) because the distortions on the periphery would lead to a very distorted and ginormous output. Some of the provided sequences are already pushing it. Finally, recall that you should rotate around the center of projection as much as possible in order to avoid parallax errors. This is especially important when your scene has nearby objects.

Turn in, via stellar, both your source images and your results.

If you need to convert images to png, one online tool that appears to work is http://www.coolutils.com/online/image-converter/

6 Halide preparation

The next two problem sets will use Halide. It will be useful if you have Halide set up so that you do not waste time with compilation or set up issues next week. If you have trouble setting it up, please ask Fredo during his office hours or ask the TAs (especially Gaurav) during the office hours of problem set 8.

6.1 Obtaining Halide

We recommend that you use precompiled binaries of Halide from https://github.com/halide/Halide/releases. Use the trunk version for your favourite operating system. Linux will be easiest, followed closely by Mac. Windows should be more troublesome, but not impossible. We strongly recommend Windows users use Athena for the Halide problem sets.

6.2 Compiled from source

If the precompiled binaries do not work right away, you can compile Halide from source, which is what most of us who actually use Halide have done. You will find instructions at https://github.com/halide/Halide (see "Building Halide").

But try the precompiled binaries first.

6.3 Testing

Copy `Makefile.halide.tutorials` into the unzipped directory (containing `bin`, `include`, `tutorial`). Run `make -f Makefile.halide.tutorials` and you should see all the tutorials compiling. You can run them from the `bin` directory.

You can find Halide documentation and examples at https://github.com/halide/Halide/wiki. Get in touch with TAs (especially Gaurav) if things don't work or post on Piazza.

7 Extra credits

If you implement any extra credit, you must include test cases in your `a8_main.cpp` file.

6.865: Stitch N images with two-scale blending (5%) That about sums up the instructions :).

Automatically crop margins (5%) It's not trivial, but not too hard. Find a reasonable way to crop out the black margins automatically. Be careful not to crop too much from the Image.

Cylindrical reprojection (5%) We can reproject our panorama onto a virtual cylinder. This is particularly useful when the field of view becomes larger. This is not a difficult task per say, but it requires you to keep track of a number of coordinate systems and to perform the appropriate conversions. For this, it is best to think of the problem in terms of 3D projection onto planes vs. cylinders.

At the end of the day, we will start with cylindrical coordinates, turn them into 3D points/rays, and reproject them onto planar coordinate systems to lookup pixel values in the original images.

The projection matrix for a planar image when the optical axis is along the z coordinates is

$$K = \begin{pmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

where f is the normalized focal length, corresponding to a sensor of width 1.0.

This projects 3D points into 2D homogenous coordinates, which need to be divided by the 3rd component to yield Euclidean coordinates. The coordinates in the sensor plane are assumed to go from -0.5 to 0.5 for the longer dimension.

We then need to convert these normalized coordinates into [0..width, 0..height]. Define `size=max(height, width)`, then the normalized coordinates

$$S = \begin{pmatrix} size & 0 & width/2 \\ 0 & size & height/2 \\ 0 & 0 & 1 \end{pmatrix}$$

In the end, for the reference image, we have

$$P_{2D} = SKP_{3D}$$

We also know that for another image

$$P_{2D}^{i} = H^{ref \to i} P_{2D}^{ref}$$

Now that we have equations for planar projections, we compute the cylindrical projection of one image. We interpret the output pixel coordinates as cylindrical coordinates y, θ (after potential scaling and translation). y is the vertical dimension of the cylinder and θ the angle in radian. We convert these into a 3D point P_{3D}, which we reproject into the source image where we perform a bilinear reconstruction.

I encourage to debug this using manually-set bounding boxes (e.g. $-\pi/2..\pi/2$ in θ, and a scaling factor that maps preserves the height of the reference image).

You can then, if you want adapt your bounding box computation. Note that cylindrical projections are not convex, and taking the projection of the 4 corners does not bound the projection. You can ignore this and accept some cropping or sample the image boundary more finely.

Horizon correction for cylindrical reprojection (5%) The y axis of the reference image is not necessarily the vertical axis of the world. This might result in some distorted reproduction where the horizon is not horizontal.

You can address this by fitting a plane onto the centers of the panorama source images in the 3D coordinate system of the reference image.

8 Submission

Turn in your images to stellar, and everything else to the submission system. 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 code must compile with all of the testers uncommented in the main function!**

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?