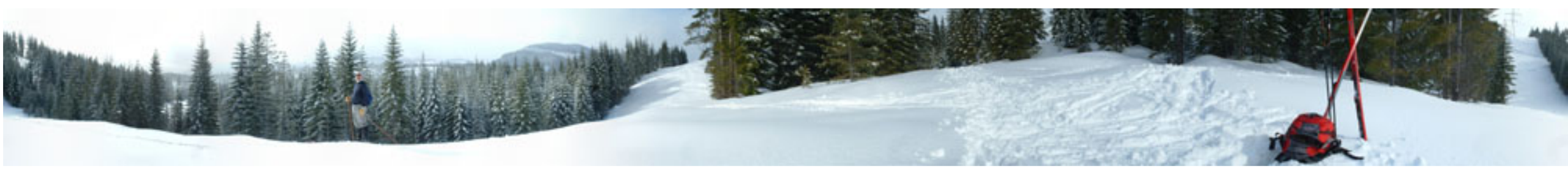
Project 2: Automatic Panoramic Mosaic Stitching

Due: 11:59pm, March 2nd, 2015

Synopsis

In this project, you will implement a system to combine a series of photographs into a panorama. Your software will automatically align the photographs (determine their overlap and relative positions) and then blend the resulting photos into a single seamless panorama. You will then be able to view the resulting panorama inside an interactive Web viewer. First, you should download the test images and skeleton code. Also read the javascript plugin of 360 degree panorama viewer.



Running from the command line

To run from the command line, click the Windows Start button and select "Run". Then enter "cmd" in the "Run" dialog and click "OK". A command window will pop up where you can type DOS commands. Use the DOS "cd" (change directory) command to navigate to the directory where Project2.exe is located. Then type "panorama" followed by your arguments. If you do not supply any arguments, the program will print out information on what arguments it expects.

Running from a shortcut

Another way to pass arguments to a program is to create a shortcut to it. To create a shortcut, right-click on the executable and drag to the location where you wish to place the shortcut. A menu will pop up when you let go of the mouse button. From the menu, select "Create Shortcut Here". Now right-click on the short-cut you've created and select "Properties". In the properties dialog select the "Shortcut" tab and add your arguments after the text in the "Target" field. Your arguments must be outside of the quotation marks and separated with spaces.

Running the skeleton program

1. Select the "ImageLib" project in the Solution Explorer (do NOT select the "panorama" project, for some reason this won't work).
2. From the "Project" menu choose "Properties" to bring up the "Property Pages" dialog.
3. Select the "Debugging" Property page.
4. Enter your arguments in the "Command Arguments" field.
5. Click "Ok".
6. Now when you execute your program from within Visual Studio the arguments you entered will be passed to it automatically (Visual Studio 2010 is preferred. If you use other versions and find problems, let the tutor know).

Note: The skeleton code includes an image library, ImageLib, that is fairly general and complex. It is NOT necessary for you to peek extensively into this library! Here is a synopsis of what you will need to understand from it.

1. The image object, *CImg<float>*, overloads the operation (). You can access a single pixel at coordinates *(x,y)* and channel *c* by *im(x,y,0,c)*, where *im* is an instance of class *CImg<float>*.
2. The image shape object, *CShape*, has three important members: *width*, *height* and *nBands*. These three members represent the width, height and the number of channels of an image. You can construct an instance of *CShape* by its constructor *CShape(int width, int height, int nBands)*.
3. The file in ImageLib you should examine more carefully is Transform.h. You will need the *CTransform3x3* class.

To Do

1. Warp each image into spherical coordinates. (file: *WarpSpherical.cpp,* routine: *WarpRDField*, *WarpSphericalField)*

   [**TODO**] Compute the inverse map to warp the image by filling in the skeleton code in the *WarpRDField* and *WarpSphericalField* routines to:

   a. *WarpSphericalField* converts the given spherical image coordinate into the corresponding planar image coordinate using the coordinate transformation equation from the lecture notes
   b. *WarpRDField* applies radial distortion using the equation from the lecture notes

   (Note: You will have to use the focal length *f* estimates for the half-resolution images provided above (you can either take pictures and save them in small files or save them in large files and reduce them afterwards) . **If you use a different image size, do remember to scale *f* according to the image size. resource).**

2. Compute the alignment of the images in pairs. (file: *FeatureAlign.cpp*, routines: *alignPair*, *countInliers*, and *leastSquaresFit*)

   To do this, you will have to implement a feature-based rotational motion estimation. The skeleton for this code is provided in *FeatureAlign.cpp*. The main routines that you will be implementing are:

   int alignPair(const FeatureSet &f1, const FeatureSet &f2, const vector<FeatureMatch> &matches, MotionModel m, float f, int width, int height,
   int nRANSAC, double RANSACthresh, CTransform3x3& M);

```
int countInliers(const FeatureSet &f1, const FeatureSet &f2, const vector<FeatureMatch> &matches, MotionModel m, float f, int width, int height,
        CTransform3x3 M, double RANSACthresh, vector<int> &inliers);

int leastSquaresFit(const FeatureSet &f1, const FeatureSet &f2, const vector<FeatureMatch> &matches, MotionModel m, float f, int width, int
        height, const vector<int> &inliers, CTransform3x3& M);
```

alignPair takes two feature sets, f1 and f2, the list of feature matches, and a *motion model* (described below), and estimates and inter-image transform matrix M. In this project, the 3D rotation motion model is being used. *AlignPair* uses RANSAC (RAndom SAmpling Consensus) to pull out a minimal set of feature matches (*two* non-identical match for this project), estimates the 3D rotation matrix, and then invokes *countInliers* to count how many of the feature matches agree with the current motion estimate. After repeated trials, the motion estimate with the largest number of inliers is used to compute a least squares estimate (by calling *leastSquaresFit* and passing in the two indices of the randomly selected points, see below) for the motion using the SVD formula presented in class, which is then returned in the motion estimate *M*.

CountInliers is similar to *evaluateMatch* except that rather than computing the average Euclidean distance, the number of matches that have a distance below *RANSACthresh* is computed uses the 3x3 rotation matrix encoded in M along with the focal length *f*. It also returns an list of inlier match ids.

LeastSquaresFit computes a least squares estimate for the *rotation* using all of the matches previously estimated as inliers. It does this using the knowledge of the focal length *f* and the SVD formula presented in class. For convenience, the formula is repeated here:

1. For every matching pair p=(x,y,f)↔p'=(x',y',f) (**where the (x',y') coordinates are centered at (0,0) at the optic center**), increment the matrix $\boldsymbol{A} = \Sigma_i \, p_i \, p_i'^{T}$

2. Take the SVD of A to obtain $\boldsymbol{A} = \boldsymbol{U}\,\boldsymbol{S}\,\boldsymbol{V}^T$

3. Set $\boldsymbol{R} = \boldsymbol{V}\,\boldsymbol{U}^T$ (return the matrix R in the CTransform3x3 M)

[**TODO**] You will have to fill in the missing code in *alignPair* to:

c. Randomly select *two* valid matching pairs and compute the rotation between the two feature images using *leastSquaresFit*.
d. Call *countInliers* to map points p=(x,y,f) to a point p'=(x',y',f) ~ Rp using the rotation and focal length before counting inliers.
e. Repeat the above random selection nRANSAC times and keep the estimate with the largest number of inliers.
f. Write the body of *countInliers* to count the number of feature matches where the Euclidean distance after applying the estimated transform is below the threshold. (Use the code in *evaluateMatch* as a guide, and don't forget to create the list of inlier ids.)
g. Write the body of *leastSquaresFit* to estimate the rotation matrix using the SVD formula presented above.

3. The routine *initGlobalAlign* in file *GlobalAlign.cpp*, which we have mostly written for you, matches all images (and their features) against each other and stores the results (lists of inlier matching pairs and inter-frame rotation estimates) in AlignMatrix &am. It then initializes the rotation estimates by finding the strongest "link" between different image pairs and factors in these rotations to the global estimate that is returned in vector<CTransform3x3> &ms. Since the logic in this latter process is a little tricky, we have written it for you.

```
int initGlobalAlign(const vector<FeatureSet> &fs, int minMatches, MotionModel m, float f,
  int width, int height, int nRANSAC, double RANSACthresh, AlignMatrix &am,
  vector<CTransform3x3> &ms);
```

[TODO] Implement the missing code in *initGlobalAlign* to compute the pairwise alignment between each two images. You'll need to call your feature matcher, as well as the *alignPair* routine.

4. Stitch and crop the resulting aligned images. (file: *BlendImages.cpp*, routines: *BlendImages, AccumulateBlend, NormalizeBlend*)

[**TODO**] Given the warped images and their relative displacements, figure out how large the final stitched image will be and their absolute displacements in the mosaic (*BlendImages*).

[**TODO**] Then, resample each image to its final location and blend it with its neighbors (*AccumulateBlend, NormalizeBlend*). Try a simple feathering function as your weighting function (see mosaics lecture slide on "feathering") (you may want to set the blend weights before you project on the sphere according to the fact that images may overlap both horizontally and vertically. The distance map is described in [Szeliski & Shum]). However, if you want nicer results, you may want to use Laplacian pyramid blending to compensate for exposure differences. In *NormalizeBlend*, remember to set the alpha channel of the resultant panorama to opaque!

Creating the Panorama

1. Use the above program you wrote to align/stitch images into the resulting panorama using the following set of steps:

    1. If your images have radial distortion that you wish to correct, use a series of commands such as:

        panorama *rdWarp img_i.tga goodimg_i.tga f k1 k2*

        This removes radial distortion and stores the results in *goodimg_i.tga*.

    2. Compute the features for each of the images. You are suggested to use SIFT executable of David Lowe http://www.cs.ubc.ca/~lowe/keypoints/ for computing the feature point.
        You can use the following command to compute the features:
            siftWin32.exe < img_i.pgm > feature_i.key
        Note that the input image should be converted to grayscale pgm format (using IrfanView, for instance).
        Let's say you end up with a file *feature_i.key* for image i.

3. Create a text file (say *image_list.txt*), which looks like this:

   goodimg_1.tga feature_1.key
   goodimg_2.tga feature_2.key
   ...
   goodimg_n.tga feature_n.key

4. To match all of the input images and compute their pairwise alignments, use

   panorama *alignAll image_list.txt orientations.txt minMatches f nRANSAC RANSACthresh bundleAdjustIters [sift]*

   For example, some reasonable values might be:

   panorama *alignAll image_list.txt orientations.txt 20 595 100 10 8 sift*

   This will output a file *orientations.txt* with the global orientations (one rotation matrix per image). Note if you do not implement *bundleAdjust* in extra credits, the value of *bundleAdjustIters* has no use.

5. Finally, stitch the images into the final panorama *outimg.tga*:

   panorama blendAll orientations.txt outimg.tga f blendRadius

   Continuing the above numerical example, this might look like:

   panorama blendAll orientations.txt outimg.tga 595 200

2. Convert your resulting image to a JPEG and paste it on a Web page along with code to run the interactive viewer. Click here for instructions on how to do this. (If it does not work for you. You can use javascript plugin mentioned at the very beginning instead.)

Debugging Guidelines

You can use the test results included in the image folders to check whether your program is running correctly.

In general, to debug your program, you should make sure that a two-image example works correctly, then test out two-frame bundle adjustment, then try with more images. In all cases, produce the blended panoramas (perhaps saving them as different filenames) to check that your algorithm is working correctly.

Turn in a zip package containing

1. The executable (panorama.exe).
2. Code that you wrote (just the .cpp files you modified and any new files you needed).
3. A directory containing an HTML report, which includes at least ***four*** panoramas: (1) the mountain test sequence, (2) two of the spherical panoramas downloadable and (3) an example that is captured by yourself. Each panorama should be shown as (1) a low-res inlined image on the web page, (2) a link that you can click on to show the full-resolution .jpg file, and (3) a viewer as described above. In the report, you also need to have a short description of what worked well and what didn't. If you tried several variants or did something non-standard, please describe them as well.
4. Submit your package via eLearn system; before the deadline. Please remember to include your name and student ID. If the package is too large, you are recommend to put it somewhere and let the tutor download it.

Extra Credit

Here is a list of suggestions for extending the program for extra credit. You are encouraged to come up with your own extensions. We're always interested in seeing new, unanticipated ways to use this program!

- The routine *bundleAdjust* in *project3.cpp* takes the current set of rotation estimates in vector<CTransform3x3> &ms, computes the linearized set of update equations needed for the least squares update, solves the least squares system, and then updates the rotation matrices.

  int bundleAdjust(const vector<FeatureSet> &fs, MotionModel m, float f, int width, int height, const AlignMatrix &am, vector<CTransform3x3> &ms, int bundleAdjustIters);

  In order to guard against "uphill" steps in the energy minimization, the Levenberg-Marquardt algorithm is used, which multiplies the diagonal of the normal equations by a factor (1+λ), with λ being increase/decreased according to the success of the previous iteration. You may implement the two missing pieces of the *bundleAdjust* routine that compute the linearized set of equations and perform the rotation matrix update For each panorama. Show the result **with and without** bundle adjustment.

  1. The set of linear equations matrix is pre-allocated for you: the number of equations is three times the number of total feature matches, and the number of variables is three times the number of frames (one 3-D incremental rotation vector per frame/rotation). Fill in the appropriate terms in A and b, i.e.,

     $$[q_{ij}]_{\times}\omega_i - [q_{ik}]_{\times}\omega_k = q_{ij} - q_{ik}, \qquad q_{ij} = R_j p_{ij}$$

  2. Update the rotation matrices R_j using Rodriguez's formula, i.e.,

$$\mathbf{R}(\hat{\mathbf{n}},\theta) = \mathbf{I} + \sin\theta\,[\hat{\mathbf{n}}]_\times + (1-\cos\theta)\,[\hat{\mathbf{n}}]_\times^2, \quad \boldsymbol{\omega} = \theta\hat{\mathbf{n}}$$

(the ω_j can be read off three at a time from the solution vector x, as given in the skeleton code).

- Specify the global orientation of the panorama. A simple way of doing this is to rotate the panorama so that the first image is at the equator. This is what the skeleton code does in the routine *fixRotations*—it premultiplies each of the rotation matrices by the inverse of the rotation matrix for the first image. You can update this method to do something fancier if you like, for extra credit. A nicer solution is described in MSR-TR-2004-92.pdf on pages 43-44.
- Sometimes, there exists exposure difference between images, which results in brightness fluctuation in the final mosaic. Try to get rid of this artifact.
- Try shooting a sequence with some objects moving. What did you do to remove "ghosted" versions of the objects?
- Implement a better blending technique, e.g., poisson image blending or graph cuts.
- Anything else?

Panorama Links

- Panoramas.dk: weekly archive of full-screen, high-quality panoramas worldwide
- VR Seattle: Seattle & Washington panorama
- Peru Panoramas
- A complete set of test images