Visual Modelling Assignment Report

Eron Steger
Kevin Forbes

Introduction

Peleg et al's 2000 paper "Mosaicing on Adaptive Manifolds" presents a method for extracting mosaics from video sequences taken from a moving camera. The fidelity of the reconstructed mosaic is increased through the use of rectification, which ensures that lines that are parallel in the scene remain parallel in the mosaic, despite distortions introduced by camera tilt.

For this project, we have created an application that implements several of Peleg's ideas. Both rotational and translation movement is handled correctly, and rectification is used to prevent "curl" in the resulting mosaics.

The program begins by acquiring a sequence of images either from a set of image files or from the frames of an AVI file. Each image is registered with its successor in the sequence, resulting in a 2x3 transformation matrix. Using these matrices, an appropriate area from the input images is warped and pasted into the mosaic. To account for changes in illumination and imperfect image registration, a linear blend is used when pasting these strips to smooth over these discontinuities.

Algorithm Details

Image Registration

Registration is done using the Lucas Kanade method, as described in section 2 of Peleg's paper. Either a 6-parameter affine transformation or a 2-parameter translation can be used for the warp function, at the user's discretion. The 2-parameter warp is incapable of properly registering frames that differ in anything other than translation, but runs much faster than the 6-parameter warp. All stages of the registration procedure operate on 32-bit floating point greyscale images, so the input images are first converted to this format before processing.

The registration procedure begins with the construction of a Gaussian pyramid from each frame. At each level in the pyramid is constructed by applying a 5x5 Gaussian blur filter, and sampling every-other row and column. The number of levels in the pyramid is determined by the following equation:

levels = ceil(log(min(w, h)) / size) / log(2)
w, h: dimensions of the image
size: the desired smallest dimension of the top level of the pyramid

The user may specify the size parameter, which has a default value of 32 pixels. The Lucas Kanade procedure is applied at each level of the pyramid, starting at the top (the smallest image). The final estimate for the warp determined at each level of the pyramid is used as the initial estimate for the next level. This coarse-to-fine procedure requires less computation by doing most of the search with lower-resolution images. It also helps to avoid local minima by dealing with the image at multiple scales, first registering large image features at the high pyramid levels before trying to match smaller details that arise in the images' textures.

The Lucas Kanade algorithm halts when the L2 norm of the update to the transform falls below a certain threshold. The user specifies the threshold value to use at the top level of the pyramid. The threshold is increased by a factor of 10 with each step in the coarse-to-fine progression. This is done for two reasons: iterations are much cheaper on the coarser levels, and it was found that running too many iterations at the finer levels was often counterproductive, as Lucas Kanade would become confused with texture details.

Strip Determination

Strip alignment is handled as described by Peleg for rectified mosaic using asymmetric strips. For each frame, there is an anchor at the same spot, representing one side of the strip. To determine the other side of the strip for a frame, we determine the location of the next frame's anchor in the current frame by using the transformation matrix calculated during image registration. This constructs a quadrilateral, which is then warped into a rectangular strip to insert into the mosaic. For horizontal mosaics, the width of the strip is the longest distance from an anchor point to the equivalent point on the transformed anchor, while the height is the height of the images. For vertical mosaics, this is reversed.

Strip Warping

Let **P**, **Q**, **P'**, and **Q'** be the points of the quadrilateral on the anchor and the points on the transformed anchor. Warping a strip from a quadrilateral to a rectangular strip is done by bilinear interpolation on these vertices. For a pixel S(x,y) on normalized strip of size 1.0x1.0:

$$\mathbf{L} = (1-y)\mathbf{Q} + y\mathbf{P}$$
$$\mathbf{R} = (1-y)\mathbf{Q'} + y\mathbf{P'}$$
$$S(x,y) = (1-x)\mathbf{L} + x\mathbf{R}$$

Strip Blending

When pasting the images into the mosaic, it is necessary to blend the strips together to account for any change in luminance between frames and to hide error due to the alignment not being exact. For each pixel in the mosaic, the pixel color is determined by a weighted average of the strips going through that pixel. This weighting linearly changes from the center of the strip to its left and right boundaries. The specifics are as follows:

Let:

$M(\mathbf{p})$ represent the mosaic image
$M_{sum}(\mathbf{p})$ represent the weighted sum of all strips in the mosaic
$M_{weight}(\mathbf{p})$ represent the sum of the weights in the mosaic
$S_i(\mathbf{q})$ represent strip i in the mosaic
$W_i(\mathbf{q})$ represent the weight of a point on strip i
$T_i(\mathbf{p})$ be a transformation from a point p in the mosaic to its location on strip i.

We calculate $M(\mathbf{p})$ by calculating the weighted average:

$$\begin{aligned} M_{sum}(\mathbf{p}) &= \textstyle\sum_i S_i(T_i(\mathbf{p}))\, W_i(T_i(\mathbf{p})) \\ M_{weight}(\mathbf{p}) &= \textstyle\sum_i W_i(T_i(\mathbf{p})) \\ M(\mathbf{p}) &= M_{sum}(\mathbf{p}) / M_{weight}(\mathbf{p}) \end{aligned}$$

where the weighting for each strip is defined as:

$$W_i(\mathbf{p}) = |(\text{width}_i)/2 - p_y| / \text{width}_i$$

Implementation

The program was written in C/C++ and developed for the Win32 environment. Intel's OpenCV imaging library was used for image processing, matrix manipulation, and file I/O. Using this library allowed us to focus upon the details of the algorithm, without worrying about details such as how to load image file formats. The library's routines are also very fast, which helps our program run at an acceptable rate. Since large monolithic operations performed within the libraries were often found to be much faster than combinations of smaller operations glued together with user code, we found that the programming style that resulted was very similar to that used in Matlab.

A good example of this is in the program's image warping function. Early in the program's development, we had implemented this function using our own code. Poor speed performance led us to profile the code, which revealed that roughly 60% of the time spent registering images was tied up in the three warping steps taken during each iteration. The limiting factor in the Lucas Kanade algorithm is supposed to be the computation of the Hessian, so we set about to optimize our warp. In the end, however, we discovered that OpenCV had a built-in image warper, and that it was very fast, so we used it instead.

User Interface

Since the mosaicing procedure requires only minimal user interaction, we decided that a command line interface would be appropriate. This also affords the program the flexibility to be called from scripts, allowing for batch processing. The mosaic created by the program is saved in a BMP file, and may be viewed using any image viewer.

Command Format:

```
mosaic -o [output] [-l|-r|-u|-d] (-t###) (-w##) (-a) [input1] [input2] ...

Files:
output: Filename of output mosaic.
input:  A list of individual image files, or a single avi file if –a is used.

Switches:
-o:     Specify the output file
-l:     Camera moves left. (default)
-r:     Camera moves right.
-u:     Camera moves up.
-d:     Camera moves down.
-q:     Use the quick (translation only) warping function.
-a:     Use AVI file input.
-w##    Specify the width of the top image in the pyramid (default 32)
-t###   Set the LK tolerance at top level of the pyramid (default .01)
```

For input consisting of a set of image files, wildcards are allowed. For an image set with rightwards camera movement, the following would create a panorama called "pano.bmp":

```
mosaic –o pano.bmp –r input*.bmp
```

Here is another example, this time using an AVI to constrct a vertical mosaic where the camera is moving upwards:

```
mosaic –a –o vertical.bmp –u input_up.avi
```

Make sure you properly specify the camera motion or else the mosaics will not come out properly.

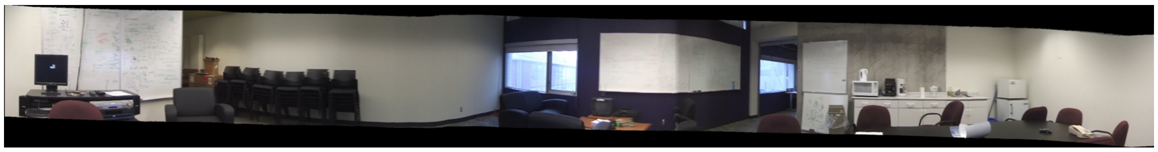
The default values are set up to give good quality results. The user may change the tolerance and width in order to generate mosaics more quickly, but with more liberal settings the algorithm is more sensitive to the quality of the input video. The user can get an idea of the quality of the mosaic by watching the program's run-time output. At each frame, the number of iterations performed by the Lucas Kanade algorithm at each level is displayed. For optimal performance, these numbers should taper off to 1 or 2 at the lowest levels of the pyramid. If a minus sign is printed after the last number, the frame's displacement was against the major direction of camera motion specified in the command line parameters, and the frame was rejected.

Results

Unless otherwise stated, the following results came from image sequences using a tripod with focus locked. While we did attempt to lock lighting on the camera, looking at the images it appears we may not have done this properly.

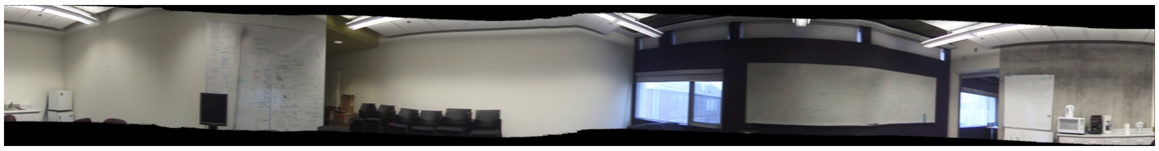
Rotational Motion

Here is a mosaic based on rotational camera movement, with motion stabilized with a tripod:



We were able to capture a full panorama in the DGP meeting area with few visual artifacts. There were some areas where the registration wasn't perfect, but they weren't very noticeable due to the way strips are blended together. Note the somewhat jarring change in lighting while moving the camera across the whiteboard becomes noticeable in the mosaic.

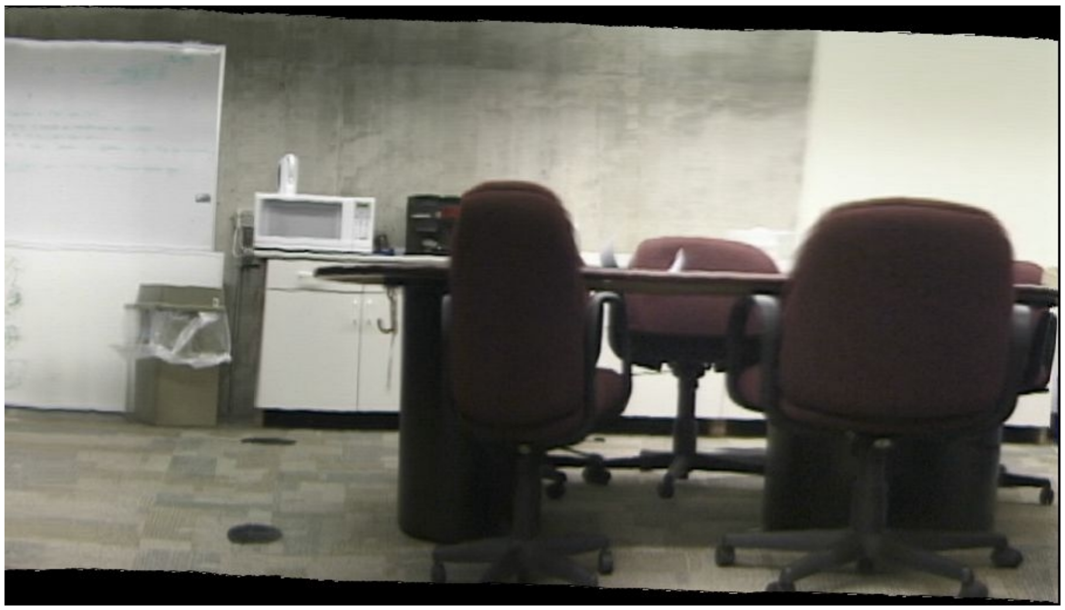
With the camera tilted slightly, we get the following results:



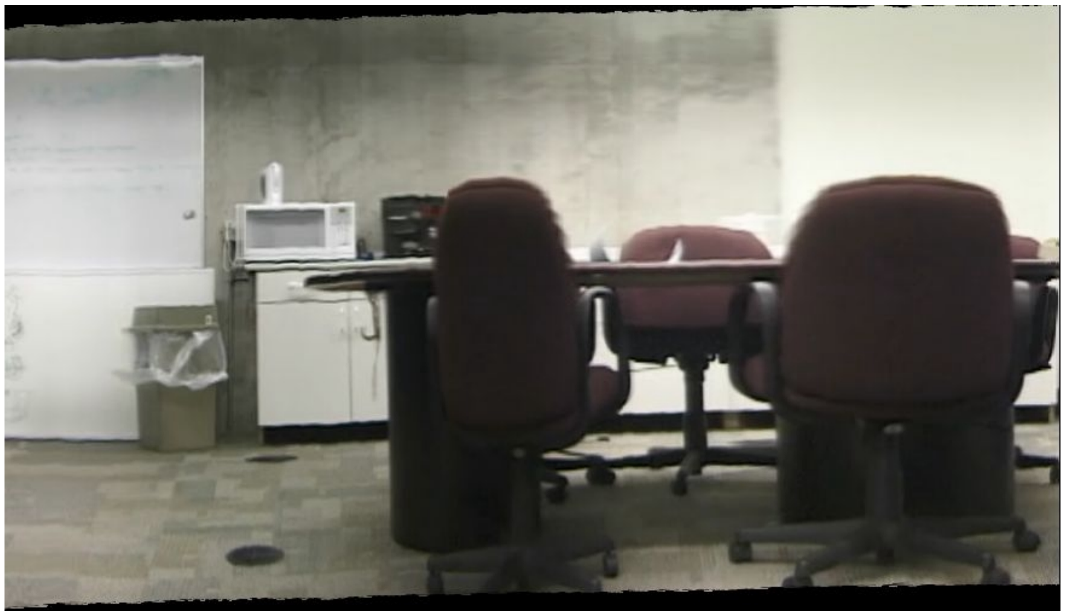
We still get very good results, with the image drifting downward only slightly. This exemplifies the robustness of the rectified mosaicing method described by Peleg. Without properly warping the strips we use from our input images, we would expect to get mosaics that curve.

Translational Motion

We also tested our method on mosaics where the camera was translated, with the camera held still in a chair and carefully slid across the room:

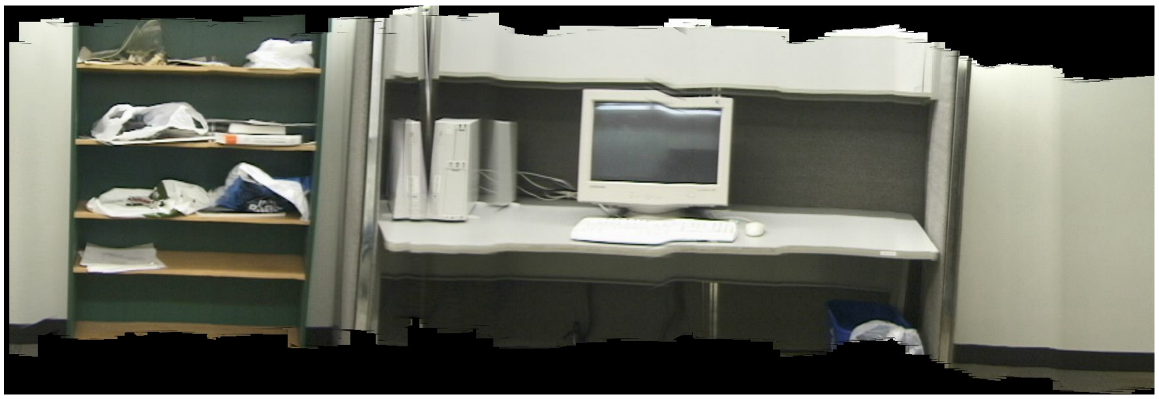


The panorama has been constructed correctly, but notice how it is on an angle. This is because the camera was not sitting flat on the seat of the chair, and the resulting video clip is tilted. This is not the same as curl – parallel lines in the video are still parallel in the mosaic. The video was rotated by 4 degrees in a video processing program, and another mosaic was made. This mosaic, shown below, doesn't suffer from tilting.



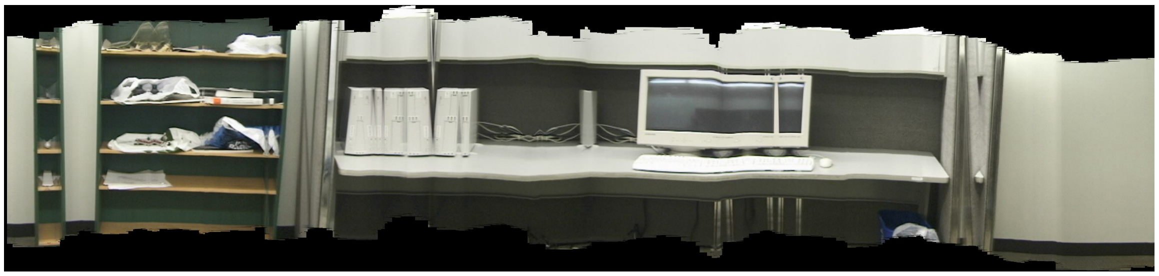
Handheld motion

We used a tripod or other stabilizing surface for most of the panoramas in this report. For this video clip, however, the camera was translated very jerkily by hand:



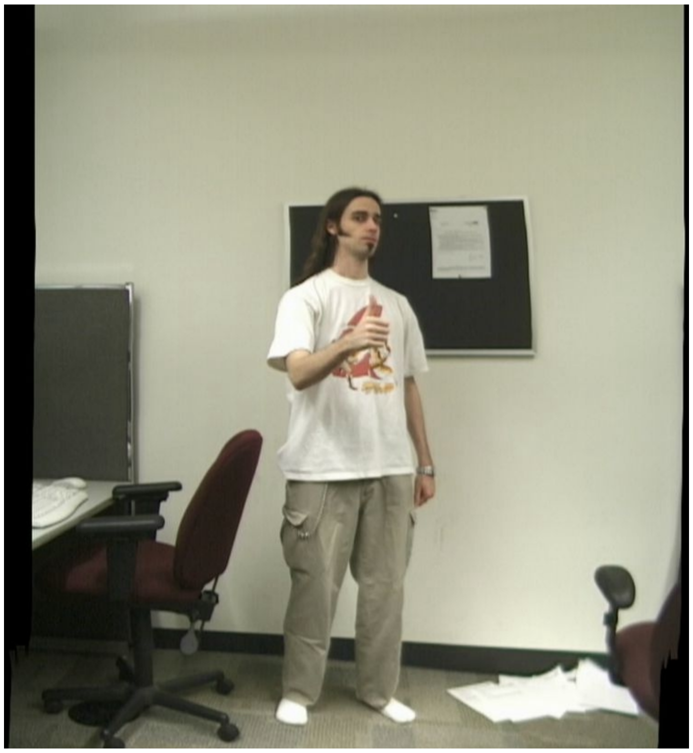
Frames that warp contrary to the major direction of camera motion are rejected, which can lead to large shifts when the correct motion is resumed. The camera made several backwards motions in the video, and these can be seen here at the distortions in the middle of the computer and its monitor. Considering the quality of the input video, we feel that the resulting mosaic is surprisingly good. With more restrictive rules for accepting a strip, we believe the mosaic would look even better.

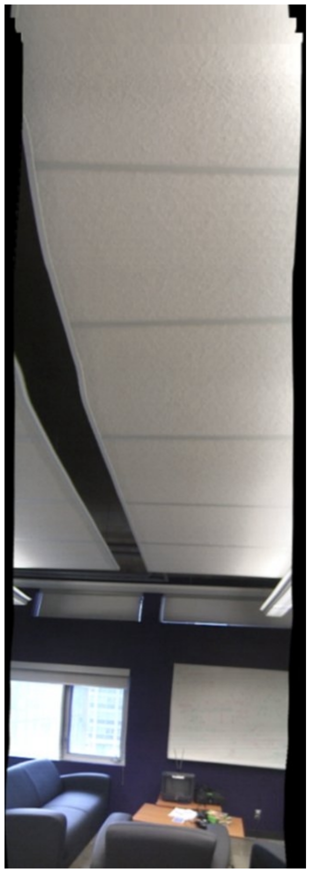
A mosaic made before frame rejection was added is shown below.



Vertical Motion

Our program can create both horizontal and vertical panoramas. Here we see a two mosaic created by tilting a camera vertically:



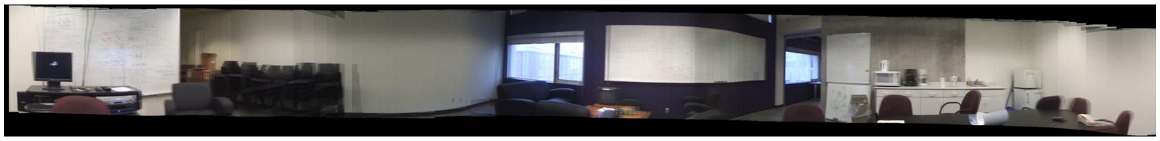


As can be seen, our program properly stitches the images together, with few artifacts.

The vertical mosaics are created in the same manner as horizontal mosaics, so as we would expect we get similar results.

Sparse Input

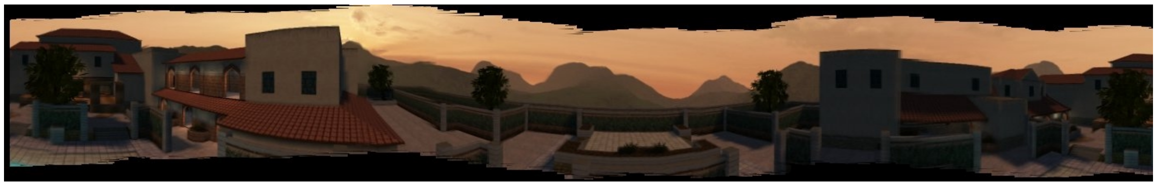
Here we can see the results using more sparse input. The following mosaic is created with every fifth frame of the video used to create the first rotational mosaic:



Unfortunately our program has trouble with image registration, resulting in ghosting effects. This is due to the fact that our registration method relies on a 6-parameter affine transformation, which can only roughly approximate camera rotation. This approximation is only appropriate when there is a small amount of movement between images.

Synthetic Input

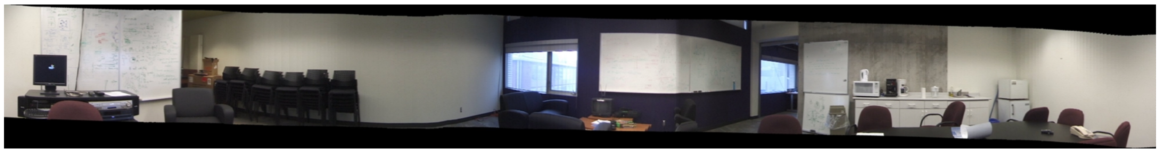
For testing purposes, we created mosaics using rendered images. This rotational mosaic was created from the game Soldier of Fortune 2:



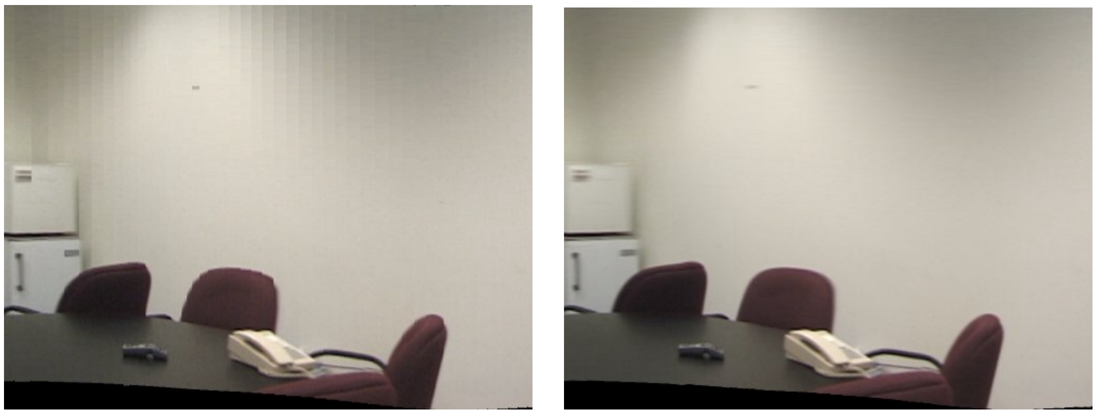
As can be seen we get very good results. In this case we were also able to get good results without image blending, since there is no change in illumination in the input.

Effects of Blending

All the images above use the strip-blending algorithm described earlier. Without it, our first rotational mosaic looks like this:



Here we compare a section of this image on the left with the smoothed version on the right:

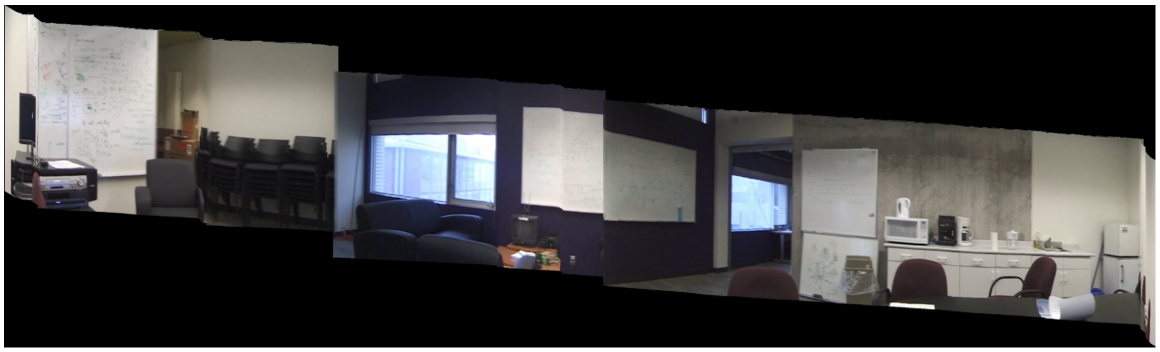


We find that changes in illumination between the input images become noticeable as discontinuities between strip boundaries. Also, problems with image registration become more apparent, as can been in the table at the center of the mosaic. With strip blending, the boundaries of the table look somewhat blurry instead of jagged.

We found that while blending to a small extent appears to give less detailed looking results, the trade-off is more than worth it due to the artifacts that would otherwise occur.

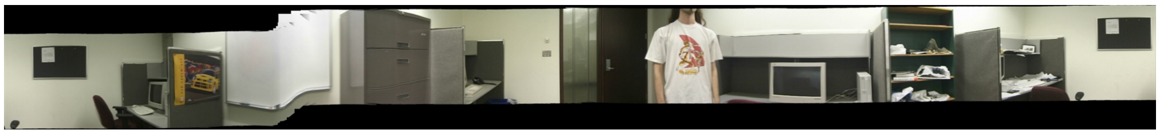
Using the 2-parameter warp

The Lucas Kanade algorithm is much easier to compute when a 2-parameter translational warp is used instead of a 6-paramater affine warp. While this greatly reduces the time to compute a mosaic, quality suffers. For example, here is the rotational example mosaiced with the 2-parameter warp:



Lucas Kanade Failures

The Lucas Kanade algorithm depends upon being able to match structures between successive frames. If the objects in the frames are featureless, this can cause registration to fail. An example of this is shown below:



Registration fails when the camera passes over the blank whiteboard, but recovers several frames later. If there had been writing on the board, this would probably not have happened.

Conclusion

We have shown an implementation of rectified mosaicing, and found that given a video with reasonably stable motion it can construct high quality output. It can handle both translational and rotational motion, along with combinations of the two.

When putting together strips from images to create the mosaic, it is necessary to smoothly blend them together or else artifacts due to changes of illumination. We found that a linear blend worked best, but perhaps different blending strategies would work better for different input.

For the purpose of creating manifold mosaics, registering images using a 6 parameter affine transformation worked well as long as there was a small amount of image movement between frames. Using a 2 parameter translation transformation does not tend to work well for rotational movement. To handle sparse input, it may be necessary to use an 8-parameter perspective warp, or in the case of a stationary camera use a rotational warp [Szelski 1997].

References

S. Baker, I. Matthews, *Lucas-Kanade 20 years on: A Unifying Framework*, International Journal of Computer Vision, 2004.

B. D. Lucas and T. Kanade, *An Iterative Image Registration Technique with an Application to Stereo Vision*, Proc. Image Understanding Workshop, pp. 121-130, 1981.

S. Peleg, B. Rousso, A. Rav-Acha and A. Zomet, *Mosaicing on Adaptive Manifolds*, IEEE Pattern Analysis and Machine Intelligence, v.22, no.10, pp. 1144-1154, 2000

S. Peleg, J. Herman: *Panoramic mosaics by manifold projection*, Computer Vision and Pattern Recognition, 1997.

S. Peleg, A. Zomet, C. Arora, *Rectified Mosaicing: Mosaics without the Curl*, Computer Vision and Pattern Recognition, v.2, 2000.

R. Szeliski and H.-Y. Shum, *Creating Full View Panoramic Image Mosaics and Environment Maps*, Proc. ACM SIGGRAPH, 1997.