What's The Big Picture?
18 - 551
Group 3
Fall 2005

Ali Naveed (anaveed@andrew.cmu.edu)
Carl Yang (carlyang@andrew.cmu.edu)
Kevin Smith (kevinsmi@andrew.cmu.edu)

December 12, 2005

Figure 1: 8 images taken on the CMU campus

Figure 2: 1 panoramic image combining all 8 using our application

Contents

1 Background **4**
1.1 History . . . 4
1.2 Applications . . . 4
1.3 Prior 551 Projects . . . 5
1.3.1 Spring 2001 . . . 5
1.3.2 Spring 2003 . . . 5

2 Implementation **5**
2.1 Overview . . . 5
2.2 Algorithms . . . 6
2.2.1 SIFT Keypoint Detector . . . 6
2.2.2 Keypoint Matching . . . 8
2.2.3 RANSAC . . . 10
2.2.4 Warping . . . 12
2.2.5 Blending . . . 14
2.3 Data Flow . . . 16
2.4 A Note About Image Formats . . . 17

3 Testing **17**
3.1 SIFT Keypoint Detector . . . 17
3.2 RANSAC . . . 18
3.3 Warping . . . 19
3.4 Blending . . . 20

4 EVM Results **22**
4.1 Test Data . . . 22
4.2 Memory Management . . . 22
4.3 Data Transfers . . . 23
4.3.1 PC ->EVM . . . 23
4.3.2 EVM ->PC . . . 23
4.4 Processing Time . . . 24
4.4.1 Optimization Level Local (-o1) . . . 24
4.4.2 Optimization Level Function (-o2) . . . 24
4.4.3 Optimization Level File (-o3) . . . 25
4.5 Profiling . . . 25
4.6 Discussion . . . 26

5 Web References **27**

1 Background

1.1 History

Cameras are restricted by their field of view, and as a result, images with a large field of view must be created by combining several smaller images. In the past, images were manually pieced together, but this gave very poor results and was very inconvenient.

After the development of airplane technology, aerophotography became an exciting new field. The limited flying heights of the early airplanes and the need for larger photo maps forced image experts to construct mosaic images from overlapping photographs.

The need for mosaicing continued to increase later in history as satellites started sending pictures back to Earth. Improvements in computer technology became a natural motivation to develop computational techniques to solve these problems.

Image mosaicing is the technique that has been introduced to address these issues. By collecting several overlapping images and combining them together one can generate a larger area of view. Also, image mosaicing is currently being widely used for research in photogrammetry, computer vision, image processing, medical imaging, real rendering, robot vision and computer graphics.

1.2 Applications

- **Texture Mapping** — Texture mapping is the method of adding realism to a computer generated graphic. By mapping the mosaic onto an arbitrary texture mapped surface we can explore the entire virtual environment.

- **Satellite Imagery** — Satellite imagery generates larger areas of view on distant moons and planets. It was widely used by the mars pathfinder to generate images of Mars.

- **Forensic Analysis** — Image mosaicing is heavily used in forensic analysis to generate larger areas for crime investigation purposes.

- **Visual Scene Representation** — The complete description of visual scene and scene models often entails the recovery of depth and parallax information as well.

- **Image Based Rendering** — These systems combine computer vision and computer graphics algorithms to create a model directly from images and then use this representation to generate new photorealistic views from viewpoints other than the original images.

1.3 Prior 551 Projects

1.3.1 Spring 2001

When we started this project we researched previous projects and found a project that was similar to ours. In Spring 2001, a group did a project on "Panoramic Image Mosaics" which had the same idea as us of combining images. The algorithms they used were completely different and they did not account for calculating any sort of correspondences between the two images.

They used an error minimization technique known as Levenberg-Marquard to come up with the best transformation matrix that relates two individual images and kept iterating over and over until the error was minimized to a certain threshold.

In addition they only accounted for four degrees of freedom which gave them limited ability in warping as it only accounted for rotation and change in focal length. Further they assumed that focal length was constant. We on the other hand accounted for all forms of transformation since our transformation matrix consisted of eight degrees of freedom.

1.3.2 Spring 2003

In spring 2003 a group worked on trying to improve the stability of a sequence of video images. Although not close to our project, they looked at different frames to calculate the similarity which they did by edge detection. They were able to detect blocks that were similar in each frame and then tried to combine them in one. We calculated similarities between two frames by calculating possible point matches and then proceeding with our task of combining them.

2 Implementation

2.1 Overview

The aim of our project was to create an image mosaicing application which utilized the C67 DSP (digital signal processor) board from TI (Texas Instruments). Figure 2 given at the beginning of this document illustrates our application. We wanted our application to be able to combine overlapping images regardless of the type of transformation between them (combining satellite imagery requires detecting a different type of transformation between images than that of the overlapping images of a panoramic image, for example). In order to combine overlapping images there are a number of steps to go through:

1. **Detect "Interesting" Points**
   In order to combine two overlapping images (We can only work on two images at a time, but after combining two images, we have a new image which we can combine with a third image. In this manner, we can combine any number of images, two at a time), we have to know where one point in the first image would appear in the second image. The first step in

this process is to detect points in both images which we should be able to detect again even if the images are changed in some manner.

2. **Find Matching Points**
   After we have found all these points in both images, we must find which of these points appear in both images. If we can do that, then we will be able to take any point in the first image and calculate its corresponding location in the second image.

3. **Generate A Mapping Function**
   Once we have generated a list of probable matches between the two images, we must use this information to develop some mapping function which can map any point in the first image to its corresponding location in the second image.

4. **Transform An Image**
   After generating our mapping function, we can begin the process of applying it to the first image (or whatever image is not acting as the reference image). After doing so, we will have a new image which properly aligns with the other image.

5. **Blend Images**
   At this point, we have two images which are properly aligned, and we could just lay one on top of the other. If we do it this way, though, it will be obvious that the resulting image was created by combining two smaller images because the edges of both images will be clearly visible. So what we do instead is "blend" one image with the other as we lay it down. By implementing some method of "blending", the edges of the two images will not be visible in the resulting image.

2.2 Algorithms

2.2.1 SIFT Keypoint Detector

SIFT, which stands for Scale Invariant Feature Transform, has been the most improved feature detection method to date. It was created by Professor David Lowe at the University of British Columbia. This method transforms an image into a large collection of local feature vectors, each of which is invariant to image translation, scaling, rotation and partially invariant to illumination changes and affine or 3D projection. Most previous methods such as the Harris Corner Detector lacked invariance to scale, other types of transformation, and were more sensitive to intensity changes. As a result, SIFT can detect many more matches between two images and allows greater flexibility in the amount of change between two images. SIFT consists of four main stages:

1. **Scale Space Peak Selection**
   Potential interest points are identified by scanning an image over location and scale. This is implemented efficiently by constructing a Gaussian

pyramid and searching for local peaks in a series of difference-of-Gaussian (DoG) images. Each point is used to generate a feature vector that describes the local image region sampled relative to its scale-space coordinate frame.

2. **Keypoint Localization**
   Candidate keypoints are localized to sub-pixel accuracy and eliminated if found to be unstable.

3. **Orientation Assignment**
   SIFT identifies the dominant orientations for each keypoint based on its local image patch. The assigned orientation's scale and location for each key point enables SIFT to construct a canonical view for the keypoint that is invariant to similarity transforms.

4. **Keypoint Descriptor**
   The final stage of the SIFT algorithm builds a representation for each keypoint based on a patch of pixels in its local neighborhood.

After running SIFT on an image, we will have a list of sub-pixel (meaning non-integer) coordinates identifying the location of all keypoints discovered by SIFT. The origin of the image coordinate system is the top left corner of the image; thus, a coordinate of $(5, 10)$ would correspond to the pixel 5 columns to the right of and 10 rows down from the top left corner of the image.

Along with the coordinates of each keypoint, SIFT also reports the scale (how many levels of the Gaussian pyramid through which the keypoint persisted) and orientation (a value from $-\pi$ to π indicating the direction of the gradient in the neighborhood of the keypoint) of each keypoint it found.

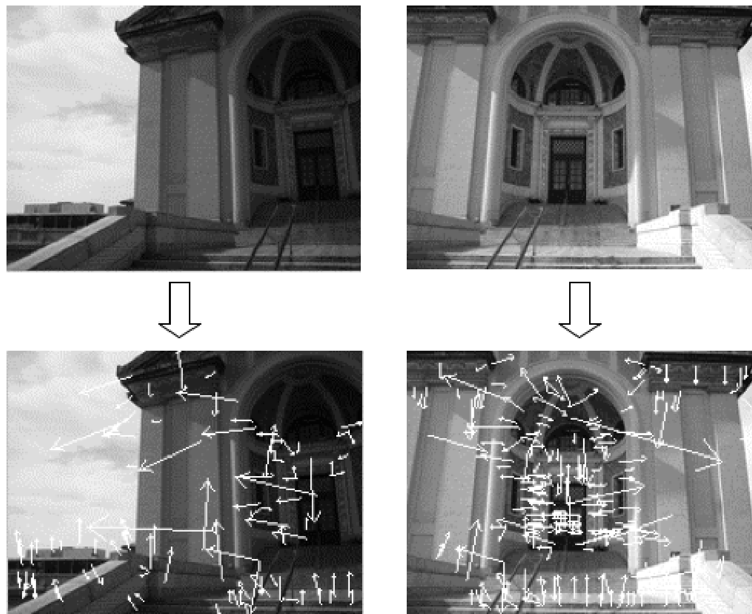
In addition to these four numbers, SIFT returns the keypoint descriptors which it calculated for every keypoint. These keypoint descriptors, which are the most important aspect of SIFT, act as unique IDs for each keypoint. Even if an image is rotated, scaled to a different size, or has its luminosity altered, the same keypoint descriptor value will be calculated for a given keypoint. This is the reason why SIFT can be used to match points in two overlapping images even if the two images are at a different scale, have been rotated, etc.

The keypoint descriptor which SIFT generates is a set of 128 numbers between 0 and 255. As you can see, this means that if two keypoints have the same descriptor, it is virtually guaranteed that they are the same keypoint. Running SIFT on an image, will generate output such as the following:

```
110 128
105.25 119.38 12.39 3.100
 0 7 25 0 0 0 0 0 0 5 11 1 5 5 2 0 0 0 0 0
 4 17 29 11 0 0 0 0 0 0 35 53 16 41 99 2 1 1 0 0
 97 44 23 0 3 5 17 23 39 0 0 0 2 17 120 109 42 0 0 0
 0 1 138 138 42 14 9 2 5 11 3 9 138 44 0 0 0 0 1 8
 138 25 0 0 0 0 9 42 138 24 0 0 0 0 25 86 12 17 30 7
 1 1 3 5 118 87 11 0 0 0 0 6 101 130 46 0 0 0 0 0
```

```
 138 132 2 0 0 0 0 0
83.92 178.24 8.05 2.912
 0 1 55 19 1 3 0 0 6 14 130 71 10 13 1 0 81 63 130 24
 0 0 0 1 130 91 16 0 0 0 2 19 0 0 2 3 5 23 3 0
 21 8 16 14 73 130 9 2 130 50 6 0 8 20 7 17 130 130 0 0
 0 0 0 1 0 0 0 0 4 2 0 0 2 0 15 8 51 83 22 6
 51 6 56 23 5 27 39 66 130 115 42 8 1 0 0 5 0 0 0 0
 0 0 0 0 0 0 53 54 1 1 1 0 6 13 123 97 5 2 4 5
 79 84 56 5 0 0 0 15
...
```

The first line tells us that it detected 110 keypoints and that the length of the descriptor for each keypoint is 128. After the first line, SIFT reports all the data it calculated for all the keypoints it found. For each keypoint, the x coordinate, y coordinate, scale, and orientation are reported on one line, and then the 128 values of the descriptor are given. Shown below is a visual representation of SIFT results:



The arrows are placed at the locations of the keypoints. The length of an arrow signifies the scale of the keypoint at that location, and the angle of the arrow represents the orientation of the keypoint at that location. The images above are images we took of the entrance to Hamerschlag Hall.

2.2.2 Keypoint Matching

Now that we have a list of keypoints for two images, we have to see if any of the keypoints in the first image are present in the second image. There are

different ways of accomplishing this, but we are using the method recommended by Professor Lowe. In this method, we compare every keypoint in the first image to every keypoint in the second image to find its two closest matches in the second image. By compare we mean take the sum of squared differences of the keypoint descriptors.

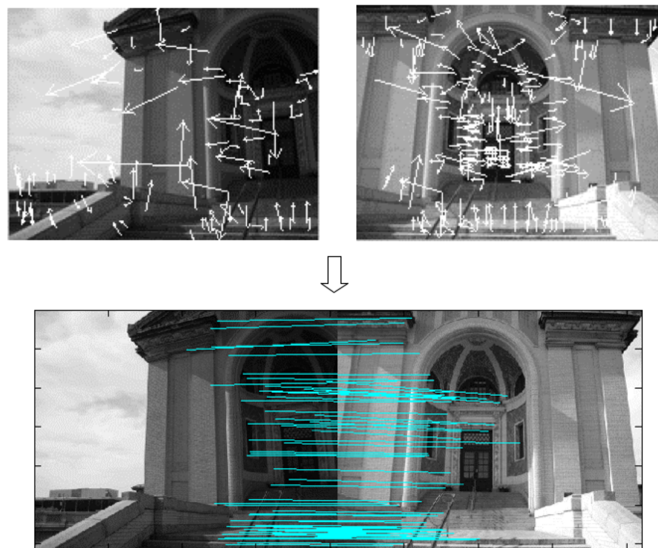
After comparing descriptors and finding the keypoint's two closest matches in the second image, we consider the closest match to be an actual match if it falls within a certain fraction of the second closest match. Professor Lowe's papers suggest using a fraction of 0.6.

This means that if the difference between the keypoint in the first image and its closest match in the second image is within 60% of the difference between the keypoint in the first image and its second closest match in the second image, then the keypoint in the first image and its closest match in the second image are considered to be a match.

This percentage can be increased or decreased to allow for more or less matches. Professor Lowe offers more details in his papers on why a value of 0.6 is chosen for the default value. Running our implementation of this matching algorithm on two sets of SIFT results will generate output such as the following:

```
119.199997 11.540000 45.240002 8.440000
152.130005 52.349998 72.309998 54.750000
152.130005 52.349998 72.309998 54.750000
167.649994 52.849998 87.190002 57.330002
...
```

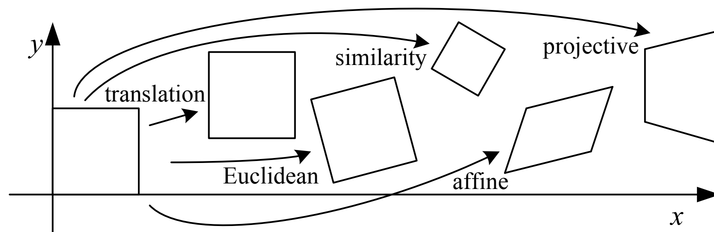
This data tells us that pixel $(119.199997, 11.54)$ in the first image can be found at $(45.240002, 8.44)$ in the second image and so on for all the matches found. Given below is a visual representation of the match results:



The resulting picture shows all matches detected based on the SIFT results given from the two input images.

2.2.3 RANSAC

Before we can go any further, we should explain the idea of image transformations. There are many types of image transformations, and several are given in the figure below:



Many transformations can be represented by a 2x2 matrix where to apply the matrix to an image, you just multiply it by every pixel in the image in the form of a 2x1 matrix:

$$\begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

However, if we want to handle projective transformations, which account for a combination of linear transformations, translations, and projective warps, then we need a 3x3 transformation matrix. To represent the coordinates in 2 dimensions with a 3x1 vector, each pixel coordinate is now in the form of homogeneous coordinates:

$$\begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} wx' \\ wy' \\ w \end{bmatrix}$$

$$where\ w\ is\ a\ scaling\ factor$$

In our case, though, we do not know what transformation was used to get from the first image to the second image. Therefore, our job is to solve for the transformation matrix H. This is why we have to find a list of matches between the two images so that we may solve for this transformation matrix. Given one match, we can derive two equations:

$$(x, y) \rightarrow (x', y')$$

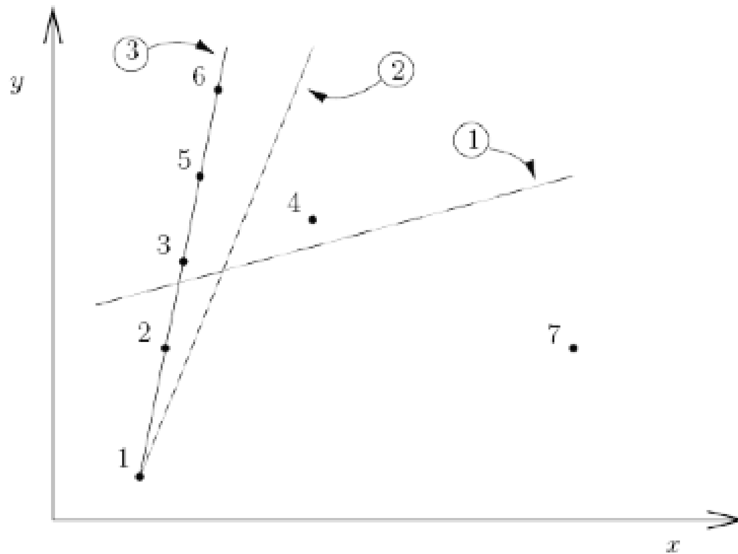
$$h_{11}x + h_{12}y + h_{13} - h_{31}xx' - h_{32}yx' - x' = 0$$

$$h_{21}x + h_{22}y + h_{23} - h_{31}xy' - h_{32}yy' - y' = 0$$

Unfortunately, this gives us eight unknowns with only two equations, but if we are given four matches, we can derive 8 equations:

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x'_1 & -y_1x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y'_1 & -y_1y'_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2x'_2 & -y_2x'_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2y'_2 & -y_2y'_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3x'_3 & -y_3x'_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -x_3y'_3 & -y_3y'_3 \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4x'_4 & -y_4x'_4 \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -x_4y'_4 & -y_4y'_4 \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} = \begin{bmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \\ x'_4 \\ y'_4 \end{bmatrix}$$

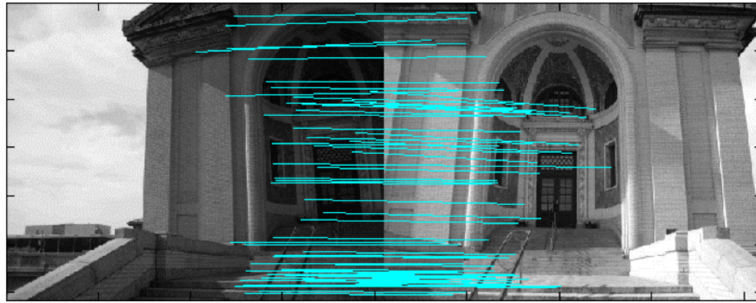
Now that we can calculate an arbitrary transformation matrix H given four matches, we need some model fitting method which will attempt to calculate that H which best fits the list of matches generated by the keypoint matching step above. One popular model fitting method is the "Least Squares" method. The problem with using this method is that if there are incorrect matches in our list, the results of the "Least Squares" method will be skewed. RANSAC (Random Sample Consensus) provides a general technique for model fitting in the presence of "outliners" (incorrect matches in our case). The figure below illustrates these differences nicely. Line 1 is the line calculated using the "Least Squares" method, line 2 is the line calculated using the "Least Squares" method if data point 7 was removed, and line 3 is the line calculated by RANSAC:



As one can see, RANSAC chooses that line which passes through the most number of points whereas the "Least Squares" method chooses that line which is the closest distance to every point. Therefore, we can see that RANSAC is a better choice for our problem.

So to implement RANSAC, we will select four matches at random from our list of matches. We will use these four matches to solve for the matrix H. We will then apply this H to all the coordinates from image1 in the match list and see how close this H gets us to the coordinates of image2 in the match list. More simply, if we apply H to (x_1, y_1) of some match in the match list, and we are within a certain distance of (x_2, y_2) of the same match, then we say this H gave us an inliner. We proceed to apply and test this H against all matches in the match list. We repeat this process of selecting random matches, calculating an H based on those random matches, and testing this H against all matches in the match list for many iterations. After doing this for many iterations, we choose that H which gave us the most number of inliners as our transformation matrix from the first image to the second image.

Below is a visual representation of RANSAC:



$$H = \begin{bmatrix} 1.473109 & -0.175044 & -118.174385 \\ 0.326636 & 1.294937 & -44.596180 \\ 0.002276 & -0.000198 & 1.000000 \end{bmatrix}$$

The picture shows that given the list of matches between our two images of Hamerschlag Hall, our implementation of RANSAC calculated the above transformation matrix.

2.2.4 Warping

Once a homography has been determined by RANSAC, it is possible to warp the first image and then lay it onto the reference image coordinate system. There are two types of warping, forward warping and inverse warping. In forward warping, each pixel from the original image is sent to its corresponding location on the warped image.

$$p_2 = Hp_1$$

$$\begin{bmatrix} wx_2 \\ wy_2 \\ w \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

$$\Rightarrow (x_2, y_2) = (\frac{h_{11}x_1 + h_{12}y_1 + h_{13}}{h_{31}x_1 + h_{32}y_1 + h_{33}}, \frac{h_{21}x_1 + h_{22}y_1 + h_{23}}{h_{31}x_1 + h_{32}y_1 + h_{33}})$$

If after applying the homography, a pixel from the original image lands between pixels on the warped image, then the color from the original pixel must be distributed to the neighboring pixels on the warped image. However, a huge drawback to forward warping is the possibility of holes. If the warped image is bigger than the original image, then all the pixels on the warped image may not be filled in.

In order to solve the problem of holes, we instead implemented inverse warping, also referred to as backward warping. This involves iterating through all the pixels of the warped image, and applying the inverse of the homography to find the corresponding point on the original image.

$$H^{-1}p_2 = p_1$$

$$\begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix}^{-1} \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} wx_1 \\ wy_1 \\ w \end{bmatrix}$$

Before traversing through the warped image, one must determine the necessary size for the final warped image. This can be accomplished quickly by applying the homography to each of the four corners of the original image to obtain the corners of the warped image. We know that all warped image points will lie within the quadrilateral defined by the four resultant corner points. The number of rows in the warped image is determined by rounding up the difference between the top-most and bottom-most corner points. The total number of columns is determined in the same way.

When applying the inverse homography to each pixel on the warped image, there would be a case that the pixel becomes mapped to a coordinate between pixels on the original image. In that case, it is best to interpolate color values from neighbors. We chose bilinear interpolation as it provides the best trade-off between speed and accuracy. Other methods of interpolation include nearest neighbor and bicubic interpolation.

Shown below is a visual representation of our implementation of warping:

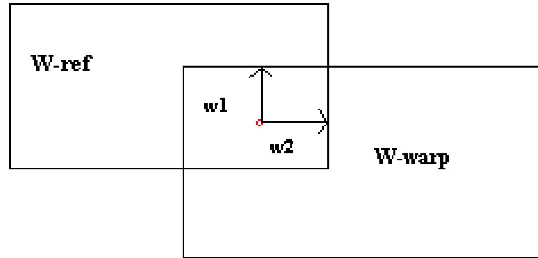
$$\begin{bmatrix} 1.473109 & -0.175044 & -118.174385 \\ 0.326636 & 1.294937 & -44.596180 \\ 0.002276 & -0.000198 & 1.000000 \end{bmatrix}^{-1}$$

Given the original image and the inverse of the H given by RANSAC, we get the resulting image shown above.

2.2.5 Blending

Once the individual images are warped to generate a larger panorama the visible seams that are present must be removed so that it looks more realistic. Simply averaging pixels in the area of overlap is not an effective means of blending. Variation in intensity may make the average value be extremely skewed from what should be an accurate value. This combined with the fact that sometimes there are misregistrations present in the warped image leads to what are known as ghosting effects. This estimate can be greatly improved by assigning weights to each pixel from the nearest edge and then averaging out the values.

The value of the RGB components of a given pixel corresponds to the weighted average of the values of the RGB components of the images that overlap in that point. The weight of the components of a pixel of each image decreases as the distance between that pixel and the center of the image increases. This situation is illustrated below:



w1 - Distance from Warped Image

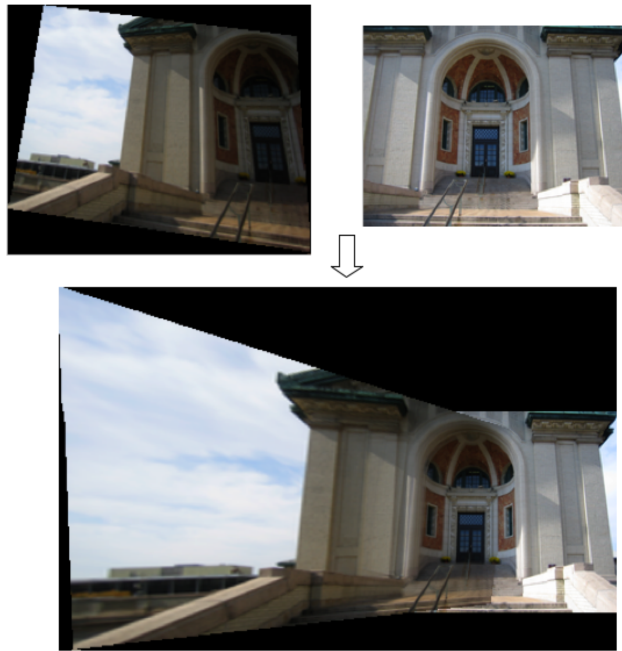
w2 - Distance from Reference Image

The simple feathering technique that we are using to blend the images works like this: Let (x, y) denote a pixel location on the final mosaic. We define the weight w_{ref} to be the minimum distance from (x, y) to any edge of the reference image or zero if (x, y) lies outside of the reference image borders. By the same virtue, we define the weight w_{warp} to be the minimum distance from (x, y) to any edge of the warped image or zero if (x, y) lies outside of the warped image borders. Let $P_i(x, y)$ be the color value at (x, y) of the ith image. The final, feathered, RGB value of the composite mosaic pixel at (x, y) is thus given by:

$$P_{mosaic}(x, y) = \frac{w_{ref} P_{ref}(x, y) + w_{warp} P_{warp}(x, y)}{w_{ref} + w_{warp}}$$

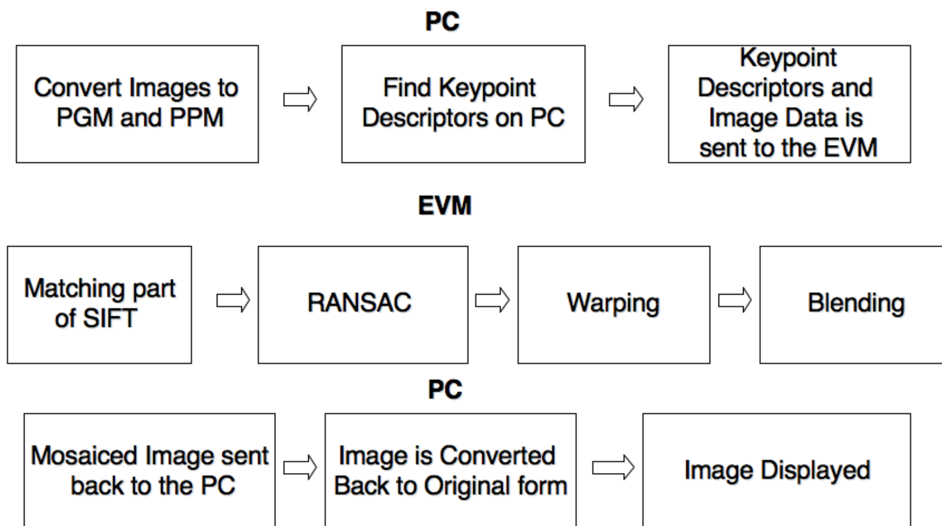
Notice that if the image lies outside the borders of the reference image, then $w_{ref} = 0$ and $P_{mosaic}(x, y) = P_{warp}(x, y)$. The opposite is true if the image lies outside of the borders of the warped image.

Given below is an example of the results of our implementation of blending:



Given the reference image and the warped image, the resulting image shown above was created. Notice that even though one image is considerably darker than the other image, the blending was able to combine the two images rather smoothly.

2.3 Data Flow



2.4 A Note About Image Formats

Images taken by our digital camera were in the JPEG (Joint Photographic Experts Group) format, and user-supplied images could be of virtually any format. However, we converted these images to Portable PixMap (PPM) format, since the PPM format is very simple to read and write.

A PPM file in RAW format is a binary file which contains some basic metadata such as number of rows and columns. This metadata is followed by a stream of bytes where the first three bytes represent the three RGB values of the first pixel, the second three bytes represent the three RGB values of the second pixel, and so on. Thus, the number of bytes is calculated as $(rows * cols * 3)$. After reading the PPM file, we represent the image with the following C structure:

```
typedef struct ImageSt {
    int rows, cols;
    unsigned char ***pixels;
} *Image;
```

The three dimensional array `pixels`, contains the row and column coordinates of each pixel in the first two dimensions and the RGB (red, green, blue) values of each pixel as an array of length 3 in the third dimension. For example, `im1->pixels[1][1][0]` would access the red value of the pixel at (2,2) in the image `im1` ((2,2) because counting starts from 0); `im1->pixels[5][3][1]` would access the green value of the pixel at (6,4) in the image and so on.

For the PPM images we are working with, these color values are an integer between 0 and 255. Thus, to save memory we can define the array using the `unsigned char` data type, which takes up only 1 byte, as opposed to integers, which occupy 4 bytes.

Using this structure makes working with the image data quite easy. However, because we only transmit one-dimensional arrays over the HPI interface between the EVM and the PC, we have a function which "flattens" an image (converts it to a one-dimensional array) before transmitting it. Once received on the other side, we can reconstruct the struct if we know the number of rows and columns.

3 Testing

3.1 SIFT Keypoint Detector

We wanted to test if SIFT was truly invariant to scale among other things. In this section we will demonstrate the capabilities of the SIFT algorithm. Figure 3 shows the images we will use to test SIFT. Notice that image 3(b) has been rotated, given a scale different from that of image 3(a), and is slightly darker. For image 3(a), SIFT detected 1007 keypoints, and it detected 129 keypoints in image 3(b). After running this set of keypoints through the matching algorithm, 50 matches were found.

After combining these two images, we get the results shown in Figure 4. We used image 3(a) as the reference image, so image 3(b) had to be rotated and

enlarged to line up properly with the first image. Clearly, SIFT had no problem with these two images despite their differences.

(a) Reference Image

(b) Pre-Warped Image

Figure 3: SIFT Test Images

3.2 RANSAC

Before beginning our discussion of RANSAC, it's important to mention that the performance of RANSAC depends heavily on the correctness of the matches generated by SIFT. If a high percentage of the matches reported by SIFT are correct, then RANSAC will need very few iterations to develop our transformation matrix.

Another important factor is the threshhold with which we test RANSAC. In our discussion of RANSAC we mention that when testing a transformation matrix, we say it gave us an inliner if the result of applying it to (x_1, y_1) is within a certain distance of (x_2, y_2). If we set this threshhold to a distance of 5 or more, RANSAC will stop improving the transformation matrix after very few iterations (assuming a high percentage of correct matches from SIFT). If, however, we set this threshhold to a value such as 0.5, then RANSAC will continue to improve the transformation matrix for many iterations. Figure 5 shows two images which we will use to test RANSAC (the image on the left, will be acting as our reference image).

These two images are of size 205 x 154, and for the image on the left, SIFT detects 273 keypoints while detecting 326 keypoints for the image on the right. After running these two sets of keypoints through the matching algorithm, 103 matches are found. We then had RANSAC run for 1,000 iterations. Figures 6(a), 6(b), and 6(c) show the results of combining the two images using different threshholds for RANSAC.

Figure 6(a) shows the results of using a value of 10. Using a threshhold

Figure 4: Combining two images of different rotation, scale, and luminosity works fine thanks to SIFT

distance of 10, RANSAC stopped improving after 2 iterations at which time it had found a transformation matrix which gave 100 inliners in the list of 103 matches.

Figure 6(b) shows the results of using a value of 2. Using a threshhold distance of 2, RANSAC stopped improving after 15 iterations at which time it had found a transformation matrix which gave 100 inliners in the list of 103 matches.

Figure 6(c) shows the results of using a value of 0.5. Using a threshhold distance of 0.5, RANSAC stopped improving after 442 iterations at which time it had found a transformation matrix which gave 86 inliners in the list of 103 matches.

The thing to note here is that we could have set the threshhold to 10 and used only a couple iterations of RANSAC. As one can see in the figures, there was no improvement by setting a tighter threshhold and running RANSAC for so many interations. The other thing to note is that this shows that SIFT gave very good results for these two images. If a large percentage of the matches had been incorrect, then the tight threshhold would have been necessary.

3.3 Warping

We said in our introduction that before the advent of computer technology, the only way to combine overlapping images was to lay them on top of one another. Figure 7 shows the results of using this method to combine some overlapping images. As one can see, this leaves a lot to be desired. If we do not warp the images, they will not line up properly when being layed on top of one another.

Figure 8 illustrates the importance of warping. If we warp the images, they

Figure 5: RANSAC Test Images

(a) Threshhold 10 (b) Threshhold 2 (c) Threshhold 0.5

Figure 6: RANSAC Results

will be properly aligned, and we can then lay the images on top of each other and get a realistic result.

Figure 7: Combining Images Without Warping

3.4 Blending

Figure 9 shows the mosaic of two images of Porter Hall without blending.

Figure 10 shows the results of mosaicing the two images with blending turned on. After blending was performed the visible seam that was present was not entirely removed but considerably improved upon. Also, the "ghosting effects" become more visible in the blended picture. The trees in the background are

Figure 8: Combining Images After Warping

clearly more hazed than the same trees in the first image. In addition there is some fuzziness across regions in the side walk. So while Weighted Average Blending considerably improves the ghosting effect which would be much more visible in normal average blending, it does not completely eliminate it.

Figure 9: Mosaic With No Blending

Figure 10: Mosaic With Blending

4 EVM Results

4.1 Test Data

The following results came from running our application on the two images shown above. Both images have dimensions of 205 x 154.

4.2 Memory Management

Given below is our *.cmd file for our project. This file tells Code Composer Studio (IDE provided by Texas Instruments) how to partition memory on the EVM.

```
-heap 0x400000
-stack 0x8000

MEMORY
{
  ONCHIP_PROG (RX) : origin = 0x00000000 length = 0x00010000
```

```
  ONCHIP_DATA (RW) : origin = 0x80000000 length = 0x00010000
  SBSRAM_PROG (RX) : origin = 0x00400000 length = 0x00014000
  SBSRAM_DATA (RW) : origin = 0x00414000 length = 0x0002C000
  SDRAM0 (RW)      : origin = 0x02000000 length = 0x00300000
  SDRAM1 (RW)      : origin = 0x03000000 length = 0x00500000
}

/* Allocates different parts into the different areas of memory */
SECTIONS
{
  .vec:          load = 0x00000000      /* Interrupt vector table */
  .text:         load = SBSRAM_PROG     /* Code */
  .const:        load = ONCHIP_DATA     /* Variables defined with const */
  .bss:          load = ONCHIP_DATA     /* Global variables */
  .data:         load = SBSRAM_DATA
  .cinit         load = ONCHIP_DATA
  .pinit         load = ONCHIP_DATA
  .stack         load = ONCHIP_DATA     /* Stack (for local variables) */
  .far           load = SDRAM0          /* Variables defined with far */
  .sysmem        load = SDRAM1          /* Heap: malloc and friends */
  .cio           load = ONCHIP_DATA
  .ipmtext       load = ONCHIP_PROG     /* Shut the linker up */
}
```

4.3 Data Transfers

4.3.1 PC ->EVM

For PC ->EVM transfers, we transmit image metadata (number of rows, number of columns, and number of keypoints), image data, and keypoint data. The metadata is 24 bytes (6 `int` values), the image data for both images is 94712 bytes (images are 205 x 154 and each pixel requires three bytes - this works out to 94710, but HPI transfers require that the number of bytes be divisble by 4 so it can fill up the 32-bit bus), the keypoint data for the first image is 200640 bytes (380 keypoints stored as `float`), and the keypoint data for the second image is 165792 bytes (314 keypoints). In total, this is 555880 bytes to be sent from the PC to the EVM. We ran this several times, and found an average transfer time of 0.168 seconds which implies an average data rate of 3.16 MB/s. We knew from previous lab work that HPI transfers typically occur at rates of 3 - 5 MB/s.

4.3.2 EVM ->PC

For EVM ->PC transfers, we transmit the resulting image metadata and the resulting image pixel values. For this test, we only timed the transfer of the pixel values and not the metadata because the wait time between these transfers

skewed the results. This amount of data (8 bytes) is negligible anyway. After timing this transfer several times, we calculated an average data rate of 3.5 MB/s.

4.4 Processing Time

Code Composer Studio has the option of setting the optimization level with which the compiler will compile code for the EVM. Below, we present our results from running our application at three different optimization levels in order of increasing optimization.

4.4.1 Optimization Level Local (-o1)

Avgerage Processing Time	67.3 s
Percentage Spent on Reconstructing Data	1.5%
Percentage Spent on FindMatches()	28.7%
Percentage Spent on RANSAC()	2.0%
Percentage Spent on Warp()	63.3%
Percentage Spent on Blend()	3.9%
Percentage Spent on “Flattening” Data	0.5%

Given below is the memory map generated by CCS at this optimization level:

```
ENTRY POINT SYMBOL: "_c_int00"  address: 00413020


MEMORY CONFIGURATION

          name            origin    length      used     attr    fill
----------------------  --------  ---------  --------  ----  --------
ONCHIP_PROG             00000000   00010000  000002c0  R  X
SBSRAM_PROG             00400000   00014000  00013240  R  X
SBSRAM_DATA             00414000   0002c000  00000000  RW
SDRAM0                  02000000   00300000  00004e54  RW
SDRAM1                  03000000   00500000  00400000  RW
ONCHIP_DATA             80000000   00010000  00008cc5  RW
```

4.4.2 Optimization Level Function (-o2)

Avgerage Processing Time	61.0 s
Percentage Spent on Reconstructing Data	2.2%
Percentage Spent on FindMatches()	21.9%
Percentage Spent on RANSAC()	2.4%
Percentage Spent on Warp()	68.9%
Percentage Spent on Blend()	3.8%
Percentage Spent on “Flattening” Data	1.1%

Given below is the memory map generated by CCS at this optimization level:

```
ENTRY POINT SYMBOL: "_c_int00"  address: 00413920


MEMORY CONFIGURATION

         name            origin    length      used     attr    fill
----------------------  --------  ---------  --------  ----  --------
ONCHIP_PROG             00000000   00010000  000002c0  R  X
SBSRAM_PROG             00400000   00014000  00013b40  R  X
SBSRAM_DATA             00414000   0002c000  00000000  RW
SDRAM0                  02000000   00300000  00004e54  RW
SDRAM1                  03000000   00500000  00400000  RW
ONCHIP_DATA             80000000   00010000  00008cc5  RW
```

4.4.3 Optimization Level File (-o3)

Avgerage Processing Time	61.0 s
Percentage Spent on Reconstructing Data	2.8%
Percentage Spent on FindMatches()	21.9%
Percentage Spent on RANSAC()	2.2%
Percentage Spent on Warp()	68.8%
Percentage Spent on Blend()	3.8%
Percentage Spent on “Flattening” Data	0.5%

Given below is the memory map generated by CCS at this optimization level:

```
ENTRY POINT SYMBOL: "_c_int00"  address: 00412800


MEMORY CONFIGURATION

         name            origin    length      used     attr    fill
----------------------  --------  ---------  --------  ----  --------
ONCHIP_PROG             00000000   00010000  000002c0  R  X
SBSRAM_PROG             00400000   00014000  00012980  R  X
SBSRAM_DATA             00414000   0002c000  00000000  RW
SDRAM0                  02000000   00300000  00004d2c  RW
SDRAM1                  03000000   00500000  00400000  RW
ONCHIP_DATA             80000000   00010000  00008b6a  RW
```

4.5 Profiling

Another useful feature of CCS is the Profiler. This feature lets you, among other things, see how many clock cycles a block of code or an entire function requires. If a function is profiled, the profiler will report how many cycles were within the function itself (Excl.) and how many cycles were within the function itself plus those within the functions which were called from the function being

profiled (Incl.). We attempted to profile the four algorithms on the EVM, but the last two took too much time. The highest optimization level (File (-o3)) was being used when obtaining the results below.

Algorithm	Excl. Avg.	Incl. Avg.
FindMatches()	38,853	338,279,409
RANSAC()	4,745	12,574,920
Warp()	N/A	N/A
Blend()	N/A	N/A

4.6 Discussion

Implementing image mosaicing on the EVM was a challenging and rewarding experience. Consequently we gained some new insight with regards to the disparity between how C code worked on a PC versus how our C code executed on the EVM. For example, because the heap is in external memory on the EVM, calls to `malloc()` are costly and require approximately 15 cycles. In our code to calculate the determinant for matrix inversion, `malloc()` is called repeatedly in a nested loop as the determinant function recursively calls itself. Therefore, we needed to revise the code in order to perform matrix inversion without calls to `malloc()`.

In addition, our warping function slowed down on the EVM dramatically because of multiple accesses to pixels in the original image in external memory for every pixel on the warped image. These multiple accesses resulted from our use of bilinear interpolation to average color values from neighboring pixels when applying the inverse of the homography gives a pixel coordinate between multiple pixels on the original image. Our code could have run faster on the EVM had we used nearest neighbor interpolation although the resulting warped image would have less smoother edges.

It's also worth mentioning that our RANSAC code ran considerably faster on the EVM than it did on a PC. Using even 100 iterations for RANSAC took 15 - 20 seconds on the PC while taking roughly 1 - 2 seconds on the EVM. The difference was even more pronounced using 1000 iterations. Using this many iterations takes minutes on the PC while requiring only several seconds on the EVM. The most costly portion of RANSAC is solving the system of 8 equations, and we must conclude that the EVM architecture and the CCS compiler were suited especially well for this type of computation.

We felt confident that our SIFT keypoint detector, RANSAC, and warping algorithms gave very accurate results. With regards to our choice of blending algorithms, although our weighted average blending worked well for most images, it may be beneficial to try out other algorithms such as image blending with Laplacian Pyramids. Overall, we were pleased with our results; our project successfully builds image mosaics without user input of correspondence points and with images taken by hand, without necessarily a tripod.

5 Web References

- Benedict Brown and Philip Shilane. *Image Mosaic*. Image mosaicing assignment from Princeton which provides a general overview.
  http://www.cs.princeton.edu/~pshilane/class/mosaic/

- Tsuhan Chen, Advanced Multimedia Processing Lab. *The Self-Reconfigurable Camera Array*. Provided test data.
  http://amp.ece.cmu.edu/projects/MobileCamArray/

- Alexei Efros. CMU 15-463: Computational Photography. Lots of useful information on many aspects of computational photography.
  http://graphics.cs.cmu.edu/courses/15-463/2005_fall/www/463.html

- Paul Heckbert. *Programming Assignment 1: IMAGE MOSAICING - Revision 2*. September 13, 1999. General overview of image mosaicing process. Suggestions on software, tools, and algorithms.
  http://www.cs.cmu.edu/~ph/869/src/asst1/asst1.html

- Paul Heckbert. *Projective Mappings for Image Warping*. September 13, 1999. Reference for image warping.
  http://graphics.cs.cmu.edu/courses/15-463/2005_fall/www/Papers/proj.pdf

- Image Mosaicing. *Image Mosaicing*. Provides information on the topic.
  http://www.cs.ust.hk/~cstws/research/641D/mosaic/

- David G. Lowe. *Demo Software: SIFT Keypoint Detector*. Professor Lowe's website provides SIFT program and sample matching code.
  http://www.cs.ubc.ca/ lowe/keypoints/

- David G. Lowe. *Object Recognition from Local Scale-Invariant Features*. Proceedings of the International Conference on Computer Vision. September 1999. Professor Lowe's original paper on his SIFT algorithm.
  http://www.cs.ubc.ca/~lowe/papers/iccv99.pdf

- Richard Szeliski. *Image Alignment and Stitching*. Discussion of different methods of image warping and image blending. Provided figure of different image transformations.
  http://www.cs.cornell.edu/courses/cs664/2005fa/Handouts/Nikos-Szeliski.pdf