RADIAL UNDISTORTION AND CALIBRATION ON AN IMAGE ARRAY

by

Charles B. Lee

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical and Computer Science

at Massachusetts Institute of Technology

May 22, 2000

Author _____
Department of Electrical Engineering and Computer Science
May 22, 2000

Certified by_____
Leonard McMillan
Thesis Supervisor

Accepted by _____
Arthur C. Smith
Chairman, Department Committee on Graduate Thesis

RADIAL UNDISTORTION AND
CALIBRATION ON AN IMAGE
ARRAY
by
Charles B. Lee

Submitted to the
Department of Electrical Engineering and Computer Science

May 22, 2000

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering and
Master of Engineering in Electrical and Computer Science

ABSTRACT

The Lumigraph is novel way of parameterizing all the light coming out of a scene. Traditional way of capturing Lumigraphs requires a camera mounted on a precision robotic gantry. This is an expensive solution that cannot be used to easily capture outdoor scenes. Recently, the idea of a Lumigraph Scanner was introduced. This is a inexpensive solution that consists of a portable scanner that would make capturing outdoor scenes possible. Scans from this scanner need to be processed before it can be used as input to a Lumigraph renderer. This paper presents a simple and straightforward process that would calculate the radial distortion coefficients and the intrinsic values of each lens of the Lumigraph Scanner. This makes it easy to capture Lumigraphs using a Lumigraph Scanner. It would therefore make it a viable alternative to the traditional method of capturing Lumigraphs.

Thesis Supervisor: Leonard McMillan
Title: Assistant Professor, MIT Electrical Engineering and Computer Science Department

TABLE OF CONTENTS

Table of Contents 3
List of Figures 4
Acknowledgments 5
1. Introduction 6
    1.1 Motivation 6
    1.2 Background 7
        1.2.1 The Lumigraph 7
        1.2.2 The Lumigraph Scanner Project 7
    1.3 Goal 8
    1.4 Thesis Outline 8
2. The Lumigraph Scanner 10
    2.1 Scanner Makeup 10
    2.2 Scans From the Lumigraph Scanner 12
3. Image Extraction 16
    3.1 Image Extraction for Scans 16
    3.2 Calculating Location of Images 16
    3.3 Extracting Images 18
4. Radial Undistortion 19
    4.1 Radial Distortion Equation 19
    4.2 Finding Radial Distortion Coefficients for Scanned Images 20
        4.2.1 Colored Triangle Pattern 20
        4.2.2 Square Grid Pattern 21
        4.2.3 Finding Coefficients By Optimization 23
    4.3 Radially Undistorting an Image 24
5. Calibration 26
    5.1 Scans Needed For Calibration 27
    5.2 Corner Detection 29
    5.3 Calculating Camera Intrinsic Values 29
6. Future Work 30
    6.1 No User Interaction for Radial Undistortion 30
    6.2 Calculating Camera Extrinsic Values 30
    6.3 Using Images As Input To Lumigraph Renderer 31
7. Conclusion 32
    7.1 Summary 32
    7.2 Achievements 33
Bibliography 34
Appendix A 35
    A.1 Image Extraction Source Code 35
    A.2 Radial Undistortion Source Code 43

LIST OF FIGURES

Number *Page*
Figure 1: Lumigraph Scanner....11
Figure 2: Camera Mounted on a Precision Robotic Gantry....12
Figure 3: Sample Scan....13
Figure 4: Color Correction And Aspect Ratio Correction....14
Figure 5: All White Background And Calculated Locations....18
Figure 6: Selecting two squares....22
Figure 7: 9 Horizontal and 9 Vertical Lines....23
Figure 8: Comparison of Distorted and Undistorted Image....25
Figure 9: Scan of Pattern Captured By Column 4....27
Figure 10: Pattern Captured By Lens 6-4....28
Figure 11: Corners Marked on the Pattern....29

ACKNOWLEDGMENTS

I would like to thank Professor Leonard McMillan for his guidance and support. I also wish to thank Jason Yang for being a great partner. Many thanks goes to Mike Bosse for letting me use some of his source code and for giving me Matlab pointers when I needed them. Same goes to Chris Buehler and Aaron Isaksen for all the help they have given me.

I would also like to thank the Eyeshake Team, for understanding that I cannot spend 24 hours a day on the startup because I had to work on this Thesis.

And of course, I would like to thank my parents for shaping me to be the way I am now. I really appreciate it. And last but not least, thanks Aileen for always supporting me through the good times and the bad times.

Chapter 1

1. INTRODUCTION

1.1 Motivation

Image-based rendering is a novel approach to computer graphics. Instead of creating images by rendering from models, the image-based rendering approach creates new images without needing to model the scene. This is done be using interpolation and morphing to create the pixels of the output image using the pixels from input images.

The Lumigraph [Levoy96, Gortler96] is a very useful image-based rendering technique, but there are many disadvantages. The system needed to capture a Lumigraph is very expensive. Currently, only the computer graphics labs at Stanford and MIT have a Lumigraph capturing device. Most places either do not have the money and/or the space to put it.

Another disadvantage is that the Lumigraph capturing device is too immobile to be able to capture outdoor scenes. So all the acquired Lumigraphs that are currently available are of indoor scenes.

We wish to make a small and inexpensive Lumigraph capturing device. This would not only allow us to capture beautiful outdoor scenes, it would also make it affordable for everyone to conduct more Lumigraph related research.

1.2 Background

1.2.1 The Lumigraph

The concept of the Lumigraph was introduced by both Levoy and Hanrahan [Levoy96] and Gortler et al. [Gortler96]. Levoy and Hanrahan call their version a lightfield but the two techniques are very similar. The papers introduced the idea of 4D parameterizing of all the light coming out or going in to a scene. This approach samples the light going through two planes: the UV-plane and the ST-plane. Once this information is captured, a novel scene can be rendered by interpolating the desired rays from the acquired images. The nice thing about this approach is that there is a very intuitive way to capture and store this 4D function. One just has to place cameras on the UV-plane and capture pictures of the ST-plane. Each pixel in each of the captured image would be a sample of a ray passing through the UV and ST plane.

Using images of a scene as input, one can create images of the same scene from a different viewpoint. This is done by interpolating the pixels of the input images to produce the pixels of the output image. The advantage of this approach is that it could produce highly realistic images in real-time. This is because rendering time is only proportional to the image size, and not the scene complexity.

1.2.2 The Lumigraph Scanner Project

This paper ties closely to the Lumigraph Scanner project by Jason Yang. [Yang00] Yang has constructed a lightfield acquisition device based on a flat-bed scanner. We have focused on correcting the distortions visible in the images captured by this device.

Yang's objective was to build an inexpensive and portable Lumigraph Scanner. The images that this scanner produces will need to be manipulated by software before it could be inputted into a Lumigraph renderer. This paper describes a simple way to radially undistort and calibrate the image array retrieved from the scans of the Lumigraph Scanner.

1.3 Goal

We need to extract the individual images from the scans, radially undistort these images, and calibrate the lenses of the Lumigraph Scanner.

The goal is to do this with as little user interaction as possible. We also want to come up with a process that is simple and quick. This way, users can take a scan, apply this simple and quick process, and immediately see the results in a Lumigraph renderer.

1.4 Thesis Outline

Chapter 2 provides an overview of the Lumigraph Scanner. It also talks about the color correction that needs to be done on the scans.

Chapter 3 describes the process of image extraction. Image extraction is basically the process of extracting the image array from the scans.

Chapter 4 describes a simple radial undistortion method that can be used to find the radial distortion coefficients.

Chapter 5 shows how we can apply an existing calibration routine to calibrate our images.

Chapter 6 describes any future work that needs to be done.

And Chapter 7 concludes this paper with a summary of the whole process.

Chapter 2

2. THE LUMIGRAPH SCANNER

The Lumigraph Scanner project introduces a new way to capture a Lumigraph. Instead of using a camera mounted on a precision robotic gantry, we can now use a simple scanner to capture Lumigraphs very easily.

2.1 Scanner Makeup

The Lumigraph Scanner is made up of a standard off-the-shelf flatbed scanner and an array of plastic lenses. The lenses used are just the top covers of "bug boxes", which are the plastic boxes used for displaying insects. The lenses are glued together in an 8 by 11 array configuration and are affixed on top of the scanner. The scanner can then be used to capture Lumigraphs by turning it on the side and scanning an image of the scene. With the process presented in this paper, each scan create by this Lumigraph Scanner can be used as input to a Lumigraph renderer.

There are many advantages of the Lumigraph Scanner over the traditional Lumigraph capture method, which uses a camera mounted on a precision robotic gantry. (See figure 2.) Since the Lumigraph Scanner uses an off-the-shelf scanner, it is definitely a much cheaper solution. In addition, the scanner (See figure 1.) is a much smaller device that can easily be used to capture outdoor scenes, which is hard for a mounted camera setup to capture.

The disadvantage of the Lumigraph Scanner is that since each image is not taken by the same camera we would need to independently calibrate each of the lenses

(virtual cameras) separately. Also, since we have used short-focal length single-lens optics, each image exhibits considerable radial distortion. The process of undistorting these images and calibrating the lenses could potentially be very tedious and take a long time. This paper presents a simple method to radially undistort and calibrate the individual images of the array.

Figure 1: Lumigraph Scanner

Figure 2: Camera Mounted on a Precision Robotic Gantry

2.2 Scans From the Lumigraph Scanner

The scanned image will contain an 8 by 11 array of images of the scene. The scan would be composed of 88 circular images. Each image would correspond to the image of the scene taken by a virtual camera at that location. It would be as if 88 different cameras were used to take an image of the scene at the same time.

From figure 3, it is obvious that the scanned images cannot easily be used as-is for input to a Lumigraph renderer. There are problems with the color, the aspect ratio, and radial distortion. The image must also be split up into 88 separate images, each representing an image taken by the virtual camera associated with the lens on the scanner. As mentioned before, these images need to be separately calibrated because, unlike a single translated camera, we do not expect that the camera intrinsics would be the same for all the images. The first two *photometric*

problems can be easily fixed by using a standard image processing methods. For instance, Paint Shop Pro can be used to color correct the image and adjust its histogram. (See figure 4.)

Figure 3: Sample Scan

Figure 4: Color Correction And Aspect Ratio Correction

However, before we can use this as input to a Lumigraph render, we must solve other geometric calibration problems. Essentially, any accurate model of a camera should provide a mapping from each point on the image place to a ray in space. The determination of this mapping is called *geometric* calibration.

The central contribution of this thesis is a system for the *geometric* calibration of a multi-lens camera array. The following steps are required to solve this problem. First, the composite image must be segmented into individual image planes. Then, the image is corrected for non-linear geometric distortions common to

spherical lens systems. Finally, an idealized pin-hole camera model is determined for each sub-image of the array.

Chapter 3 describes how images are extracted from scans. Chapter 4 describes how each image is radially undistorted. Chapter 5 describes how each image is calibrated.

3. IMAGE EXTRACTION

Images must be extracted from the image array in a consistent manner. This way the process can be repeated for different scenes in such a way that the extracted images correspond to different images taken by the same camera.

3.1 Image Extraction for Scans

If we knew the exact location of each image in the image array, then it would be easy to extract the images. For the scans taken by the Lumigraph Scanner, the locations of the images are approximately located on an 8 by 11 grid. The images might be a little off from the grid because the bug box lenses are mounted on the scanner using glue. This could lead to small holes between the lenses that would make the location of the images deviate from a perfect grid.

3.2 Calculating Location of Images

The problem now is to find the image location of each virtual camera in the scan. It is possible to manually figure out the locations for each of the 88 lenses, but this would be a very tedious task and it could take a long time. Therefore, we would like to come up with a solution that requires little or no user interaction. Fortunately, we would only need to do this calculation once for a given Lumigraph Scanner. This is because the image locations would be the same for each scan.

The solution that we came up with takes advantage of the fact that the locations of the images are very close to a grid, and the fact that the images consist of a circle that represents the border of the lens. The idea is very simple. Have the user specify the location of the grid that approximates the actual location. Then use a circle finding algorithm to find the exact location of each circle. The center and the radius of each circle would then be stored and used later on to extract the image that correspond to this virtual camera.

It is best to use a scan of an all-white background. This way, the circle-finding algorithm would less likely err by finding a circle that is not the boundary circle.

The implementation of this location calculation algorithm is written as a Matlab program. The circle calculation algorithm is just an implementation of the Hough Transform circle-finding algorithm. It searches for the circle in a 10-pixel range for the x, y, and radius value.

We also provide a way for the user to manually specify the circle if the circle-finding algorithm is unable to correctly locate the boundary of a lens.

See Figure 5 for a scan of an all-white background and the location of the found boundary circles.

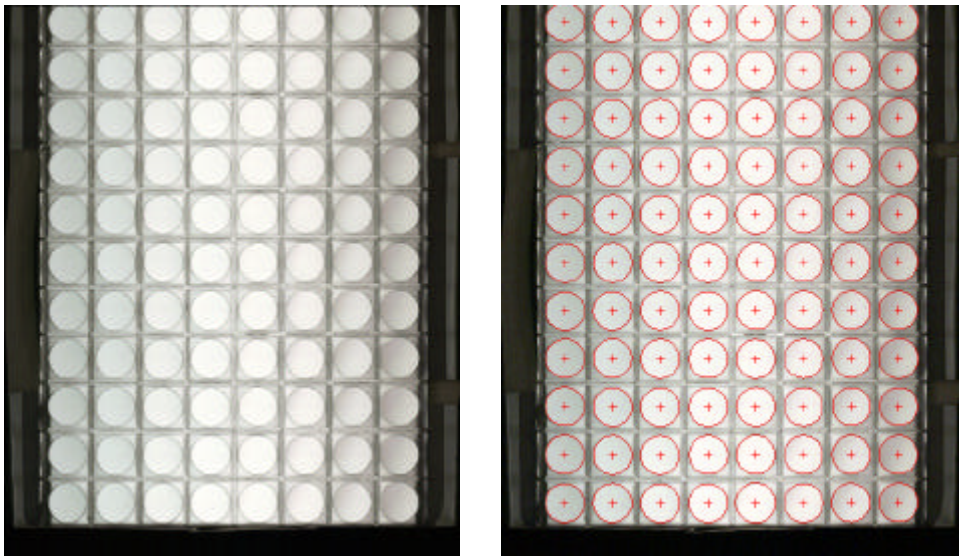

Figure 5: All White Background And Calculated Locations

3.3 Extracting Images

Once the image locations are calculated, images can be easily extracted. Since we would know the center and the radius of the circle that corresponds to the boundary of the corresponding lens, all the pixels within this circle would belong in the extracted image.

The implementation of this part is also written as a Matlab program. This program takes as input the scan image and the location file, and for each center and radius value in the location file, it would extract the circular image and create a new image file. See Appendix A for the Matlab code.

Chapter 4

4. RADIAL UNDISTORTION

One the most common visual distortion of images seen through a lens is radial distortion. This occurs because the magnification of the lenses is different at the edge of the lenses versus the center of the lenses. There are two kinds of radial distortion: pincushion and barrel. Like their names suggest, pin-cushion radial distortion distorts a square object into pin-cushion shaped object, while barrel radial distortion distorts a square object into more of a barrel shaped object.

4.1 Radial Distortion Equation

Radial distortion just means that each point is radially distorted from a certain point, that we call the center of radial distortion.

Radial distortion is governed by the equation: [Weng92]

$$r' = r + k_1 r^3 + k_2 r^5 + k_3 r^7 + \ldots$$

But according to Tsai [Tsai87], for practical purposes, we can safely approximate the radial distortion equation by using only the first term of the infinite series. So for this thesis, we use this simplified radial distortion equation:

$$r' = r + kr^3$$

r' is the distorted radius and r is the original radius. k is the coefficient of radial distortion. This shows that the coefficient k affects how much a point is radially

distorted. The sign of k affects the type of radial distortion. If k is negative, it is a barrel radial distortion. If k is positive, it is a pincushion radial distortion.

In order to undistort an image, we need to find three variables: the x and y values of the center of radial distortion and the coefficient of radial distortion, k.

4.2 Finding Radial Distortion Coefficients for Scanned Images

Since each lens on the Lumigraph Scanner is different, the radial distortion parameter for each sub-image would vary. We need to calculate the radial distortion coefficients for each of the 8 by 11 images. It is important that our radial undistortion algorithm requires little or no user involvement. We came up with a simple algorithm to undistort images that requires very little user interaction.

The idea is that straight lines in real life should remain as straight lines in an image. This is because the image just shows a projection of the scene onto the image plane, and a projection matrix will always preserve straight lines. Radial distortion will tend to curve straight lines. So if we can detect lines that should be straight in the images, then we can use an optimization routine to try to find the distortion coefficients that will make these lines straight.

4.2.1 Colored Triangle Pattern

We need to come up with a pattern that makes detecting lines easy. Originally we came up with a pattern that had 5 differently colored straight lines that forms a star. We would then take a picture of this pattern and use a color separating method to extract the pixels that belong to each line. We can then try to straighten out these lines. As it turned out, the colors in the scanned image vary too much from the expected color. So the color separating method yielded too

many error pixels. We had to give up this approach and find a better pattern to use.

4.2.2 Square Grid Pattern

Our second approach was a little different. Instead of trying to detect lines, we would detect objects. The pattern we used is a 9 by 9 grid of squares. We chose to use this pattern mainly because it is the same pattern used by the calibration process described in Chapter 5. This way, we would save some work by being able to use the same scans to do both radial undistortion and calibration.

In order to locate the grid in the image, we need to find the squares. This is done using a standard connected components labeling algorithm, specifically the *bwlabel* function in Matlab. But before we can apply *bwlabel* to the image, we must first make sure that the image has the square objects separated from the background. This is done using a threshold algorithm created by Michael Bosse that separates objects from background based on local edge intensities. See Appendix A for the Matlab source code.

Once the square objects are located, we need to pinpoint the 9 by 9 grid. To do this, we require the user to specify two of the squares next to each other. The user just needs to click on the two squares. (See Figure 7.) This helps the program figure out the size of the square and the location and orientation of the grid.

The program would then recursively move outwards from one of the two selected squares in all four directions to find the whole grid of squares. The Matlab source code can be found in Appendix A.

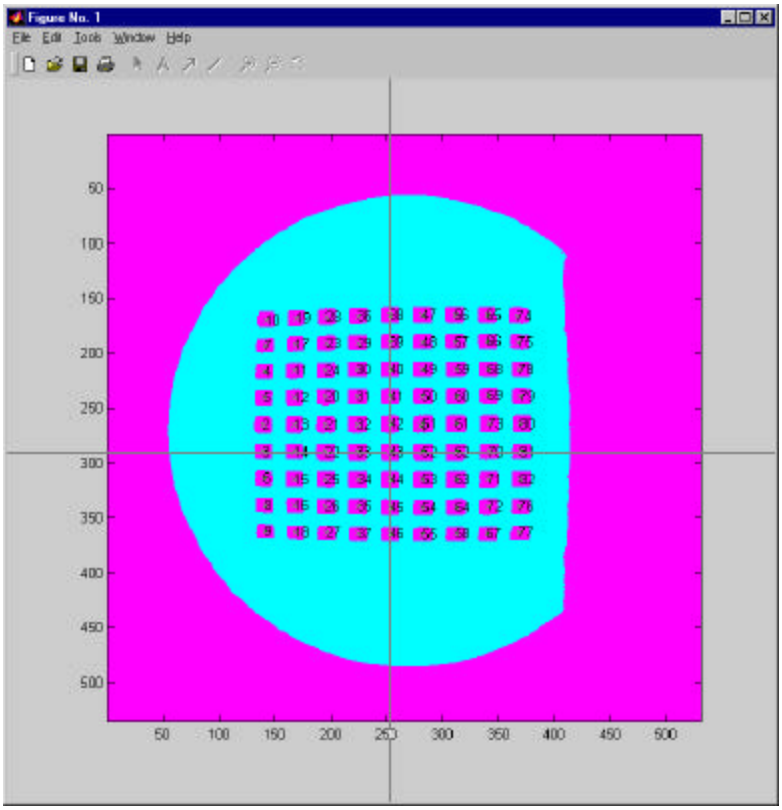

Figure 6: Selecting two squares

Once the grid of squares is located, we can calculate the location of the centroid of each square. Since we know that the squares lie on a grid, the centroids of the squares must also lie on a grid. This would means these centroids are actually points that lie on a group of straight lines. Specifically, there are 18 lines, 9 vertical and 9 horizontal, which we know should be straight. Each line is made up of 9 points. See Figure 6.

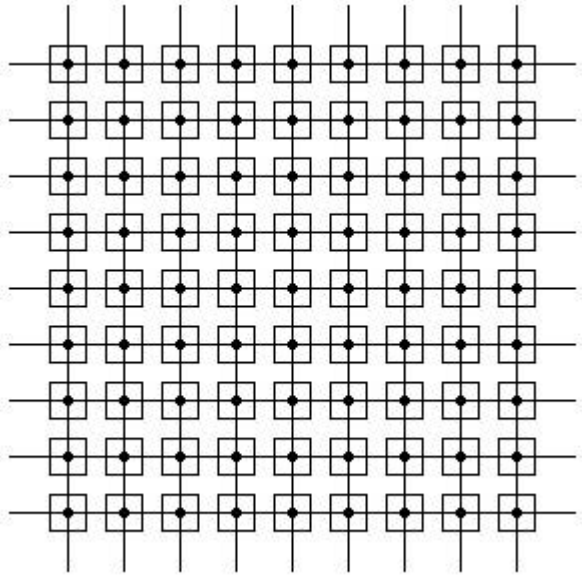

Figure 7: 9 Horizontal and 9 Vertical Lines

4.2.3 Finding Coefficients By Optimization

Once the points of the lines are found, we can just use a least square non-linear optimization algorithm to fit these 18 horizontal and vertical lines into straight lines. The initial guess at k, the radial distortion coefficient, is 0. And the initial guess at the center of radial distortion is just the center of the image. The least square non-linear optimization algorithm we used is the *lsqnonlin* function in MATLAB. And we also used a standard line-fitting algorithm to fit the points to a straight line. The deviation of the points from the best-fit line is used as the error values to the *lsqnonlin* function. This optimization routine finishes in about 50 iterations and finds the radial distortion coefficient and the center of radial distortion.

As mentioned before, the process of finding these coefficients needs to be repeated for each of the 88 images. This whole process is done once per Lumigraph Scanner, because images in each successive scan should be radially distorted the same way.

4.3 Radially Undistorting an Image

To radially undistort an image, we just bi-linearly interpolate the mapped points in the distorted image and copy them to the undistorted image.

Specifically, we know that each point in the undistorted image corresponds to a point in the distorted image. So we just apply the radial distortion function to each point of the desired image to get the point in the distorted image it corresponds to. Specifically, these equations are used to calculate the distorted coordinates:

$$x' = cx + (x - cx)(1 + kr^2)$$

$$y' = cy + (y - cy)(1 + kr^2)$$

$$\text{where } r = \sqrt{(x - cx)^2 + (y - cy)^2}$$

The calculated points will be non-integer. Therefore, we would bi-linearly interpolate the four closest points in the distorted image and copy this pixel to the desired undistorted image. This process is applied to the whole image. Using Matlab, it is all done very simply using matrix manipulations. See Appendix A. for the code. Figure 8 shows an example of a distorted image and the undistorted version of it.

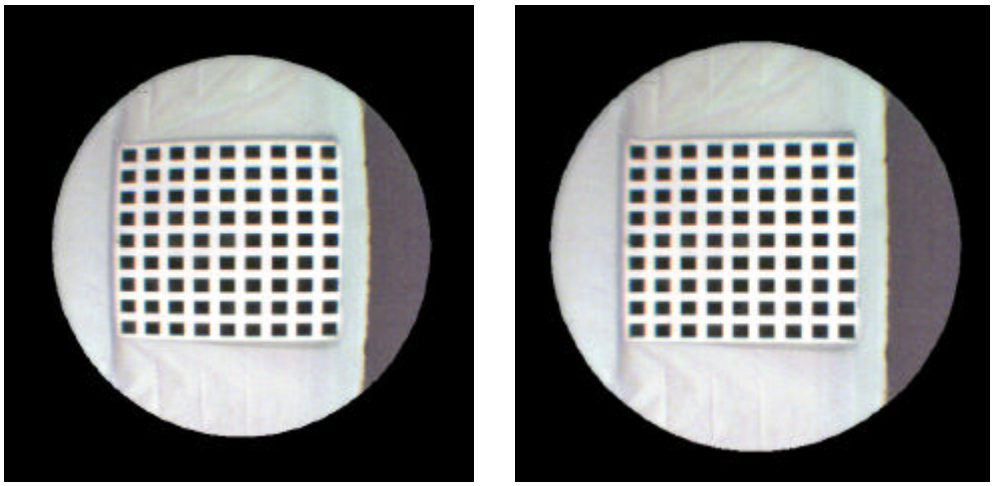

Figure 8: Comparison of Distorted and Undistorted Image

5. CALIBRATION

Calibration is an important step in any computer graphics application that uses lenses/cameras. Lenses project a real world scene onto a flat image. This projection is different for each lens and it depends on many intrinsic values for the lens. Calibration is the process of finding these intrinsic values. Once we have found these values, we could figure out how the pixels in the image correspond to points in the real world.

There have been many different proposed methods of calibration. Many of these methods require you to know the precise location of a few points in the real world. This is a very cumbersome task, and it also requires the use of a precise location-finder device, like a Faro arm, or a precision calibration object. This makes performing calibration inconvenient for someone without access to such a device.

Zhang recently published a paper [Zhang99] describing a calibration method that is very easy to do and only requires software processing. This method needs at least two images of a planar pattern. This pattern consists of a 9 by 9 grid of squares, which we also used for the Radial Undistortion method described in Chapter 4.

Zhang calculates the intrinsic values by using a close-form solution with a non-linear refinement using maximum likelihood estimation.

We believe that Zhang's calibration routine is very well suited for our needs. This would allow us to just take scans of the pattern and use software to calibrate all the lenses.

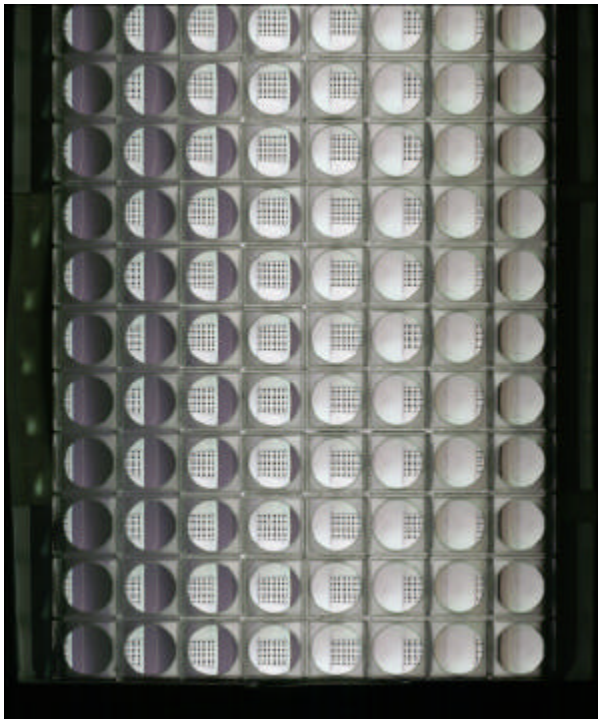

Figure 9: Scan of Pattern Captured By Column 4

5.1 Scans Needed For Calibration

We decided to use 4 different images of the pattern for each lens: straight on, tilted left, tilted right, and tilted upwards. So for each of the lens on the Lumigraph Scanner, we would need 4 images taken by that lens of the pattern at these 4 different angles. Of course, we would like to get all these images with as few scans as possible. We were able to get all the lens of a certain column to see the whole pattern, (See Figure 9.) so this means we would only need to take 4

scans per column, or in other words, 32 scans. This is still quite a lot of scans, but it is a far less than 352 scans, which is the number of scans needed if we needed to take 4 scans for each lens.

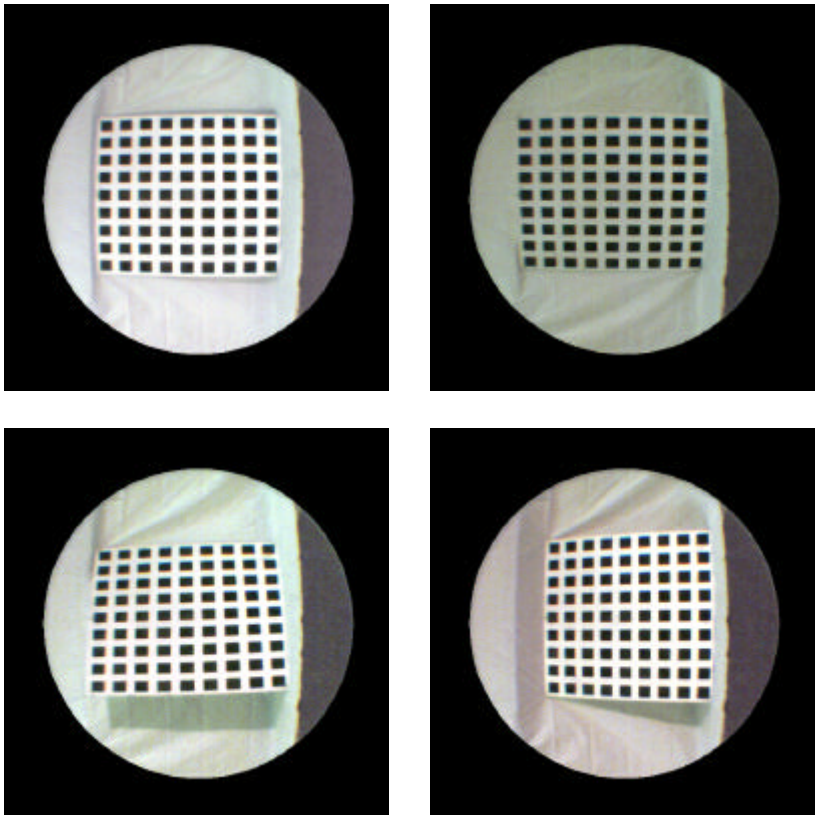

Figure 10: Pattern Captured By Lens 6-4

See Figure 10 for the 4 scans from the lens on row 6, column 4. These are images before radial undistortion. We would actually first apply our radial undistort software (See Chapter 4.) to it before we calibrate them.

5.2 Corner Detection

As required by Zhang's method, we need to find the corner of each square in the grid. That is a total of 324 points. These corners are detected using a standard corner-detection software that finds the corner as the intersection of the straight lines that are the edges of the squares. The corner points are used because we can find the corner points with a sub-pixel level accuracy. Figure 11 shows an image that the corner-finding software produces. This image has the corners of all the squares marked.

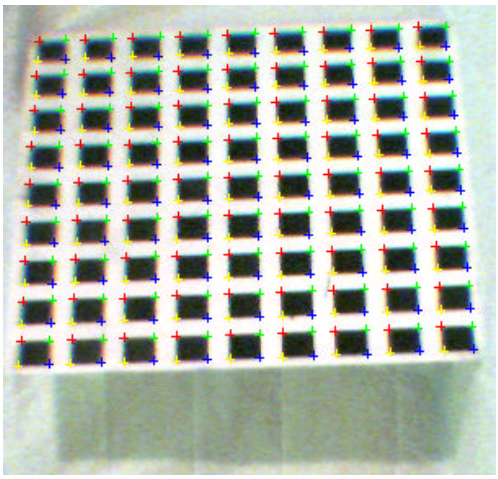

Figure 11: Corners Marked on the Pattern

5.3 Calculating Camera Intrinsic Values

The corner points are then used as input to the *EasyCalib* software that Zhang provides at ftp://ftp.research.microsoft.com/users/easycalib/EasyCalib.zip.

The *EasyCalib* program calculates the les intrinsic values α, c, b, $u0$, and $v0$. It would also calculate the rotation and translation matrix for each of the input images.

6. FUTURE WORK

Although we have a working solution that requires little user interaction to radially undistort and calibrate an image array, there is still a lot of future work that can be done.

6.1 No User Interaction for Radial Undistortion

Currently the radial undistortion process requires a bit of user interaction to pick out two of the squares so that the system can find the rest of the squares in the grid. It would obviously be better if the system could figure out the location of the grid without any user interaction. This is possible, but we would need to come up with a way for the system to correctly locate the grid of squares in any given image.

6.2 Calculating Camera Extrinsic Values

At the time of this writing, we have not gotten a chance to calculate extrinsic values for the lenses. Extrinsic values specify the location of the lens relative to the other lenses. Extrinsic values are crucial if we need to know where our virtual cameras are located with respect to each other.

We might be able to use Zhang's algorithm to solve this problem. Since Zhang's algorithm calculates the location of the camera relative to the planar grid of squares, if two lenses see the same planar grid, we will know the two lenses'

rotation and translation matrix relative to the planar grid. We can then calculate the rotation and translation matrix from one lens to the other. We can do this repeatedly until we figure out the relative locations of all the lenses. This would definitely work for the lenses that lie in the same column, because already have scans in which the images in the same column all contain the planar grid. We have not yet captured scans that have two rows both containing the planar grid, but we do foresee any problems.

Another approach would be to use the method published by Tsai [Tsai86] or something similar.

6.3 Using Images As Input To Lumigraph Renderer

The next step would be to use the undistorted and calibrated images as input to a Lumigraph Render. This would show whether or not the Lumigraph Scanner is indeed a viable alternative to the traditional Lumigraph capture method of using a camera mounted on a precision robotic gantry.

Chris Buehler, of MIT's Computer Graphics Lab, recently created a Lumigraph renderer that is very flexible in terms of the location and orientation of the cameras used to create the input images. Given the calibration data and the undistorted images, we believe we would have no difficulty in using Buehler's Lumigraph render.

Chapter 7

7. CONCLUSION

7.1 Summary

This paper introduces a straightforward process, which can be applied to a scan from a Lumigraph Scanner in order to use it in Lumigraph renderer.

First, the exact location of the lenses is found. This location calculation algorithm, which uses a circle-finding algorithm, is straightforward and requires the user to just specify where to look for the circles. This needs to be done only once, because the location of the lenses does not change for each successive scan.

Next, the radial distortion coefficients of each image are estimated. This requires very little user interaction, and it can find the coefficients pretty efficiently. This also only needs to be done once, because the radial distortion coefficients for each lens should not change for each successive scan.

Next, you need to use Zhang's calibration method to calibrate each lens. This is a fairly tedious task, but it also only needs to be done once.

Although not explored in this paper, you need to find the extrinsic values of the lenses by using Tsai's method or something similar.

Finally, for each scan you take, the software will extract each image, and apply the correct radial undistortion to each image. This requires no user interaction at all, because both the location of the lenses and the radial distortion coefficients are

previously calculated. So the software only needs to apply the image extraction procedure and then the radial undistortion procedure. The resultant images and the calibration data (which were previously calculate) can be plugged into a Lumigraph renderer so that you can view this captured scene from any location and orientation.

7.2 Achievements

We believe that our solution is a fairly simple and straightforward process. The initial calibration process would take some time though, because we need to calibrate all of the 88 lenses. Fortunately, we only need to do it once for each Lumigraph Scanner. After the calibration is complete, each successive scan would require no user interaction at all. This is very valuable, because once a Lumigraph Scanner is calibrated, capturing a Lumigraph using that Lumigraph Scanner will be very easy.

BIBLIOGRAPHY

[Devernay95] Devernay and Faugeras. Automatic Calibration and Removal of Distortion from Scenes of Structured Environments. SPIE 2567. pp. 62-72, 1995.

[Gortler96] Steven Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. In *Computer Graphics, Annual Conference Series, 1996*, pp. 43-54

[Levoy96] Mark Levoy and Pat Hanrahan. Light-field rendering. In *Computer Graphics, Annual Conference Series, 1996*, pp. 31-42

[Tsai86] Roger Y. Tsai. An Efficient and Accurate Camera Calibration Technique for 3-D Machine Vision. In CVPR, pp. 364-374, 1986.

[Tsai87] Roger Y. Tsai.A versatile Camera Calibration Technique for High-Accuracy 3D Machine Vision Metrology Using Off-the-Shelf TV Cameras and Lenses. In *IEEE Journal of Robotics and Automation*, Vol. RA-3, No. 4, August 1987, pp. 323-345

[Weng92] Juyang Weng, Paul Cohen, and Marc Herniou. Camera Calibration with Distortion Models and Accuracy Evaluation. In *IEEE Transitions on Pattern Analysis And Machine Intelligence*, Vol.14, No. 10, October 1992, pp. 965-980

[Yang00] Jason C. Yang. A Light Field Camera for Image Based Rendering.

[Zhang99] Zhang, Z.Y. Flexible Camera Calibration by Viewing a Plane from Unknown Orientations. In ICCV, pp. 666-673, 1999.

APPENDIX A

A.1 Image Extraction Source Code

PROG_FIND.M

```
  disp('Loading image...')

  Img = imread('NEW_WHITE.tif');

  fit_all_circles;
```

PROG_CONFIRM.M

```
  disp('Loading image...')

  Img = imread('NEW_WHITE.tif');

  theta = linspace(0,2*pi,100);
  costheta = cos(theta);
  sintheta = sin(theta);

  fid = fopen('result.txt','r');
  while (~feof(fid))
      j = fscanf(fid,'%d',1);
      i = fscanf(fid,'%d',1);
      x(j,i) = fscanf(fid,'%f',1); % top left x
      y(j,i) = fscanf(fid,'%f',1); % top left y
      lengthx(j,i) = fscanf(fid,'%f',1); % length x
      lengthy(j,i) = fscanf(fid,'%f',1); % length y
      sx(j,i) = fscanf(fid,'%f',1); % center x
      sy(j,i) = fscanf(fid,'%f',1); % center y
     sr(j,i) = fscanf(fid,'%f\n',1); % radius
  end
  fclose(fid);

  quit = 0;

  while(quit == 0)

      % draw circles
      imshow(Img);
      for j=1:11
            for i=1:8
            hold on; plot(sx(j,i) + sr(j,i)*costheta, sy(j,i)
  + sr(j,i)*sintheta, 'r'); hold off
```

```
         hold on; plot((sx(j,i)), (sy(j,i)), 'r+'); hold
off
    end
   end
   drawnow;

    disp('Press a key when you have found a error or there
is no more.')
    pause

    disp('If a circle you see is off, then click inside the
circle')
    disp('Otherwise click anywhere that is not inside any
circle')

    pt = ginput(1);
    ptx = pt(1);
    pty = pt(2);

   jj = 0;
   ii = 0;
    for j=1:11
      for i=1:8
         if (((sx(j,i) - ptx)*(sx(j,i) - ptx) + (sy(j,i) -
pty)*(sy(j,i) - pty)) < sr(j,i)*sr(j,i))
            [j,i]
            jj = j;
            ii = i;
         end
      end
   end

   if (jj~=0)
         disp('Click on the top left point and bottom right
point to define the new circle')
         tl = ginput(1);
         br = ginput(1);

         tlx = tl(1);
         tly = tl(2);
         brx = br(1);
       bry = br(2);

    sx(jj,ii) = (tlx+brx)/2;
       sy(jj,ii) = (tly+bry)/2;
      sr(jj,ii) = (((brx-tlx)/2)+((bry-tly)/2))/2;

   else
      quit = 1;
   end
```

```
  end

  fidw = fopen('result-confirmed.txt','w');
  for j=1:11
      for i=1:8
            fprintf(fidw,'%d %d %f %f %f %f %f %f %f\n',
  [j,i,x(j,i),y(j,i),lengthx(j,i),lengthy(j,i),sx(j,i),sy(j,i)
  ,sr(j,i)]);
      end
  end
  fclose(fidw);

  disp('Written to result-confirmed.txt');
```

PROG_GET_REF_POINTS.M

```
  disp('Loading image...')

  Img = imread('NEW_WHITE.tif');

  figure;
  imshow(Img);
  drawnow

  [tl, bl, tr, br] = find_four_points;

  fid = fopen('refpoints.txt','w');

  fprintf(fid,'%f %f\n', [tl(1), tl(2)]);
  fprintf(fid,'%f %f\n', [bl(1), bl(2)]);
  fprintf(fid,'%f %f\n', [tr(1), tr(2)]);
  fprintf(fid,'%f %f\n', [br(1), br(2)]);

  fclose(fid);
```

PROG_EXTRACT.M

```
  function prog_extract(name)

  mkdir(name);

  filename = strcat(name,'.tif');


  % load ref points
  fid = fopen('refpoints.txt','r');

  x = fscanf(fid,'%f',1);
  y = fscanf(fid,'%f',1);
```

```
ref_tl = [x y];

x = fscanf(fid,'%f',1);
y = fscanf(fid,'%f',1);
ref_bl = [x y];

x = fscanf(fid,'%f',1);
y = fscanf(fid,'%f',1);
ref_tr = [x y];

x = fscanf(fid,'%f',1);
y = fscanf(fid,'%f',1);
ref_br = [x y];

fclose(fid);
% finish loading ref points


disp('Loading image...')

Img = imread(filename);
figure;
imshow(Img);
drawnow;

[tl, bl, tr, br] = find_four_points;

% average the difference of the four points
diff = ( (tl + bl + tr + br) - (ref_tl + ref_bl + ref_tr +
ref_br) ) / 4

%x_diff = (tl(1) - ref_tl(1) + bl(1) - ref_bl(1) + tr(1) -
ref_tr(1) + br(1) - ref_br(1)) / 4
%y_diff = (tl(2) - ref_tl(2) + bl(2) - ref_bl(2) + tr(2) -
ref_tr(2) + br(2) - ref_br(2)) / 4

disp('Resize so that you see all the lens');
pause

fid = fopen('result.txt','r');

cd(name);

theta = linspace(0,2*pi,100);
costheta = cos(theta);
sintheta = sin(theta);

while (~feof(fid))
    j = fscanf(fid,'%d',1);
    i = fscanf(fid,'%d',1);
```

```
    x = fscanf(fid,'%f',1); % top left x
    y = fscanf(fid,'%f',1); % top left y
    lengthx = fscanf(fid,'%f',1); % length x
    lengthy = fscanf(fid,'%f',1); % length y
    sx = fscanf(fid,'%f',1); % center x
    sy = fscanf(fid,'%f',1); % center y
   sr = fscanf(fid,'%f\n',1); % radius


   cx = sx - x;
   cy = sy - y;

   % modify x,y to reflect the shift of the image
   x = x + diff(1);
   y = y + diff(2);
   sx = sx + diff(1);
   sy = sy + diff(2);

   % draw the circle (adjusted)
   hold on; plot(sx + sr*costheta, sy + sr*sintheta, 'r');
hold off
   hold on; plot((sx), (sy), 'r+'); hold off
   drawnow

   if (y<0)
      SubImg = uint8(zeros(round(lengthy), round(lengthx),
3));
      SubImg(round(2.0-y):round(lengthy), 1:round(lengthx),
:) = Img(1:round(y)+round(lengthy)-1,
round(x):round(x)+round(lengthx)-1, :);
   else
      SubImg = Img(round(y):round(y)+round(lengthy)-1,
round(x):round(x)+round(lengthx)-1, :);
   end

   outfile1 = strcat('img',int2str(j),'-
',int2str(i),'.tif');

   % now, mask it with a circle at sx,sy with radious sr

   disp('Creating image...')

   numpoints = 50;
   increment = (2*pi)/numpoints;
   angles = linspace(increment, 2*pi, numpoints);

   col = cos(angles)*sr+cx;
   row = sin(angles)*sr+cy;

   bw = roipoly(SubImg,col,row);
```

```
      bw2 = SubImg;
      bw2(:,:,1) = bw;
      bw2(:,:,2) = bw;
      bw2(:,:,3) = bw;

       SubImg(bw2==0) = 0;

      disp(strcat('Writing...',outfile1,'...'))
       imwrite(SubImg, outfile1);
  end

  fclose(fid);

  cd('..');
```

FIND_FOUR_POINTS.M

```
  function [tl, bl, tr, br] = find_four_points()

  % Assumes image is already loaded and is displayed as a
  figure

  disp('Please resize window and zoom close to the top left
  point and press any key')
  pause
  disp('Click on the top left point')
  tl = ginput(1);

  disp('Please resize window and zoom close to the bottom left
  point and press any key')
  pause
  disp('Click on the bottom left point')
  bl = ginput(1);

  disp('Please resize window and zoom close to the top right
  point and press any key')
  pause
  disp('Click on the top right point')
  tr = ginput(1);

  disp('Please resize window and zoom close to the bottom
  right point and press any key')
  pause
  disp('Click on the bottom right point')
  br = ginput(1);
```

FIT_ALL_CIRCLES.M

```
  figure;
```

```
imshow(Img);
drawnow

disp('This program assumes that the image array is 11x8')
disp('Please resize window and zoom close to the 4 top left
corner lens, and press a key')
pause
disp('Please pick the center of the 4 top left corner lens')
tl = ginput(1);
disp('Please resize window and zoom close to the 4 bottom
right corner lens, and press a key')
pause
disp('Please pick the center of the 4 bottom right corner
lens')
br = ginput(1);

disp('Please resize window so that everything is in view,
and press a key')
pause

tlx = tl(1)
tly = tl(2)
brx = br(1)
bry = br(2)

lengthx = (brx-tlx)/6
lengthy = (bry-tly)/9

x = tlx - lengthx
y = tly - lengthy

theta = linspace(0,2*pi,100);
costheta = cos(theta);
sintheta = sin(theta);

fid = fopen('result.txt','w');
for j=1:11
    for i=1:8
      disp(sprintf('Fitting circle (%d,%d)...', j, i))

      if (y<0)
          SubImg = Img(1:round(y+lengthy-1),
round(x):round(x+lengthx-1), :);
      else
         SubImg = Img(round(y):round(y+lengthy-1),
round(x):round(x+lengthx-1), :);
      end

      [ex,ey,er] = fit_one_circle(SubImg);
```

```
        if (y<0)
           sy = ey;
        else
           sy = y + ey - 1;
        end
        sx = x + ex - 1;
        sr = er;

            fprintf(fid,'%d %d %f %f %f %f %f %f %f\n',
  [j,i,x,y,lengthx,lengthy,sx,sy,sr]);

        hold on; plot(sx + sr*costheta, sy + sr*sintheta,
  'r'); hold off
        hold on; plot((sx), (sy), 'r+'); hold off
        drawnow

        x = x + lengthx;
     end
     x = tlx - lengthx;
     y = y + lengthy;
  end
  fclose(fid);

  disp('Written to result.txt');
```

GRAD_IMG.M

```
  function E = grad_img(I);

  H = fspecial('sobel');
  E = sqrt(filter2(H,I).^2+filter2(-H',I).^2);
```

CIRCLE_HOUGH.M

```
  function Counts = circle_hough(I, cx_range, cy_range,
  r_range)

  ncx = length(cx_range);
  ncy = length(cy_range);
  nr = length(r_range);

  Counts = zeros([ncx ncy nr]);

  for i=1:nr
     r=r_range(i);
     theta = linspace(0,2*pi,round(2*pi*r));
     rcthe = r*cos(theta);
     rsthe = r*sin(theta);

     [CX,CY] = ndgrid(cx_range,cy_range);
```

```
   CX=CX(:);
   CY=CY(:);
   X = round(repmat(rcthe,length(CX),1) +
repmat(CX,1,length(theta)));
   Y = round(repmat(rsthe,length(CY),1) +
repmat(CY,1,length(theta)));
   good = find( X>0 & X < size(I,2) & Y > 0 & Y < size(I,1)
);

   ind = sub2ind(size(I),Y(good),X(good));

   Samples = zeros(size(X));
   Samples(good) = I(ind);
   cnts = sum(Samples,2);
   cnts = reshape(cnts, [ncx ncy]);

%   imagesc(cnts),colorbar
%   drawnow
   Counts(:,:,i) = cnts;

end
```

A.2 Radial Undistortion Source Code

PROG_UNDISTORT4.M

```
function prog_undistort4(name)

prog_undistort('cal0', name);
prog_undistort('cal1', name);
prog_undistort('cal2', name);
prog_undistort('cal3', name);
```

PROG_UNDISTORT.M

```
function prog_undistort(DIR, name)

IMAGE = strcat(name,'.tif');
OUTIMAGE = strcat(name,DIR,'.tif');
%BMPIMAGE = strcat(name,DIR,'.bmp');

% maximum size of grid
GRID_SIZE = 20;
THRESH_HOLD = 2;

img = imread(strcat(DIR,'\\',IMAGE));

ss = size(size(img));
```

```
if (ss(2)==2)
   grayimg = img;
else
   grayimg = rgb2gray(img);
end

I = -double(grayimg);
disp('thresholding image...')
T = threshold_img(I,THRESH_HOLD);

% guess the labels
disp('labeling objects...')
pts = label_objects(T);
numplot(pts);

% ask the user to pick two points
colormap cool;
imagesc(T);
numplot(pts);
disp('Please pick the center of 2 squares that are side by
side to each other')
two_points = ginput(2);

% find the closest pts to the ones the user selected
D = sqdist(pts(:,1:2)',two_points');
[val, marker_ind] = min(D);

% calculate min and max areas for the circles
point1 = pts(marker_ind(1),1:2);
point2 = pts(marker_ind(2),1:2);
area1  = pts(marker_ind(1),3);
area2  = pts(marker_ind(2),3);
dist = sum((point1-point2).^2).^0.5;
area = (area1 + area2) / 2;
maxarea = area * 1.4
minarea = area * 0.6

% label with the correct min and max areas
disp('labeling objects again...')
pts = label_objects(T,minarea,maxarea);
numplot(pts);

% Use only the y and x coordinates
points = pts(:,[2 1]);
% Calculate the grid
grid = points2grid(points, dist, GRID_SIZE, size(img))

%do the optimization
dimension = size(I);
height = dimension(1);
```

```
  width = dimension(2);
  %k = -.0000015;
  k = 0;
  guess = [(height+1)/2 (width+1)/2 k];

  % optimize
  optimized = lsqnonlin('vector_optimize', guess, [ ], [ ], [
  ], dist, points, grid);

  cy = optimized(1);
  cx = optimized(2);
  k = optimized(3);

  % write distortion values
  fid = fopen(strcat(DIR,'\\',name,'_coeff.txt'),'w');

  fprintf(fid,'%f %f %0.12f', [cx, cy, k]);

  fclose(fid);

  % undistort image
  outimg = undistort_image(img, cx, cy, k);

  %figure; imshow(img);
  %figure; imshow(outimg);

  disp('Writing output images...')
  imwrite(outimg, strcat(DIR,'\\',OUTIMAGE));
  %imwrite(outimg, strcat(DIR,'\\',BMPIMAGE));
```

THRESHHOLD_IMG.M

```
  function [Tl,Th] = threshold_img(img, nblocks)
  %THRESHOLD_IMG  Automatically threshold image in to objects
  and background based on local edge intensity.
  %[Tl,Th] = threshold_img(img, nblocks)
  %  returns high and low thresholds for image segmentation
  %G = threshold_img(img, nblocks)
  %  returns thresholded image

  if nargin < 2, nblocks = 10; elseif isempty(nblocks),
  nblocks = 10, end;

  %find threshold by looking at average edge intensity
  I = imresize(img,.25);
  disp('edge detection')
  [bw,thresh] = edge(I,'zerocross');
  E = I;
  E(~bw)=0;
  bsiz = bestblk(size(I), max(size(I))/nblocks);
```

```
  tm = blkproc(E,bsiz,'mean([x(find(x));0])');
  tm(find(tm==0))=max(tm(:));

  if(any(size(tm)>5)),
     tm = conv2(tm,fspecial('gaus',1.5),'same');
  end

  disp('resize and renorm')
  tm = imresize(tm,size(img),'bilinear');
  %tsdv =
  blkproc(E,bsiz,'repmat(std([x(find(x));0]),size(x))');
  %tm = mean(img(bw));
  tsdv = std(img(bw));
  Tl = tm - tsdv./2;
  Th = tm + tsdv./2;
  %Tl = kron(Tl,ones(2,2));
  %Th = kron(Th,ones(2,2));

  if nargout == 1
     %normalize
     G = (img-Tl)./(Th-Tl);

     %set all pixels greater than Th to 1
     G(find(G > 1)) = 1;
     %set all pixels less than Tl to 0
     G(find(G < 0)) = 0;

     [m,n] = size(G);
     G(1,:) = 1;
     G(m,:) = 1;
     G(:,1) = 1;
     G(:,n) = 1;

     Tl = G;
     Th = [];
  end
```

LABEL_OBJECTS.M

```
  function pts = label_objects(img, Amin, Amax)
  % coords = label_objects(img, Th, Tl, Amin, Amax)
  %         img is the image of the test pattern,
  %   Amin and Amax are the min and max area of an object.
  %     returns the coords of the centriods of the objects.

  if (nargin < 3), Amax = inf, end
  if (nargin < 2), Amin = 0, end

  disp('bwmorph clean')
  B = img > 0;
```

```
  B = bwmorph(B,'clean');
  imagesc(B)
  drawnow;

  disp('bwlabel')
  L = bwlabel(B,8);
  imagesc(L) %, colorbar
  drawnow

  disp('index reorder')
  K = find(L(:));
  %create matrix indexed by label
  Lind = sparse(K,L(K),img(K));

  Avec = sum(Lind,1);
  good = find(Avec > Amin & Avec < Amax);

  %preallocate some of the result structure in advance
  %the colums are xcoord, ycoord, and Area, and moments.
  pts = zeros(length(good),9);

  disp('moment calculations')
  num=0;
  for i = good
     num = num+1;

     ind = find(Lind(:,i));
     [I,J] = ind2sub(size(img),ind);

     %calculate area and centroid
     A = Avec(i);
     x = sum(img(ind).*J/A);
     y = sum(img(ind).*I/A);

     %find the higher order invariant moments
     %ivm =
  affine_invariant_moments(img(min(I):max(I),min(J):max(J)))'
     %ivm2 = affine_invariant_moments(img(ind)>0,J,I)';

     %pts(num,1:9) = [x,y,A, ivm2(:)'];
     pts(num,1:3) = [x,y,A];
  end

  pts = pts(1:num,:);
  num
```

SQDIST.M

```
  function D = sqdist(X1,X2)
```

```
  % computes the distance b/w every pair of column vectors in
  X1, and X2;

  if nargin == 1
     X2 = X1;
  end

  N = size(X1,2);
  M = size(X2,2);

  X11 = repmat(sum(X1.^2)',1,M);
  X22 = repmat(sum(X2.^2) ,N,1);
  X12 = X1'*X2;

  D = X11 - 2*X12 + X22;
```

POINTS2GRID.M

```
  function grid = points2grid(pts, distance, gridsize,
  imgsize)

  % pts is [y x]

  center = [imgsize(1)/2 imgsize(2)/2];
  center_grid = [round(gridsize/2) round(gridsize/2)];

  D = sqdist(pts',center');
  [val, center_marker] = min(D);

  % Initialize grid to all zeroes of gridsize by gridsize
  grid = zeros(gridsize,gridsize);
  % Set the center position to center_marker
  grid(center_grid(1), center_grid(2)) = center_marker;

  % Recurse right
  grid = findnext(pts, grid, distance, center_grid, [0 1],
  pts(center_marker,:), [0 distance]);
  % Recurse right
  grid = findnext(pts, grid, distance, center_grid, [0 -1],
  pts(center_marker,:), [0 -distance]);

  % For each grid position found in the center horizontal line
  for x = 1:gridsize
     marker = grid(center_grid(1), x);
     if (marker~=0)
        % Recurse up
            grid = findnext(pts, grid, distance,
  [center_grid(1) x], [1 0], pts(marker,:), [distance 0]);
        % Recurse down
```

```
      grid = findnext(pts, grid, distance, [center_grid(1)
x], [-1 0], pts(marker,:), [-distance 0]);
   end
end
```

FINDNEXT.M

```
function newgrid = findnext(pts, grid, distance, gridpos,
grid_direction, imgpos, img_direction)

newgrid = grid;
gridsize = size(grid);

new_gridpos = gridpos + grid_direction;
new_imgpos = imgpos + img_direction;

% Only do it if the new grid position is inside the grid
if (new_gridpos(1)>=1 & new_gridpos(1)<=gridsize(1) &
new_gridpos(2)>=1 & new_gridpos(2)<=gridsize(2))
    % find the closest point to the estimated point
   D = sqdist(pts',new_imgpos');
   [val, marker] = min(D);

   % if the marker of the new point is not equal to the
previous one.
   if (marker ~= grid(gridpos(1), gridpos(2)))
    point_marker = pts(marker,:);
       d = sum((point_marker-new_imgpos).^2).^0.5;
       % if the new point is close enough to the estimated
one
      if (d < distance*0.6)
         % Set the new grid point
         newgrid(new_gridpos(1), new_gridpos(2)) = marker;
         % keep on recursing in the same direction
         % but this time use point_marker-imgpos as the
image direction
         % Because of the distortion in the image
point_marker-imgpos is going to be more slanted than
img_direction
         %           which is what we want. it will be a
better estimate for the next point
         newgrid = findnext(pts, newgrid , distance,
new_gridpos, grid_direction, point_marker, point_marker-
imgpos);
      end
   end
end
```

VECTOR_OPTIMIZE.M

```
function error = vector_optimize(in, distance, pts, grid)

cy = in(1);
cx = in(2);
center = [cy cx];
k = in(3);

gsize = size(grid);

% undistort the points
points = mat_undistort_points(pts, center, k);

error = [];

% for each horizontal line in the grid
for y = 1 : gsize(1)
   line = grid(y,:);
   % get all non zero elements
   line = line(line>0);
   line = points(line,:);
   s = size(line);

   % if at least 5 elements
   if (s(1)>=5)
      % throw away the 2 end points (they might be outliers)
%      line = line(2:s(1)-1,:);

      % fit line and grab the error (residue)
    [coeff res] = linefit(line);
      error = [error; res];
   end
end

% for each vertical line in the grid
for x = 1 : gsize(2)
   line = grid(:,x);
   % get all non zero elements
   line = line(line>0);
   line = points(line,:);
   s = size(line);

   % if at least 5 elements
   if (s(1)>=5)
      % throw away the 2 end points (they might be outliers)
%      line = line(2:s(1)-1,:);

      % fit line and grab the error (residue)
      [coeff res] = linefit(line);
      error = [error; res];
   end
```

```
  end

  %disp(sprintf('func: center = (%0.3f, %0.3f), k = %0.9f,
  error = %d', cy, cx, k, sum(error)))
```

LINEFIT.M

```
  function [l,residue] = linefit(P)
  % P is a matrix of size n by 2, which is of the form
  % [x1 y1; x2 y2; ... xn yn] which are the points you want
  % to fit

  % l is a vector [a b c] which is the best fit line
  % of the form ax+by-c=0

  % residue is proportional to the error (variance?)

  [m n] = size(P);
  if n ~= 2, error('matrix P must be m x 2'),end
  if m < 2, error ('Need at least two Points'), end
  one = ones(m,1);
  % p = centroid of all the points in P
  p = (P'*one)/m;
  % matrix of centered coordinates
  Q = P-one*p';
  [U Sigma V] = svd(Q);
  n = V(:,2);
  l = [n;p'*n];
  residue = Sigma(2,2);
```

UNDISTORT_IMAGE.M

```
  function outimg = undistort_image(img, cx, cy, k)

  dimension = size(img);
  height = dimension(1);
  width = dimension(2);
  colordepth = dimension(3);

  center = [cy cx];

  all = img | 1;
  all = all(:,:,1);
  points = line2points(all);

  outimg = im2uint8(ones(height, width, colordepth));

  points_before = mat_distort_points(points, center, k);

  % remove all the indices that are out of bounds
```

```
  points = points(1<=points_before(:,1) &
  points_before(:,1)<=height & 1<=points_before(:,2) &
  points_before(:,2)<=width,:);
  points_before = points_before(1<=points_before(:,1) &
  points_before(:,1)<=height & 1<=points_before(:,2) &
  points_before(:,2)<=width,:);

  y_before = points_before(:,1);
  x_before = points_before(:,2);

  y = points(:,1);
  x = points(:,2);

  y_before_floor = floor(y_before);
  x_before_floor = floor(x_before);
  y_before_ceil  =  ceil(y_before);
  x_before_ceil  =  ceil(x_before);

  ind = sub2ind(dimension, y, x);

  ind_ff = sub2ind(dimension, y_before_floor, x_before_floor);
  ind_fc = sub2ind(dimension, y_before_floor, x_before_ceil);
  ind_cf = sub2ind(dimension, y_before_ceil,  x_before_floor);
  ind_cc = sub2ind(dimension, y_before_ceil,  x_before_ceil);

  ind_4 = [ind_ff, ind_fc, ind_cf, ind_cc];

  x_f = x_before  - x_before_floor;
  x_c = 1              - x_f;
  y_f = y_before  - y_before_floor;
  y_c = 1              - y_f;

  factor_4 = [y_c, y_c, y_f, y_f] .* [x_c, x_f, x_c, x_f];

  for i = 1:colordepth
     img_temp = img(:,:,i);

      img_4 = double(img_temp(ind_4));

      % linearly interpolate
      outimg_temp = im2uint8(ones(height, width));
     outimg_temp(ind) = uint8(round(sum(img_4.*factor_4, 2)));

     outimg(:,:,i) = outimg_temp;
  end
```

MAT_UNDISTORT_POINTS.M

```
  % Radial undistortion for a matrix of points
```

```
function points_out = mat_undistort_points(points_in
,center, k)

dirs = points_in - repmat(center,size(points_in,1),1);

sqr_dirs = dirs.^2;

rp = sqrt(sqr_dirs(:,1) + sqr_dirs(:,2));

r = mat_undistort(rp, k);

f = (r ./ rp);

f = repmat(f,1,2);

dirs = dirs .* f;

points_out = repmat(center,size(dirs,1),1) + dirs;
```

MAT_DISTORT_POINTS.M

```
% Radial distortion for a matrix of points
function points_out = mat_distort_points(points_in ,center,
k)

points_out = points_in;

dirs = points_in - repmat(center,size(points_in,1),1);

sqr_dirs = dirs.^2;

rp_t = sqrt(sqr_dirs(:,1) + sqr_dirs(:,2));

rp = rp_t(rp_t>0);
dirs = dirs(rp_t>0,:);

r = mat_distort(rp, k);

f = (r ./ rp);

f = repmat(f,1,2);

dirs = dirs .* f;

points_out(rp_t>0,:) = repmat(center,size(dirs,1),1) + dirs;
```

MAT_UNDISTORT.M

```
% Radial undistortion for a matrix
function r = mat_undistort(rp, k)
```

```
  if k == 0
     r = rp;
  elseif k > 0
     nrp = rp * 3.0;
     t = 1.0 / (6.0 * k);
     d = sqrt(nrp .* nrp + 8.0 * t);
     r = ((nrp + d) * t).^(1/3) - ((d - nrp) * t).^(1/3);
  else
     t = sqrt(-1.0 / (3.0 * k));
     d = -1.5 * rp / t;
     for i = 1:size(d,1),
        if abs(d(i)) > 1.0
  %      disp('ERROR: undistort error');
           if d(i) >= 0.0
           d(i) = 1.0;
           else
           d(i) = -1.0;
           end
     end
     end
     r = -2.0 * t * cos((acos(d) + pi) / 3.0);
  end
```

MAT_DISTORT.M

```
  % Radial distortion
  function rp = mat_distort(r, k)

  rp = r + k * r.^3;
```

LINE2POINTS.M

```
  function points = line2points(line)

  [a b] = find(line);
  points = [a,b];
```