(11)

The Design and Construction of a 3-D Scanner from Off-the-Shelf Components

By

Wandy Sae-Tan

Submitted to the
Department of Electrical Engineering and Computer Science
In Partial Fulfillment of the Requirements for the
Degree of Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

May 27, 1998
[June 1998]

Author _____
Department of Electrical Engineering and Computer Science
May 27, 1998

Certified by _____
Dr. Amar Gupta
Thesis Supervisor

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

The Design and Constructor of a 3-D Scanner from Off-the-Shelf Components

By

Wandy Sae-Tan

Submitted to the Department of Electrical Engineering and Computer Science

May 27, 1998

In Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

We present the design and prototype implementation of a low-cost 3-D scanning system using off-the-shelf personal computer, digital camera, and computer projector. In this system, we project a fixed grid onto the subject being scanned. By placing a camera at an angle, we can capture the distortions of the grid caused by the scanned subject. These horizontal distortions can then be converted into a collection of depth points. The precision and resolution of this system will not be comparable to those of existing laser scanners, but the affordability of this design could make it very attractive to an entirely new class of applications. Our prototype has served well as a "proof-of-concept". With further work, we believe it is possible to turn this design into a practical implementation.

Thesis Supervisor: Dr. Amar Gupta

Title: Co-director, PROFIT Program, MIT Sloan School of Management

TABLE OF CONTENT

1 INTRODUCTION 5

1.1 3-D Scanner from Off-the-Shelf Components 5
1.2 Project Scope 6
1.3 Paper Roadmap 6

2 BACKGROUND 7

2.1 Basic Approaches 7
2.2 Commercially Available Systems 8

3 DESIGN OVERVIEW 11

3.1 Considerations and Criteria 11
3.2 The Idea 11
3.3 Components 12
3.4 Assumptions 13
3.5 Scanning 14
3.6 Processing 15

4 QUANTIFYING GRID DISTORTIONS 17

4.1 The Goal 17
4.2 Extracting Intersections 17
4.3 Pixel Filter 19
4.4 Identify Intersections 23

5 CONVERTING DISTORTIONS TO DEPTHS 26

5.1 Optical Model 26
5.2 Model Verification 29

6 IMPLEMENTATION OVERVIEW 32

6.1 Setup And Functionality 32
6.2 Process Flow 33

7 DETAILS OF PROCESSING COMPONENTS 36

7.1 Image Subtractor 36
7.2 Intersection Extractor 36
7.3 Point Mapper 40

8 EVALUATION 43

8.1 Fulfillment of Design Goals 43
8.2 Accuracy 44
8.3 Robustness 45
8.4 Ease of Setup 46
8.5 Range of Applications 46

9 CONCLUSION 48

9.1 Future Work 48
9.2 Summary 49

10 REFERENCES 50

11 APPENDICES 51

11.1 Class Prototype 51
11.2 Class PrototypeFrame 61
11.3 Class FileIO 62
11.4 Class FindPoint 64
11.5 Class GridPointProcessor 66
11.6 Class ImageProcessor 69
11.7 Class MapPoints 73
11.8 Class OpticCalculator 78
11.9 Class Pattern 82
11.10 Class PatternFilter 84
11.11 Class PatternParser 86
11.12 Class GridGenerator 88
11.13 Class GridGeneratorFrame 89
11.14 Class Square 94

1 INTRODUCTION

Have you ever heard of a disposable 3-D point-and-shoot camera?

No one has. The techniques for capturing two-dimensional data have been so well developed that a five-year old and pick up a $10 camera and shoot a picture. But when it comes to capturing three-dimensional objects, no one has figured out how to do it cheaply and with the quality that we see in photographs produced by an ordinary camera. This paper attempts to address some of these concerns in the hope of bringing 3-D scanners one-step closer to the mainstream market.

The main concern addressed in this project is cost. 3-D scanners today are built from proprietary hardware and software, making them highly customizable to the buyer's specific needs. However, the high costs of these systems have limited their applications to the few domains that need realistic surface digitizing capability. What if we could build a 3-D scanner that is very low in cost and maintenance? Could people dream up applications for affordable 3-D data?

1.1 3-D SCANNER FROM OFF-THE-SHELF COMPONENTS

In this paper, we explore a possible design of a 3-D scanner from off-the-shelf components. We also present the results and evaluation of a prototype unit that we built based on the design: using a digital camera, a computer projector, and a personal computer.

How could this be done? We project a pre-determined grid onto the scanned subject, and observe the grid distortion through the digital camera. We then convert the horizontal discrepancies into depth information, which we can assemble to build a three-dimensional model.

The design is simple. The cost is low. But as our prototype implementation shows, it certainly has its share of limitations and problems. We will present the details of the system and the difficulties we encountered in the rest of this paper.

1.2 Project Scope

This project covers the following aspects of creating the 3-D scanning system:

- Design of system setup and algorithms needed for processing
- Implementation of the processing algorithms and optic model that takes 2-D images and produces 3-D (x, y, z) model file.

We decided to leave the following out of the project, because we feel that they are not essential for constructing our "proof-of-concept" system:

- Completely automated control of digital camera and computer projector.
- Visualization of the 3-D model file.

1.3 Paper Roadmap

In this next chapter, we will provide background information on the field of three-dimensional scanning. We will describe the basic concepts and detail the state-of-the-art commercial systems. We will also discuss the advantages, limitations, and current applications of these scanning systems.

In Chapter 3, we will give a functional and design overview of our 3-D scanning system made from off-the-shelf components. We will describe how the system works from a high level point of view.

The next three chapters discuss different aspects of system design in detail. Chapter 4 covers the problem and solution to describing grid distortion in a quantitative manner. Chapter 5 shows the optical model and the mathematics needed to convert horizontal grid distortions into depth information.

Chapter 6 and 7 are devoted to the prototype implementation we have completed. We will first present the structure and flow of our processing application from a high level. Then, we will examine the implementation details of the individual software components.

Once we have completed the presentation of all the technical details of the system, we will present an in-depth evaluation of our efforts in Chapter 8. And finally, we will conclude the paper with Chapter 9, where we will present a summary and a few possibilities for future work that could be extended from this project. We have also included an appendix, which lists the source code for the prototype programs written to demonstrate the design.

2 BACKGROUND

2.1 BASIC APPROACHES

The basic techniques for capturing three-dimensional information can be divided up into two groups: contact and non-contact. These approaches have their tradeoffs.

2.1.1 Contact

The contact approach is often referred to as "surface digitizing". It involves taking discrete points from a surface using touch trigger probe. This approach has some fundamental advantages over non-contact systems:

- Treatment of surfaces to prevent reflections is not required.
- Vertical faces can be accurately scanned.
- Data density is not fixed and is automatically controlled by the shape of the component.
- Time consuming manual editing of the data to remove stray points is not required.

However, this approach can be very time-consuming, since the probe must be physically moved over the entire surface. For this reason, we ruled out the contact approach when designing our off-the-shelf components 3-D scanning system.

2.1.2 Non-contact

The non-contact approach is often referred to as "scanning". It typically involves emitting a light (laser or white light) onto the subject. The surface is recorded via the behavior of the lights. In the case of a laser-based scanner, a CCD screen is used to capture the reflectance of laser that bounces off the surface of the subject. There are also products that project modulated white light to capture 3-D data (see Section 2.2.2). The non-contact system offers the following advantages over mechanical contact-based systems:

- Since all that touches the model is a beam of light, the laser system allows fragile, delicate, and highly detailed surfaces to be scanned without damage.
- The model may be made of metal, wood, plastic, plaster, foam or even clay.
- The data collected are true surface data without stylus offsets or deflection.

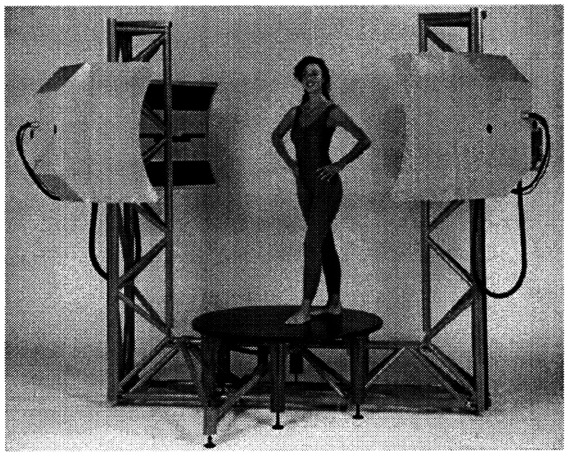
- Since no mechanical movement is required, non-contact systems are usually faster than its contact-based counterparts.

Non-contact systems also have their problems. The biggest drawback being that light beams cannot reach all places of the subject, so often times a scan will have "shadow" regions or "holes" where the light could not reach. These regions must be extrapolated from nearby surfaces or filled-in manually. However, speed and ease of operation are important to the design of our 3-D scanner, so we have adopted a non-contact design.

2.2 Commercially Available Systems

In this section we will introduce other companies and produces in the field of 3-D scanning. The companies and products profile will put our project in a broader market context and help us understand the current technologies and applications.

2.2.1 Cyberware



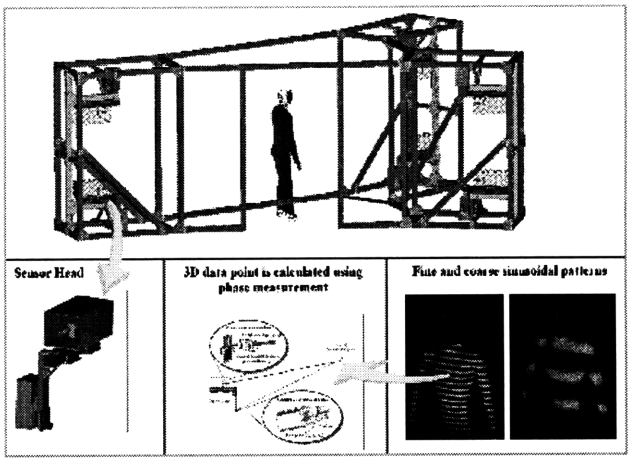
Cyberware (www.cyberware.com) manufactures a variety of standard and custom design instruments for three-dimensional scanning. Its most differentiating products are the head scanners and full body scanners (as shown in picture). Cyberware pioneered the techniques for scanning human faces, and extended 3-D applications from industrial design and manufacturing to anthropology and entertainment. The head scanner produces a detailed three-dimensional data set of facial features and a detailed texture map of the surface color. It has been used in special effects production of Terminator II, Jurassic Park, and many other popular movies.

The full body scanner is a highly respectable product, aiming to bring 3-D scanning to an even wider market. The scanner has four moving scanning heads that travels up and down along the axis. A complete full-body scan is completed in 18 seconds (scan data processing takes significantly longer, depending on the desired output). The scanner is a great eye-catcher – spanning an area of 15'x15' and producing a 3-D scan on screen within a minute. However, this amazing piece of equipment does not come cheap. Ticket price is on the

order of $250,000. Beyond the scanner, the buyer also needs to purchase high-end SGI workstations to run the processing software, which can easy cost another tens and thousands of dollars. The high cost of this scanner is in fact the inspiration for this project. Getting a 3-D scan *is fun* – wouldn't it be great if more people could afford to do it?

2.2.2 [TC]^2

[TC]^2 Inc. (www.tc2.com) is member-driven organization aiming to create innovative technical solutions to enhance the future of apparel manufacturing in the United States. One of its projects under research and development is a 3-D non-contact full-length body measurement system. The goal of this project is to develop a kind of "automated tailor", capable of measuring the human body in a snap-shot and using the information collected to produce custom-tailored clothes.

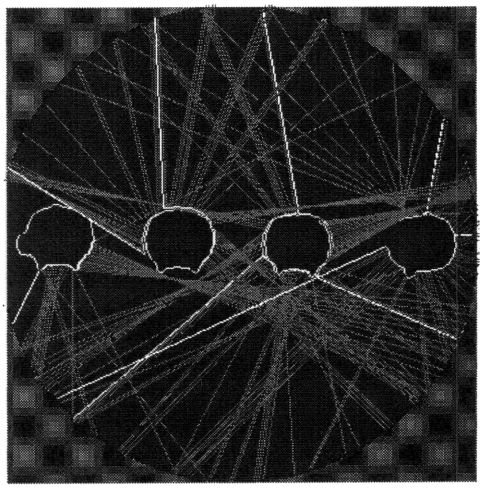


In the 3-D capture respect, the [TC]^2 system is functionally identical to the Cyberware full body scanner. However, its approach to scanning is a unique departure from other scanner systems. Instead of recording laser reflectance, [TC]^2 uses specially modulated *white* lights to produce sinusoidal patterns on the scanned subject. Not using laser allows them to not have any moving parts in the system to scan a large subject. It also brings down the cost to less than ½ of the Cyberware scanner. The figure above shows a high-level functional view of the system. If [TC]^2 successfully develops versions of this product with a smaller footprint and lowers the cost of light modulation components, you could be finding one of these systems at an apparel store near you.

2.2.3 Digibotics

Digibotics (www.digibotics.com) manufactures a variety of advanced laser-based 3-D scanners, specializing in engineering design, rapid prototyping, modeling, and animation. Its most differentiating technology is its adaptive scanning algorithm. The figure on the next page demonstrates the workings of this algorithm. In the picture, we are looking at the object being scanned from above. The scanned objects are the four cylinders (appear as

circles in the picture). The red lines are the laser beams generated by the algorithms to complete the scanning of the subject. Adaptive scanning algorithm allows the scanner to measure points in the deepest cavities and completely eliminates the shadowing effects normally found on other scanning systems. This adaptive scanning technique truly takes advantage of the fact that laser emitters can be actively controlled in a closed setting. Such techniques to remove "shadows" will be very difficult to implement in our off-the-shelf component design, since we have a single light source whose location can not be easily adjusted.



2.2.4 Faro Technologies

Faro Technologies (www.faro.com) develops of portable and precise measurement solutions. Its multi-axis, articulated arms along with a CAD-based measurement software program provide 3D solutions for a variety of manufacturing and design applications. It uses probes to digitize the surface by contact. Its leading product, FaroArm, has a portable design that allows users to bring the probe to anything they need to measure or digitize. This approach is excellent for capturing 3-D information for industrial design and manufacturing purposes, since it can digitize anything, anywhere. The figure above demonstrates a FaroArm unit being used to measure the inside cavity of a jet engine. However, as we discussed at the start of the chapter, the drawbacks to the contact approach is that it is time consuming since the probe needs to be physically moved over the entire surface, and the operator needs to be trained to use the probe correctly.

3 DESIGN OVERVIEW

3.1 CONSIDERATIONS AND CRITERIA

The most important considerations in the design of our 3-D scanning system are:

- **Affordability:** The system must use off-the-shelf components whenever possible, so that we could take advantage of the economy of scale.
- **Small Footprint:** The system must occupy as small an area as possible, so that we could minimize the use of valuable office or retail floor space.
- **Simple Operation:** The system must be simple to operate, so that any average user can take advantage of the system.

These goals are dramatically different from the other commercial systems we examined in the earlier chapter, which are very high-end systems that scan objects with great resolution and accuracy. The goal of this system is to attempt to bring 3-D scanner to "the-rest-of-us". In doing so, we hope to stimulate imaginative applications of affordable 3-D data. In fact, we have already received ideas from a variety of fields, ranging from insurance to shipping, and we will explore them further in Section X. In the rest of this chapter, we will present a design and functional overview of our 3-D scanning system.

3.2 THE IDEA

3.2.1 Why Can't Computers See like Humans Do?

Most 3-D scanning system or range finders today use laser to capture depth information. However, these laser scanners are not mass-produced components and are therefore costly to obtain and assemble. What would be a good alternative to laser? Of all the off-the-shelf components available today, digital camera stands out as an obvious potential candidate.

One could easily imagine reproducing stereo vision with a couple of cameras. After all, this is how we humans see the world, and we can certainly see in three dimensions without lasers of any sort. Our left and right eye each captures a slightly different two-dimensional view. Then, our brain assembles the two pictures, and we gain depth information by interpreting the slight discrepancies between our left-eye-view and our right-eye-view.

Our brain interprets these different views to give us stereo vision effortlessly – but can we write the software that is able to carry out this task? Unfortunately, we have not understood our brain well enough to reproduce it in software. Somehow, our brain is capable of recognizing arbitrary features from the two views and *matches* them together. But we simply have not found a way to write software that robustly recognizes arbitrary features from a two-dimensional image, let alone matching them together to calculate depth from the differences in their locations.

So how to we plan to solve this problem in this thesis? By simplifying the problem.

3.2.2 Projecting a Grid

If the software that recognizes *arbitrary* features does not yet exist, why don't we give our scanner a *pre-determined* feature that it can identify from an image and use as reference to calculate depth?

In our design, we use a computer projector to project a grid. A camera is located at an angle, so that when the grid is projected over the subject with 3-D volume, the grid seems "distorted" to the camera. Now, instead of looking for arbitrary features in the picture, the software only needs to look for the grid lines, and the amount of distortion in the grid lines can then be translated into depth information!

By projecting a grid over the scanned subject, we have moved the problem of creating stereo vision from digital cameras from "quite impossible" to "probably feasible". In later chapters, we will explore the technical difficulties that we had to overcome to turn the idea into a working solution. But first, we will present a high level tour of our proposed design.

3.3 COMPONENTS

The design of this low-cost 3-dimensional scanner requires four pieces of equipment, all of which are commercially available off-the-shelf components:

- **Computer Projector:** To project grid patterns on to the scanned subject.
- **Digital Camera:** To capture images.
- **Personal Computer:** To interpret images and create 3-dimensional model.
- **Back Plane:** Any white flat surface. It serves as a geometrical reference.

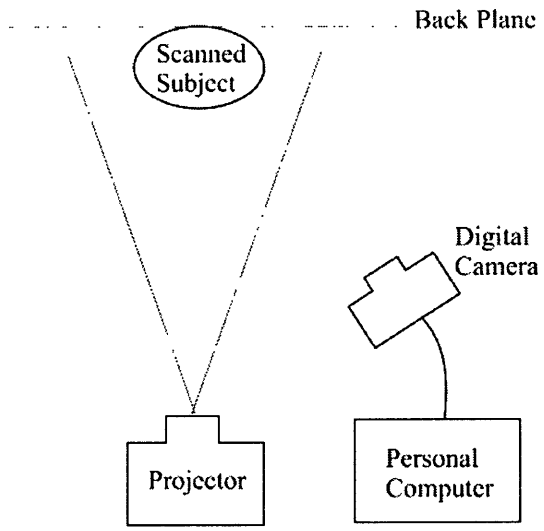


Figure 1. Components Overview

The above figure is a highly simplified view of the system that demonstrates how the different components fit together from a high level point of view. There are two important things to note about the picture:

- The projector is located directly in front of the back plane, but the digital camera captures the image from an *angle*.
- Humans use two eyes to see in three dimensions. In this design however, only *one* camera is needed for recording 3-D data from a given view: the projected grid serves as the "second eye" by acting as a known reference.

If the scanned subject is symmetrical, then one camera could suffice. However, more cameras could be easily added if more than one view is required. The actual quality and the cost of projector, digital camera, and computer will vary depending on the requirement for resolution, processing speed, and size of the scanned subject. We will give further detail on the prototype constructed in Chapter 6.

3.4 Assumptions

Before we discuss the actual scanning process, it is important to understand that the proper working of our design requires the following assumptions to be met:

- The position of the cameras and projector are fixed.
- The scanned object does not move during the scanning process.

- The lighting of the scanning booth is constant during the scanning process.
- System calibration numbers are collected according to Chapter 5.

The most important limitation imposed by these assumptions is that our system will not be portable. But since portability was not one of our design goals, these assumptions are acceptable to us.

3.5 SCANNING

The scanning consists of taking the following four pictures:

- **Reference Image:** The projector casts the grid onto the empty back plane, and the digital camera captures the image. This image is only taken during system calibration.
- **Base Image:** Scanned subject is positioned in front of the back plane, and the projector casts white light onto the subject (no grid), and the digital camera captures the image.
- **Grid Image:** The projector casts the pre-determined grid onto the subject, and the digital camera captures the image.
- **Synchronize Image:** The projector casts a single square onto the subject, and the digital camera captures the image. This square will help us synchronize the grid lines from the grid image with the reference image.

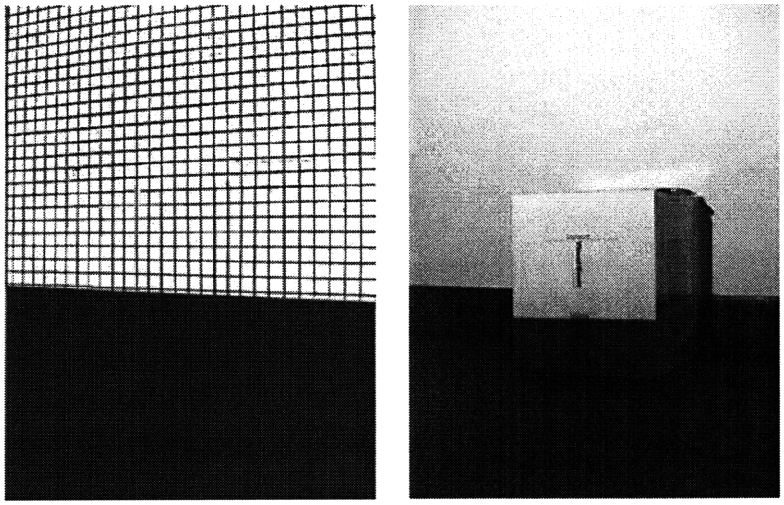


Figure 2.
Sample Images.

Left: Reference Image,

Right: Base Image,

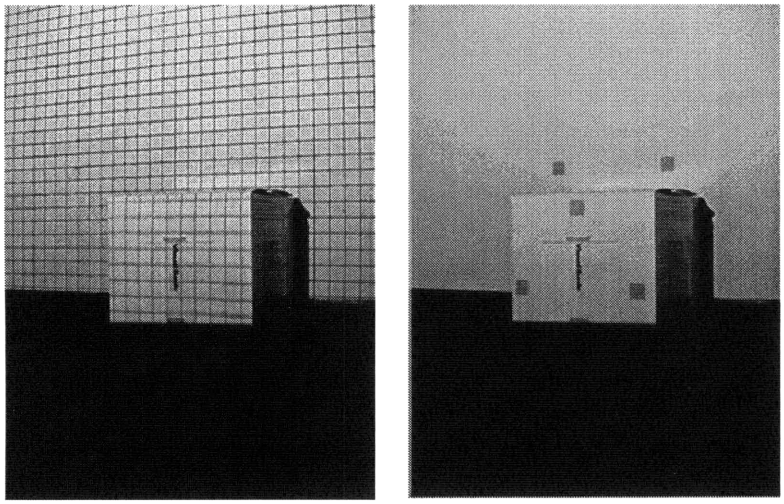


Figure 3.
Sample Images.

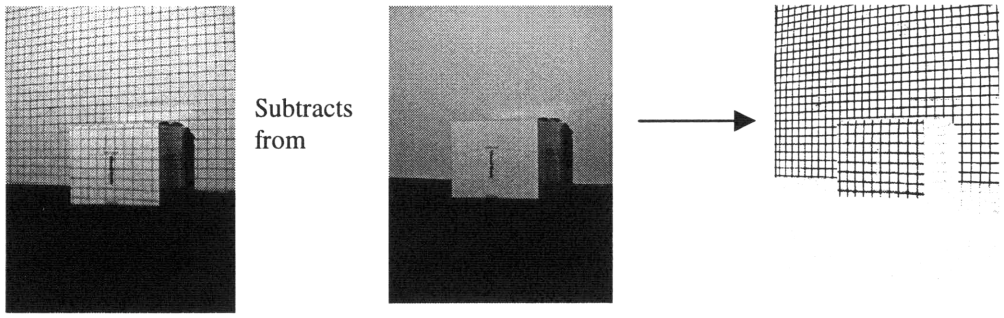
Left: Grid Image,

Right: Synchronization Image,

3.6 PROCESSING

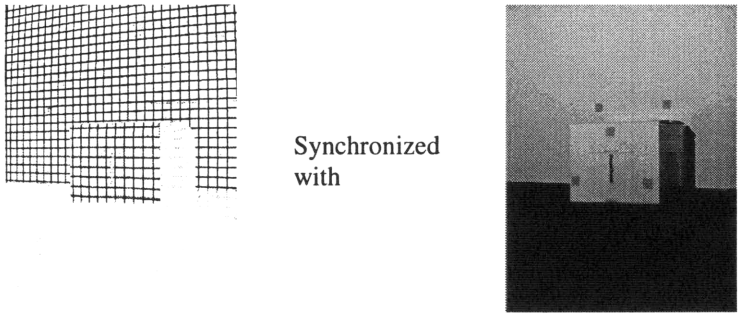
Once we have captured all the images, we process them to produce the three-dimensional model:

- **Extract the grid lines** by subtracting the grid image from the base image. This gives us an image with *just the grid lines* projected over the scanned subject.

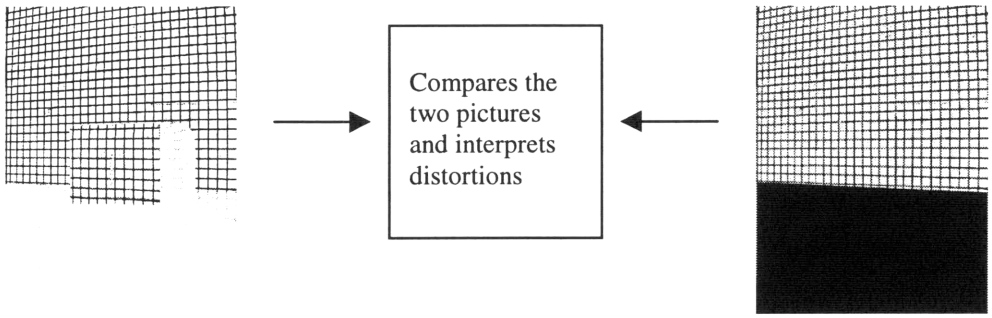


- **Extract the grid intersections** by putting the image of grid lines through a pixel filter (which will be discussed in Chapter 4.3). This gives an image of the grid intersections only.

- **Identify the intersections** by referencing from the synchronization image. We need to map the points in the grid image to their original locations in reference image, so that we can calculate how the points have moved. (Chapter 4.4)



- **Interpret the distortions** to produce depth information by comparing the locations of the intersections to the ones in the reference image using our optical model (which will be discussed in Chapter 5.1).



In the rest of this paper, we will discuss each step in the processing in much greater detail. We will cover the technical problems and discuss possible designs and solutions.

4 QUANTIFYING GRID DISTORTIONS

4.1 THE GOAL

Quantifying grid distortion is crucial for the completion of our 3-D scanner. After all, it is the distortion of the grid that will be translated into depth information (details of this "translation" are explained in the next chapter "Converting Distortions to Depths").

So how can one quantify grid distortion? There are, of course, many different answers to this question. For our 3-D scanner, we have decided to track how much the *grid intersections* move when it is cast on the scanned subject. We quantify grid distortions by calculating the horizontal pixel distance by which the intersections move.

We can break down this goal into two sub-problems:

- **Extraction of grid intersections:** Given the grid image, we have to be able to pin point the intersections.
- **Identification of grid intersections**: Once we have the locations of the intersections, we need to know how to "map" these intersections to the ones from the reference image (where grid is projected on the empty back plane). Then we can calculate the pixel distances by which the intersections have moved.

In the rest of this chapter, we will be describing how these two problems are solved to quantify grid distortion.

4.2 EXTRACTING INTERSECTIONS

4.2.1 The Problem

We have learned in the previous section "Design Overview" that no one has been able to create software that recognizes arbitrary features from an image, and that is why we are simplifying the problem by projecting a pre-determine feature – grid. Unfortunately, working with grid lines is not a trivial task either.

A sample grid image is shown in the following figure uses a ball and a chair as the scanned objects. We as humans can easily follow the grid lines from top to bottom or from left to right. However, a computer looking at this picture faces many difficulties. Our original approach to the problem of extracting intersection was to "trace" the grid lines, hoping to find the intersections from the traces. We attempted to find the start of the grid line along the top edge of the image, then scan the three pixels right below and choose the darkest pixel to be the next point in our growing "line" trace. However, this approach suffered many serious drawbacks and often failed in the following cases:

- Shadows: The algorithm could not tell that it should terminate when it runs into shadow, which are large patches of black where no grid lines can be seen.
- Varying Brightness: The color of the grid lines could vary between pitch black and very light gray due to uneven lighting, which often resulted in micro "breaks" in the grid lines. Although people can easily tell how the line continues, the algorithm could not.
- Varying Contrasts: The contrast between the grid line and the background on which it was cast could vary greatly due to uneven lighting and the color of the scanned subject.
- Grid Intersections: When tracing a vertical line downwards, the algorithm had problems at grid intersections, because following the darkest pixel often led it to follow the horizontal line instead.

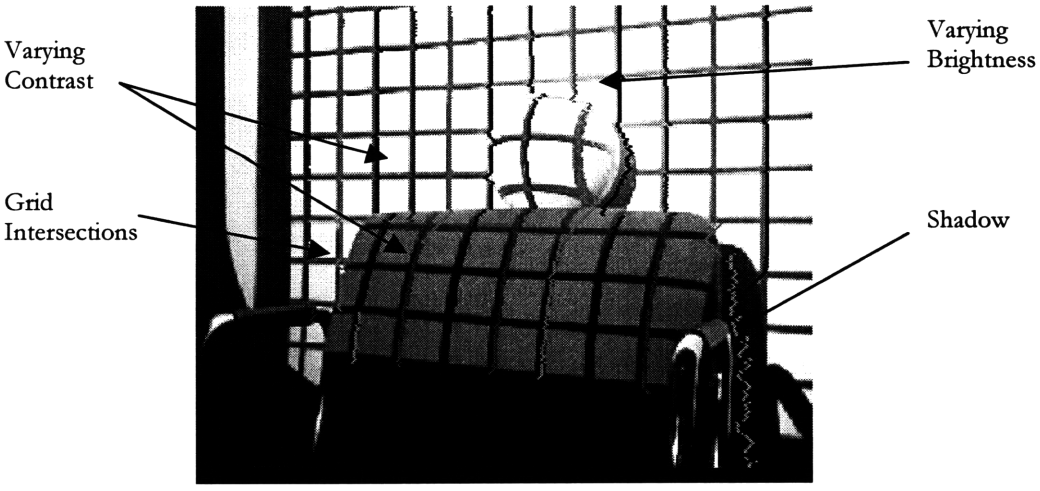


Figure 4. Difficulties of tracing grid lines. These problems forced us to abandon the "tracing approach" to finding grid intersections.

Many "special case" solutions were added to the algorithm in an attempt to solve the problems outlined above. If this algorithm had worked, we would need to take just one single picture of the scanned subject – which would mean that we could decrease the cost of the system even further by using a simple overhead projector. But in the end we felt that this "tracing" algorithm was simply not workable, and were forced to abandon it in search of a better design.

4.2.2 The Solution

In Section 3.6 "Design Overview: Processing" we have already seen a high level tour of the final processing solution. Instead of recapping the processing steps here, we will just highlight the two major differences from our original "line tracing" concept:

- **Base Image:** We take an additional picture call the base image, which is picture of the subject without the grid line projected over it. Instead of trying to trace the grid lines from the grid image (where the grid is projected over the subject), we will subtract the grid image from the base image. The resultant image contains only the difference of the two – the grid lines.
- **Pixel Filter:** Instead of tracing the lines to find the grid intersection, pixel filtering is a radically different approach that looks for grid intersections by picking the pixels that matches a certain pattern.

This solution proves to be much more elegant and reliable than the "trace" approach. By subtracting the grid image from the base image, we no longer have problems with shadows, because it is present in both images and are removed. We no longer have problems with varying contrasts, since the background and the scanned subject are not present in the result image. By using a pixel filter to identify the grid intersections, we no longer need to worry about "tracing" the lines correctly. The next section will be devoted to the working of the pixel filter.

4.3 PIXEL FILTER

4.3.1 Introduction

The pixel filter alters the color of an image by examining the color of each pixel and transforming its color using the filter function.

$F(x) = x'$ where F is the filter function, x is the color of the pixel, and x' is the new color of the pixel

The simplest example would be the "identity" function, where $x = x'$ (input color is the same as the output color) and the image is not modified. Another example could be the "map-black-to-white" function, which says:

IF x = black THEN
x' = white
ELSE $x' = x$

This function turns all the black pixels in the image into white ones, while leaving all other colors unchanged. Pixel filter could be arbitrarily complex. The filter function could take many arguments and calculate the transformation based on multiple input values.

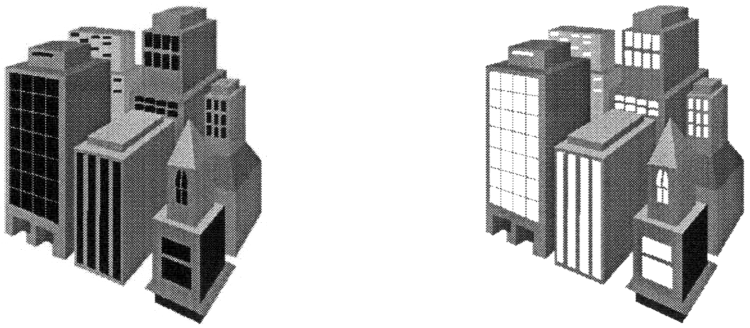


Figure 5. Demonstration of a simple "map-black-to-white" pixel filter. The original picture is on the left. The filtered picture is on the right, whose windows have been filtered to the color white.

4.3.2 Grid Intersection Filter

In our 3-D scanning image processing, we can use pixel filter to find out where the grid intersections are. Looking at the grid intersections, we notice that all intersection points share roughly the same pattern. As illustrated in the following figure, the three different grid

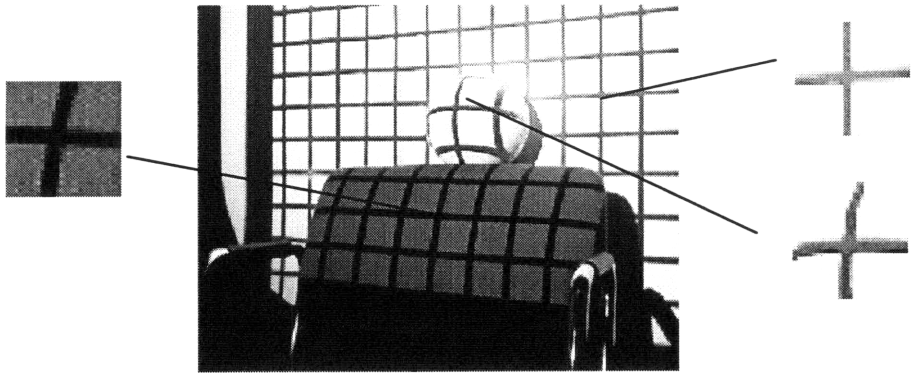


Figure 6. Close-ups of grid intersections show that they share a highly similar "cross" pattern.

intersections look like they have very different curvatures in the picture, but when we enlarge the intersections by themselves, they share a highly similar "cross" pattern.

To capture the intersection points, we created a 5-pixel by 5-pixel template to describe this "cross" pattern, and then constructed a filter to find pixels whose surrounding region matches the template. The pixels that match the pattern will remain black; otherwise they will be colored white. The matching template works as follows:

- The middle cell of the matrix corresponds to the pixel being transformed / filtered.
- Each cell in the matrix may contain a value that describes the weight of that pixel. If the cell has no value, then that pixel is completely ignored.
- If the pixel's surrounding region matches the template, then its color is unchanged. Otherwise, the pixel gets filtered out. In our case, the pixel is colored white.

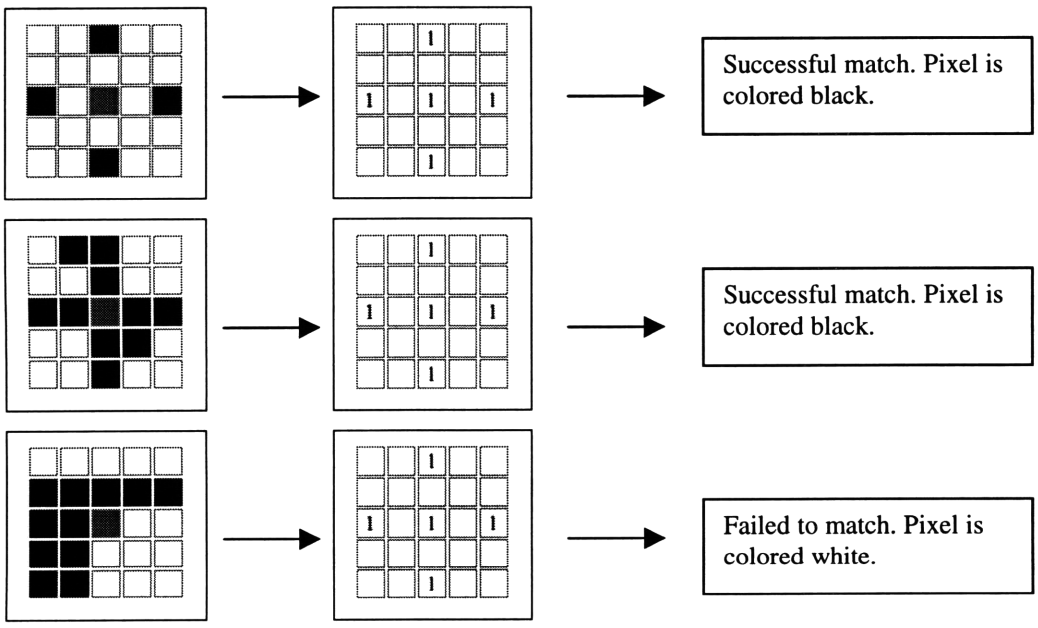


Figure 7. Pattern matching with pixel filter. Here we demonstrate three scenarios and the filter results.

The above figure illustrates how our pixel filter works. We see three sample pixels being filtered (highlighted in red). We need to decide whether these pixels should be colored black or white. The left column shows the colors of the 25 pixels surrounding the pixels in question. We match the 25 pixels against our template (shown in middle column). A match is successful only if for all the value "1" cells in the template, the corresponding pixel has the color black. If the template cell has no value, the corresponding pixel is ignored.

4.3.3 Implementation

We have implemented this pixel filter in Java, and the algorithm works remarkably well. The following picture shows the filtered image (with filtered grid intersections points in red) layered over the original grid line picture. The pixel filter has highlighted all the intersection points.

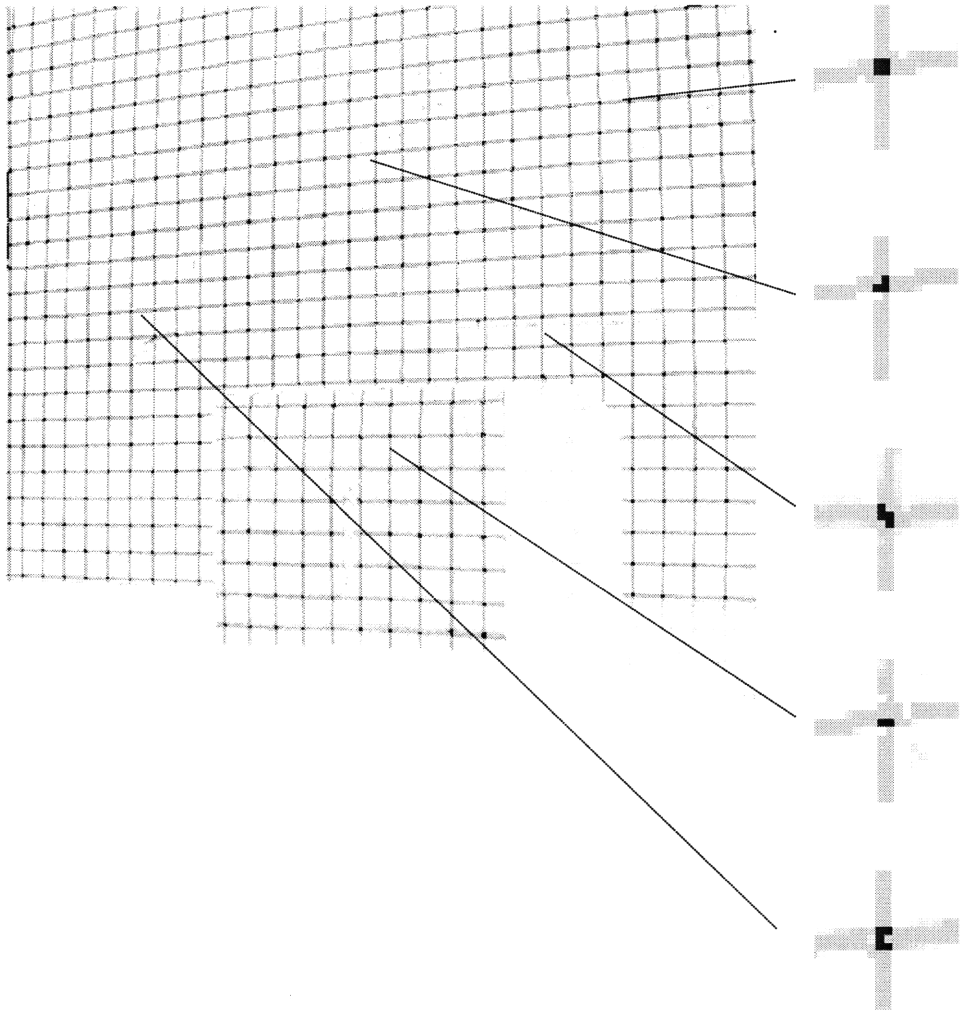


Figure 8. Result of pixel filter and sample enlargements to show details at grid intersections. The pixels selected by the filter are highlighted in red.

In the figure above, notice in the enlargements that usually the filter selects a cluster of 2 to 5 pixels for each intersection. In our prototype implementation, the program processes these clusters further and selects one pixel from each cluster. We will discuss the details in Chapter 6. The complete listing of the program is available in the appendix.

4.4 IDENTIFY INTERSECTIONS

4.4.1 The Problem

In the previous section we saw that we can use pixel filter to extract the intersection points from our images. However, these intersection points extracted are meaningless by themselves. We need to be able to map all intersection points in the grid image at (x', y') to their original locations in the reference image at (x, y). Then, we can calculate the displacement of each intersection: (x'-x, y'-y), and complete the quantification of grid distortion.

We call this mapping problem the "identification of intersections". This problem is difficult because the scanned subject's three-dimensional physical volume and its shadow obstruct some grid lines from the camera's view, so we do not have a straightforward 1-to-1 mapping from the grid image to the reference image. Furthermore, from the grid image alone, we cannot determine which grid lines are obstructed or how the grid lines have shifted compared to the reference image. Why? Because all the grid lines look identical to each other.

In the rest of this section, we will describe our first two failed attempts at solving this problem, and then we will present our final solution.

4.4.2 Two Failed Attempts

Attempt #1: Grid Line Tracing. Originally, we had hoped to be able to trace the grid lines and be able to tell how the line shifts when it hits the scanned subject. If we could follow all the lines correctly, we could reconstruct the grid and determine where the intersections from the reference image have move to in the grid image. Unfortunately, this approach failed for the following reasons:

- Difficulties with Grid Line Tracing: As we have outlined in section 4.2.1, writing software to trace lines in a picture is ridden with difficulties. We were not able to solve these problems.
- Grid Line Shifts Arbitrarily at the Border: When the vertical grid line falls on the subject, it "breaks" at the border and shifts horizontally. It is impossible to tell how much the line has shifted by examining the grid image alone, as illustrated in the following figure.

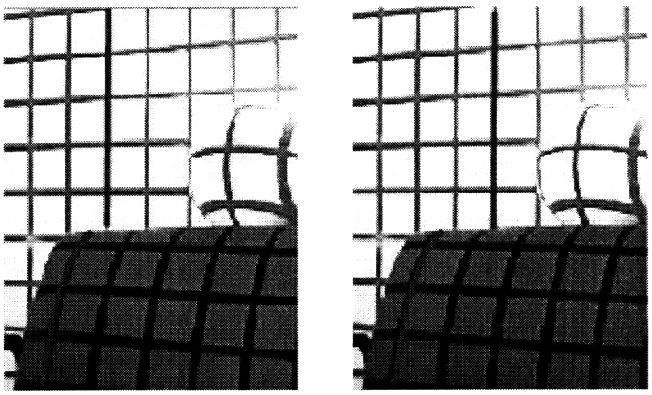


Figure 9. The problem caused by breaks and shifts of grid lines: In the pictures above, we try to reconstruct the grid line in two different ways, but it is impossible to tell which one is correct from the images alone. The one on the right is actually correct, demonstrating that grid lines can shift significantly when they land on the scanned subject.

Attempt #2: Colored Grid Lines. We also considered marking the grid lines with alternating colors (for example, red, green, blue, and yellow) so that when the line breaks and shifts we could identify the lines by its color. Unfortunately this approach was not a workable solution either for the following reasons:

- The Projected Color is Imperfect: When the project casts a black line onto the white back plane, the line looks consistently black to the human eye. However, when the digital camera captures the image, we see that the "black" line actually has various hints of blue, green, and even yellow. We believe that the uneven projected lighting at different parts of the image causes the camera to capture different colors for the "black" line.
- Change in Background Color Changes the Line Color: Even with the imperfection of the projected color, we could still tell the colors apart pretty well when the lines are projected on a white background. However, the line colors are affected dramatically when the lines are projected onto the scanned subject, which could have all kinds of different colors. The blending of line colors and background colors made it very difficult to identify lines based on their original colors.

4.4.3 The Solution

After the two failed attempts, we decided to take another picture so that we could collect more information to help us map the intersection points from the grid image to reference image. We call this the synchronization image.

The synchronization image has a number of squares positioned roughly to cover the size of the subject. We project this image on to the scanned subject and record where the five squares are located. Since the positions of these squares are predetermined, we know where they should be in the reference image, and we can use them to map intersection points in the grid image to their original locations in the reference image.

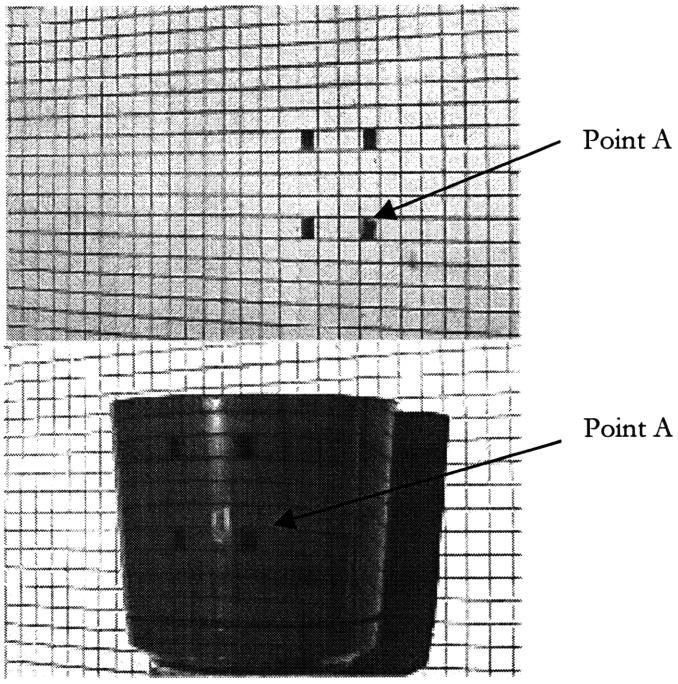


Figure 10. Identify Intersections: By following the black squares, we can tell where point A in the reference image (top) have move to in the grid image (bottom)

The figure above illustrates this procedure: In the reference image on top (where the grid is not distorted), we know that intersection point A is located at the top right corner of the synchronizing square. Now, how do we find point A in the grid image? We first find the location of the synchronizing square, then we know that point A is the point located at the top right corner of the square. Once we know point A, all other points can be identified *relative* to A. We will be discussing the details of implementing this procedure in Chapter 6.

5 Converting Distortions to Depths

5.1 Optical Model

In this section, we will present the optical model used to convert horizontal distortions to depth information. The mathematics is fairly straightforward trigonometry. The greatest difficulty was in constructing the actual model and verifying its correctness. The following figure is a graphical summary of the model. We will be explaining the setup, the assumptions, the requirements, and the calculations.

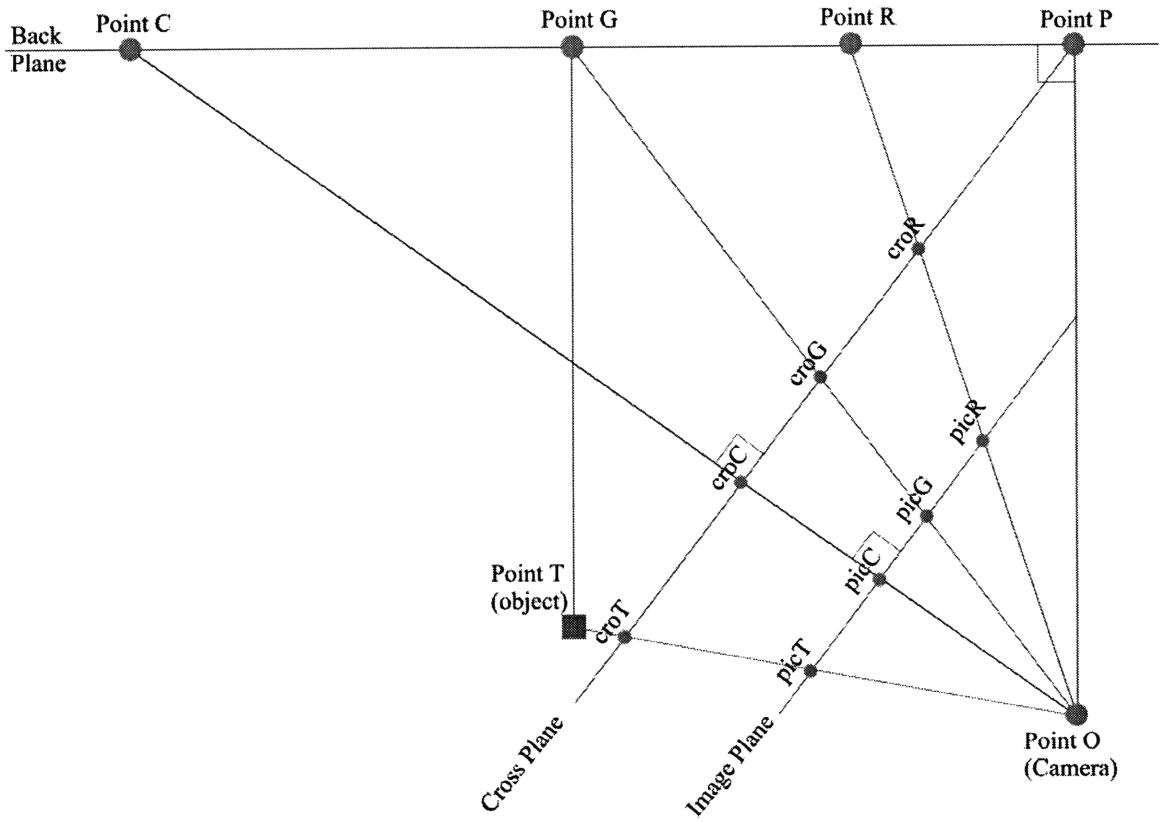


Figure 11. Diagram of Optic Model

5.1.1 Setup

The optic diagram models the view of the camera (point O) when it is capturing an image of the back plane. The projector (not shown in figure) casts a grid onto the back plane. We choose a single vertical grid line, which is point G in the diagram. When we place a subject in front of the plane, our chosen grid line lands on the subject (point T).

Then, we choose another grid line that lands on the back plane as point R. This point serves as a reference point so that we can calculate the ratio between real distances and pixel distances. Point R can be any grid line as long as it is within the view of the camera.

Next, we fixe the camera so that the *center* of its view corresponds to point C on the back plane. When the camera captures an image, the particular view can be represented as a line (picture line) that is perpendicular to the line-of-sight from the camera to point C.

Now we can locate the perpendicular point P on the back plane. Line PO is the shortest distance between the camera and the back plane. We extend another imaginary line from point P (cross line) which is perpendicular to line CO. This line is needed to complete the trigonometry calculations. Note that point P does not need to be in the view of the camera.

5.1.2 Given

From the images, we will have the distortion of grid lines in terms of pixel distances. In the figure, this is the distance croGcroT. (Chapter 4 details the quantification of grid line distortion). We are also given the pixel distance between point R and point C (in the figure this is the line picRpicC). This is a one-time measurement needed to find the ratio between real and pixel distances.

5.1.3 Objective

We want to calculate the length of GT, which is the distance between the object and the back plane (i.e. the depth of the subject at that point).

5.1.4 Measurements

The following distances are required to bootstrap the calibrations and calculations.

- Length of PO: The shortest distance between the camera (point O) and the back plane, which is the perpendicular point P.

- Length of PR: The distance between the perpendicular point P and the reference point R.
- Length of PC: The distance between point P and point C, which is the center point in the camera's view.
- Width of grid: The horizontal distance between two adjacent vertical grid lines.

5.1.5 Calibrations

The four measurements alone would have given us the parameters necessary to complete the model – if the projector were *not* a point source of light. Unfortunately, the point source nature of the project means that light travels away from the project at an angle (∠RGT), which must be measured to ensure the accuracy of the model. However, hand-measuring this angle for every grid line is a very time-consuming and error prone process. Therefore a calibration run is needed to calculate ∠RGT for each grid line.

5.1.6 Calculations

Once we have taken the measurements and calibrated the angles, we can calculate our objective, distance GT.

First, we find the real distance to picture ratio.

$$\angle PCO = \arctan\left(\frac{\overline{PO}}{\overline{CP}}\right)$$

$$\overline{RcroC} = \overline{CR} \cdot \sin(\angle PCO)$$

$$ratio = \frac{\overline{RcroC}}{\overline{picRpicC}}$$

Using the ratio, we can begin to calculate all the other distances and angles in the model. We start with finding ∠TOC.

$$\overline{croTcroC} = (\overline{picRpicC} - \overline{picRpicT}) \cdot ratio$$

$$\overline{croCO} = \sqrt{\overline{RO}^2 - \overline{RcroC}^2}$$

$$\angle TOC = \arctan\left(\frac{\overline{croTcroC}}{\overline{croCO}}\right)$$

Then, we find ∠PGO and ∠ROC.

$$\overline{GP} = \overline{CP} - \overline{CR} + \overline{GR}$$

$$\angle PGO = \arctan\left(\frac{\overline{PO}}{\overline{GP}}\right)$$

$$\angle POG = \frac{\pi}{2} - \angle PGO$$

$$\angle ROC = \frac{\pi}{2} - \angle PCO$$

Now we can completely characterize the triangle OGT.

$$\angle GOT = \angle ROC - \angle POG - \angle TOC$$

$$\angle GTO = \pi - \angle OGT - \angle GOT$$

$$\overline{GO} = \sqrt{\overline{GP}^2 + \overline{PO}^2}$$

$$\angle OGT = \angle RGT - \angle PGO$$

Finally, we will use the sine rule to find GT.

$$\overline{GT} = \overline{GO} \cdot \frac{\sin(\angle GOT)}{\sin(\angle GTO)}$$

5.2 Model Verification

We have verified the model using a simple test setup. In this experiment, we setup the camera and projector and took the four required measurements.

$$\overline{PO} = 815.98\text{mm}$$

$$\overline{PC} = 488.00\text{mm}$$

$$\overline{PR} = 205.00\text{mm}$$

$$\text{GridWidth} = 30.71\text{mm}$$

Next, we calibrated seven grid lines by calculating their ∠RGT angle. We took images of an object of known and uniform depth and then calculated the angles. The next table summarizes data from our test with seven points.

GR (mm)	61.43	92.14	122.86	153.57	184.29	215.00	245.71
picRpicT (pixel)	124	151	176	201	224	247	269
∠RGT (degrees)	95.23	94.95	94.32	94.11	93.48	93.24	92.96

$\overline{GT}$ (depth of rectangular box) = 177mm

Table 1: Summary data from test calibration

Then, we placed a cylindrical object and calculated its depths along the surface based on the ∠RGT from our calibration run. We also hand measured the depth of the cylinder at those points so that we could compare the magnitude of error.

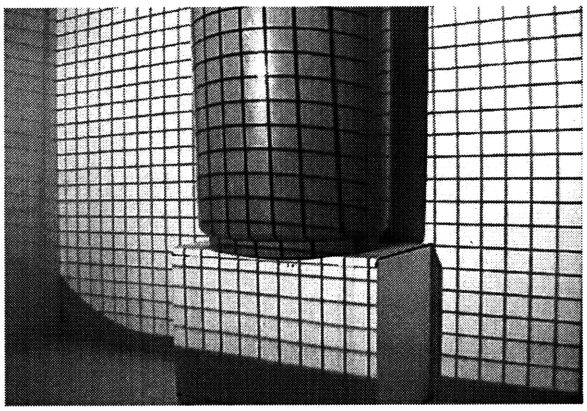


Figure 12. An image of our test setup. The cylindrical object was our scan subject. The white box below (with known and uniform depth) was used for calibration.

∠RGT (degrees)	95.23	94.95	94.32	94.11	93.48	93.24	92.96
picRpicT (pixel)	124	167	207	240	267	290	308
GT (mm)	177.00	207.41	231.51	245.00	250.67	249.43	242.26
Real depths (mm)	177.00	208.00	232.00	245.00	250.00	249.00	242.00
Error	0.00%	-0.28%	-0.21%	-0.00%	0.27%	0.17%	0.11%

Table 2: Summary data from scanning a cylindrical object.

We were pleased to find that the error margins to be ±0.30%. This simple experiment confirms that our optic model is correct (other earlier but incorrect versions of the model had errors on the order of 30%!) However, having the correct model, we still have to deal with certain calibration and hardware inaccuracies:

- Measurement from Center of Camera: Our measurement requires that we take the shortest distance from the camera to the back plane. However, the center of the camera lens is difficult to pinpoint, as the lens is encased in plastic shell.
- Camera Resolution: When extracting grid intersection from the images, we are limited to the resolution that the camera can provide. Having a higher resolution would let

us calculate grid distortions with higher precision, and thereby increasing the accuracy of the depth data.

- Other Camera and Projector Distortions: The spherical nature of the camera lens can cause distortion in the image. This problem is especially significant at the outer border of the image: instead of capturing straight grid lines, we could have lines that seem bowed. The projector can also contribute to this problem. Much like the way a monitor could cause distortion of image on the screen, the projector can adjust pincushion, trapezoid, and orthogonality properties. A faulty or incorrectly adjusted projector could cast a distorted grid.

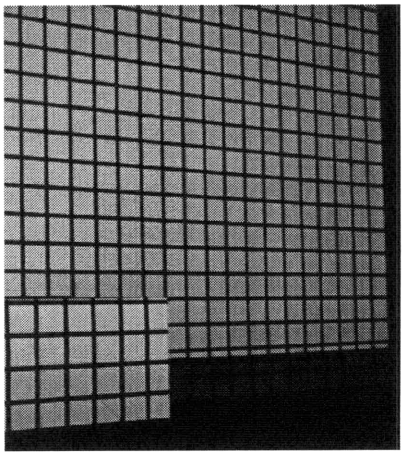


Figure 13. Camera and projector distortion. In the image above, the vertical lines on the left are parallel to the vertical edge of the image, but the vertical lines on the right are seriously slanted.

Can these problems be solved? The answer isn't clear. Fundamental hardware problems can be fixed by using better hardware, but finding software approaches to compensate for hardware inadequacies can be very difficult. Overall, we believe that our model is correct, as we were able to achieve the ±0.3% error margin.

6 IMPLEMENTATION OVERVIEW

6.1 SETUP AND FUNCTIONALITY

For the purpose of demonstration, we have implemented the algorithms in Chapter 4 and the optical model in Chapter 5 to produce a prototype scanner. In this chapter, we will first give an overview of the setup and functionality of the prototype unit. Then we will focus on the details of each of the functional components in Chapter 7.

For our prototype setup, we used a Polaroid Polaview 201 projector to cast the grid and a Polaroid PDC-2000 digital camera to capture the images. Both the camera and the projector operated at resolution of 800x600 pixels. The processing application was implemented in about 2,500 lines of Java code and ran on a Pentium 166MHz PC running Windows 95 with 32 MB of memory. The wall served as the geometric back plane and was assumed to be perpendicular to the floor.

To get our prototype up and running, we first took the required measurements to setup the optical model.

- PO — Shortest distance from camera to point P on the back plane
- PC — Distance from point P to point C, center of camera's view
- PR — Distance from point P to point R, reference point.
- Grid width — Horizontal distance between two adjacent vertical grid lines

Then we took the following reference images. Notes that Wall denotes the back plane, Grid denotes the grid lines, and Sync denotes the synchronization square. Wall + Grid means that the image is taken with the grid projected onto the wall.

- Wall
- Wall + Grid
- Wall + Sync

Next, we placed the calibration object and took three more images.

- Wall + Object
- Wall + Grid + Object
- Wall + Sync + Object

We then fed all the data and images to our processing application for a calibration run. The program produces a calibration data file.

Once we have completed calibration, we can now take images of other objects and feed them into the applications. The program will produce a 3-D model file containing (x, y, z) points of the object. This 3-D model file can then be imported into a variety of graphics programs for viewing.

6.2 Process Flow

Our processing application processes the six images following the sequential order of steps below to either produce the calibration angle data or the (x, y, z) depth points of the scanned object:

- Process Reference Images
- Process Object Images
- Synchronization Mapping
- Run Optical Model

In the rest of this section, we will discuss these four steps individually and see how we can go from 2-D images to 3-D data points. For clarity, we will be explaining the intermediate steps an "input →[procedure]→ output" notation. In this chapter, we will treat the "procedures" as black boxes with the specified functionality. In Chapter 7, we will look inside these black boxes and discuss how they are actually implemented.

6.2.1 Process Reference Images

The reference images are the ones taken without the scanned object. We need two pieces of data from these images:

- Ref_Intersect_Pts: Array of intersection points location (x, y).
- Ref_Sync_Pt: Location of the synchronization point.

The processing consists of four intermediate steps:

(Image of Wall+Grid, Image of Wall) → [Image Subtractor] → (Image of Reference Grid)

(Image of Reference Grid) → [Intersection Extractor] → (Ref_Intersect_Pts)

(Image of Wall+Sync, Image of Wall) → [Image Subtractor] → (Image of Reference Sync)

(Image of Reference Sync) → [Sync Extractor] → (Ref_Sync_Pt)

We will briefly describe the functionality of the black box procedures mentioned above: *Image subtractor* takes two input images and returns a new image that contains the difference of the two. *Intersection extractor* takes a grid image and returns an array of intersection points at (x, y). *Sync extractor* takes a sync image and returns the location of the synchronization square found in the image. We will discuss their implementation details in the next chapter.

In practice, the processing of reference image only needs to be done once for every calibration. However, in our prototype implementation we have included the processing in every run for demonstration purposes.

6.2.2 Process Object Images

The object images are the ones taken with the scanned object. We need two pieces of data from these images:

- Obj_Intersect_Pts: Array of (x, y) which are locations of intersection points covering the scanned object.
- Obj_Sync_Pt: Location of the synchronization point.

The processing consists of six intermediate steps that give us the two items that we need:

(Image of Wall+Grid+Obj, Image of Wall+Obj) → [Image Subtractor] → (Image of Object Grid)

(Image of Wall+Obj, Image of Wall) → [Image Subtractor] → (Image of Object Outline)

(Image of Object Grid, Image of Object Outline) → [Image Subtractor] → (Image of Obj Outline Grid)

(Image of Object Outline Grid) → [Intersection Extractor] → (Obj_Intersect_Pts)

(Image of Wall+Obj+Sync, Image of Wall+Obj) → [Image Subtractor] → (Image of Object Sync)

(Image of Object Sync) → [Sync Extractor] → (Obj_Sync_Pt)

Notice that we have added a step to find the outline of the object. The object outline helps us eliminate the intersection points that are not on the object, and therefore reduces the processing work significantly.

6.2.3 Synchronization Mapping

Now that we have processed the images and collected the intersection points into arrays, we are ready to synchronize the object image with the reference image. With the help of the

synchronization points, we will map all the points in the object image to their original location in the reference image. We produce the Mapped_Array, whose elements are the coordinates (Obj_x, Obj_y, Ref_x, Ref_y).

(Ref_Intersect_Pts, Ref_Sync_Pt, Obj_Intersect_Pts, Obj_Sync_Pt) → [Point Mapper] → (Mapped_Array)

6.2.4 Run Optical Model

We have now completed the entire image processing work. The Mapped_Array tells us quantitatively how the intersection point have moved from their original location in the reference image to their 'distorted' location in the object image. Having this data, we can finally run the optical model. There are two different ways to use the optical model., depending on whether this is a "calibration" run or an actual "scan" run.

First, we can use the model to calculate the RGT angles for calibration. Recall from Chapter 5 that these angles are formed by the light rays traveling from the projector to the back plane, and that for every vertical grid line, we need to calculate its RGT angle. For calibration, the object being scanned must be of known and uniform depth and we must provide the pre-measured depth of the object. The program will give us a calibration data file containing the angle data for each of the vertical grid lines.

Calculate angles: (Mapped_Array, Real_Object_Depth) → [Optical Model] → (Ref_x_Data)

(Ref_x_Data) → [File Writer] → Calibration Data File

Once we have the calibration data in place, we can give the program images of arbitrary objects, and the program will give us a 3D Model File containing the (x, y, z) data points of the object.

Calculate depths: Calibration Data File → [File Reader] → (Ref_x_Data)

(Mapped_Array, Ref_x_Data) → [Optical Model] → (real_x, real_y, z)

(real_x, real_y, z) → [File Writer] → 3D Model File

We have finally produced a 3-D data file from the 2-D images! In the next chapter, we will discuss the implementation of the black-box procedures used in all the intermediate steps in detail.

7 DETAILS OF PROCESSING COMPONENTS

In the previous section, we explained the processing flow with a handful of procedures which we treated as "black boxes". In this section, we will look inside these black boxes and see how they are actually implemented. The three we will examine are: image subtractor, intersection extractor, and point mapper. The optic model component is not explained here because we have already detail the optic mathematics in chapter 5.

7.1 IMAGE SUBTRACTOR

The functionality of the imager subtractor is very simple: take two input images, subtract the color on a pixel by pixel basis, and return the resultant difference. In our implementation, we go one step further and turn the images from "red, green, and blue" colors into black and white. This is to facilitate processing by other components that will use the image as their inputs.

We ended up with a simple implementation of the image subtractor because the input images from our prototype setup subtract very cleanly. If the input images are noisy or if the camera performs any kind of automatic image adjust, this component will need to be extended to handle the additional processing.

7.2 INTERSECTION EXTRACTOR

The goal of the Intersection Extractor is to take the image of the grid lines, and produce an array of (x, y) points that are the grid line intersections. Chapter 4.2 gave detailed description of this problem and the design of the solution. Here, we will focus on the implementation of the solution.

The task of extracting intersections can be broken into two smaller steps:

Step 1: {Image of grid lines} → [Pattern Filter] → {Image of intersection points}

Step 2: {Image of intersection points} → [Grid Point Processor] → {Array of (x, y) points}

We will now describe the two components, Pattern Filter and Grid Points Processor, that help us extract intersection points.

7.2.1 Pattern Filter

The pattern filter component takes a grayscale image of the grid lines, and outputs an image of grid line intersection points. Recall from Chapter 4.3 that filtering works on a pixel by pixel basis, assigning a new color to each pixel based on how well its surrounding pixels match a given pattern.

Pattern description file: In our prototype implementation, we let the user describe the desired pattern in an external file. Below is the pattern file used in our prototype to filter out the grid intersection points. The first five lines specify parameters about the pattern, followed by the coefficient data for the actual pattern itself:

```
row 5
col 5
targetrow 3
targetcol 3
threshold 75
0 0 1 0 0
0 0 0 0 0
1 0 1 0 1
0 0 0 0 0
0 0 1 0 0
```

"Row" and "col" specify the dimension of the pattern. "Targetrow" and "targetcol" specify the location of the pixel to be filtered in the pattern. "Threshold" specifies the upper bound for the color of a "matching" pixel, ranges from 0 (black) to 255 (white).

Filtering process: The pixel to be filtered is called the target pixel. The pattern is matched against the area of the same dimension surrounding the target pixels. Wherever the coefficient is '1' in the pattern, the corresponding pixel must have a color value that is less than the threshold value. If all the pixels marked with '1' in the pattern have color value less than the threshold, then the target pixel is colored black. Otherwise, the target pixel is colored white. For a more graphical explanation, please see Chapter 4.3.

Image boundary: The filtering process is very straightforward, but it does have its tricky boundary cases. As we move our pattern template across the image a pixel at a time, we will have problems with our calculations whenever we come to the borders of the image. As shown in the picture on the right, this problem is caused by the pattern "hanging over"

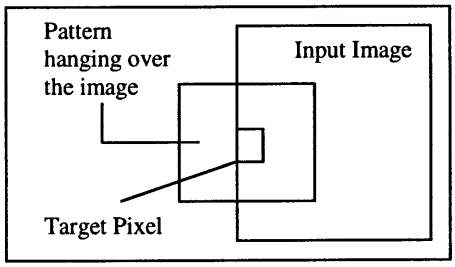


Figure 14. Pixel filter "over hanging" at the edges of the image. We need to take special care of these boundary conditions.

the edge of the input image and there are no pixels to match the pattern. This problem occurs at the top, left, right, and bottom borders of the image. We considered some different methods to cope with this problem:

- Completely ignore the data at the edges of the image and do not filter those problematic pixels.
- Replicate data at the edges of the image, or give the missing data a default "fill-in" value. Then perform filtering as usual.

In our implementation, we decided to completely ignore the data at the edges of the image. Our choice means that we lose all that data at the edges of the image and that the scanned subject cannot be placed at the edges. However, since the edges often contain noisy or distorted data, we feel that it is advantageous to lose the edges anyway.

These components are implemented in the classes `PatternFilter`, `PatternParser`, and `Pattern`. For code listing please see appendix.

7.2.2 Cluster Processor

The pattern filter gives us an image of the intersection points. To know exactly where these intersection points are, all we have to do is pick them off the image and record their (x, y) locations. How difficult can this be?

Indeed, the problem seems easy at the first sight. The pattern filter does give us a clean image of the intersection points. But when we zoom in and examine the image at the pixel level, we see for every intersection, the filter gives a cluster of pixels, as shown in the figure on the right. This is because the grid lines are more than one pixel thick, and the "clusters" in the image reflect the thickness of the grid lines.

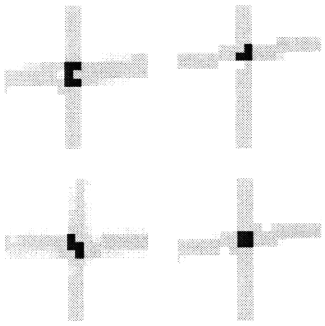


Figure 15. Close-ups of the image show that the pixel filter gives irregularly shaped clusters of pixels.

These "clusters" poses significant problems for several reasons:

- For every grid intersection, we only want <u>one</u> pixel representing that intersection. This means that for every cluster, we must choose a pixel from it, using a consistent heuristic. In our prototype, we choose the lower-rightmost pixel from every cluster.

- To make a consistent pixel choice, we need to locate the cluster in its entirety so that we can find, say, its lower-rightmost pixel. However, these cluster come in completely unpredictable shapes, so profiling them is difficult.
- To make things even more complicated, the locations of these clusters are not predictable or known, so we need an algorithm can locate clusters in arbitrary locations.

Cluster locating algorithm: Fortunately, we have found an algorithm that can locate clusters reliably and quickly. Scan across the image from top to bottom, starting at the upper-left hand corner and sweeping to the right. To keep track of which pixels have been examined, we attached a "visited" flag to all the pixels we have processed. The description of the algorithm is organized into the following pseudo code format for clarity:

```
Loop over the entire image. For every pixel at x, y:
{
        If (pixel color = white)
                Then pixel does not belong to a cluster. So skip to the next pixel.
        If (pixel color = black && pixel is marked "visited")
                Then pixel already belongs to a cluster that we have processed.
                So skip to the next pixel.
        If (pixel color = black && pixel is not marked "visited")
                Then pixel is part of a new cluster. Call ProcessCluster (x, y)
        Mark the pixel as "visited"
}
```

Finding cluster's bounding box: The ProcessCluster function locates the entire cluster by incrementally exploring its neighboring pixels. Since we are scanning the image from top to bottom, we know that the new pixel we just found must be in the top row of the cluster. Therefore we only need to look in three directions (left, right, and below) until all the neighboring pixels are white. In our implementation, we keep track of the bounding box of the cluster, and we expand incrementally expand the box to enclose the entire cluster.

```
ProcessCluster (x, y)
{
        // Initialize bounding box variables
        box_x_left = box_x_right = x
        box_y_bottom = box_y_top = y
        saw_more_black_pixel = true
        // Loop as long as we are still finding black pixels
        While (saw_more_black_pixel = true) {
                // Expand bounding box to the left
```

```
            Let (y = box_y_bottom) and examine pixel at (box_x_left-1, y). Keep on
                decreasing box_x_left by 1 until we see a white pixel.
            // Expand bounding box to the right
            Let (y = box_y_bottom) and examine pixel at (box_x_right+1, y). Keep on
                increasing box_x_right by 1 until we see a white pixel.
            // Expand bounding box downwards
            Increase box_y_bottom by 1
            saw_more_black_pixel = false
            Let (x = box_x_left TO box_x_right). Examine pixel at (x, box_y_bottom).
                If pixel is black, set (saw_more_black_pixel = true).
        }
        // Mark all the pixels within the bounding box as "visited"
        Let (x = box_x_left TO box_x_right) and (y = box_y_top TO box_y_bottom). Mark
            pixels at all (x, y) as "visited".
        // Select the representing pixel from the cluster
        Call ChoosePixel (box_x_left, box_x_right, box_y_top, box_y_bottom)
}
```

This "expanding bounding box" strategy reliably and quickly locates the entire cluster. The "visited" tag makes sure that we avoid processing pixels that already belong to a cluster. Once ProcessCluster figures out the bounding box, all that is left to do is to select the lower-rightmost pixel from the cluster, which is very straightforward. The location (x, y) of the chosen pixel is inserted into the array, where all the other representing pixels of the clusters are collected.

Finally, we have completed our task of extracting grid intersection! We have taken an input grid line image and produced an array of (x, y) points of grid intersections.

7.3 Point Mapper

The goal of the Point Mapper is to map intersection points from the object image to their original location in the reference image. It takes the two synchronization points and two arrays of intersection points from the object image and the reference image, and it produces an array that contains pairs of intersection points. Each pair of points contains an intersection point from the object image, and its original location point from the reference image.

In Chapter 4.4, we had given detailed description of the problem of identifying intersection and addition of synchronization squares to solve the problem. To recap briefly, the sync square serves as an "anchoring point". By following the movement of the synchroization

square, we can establish the identity of its neighboring intersection point. Once we have the first point mapped, we can map all the other points by their relative position to our first point.

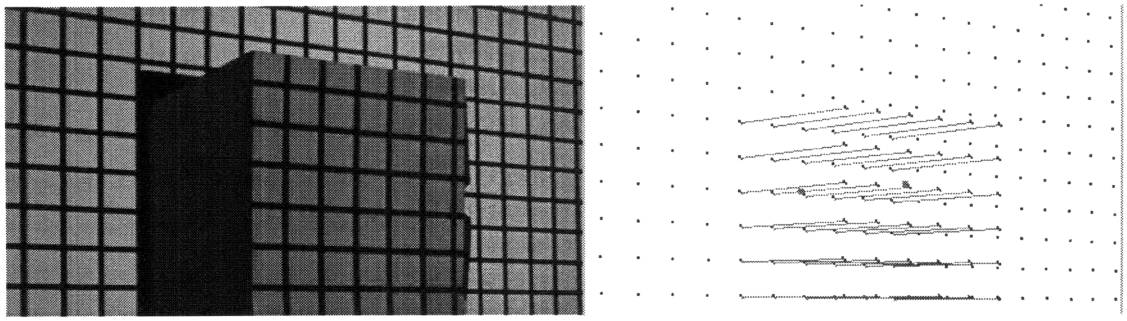


Figure 16. The image on the left shows a box being scanned. The image on the right shows its point mapping result. The lines connect points on the box (in red) to their original location in the reference image (in blue). The green dots are the anchor points.

In implementation, we found that synchronization square does serve as an effective anchor point. However, given the anchor point, locating points relative to each other turned out to be a difficult problem as well. In the rest of this section, we will focus on the algorithm that locates and maps all the points using their relative location to their neighboring points.

Problems with locating points relatively: At first glance, this problem does not appear to be too difficult. After all, the points are not in completely random positions – they are connected by grid lines. This relative "predictability" of location is true for intersection points in the reference image, where the grid is not distorted. However, when the grid is projected over an object, the grid lines can bend and move dramatically, which means we have no idea what the distances between grid points may be, or how their locations have changed.

Furthermore, we must cover all the points covering the scanned subject, which can take on any arbitrary shape. This means that we do not have a rectangular bounding box that could help us work with points at the edges of the subject.

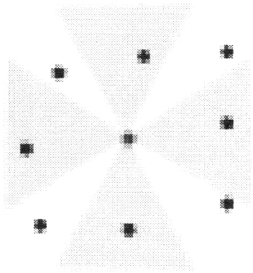


Figure 17. Finding the next point on the grid line by looking for the closest point that lands inside a 30° cone in the specified directions.

The point mapping algorithm: After many different iterations of design, we adopted an algorithm that searches for points outwards in four different direction from our anchoring point. The next point in the grid in the direction

up, down, left, or right is defined to be the closest point that lands inside a 30° cone in the specified direction from the anchor point. We cover the entire collection of grid points by a finding the left and right points from the anchor points, and then finding the points above and below these neighboring points.

Limitation of this algorithm: This point mapping algorithm isn't 100% fool proof. In fact, it will break whenever there are "missing intersection points" in the image (i.e. the pixel filter missed those intersection either because the image was somehow degraded or the grid lines do not show up well at the edges of the object). Why does this happen? Because our algorithm uses the 30° cone to find the next *closest* point, but it cannot make judgement on the how far that closest point can or should be. We believe that this problem can only be solved by some kind of "cross checking" mechanism to verify these grid points against the original image. How exactly this cross checking will work will have to be left to future work.

We have now completed the presentation of the technical details of the system. In the next chapter, We will perform an in-depth evaluation of our design and implementation effort, and see where improvements can be made.

8 EVALUATION

In this chapter, we will evaluate our design and implementation by seeing how well we fulfilled our original design goals: affordability, ease of operation, and small footprint (as presented in Chapter 3). Beyond the design goals, there are certain criteria that all systems have to fulfill to be successful. We will further evaluate our system in these four important areas: accuracy, robustness, ease of setup, and range of applications.

8.1 FULFILLMENT OF DESIGN GOALS

The original design goals of our 3-D scanner were affordability, ease of operations, and small footprint. However, as we went through the actual implementation, we had to make some compromises to make the system a reality. How far did we stray from our original goals? We will discuss each of them below:

Affordability

How much would our prototype setup cost? Here is the listing of the latest street price from www.buycomp.com:

Pentium 166MHz PC w/ 32MB RAM	$700.00
Polaroid PDC-2000 Digital Camera	$1,300.00
Polaroid Polaview 201 Projector	$2,800.00
Total	$4,800.00

This total is higher than what we were expecting, because we were not planning on using such high quality components when we first conceived the idea for this project! In fact, many of our earlier tests were conducted using a $400 overhead projector and a $500 digital camera. However, as we went through iterations of the design, we had to make digital projector a mandatory component, which increased the total price by a significant amount.

Also, we must keep in mind that the prices of all three components are falling rapidly, so as time goes by, our system will do better and better in terms of affordability.

Ease of Operation

We found the system very easy to use. Although we did not automate the control of the camera, the image capture process was very straightforward. If the integration with the camera and projector were completed, the user would only need to press a key, and the processing program will take care of the rest.

Small Footprint

The footprint of our prototype implementation measured 3 feet wide, 6 feet deep, and 4 feet tall. It could measure objects that are up to 2 feet wide, 2 feet deep, and 3 feet tall. Of course, the footprint of our scanner depends on the size of the object being scanned; but overall, we feel that the size of the footprint is reasonably in-lined with what we were expecting. For example, to measure a human-size object, our setup would occupy a space that is roughly the same size as an average dressing room.

8.2 Accuracy

The resolution of the images and the dimensions of the actual measured space determine how sensitive the system is to processing error. For example, let's say the camera gives images that are 800 pixels wide. If we use the camera to capture an 800mm wide space, the width of each pixel translates into real width of 1mm. So our error rate is 1mm * "pixel distance from chosen pixel to the correct one". On the other hand, if we use the camera to capture a 4000mm wide space, then the width of each pixel translates into 5mm, so if we are off by just a single pixel, we are penalized with 5mm of error!

The simplicity of this error sensitivity relationship between the resolution of the image and the dimensions of the actual measured space is both an advantage and a disadvantage. It gives us the flexibility to set the required image resolution and the width of the space we want to measure, depending on the precise amount of error we can tolerate. However, it also sets a fundamentally limit on the accuracy performance of the system for a given image resolution and measured width. Unfortunately, we have not been able to think of any means to bypass this limit.

8.3 ROBUSTNESS

The algorithms for extracting intersections and mapping points are designed to be as robust as possible and they work very well when given clean input images. However, while testing with our prototype setup, we have found plenty of ways to break these algorithms. We will present some of these robustness issues:

Erroneous Pixel Filter Result

The pixel filter used to locate the intersection works very efficiently. However, our current implementation of the filter cannot tell whether the filtered pixels are really part of an intersection, or just erroneous noise in the image.

Object Surface Shine and Texture

If the object has a glossy surface that reflects light, the image capture will have "shine spots" on the object that can cause problem when processing the image. The color of the "shine spots" are often very close to the color of the wall, which means that our program will fail to recognize those areas are part of the object.

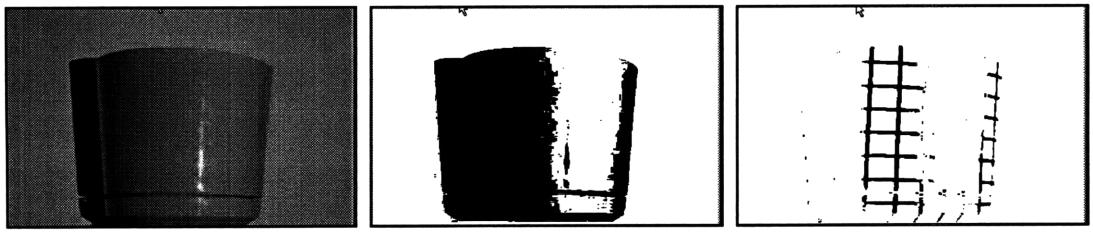


Figure 18. Demonstration of surface shine problem. Image on the left shows a glossy object being scanned. The middle image shows that the glossiness causes problem when finding the outline of the object, resulting in seriously degraded image of the object grid (right)

Object with Dark Color

In our current implementation, black grid lines are projected onto the object. If the object has dark color, then the grid lines will show up poorly in the images, causing the processing algorithms to break. We can solve this problem by first checking the color of the object. If the color is dark, then we can project the grid in "video reverse" mode – white grid lines on black background. Then we can subtract out the grid lines cleanly and proceed with the rest of the processing as usual.

8.4 Ease of Setup

Putting the hardware components in place for our prototype setup was easy. We just put the projector on the floor and the digital camera on an elevated platform. However, taking measurements that is required for calibration was very difficult. Recall that the four required measurements are:

- PO: The shortest distance between the camera (O) and the back plane, which is the perpendicular point P.
- PR: The distance from the perpendicular point P and the reference point R.
- PC: The distance from point P to point C, the center point in the camera's view.
- Width of grid: The horizontal distance between two adjacent vertical grid lines.

Measuring PO was the most difficult of all. First, we could only guess where the center of the camera lens is; and secondly, we could not easily make sure that we were measuring the shortest line that is perpendicular to the wall. Finding the center of the camera's view was also quite difficult. We could easily find the center of the image, but finding that point on the wall took some time and patience.

However, the setup process *can* be improved. For example, we could find the length of PO by doing double calibrations with objects of two different depths (remember that we are already doing one calibration run to find the $\angle RGT$, the angle at which the gridline travels from the projector to the wall). Other techniques to further simplify the process can surely be developed.

8.5 Range of Applications

We took this idea of building a scanner for "affordable but not great quality 3-D data" to many people from different industries, and to our delight, they all met the possibility with enthusiasm. We will outline three of the suggested applications:

Custom-tailored Clothes

A representative from Levi Strauss suggested that such a system could be used to collect measurements of a human body for making custom-tailored jeans. When we go to the store today, we only have a limited selection of sizes to choose from, and they are determined by waist and inseam only. However, the trend for

mass-custom-tailored clothes is growing, as demonstrated by Levi's own "Personal Pair" program, which allows female shopper to choose from more than four hundred sizes. However, measuring shoppers for their sizes in a retail store can be very time consuming, so an automated system that will pick out the shopper's measurement in a flash is very appealing.

Insurance Claims

A medical doctor suggested that affordable 3-D data could greatly assist the medical community to collect quantified data for insurance claims. For example, physical therapists could take a "before treatment" and an "after treatment" image, and the effect of the therapy would be visible from the 3-D image. The low resolution would be an advantage in this application, since capturing the patience exact skin texture and muscle tone is not only unnecessary, but also undesirable.

Shipping Boxes

A representative from Federal Express suggested that such a scanner could be used to measure the shape and size of the shipping boxes. They can then use the data for everything from routing and packing items to collecting shipping statistics. In this application, only low-resolution data is needed, and the affordability of the off-the-shelf components would allow multiple installations at many different sites.

9 CONCLUSION

9.1 FUTURE WORK

Beyond solving the problems discussed in the previous chapter, there are many possible ways to extend the capability of the system. We will outline a few in this section:

Multiple Cameras

With our single camera implementation, we only have a limited view of the scanned subject. By placing another multiple cameras at different angles, we can increase our coverage of the scanned subject and produce a complete 3-D model. Having multiple cameras could also allow us to use the overlapped data to "cross-check" confidence level in the processing output, and simply discard the data points with low confidence level. For example, a camera placed directly in front of the object would reliably tell us the maximum width and height of the object, so we can safely discard any data points that land outside of that bounding box.

Increased Resolution with Moving Grid

How could we increase the resolution of our scan? One solution is to make the grid narrower, giving us more data points to work with. However, the smaller the grid gets, the harder it is to process the images. This is because thinner grid lines will not be as well defined in the images. Dr. Yibing Yang of Polaroid Corporation suggested a different solution to this problem: Move the grid across the subject and take images of the grid at each location. This will allow us to greatly improve accuracy without risking processing error.

Capture Color Information

Many commercial laser scanners today are able to capture color information using an additional digital camera. Obviously, color information can be enormously useful in many applications, and processing the images to capture color information of the scanned subject would be an excellent way to extend the functionality of the system.

9.2 SUMMARY

We have presented the design and prototype implementation of a low-cost three-dimensional scanning system. In this system, we project a fixed grid onto the subject being scanned. By placing a camera at an angle, we can capture the distortion of the grid caused by the scanned subject. We then process the image to quantify the grid distortion, which can then be converted into a collection of depth points.

This system uses off-the-shelf personal computer, digital camera, and computer projector and requires no special lighting equipment. Although its precision and resolution will not be comparable to those of existing laser scanners, the affordability and small footprint of this design could make it very attractive to an entirely new class of applications.

Our current design and implementation is very far from being suitable for commercial use, and we have detailed the many inadequacies in Chapter 8. However, the prototype setup has served well as a "proof-of-concept", and as we have outlined in this paper, there are many different ways to improve the system. With further work, we believe it is possible to turn this design into a practical implementation.

10 REFERENCES

Lindley, Craig (1995). Photographic Imaging Techniques in C++, John Wiley & Sons, New York, New York.

Phillips, Dwayne (1994). Image Processing in C: Analyzing and Enhancing Digital Images, R & D Publications, Lawrence, Kansas.

Karlsruhe J (1989). Optical 3-D measurement Techniques : Applications in inspection, quality control, and robotics, Wichmann, Vienna, Austria.

Heard, Harry Gordon (1968). Laser Parameter Measurements Handbook, John Wiley & Sons, New York, New York.

Clarke, D. (1971). Polarized Light and Optical Measurement, Pergamon Press, Oxford, New York.

Bechek, Robert I. (1981). Robot end Effector Position and Orientation Measurement Using Laser Triangulation for Improved Accuracy, Thesis M.E., MIT, Cambridge, Massachusetts.

Bellingham, Wash. (1989) Surface Measurement and Characterization: Proceedings of SPIE – The International Society for Optical Engineering, Hamburg, Federal Republic of Germany.

11 APPENDICES

11.1 CLASS PROTOTYPE

```
// ********************************************************************************
// Prototype                                              Wandy Sae-Tan Spring 1998
//
//
// ********************************************************************************
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
import java.util.*;
import java.io.*;

import PrototypeFrame;
import MapPoints;      // originally SyncPoints
import OpticCalculator;
import GridPointProcessor;
import PatternFilter;
import DesaturationFilter;
import FileIO;
import ImageProcessor;

public class Prototype extends Applet
{

    // ********************************************************************************
    // S E T U P   A R E A
    // ********************************************************************************
    // These numbers are measured by hand

    // Data for "Polaroid 2"
    int img_w = 800,
        img_h = 600,
        crop_w = 450,
        crop_h = 280;

    int picRx = 78,  //73 old
        picRy = 258; //316 before crop

    double PO = 1088.53,
           PR = 276,  //271 old
           RC = 443,  //448 old
           PC = PR+RC,
           gridwidth = 32.5,
           picRpicC = (double)(img_w/2 - picRx);

    private double    m_cali_depth = 0;//350.0;

    // Default Filenames
    String m_directory         = "",
           m_file_wall         = "w.jpg",
           m_file_wall_grid    = "w_g.jpg",
           m_file_wall_sync    = "w_s.jpg",
           m_file_wall_obj        = "w_o.jpg",
           m_file_wall_obj_grid = "w_o_g.jpg",
           m_file_wall_obj_sync = "w_o_s.jpg";

    String patternFile    = "7crosspattern.txt",
           angleDataFile  = "AngleData.txt",
           xyzDataFile    = "XYZ.txt";

    String  refVFile       = "RefV.txt",
           objVFile       = "ObjV.txt",
           refSyncPtFile  = "RefSyncPt.txt",
```

```
        objSyncPtFile  = "ObjSyncPt.txt",
        mappedGridPtsFile = "MappedPts.txt";

String  refGrid_file   = "refGrid.jpg",
        objAllGrid_file = "objAllGrid.jpg",
        objOutline_file = "objOutline.jpg",
        refSync_file   = "refSync.jpg",
        objSync_file   = "objSync.jpg";

// ************************************************************************
// C O N S T A N T S
// ************************************************************************

// Constants for referencing mappedGridPts
final int GRIDX = 0, GRIDY = 1, REFX = 2, REFY = 3;
// Constants for referencing angleData
final int ANGLEX = 0, GR_LEN = 1, ANGLE = 2;

// ************************************************************************
// M A I N   P R O G R A M
// ************************************************************************

Image offscreen;
ImageProcessor imager;
GridPointProcessor gridPtProcessor;
FileIO filer;
OpticCalculator opticCalc;
Vector  refV = new Vector(),
        objV = new Vector();
int[]   refSyncPt = new int[2],
        objSyncPt = new int[2];
int     mappedGridPtsSize=0;
int[][] mappedGridPts = new int[1][4];

// INITIALIZATION
//------------------------------------------------------------------------
public void init()
{
    if (!m_fStandAlone)    GetParameters(null);
    resize(crop_w, crop_h);
    requestFocus();

    imager = new ImageProcessor();
    filer  = new FileIO();
    gridPtProcessor = new GridPointProcessor();
    opticCalc = new OpticCalculator (PO, PC, RC, gridwidth, picRx, picRy, picRpicC);

    offscreen = createImage(crop_w, crop_h);
}

// RUN THE MAIN PROGRAM
//------------------------------------------------------------------------
public void go() {

    ProcessRefImages();
    //LoadProcessedData();
    ProcessObjImages();

    SynchronizationMapping();
    //LoadMappedData();
    RunOpticalModel();
    showBlank(); waitforkey();
    System.exit(0);
}
public boolean userwait = false;

public void waitforkey() {
    userwait = true;
    debug ("");
    debug ("*** Press a key to continue to the next step ... ***");
    debug ("");
```

```
        while (userwait);
    }

    //************************************************************************
    // Main Program Subroutines
    //************************************************************************

    // ProcessRefImages
    //------------------------------------------------------------------------
    public void ProcessRefImages() {
        try {
            //Image refGrid = imager.imageFileCropSubtract(m_file_wall_grid, m_file_wall,
crop_w, crop_h);
            Image img1 = imager.cropImage(imager.loadImageFile(m_file_wall), crop_w,
crop_h);
            showImage(img1); waitforkey();
            Image img2 = imager.cropImage(imager.loadImageFile(m_file_wall_grid), crop_w,
crop_h);
            showImage(img2); waitforkey();
            Image refGrid = imager.imageSubtract(img1, img2);
            showImage(refGrid); waitforkey();
            refV = gridPtProcessor.extractIntersections(refGrid, patternFile);
            showIntersections(refV); waitforkey();

            img1 = imager.cropImage(imager.loadImageFile(m_file_wall), crop_w, crop_h);
            img2 = imager.cropImage(imager.loadImageFile(m_file_wall_sync), crop_w,
crop_h);
            showImage(img2); waitforkey();
            Image refSync = imager.imageFileCropSubtract(m_file_wall_sync, m_file_wall,
crop_w, crop_h);
            refSyncPt = extractSyncPt(refSync);
            showImage(refSync); waitforkey();
            showPoint(refSyncPt); waitforkey();

            filer.writeFileWithIntColumns(refVFile, refV, 2);
            filer.writeFileWithIntColumns(refSyncPtFile, refSyncPt, 2);
            //System.gc();
        } catch (Exception e) {
            debug ("Problem in ProcessRef!");
            errorhandle(e);
        }
    }

    // ProcessObjImages
    //------------------------------------------------------------------------
    public void ProcessObjImages() {
        try {
            //Image objAllGrid = imager.imageFileCropSubtract(m_file_wall_obj_grid,
m_file_wall_obj, crop_w, crop_h);
            imager.threshold = 65;
            Image img1 = imager.cropImage(imager.loadImageFile(m_file_wall_obj), crop_w,
crop_h);
            showImage(img1); waitforkey();
            Image img2 = imager.cropImage(imager.loadImageFile(m_file_wall_obj_grid),
crop_w, crop_h);
            showImage(img2); waitforkey();
            Image objAllGrid = imager.imageSubtract(img1, img2);
            showImage(objAllGrid); waitforkey();

            imager.threshold = 40;
            img1 = imager.cropImage(imager.loadImageFile(m_file_wall), crop_w, crop_h);
            showImage(img1); waitforkey();
            img2 = imager.cropImage(imager.loadImageFile(m_file_wall_obj), crop_w,
crop_h);
            showImage(img2); waitforkey();
            Image objOutline = imager.imageSubtract(img1, img2);
            showImage(objOutline); waitforkey();
            Image objGrid = imager.maskImage (objAllGrid, objOutline);
            showImage(objGrid); waitforkey();
            objV = gridPtProcessor.extractIntersections(objGrid, patternFile);
            showIntersections(objV); waitforkey();
```

```
            img1 = imager.cropImage(imager.loadImageFile(m_file_wall_obj_sync), crop_w,
crop_h);
            img2 = imager.cropImage(imager.loadImageFile(m_file_wall_obj), crop_w,
crop_h);
            showImage(img1); waitforkey();
            Image objSync = imager.imageSubtract(img1, img2);
            objSyncPt = extractSyncPt(objSync);
            showImage(objSync); waitforkey();
            showPoint(objSyncPt); waitforkey();

            filer.writeFileWithIntColumns(objVFile, objV, 2);
            filer.writeFileWithIntColumns(objSyncPtFile, objSyncPt, 2);
            //System.gc();
        } catch (Exception e) {
            debug ("Problem in ProcessObj!");
            errorhandle(e);
        }
    }

    // SynchronizationMapping
    //-------------------------------------------------------------------------
    public void SynchronizationMapping() {
        try {
            mappedGridPtsSize = objV.size();
            mappedGridPts = new int[mappedGridPtsSize][4];
            MapPoints mapper = new MapPoints(objV, refV);
            mappedGridPts = mapper.doMapping(objSyncPt, refSyncPt);

            showBlank(); waitforkey();
            showIntersections(objV, Color.red,  2); showPoint(objSyncPt); waitforkey();
            showIntersections(refV, Color.blue, 2); showPoint(refSyncPt); waitforkey();

            filer.writeFileWithIntColumns(mappedGridPtsFile, mappedGridPts,
mappedGridPtsSize, 4);
            showMappedGridPts(mappedGridPts, mappedGridPtsSize); waitforkey();

        } catch (Exception e) {
            debug ("Problem in SynchronizationMapping!");
            errorhandle (e);
        }
    }

    // LoadProcessData
    //-------------------------------------------------------------------------
    public void LoadProcessedData() {
        try {
            refV = filer.readFileWithIntColumns(refVFile,2);
            objV = filer.readFileWithIntColumns(objVFile,2);
            Vector tempV1 = filer.readFileWithIntColumns(refSyncPtFile,2);
            Vector tempV2 = filer.readFileWithIntColumns(objSyncPtFile,2);

            refSyncPt = (int[]) tempV1.elementAt(0);
            objSyncPt = (int[]) tempV2.elementAt(0);

            showIntersections(refV); //showPoint(objSyncPt);
        } catch (Exception e) {
            debug ("Problem in LoadProcessedData!");
            errorhandle(e);
        }
    }

    // LoadMappedData
    //-------------------------------------------------------------------------
    public void LoadMappedData() {
        try {
            Vector V = filer.readFileWithIntColumns(mappedGridPtsFile,4);
            mappedGridPtsSize = V.size();
            mappedGridPts = new int[V.size()][4];
            for (int i=0; i<V.size(); i++) {
                int[] p = (int[])V.elementAt(i);
```

```
                mappedGridPts[i][0] = p[0];
                mappedGridPts[i][1] = p[1];
                mappedGridPts[i][2] = p[2];
                mappedGridPts[i][3] = p[3];
            }
            showMappedGridPts(mappedGridPts, mappedGridPtsSize);
            System.gc();
        } catch (Exception e) {
            debug ("Problem in LoadMappedData!");
            errorhandle(e);
        }
    }

    // RunOpticalModel
    //-------------------------------------------------------------------------
    public void RunOpticalModel() {

        if (m_cali_depth == 0)
        {
            // Calculate depths
            try {
                Vector angleDataVec = new FileIO().readFileWithColumns(angleDataFile, 3);
                double[][] xyz = opticCalc.findDepthData(mappedGridPts, mappedGridPtsSize,
angleDataVec);
                new FileIO().writeFileWithColumns(xyzDataFile, xyz, mappedGridPtsSize, 3);

            } catch (Exception e) {debug(e);}
        }
        else
        {
            // Calculate Calibration
            try {
                Vector angleData = opticCalc.findAngleData(mappedGridPts,
mappedGridPtsSize, m_cali_depth);
                new FileIO().writeFileWithColumns(angleDataFile, angleData, 3);}

            catch (Exception e) {debug(e);}
        }
    }

    //*************************************************************************
    // Functional Procedures
    //*************************************************************************

    // extractSyncPt
    //-------------------------------------------------------------------------
    public int[] extractSyncPt (Image img) {

        int matchcolor = 0xff000000;

        // get image ready
        int w = img.getWidth(this);
        int h = img.getHeight(this);
        int[] pix = new int[w*h];
        try {pix = new ImageProcessor().intoArray(img);} catch (Exception e) {
errorhandle(e); }

        // find the start point - simple strategy scan until hits something
        // we are assuming that the sync point is the only black pixel in image!
        int[] p = new int[2];
        for (int x=0; x<w; x++) {
            for (int y=0; y<h; y++) {
                if (pix[y*w+x] == matchcolor) {
                    p[0] = x; p[1] = y;
                    y = h; x = w;//exit for loop
                }
            }
        }
        debug(p[0] +" " + p[1] + "!");

        // find the bounding box, then choose the lower right corner
```

```
    // locate the cluster by scanning right, left, and down
    boolean atleastone = true;
    int currentx = p[0], currenty = p[1];
    int minx=currentx, maxx=currentx, realmaxx=maxx;

    while (atleastone) {

        // Look to the left and to the right
        while (minx>0   && pix[currenty*w+minx] == matchcolor) minx--;
        while (maxx<w-1 && pix[currenty*w+maxx] == matchcolor) maxx++;
        realmaxx=maxx-1;
        if (minx>0) minx--;
        if (maxx<w-1) maxx++;

        // Now we look down a row. If there are no good pixels in
        // the next row, then we have found the bottom boundary of
        // this cluster
        atleastone=false;
        if (currenty < h-1) {
            currenty++;
            int oldminx = minx, oldmaxx = maxx;
            for (int x=oldminx; x<=oldmaxx; x++) {
                if (pix[currenty*w+x]==matchcolor) {
                    atleastone = true;
                    if (minx == oldminx) minx=x; // we only want to set minx once
                    maxx = x;                    // keep on incrementing maxx
                    realmaxx = x;
                }
            }
        }
    }
    p[0]=realmaxx;
    p[1]=currenty;
    debug("sync x = " + p[0] + " y = " + p[1]);
    return p;
}

//************************************************************************
// Graphics Routines
//************************************************************************

public void showImage(Image img) {
    offscreen.getGraphics().drawImage(img,0,0,this);
    repaint();
}

public void showIntersections (Vector V) {
    showIntersections(V, Color.red, 4);
}
public void showIntersections (Vector V, Color c, int size) {
    Graphics g = offscreen.getGraphics();
    g.setColor(c);
    for (int i=0; i<V.size(); i++) {
        int[] p = (int[])V.elementAt(i);
        g.fillRect(p[0]-size, p[1]-size, size, size);
    }
    repaint();
}
public void showPoint (int[] p) {
    Graphics g = offscreen.getGraphics();
    g.setColor(Color.green);
    g.fillRect(p[0], p[1], 5, 5);
    repaint();
}
public void showPoint (int x, int y) {
    Graphics g = offscreen.getGraphics();
    g.setColor(Color.green);
    g.fillRect(x, y, 5, 5);
    repaint();
}
```

```
public void showMappedGridPts (int[][] a, int s) {
    Graphics g = offscreen.getGraphics();
    for (int i=0; i<s; i++) {
        g.setColor(Color.red);
        g.fillRect(a[i][0], a[i][1], 2, 2);
        g.setColor(Color.green);
        g.fillRect(a[i][2], a[i][3], 2, 2);
        g.setColor(Color.gray);
        g.drawLine(a[i][0], a[i][1], a[i][2], a[i][3]);
    }
    repaint();
}
public void showBlank() {
    Graphics g = offscreen.getGraphics();
    g.setColor(Color.white);
    g.fillRect(0,0,crop_w,crop_h);
    repaint();
}

//************************************************************************
// U T I L I T I E S
//************************************************************************

public void waitForImage(Component component, Image image) {
  MediaTracker tracker = new MediaTracker(component);
  try {
     tracker.addImage(image, 0);
     tracker.waitForID(0);
  } catch(Exception e) {
    errorhandle(e);
  }
}
public void debug(Object thing) {
    System.out.println("APP MSG: "+ thing);
}
public void debug(String s, int i) {
    System.out.println(s + " " + i);
}
public void debug(String s, byte b) {
    System.out.println(s + " " + b);
}
public void errorhandle (Exception e) {
    System.out.println("APP ERROR: "+ e); //System.exit(1);
}

//************************************************************************
// S U P P O R T   C O D E
//************************************************************************

// Prototype Paint Handler
//------------------------------------------------------------------------
public void paint(Graphics g)
{
    g.drawImage(offscreen,0,0,this);
}

// STANDALONE APPLICATION SUPPORT:
//     m_fStandAlone will be set to true if applet is run standalone
//------------------------------------------------------------------------
private boolean m_fStandAlone = false;

// PARAMETER SUPPORT:
// <type>       <MemberVar>    = <Default Value>
//------------------------------------------------------------------------


// Parameter names.  To change a name of a parameter, you need only make
// a single change.  Simply modify the value of the parameter string below.
//------------------------------------------------------------------------
private final String PARAM_cali_depth = "cali_depth";
private final String PARAM_directory = "directory";
```

```
private final String PARAM_file_wall_grid = "file_wall_grid";
private final String PARAM_file_wall = "file_wall";
private final String PARAM_file_wall_sync = "file_wall_sync";
private final String PARAM_file_wall_obj_grid = "file_wall_obj_grid";
private final String PARAM_file_wall_obj = "file_wall_obj";
private final String PARAM_file_wall_obj_sync = "file_wall_obj_sync";

// STANDALONE APPLICATION SUPPORT
//----------------------------------------------------------------------------
String GetParameter(String strName, String args[])
{
    if (args == null)
    {
        // Running within an HTML page, so call original getParameter().
        //-------------------------------------------------------------------
        return getParameter(strName);
    }

    // Running as standalone application, so parameter values are obtained from
    // the command line. The user specifies them as follows:
    //
    //  JView Prototype param1=<val> param2=<"val with spaces"> ...
    //-------------------------------------------------------------------------
    int    i;
    String strArg  = strName + "=";
    String strValue = null;
    int    nLength  = strArg.length();

    try
    {
        for (i = 0; i < args.length; i++)
        {
            String strParam = args[i].substring(0, nLength);

            if (strArg.equalsIgnoreCase(strParam))
            {
                // Found matching parameter on command line, so extract its value.
                // If in double quotes, remove the quotes.
                //-----------------------------------------------------------
                strValue = args[i].substring(nLength);
                if (strValue.startsWith("\""))
                {
                    strValue = strValue.substring(1);
                    if (strValue.endsWith("\""))
                        strValue = strValue.substring(0, strValue.length() - 1);
                }
                break;
            }
        }
    }
    catch (Exception e)
    {
    }

    return strValue;
}

// STANDALONE APPLICATION SUPPORT
//----------------------------------------------------------------------------
void GetParameters(String args[])
{
    // Query values of all Parameters
    //-------------------------------------------------------------
    String param;

    // directory: Parameter description
    //-------------------------------------------------------------
    param = GetParameter(PARAM_cali_depth, args);
    if (param != null) m_cali_depth = Integer.parseInt(param);

    // directory: Parameter description
```

```
    //---------------------------------------------------------------
    param = GetParameter(PARAM_directory, args);
    if (param != null) m_directory = param;

    // file_wall_grid: Parameter description
    //---------------------------------------------------------------
    param = GetParameter(PARAM_file_wall_grid, args);
    if (param != null) m_file_wall_grid = param;

    // file_wall: Parameter description
    //---------------------------------------------------------------
    param = GetParameter(PARAM_file_wall, args);
    if (param != null) m_file_wall = param;

    // file_wall_sync: Parameter description
    //---------------------------------------------------------------
    param = GetParameter(PARAM_file_wall_sync, args);
    if (param != null) m_file_wall_sync = param;

    // file_wall_obj_grid: Parameter description
    //---------------------------------------------------------------
    param = GetParameter(PARAM_file_wall_obj_grid, args);
    if (param != null) m_file_wall_obj_grid = param;

    // file_wall_obj: Parameter description
    //---------------------------------------------------------------
    param = GetParameter(PARAM_file_wall_obj, args);
    if (param != null) m_file_wall_obj = param;

    // file_wall_obj_sync: Parameter description
    //---------------------------------------------------------------
    param = GetParameter(PARAM_file_wall_obj_sync, args);
    if (param != null) m_file_wall_obj_sync = param;

}

// STANDALONE APPLICATION SUPPORT
//-------------------------------------------------------------------------
public static void main(String args[])
{
    // Create Toplevel Window to contain applet Prototype
    //-----------------------------------------------------------------------
    Prototype applet_Prototype = new Prototype();
    PrototypeFrame frame = new PrototypeFrame("Prototype", applet_Prototype);

    // Must show Frame before we size it so insets() will return valid values
    //-----------------------------------------------------------------------
    frame.show();
     frame.hide();
    frame.resize(frame.insets().left + frame.insets().right  + 450,
                 frame.insets().top  + frame.insets().bottom + 450);

    frame.add("Center", applet_Prototype);
    applet_Prototype.m_fStandAlone = true;
    applet_Prototype.GetParameters(args);
    applet_Prototype.init();
    applet_Prototype.start();
     frame.show();
    applet_Prototype.go();
}

// Prototype Class Constructor
//-------------------------------------------------------------------------
public Prototype()
{
}

// APPLET INFO SUPPORT:
//-------------------------------------------------------------------------
public String getAppletInfo()
{
```

```
        return "Name: Prototype\r\n" +
               "Author: Wandy Sae-Tan\r\n" +
               "Created with Microsoft Visual J++ Version 1.1";
    }

    // PARAMETER SUPPORT
    //----------------------------------------------------------------------
    public String[][] getParameterInfo()
    {
        String[][] info =
        {
            { PARAM_cali_depth, "int", "Parameter description" },
            { PARAM_directory, "String", "Parameter description" },
            { PARAM_file_wall_grid, "String", "Parameter description" },
            { PARAM_file_wall, "String", "Parameter description" },
            { PARAM_file_wall_sync, "String", "Parameter description" },
            { PARAM_file_wall_obj_grid, "String", "Parameter description" },
            { PARAM_file_wall_obj, "String", "Parameter description" },
            { PARAM_file_wall_obj_sync, "String", "Parameter description" },
        };
        return info;
    }
}

// System Setup Procedure
// ----------------------------------------------------------------------------
/*
    1)  - project grid and take picture
        - find picCenter and mark it on wall (record picCx, picCy)
        - find picR and mark it on wall (record picRx, picRy)

    2)  - measure PO and mark P on wall
        - measure PC and PR
        - measure gridwidth

    3)  - take 3 ref images
        - take 3 calib obj images (record calib depth)
        - take 3 test obj images (record test depth)
        - take 3 generic obj images
  */
```

11.2 CLASS PROTOTYPEFRAME

```
//*************************************************************************
// PrototypeFrame.java:
//
//*************************************************************************
import java.awt.*;
import Prototype;

//=========================================================================
// STANDALONE APPLICATION SUPPORT
//  This frame class acts as a top-level window in which the applet appears
// when it's run as a standalone application.
//=========================================================================
class PrototypeFrame extends Frame
{
    Prototype app;

    // PrototypeFrame constructor
    //-------------------------------------------------------------------------
    public PrototypeFrame(String str, Prototype app)
    {
        super (str);
        this.app = app;
    }

    // HANDLE EVENT
    //-------------------------------------------------------------------------
    public boolean handleEvent(Event evt)
    {
        switch (evt.id)
        {
            // Application shutdown (e.g. user chooses Close from the system menu).
            //-----------------------------------------------------------------

        case Event.KEY_PRESS:
            app.userwait = false;
            return true;

        case Event.WINDOW_DESTROY:
            dispose();
            System.exit(0);
            return true;

        default:
            return super.handleEvent(evt);
        }
    }
}
```

11.3 CLASS FILEIO

```
// ********************************************************************************
// FileIO                                                  Wandy Sae-Tan Spring 1998
//
// Simple File utility for prototype
//
// void writeFileWithColumns(String filename, Vector v, int col) throws Exception
//
//   This procedure writes the data from v to filename. Vector v must have
//   double[col] as its elements. col must be >= 1.
//
// void writeFileWithColumns(String filename, double[][] data, int dim1, int dim2) throws
Exception
//
//   This procedure writes the array from data to filename. dim1 is the first
//   dimension of the array, and dim2 is the second dimension. Both must be >= 1.
//
// Vector readFileWithColumns (String filename, int col) throws Exception
//
//    This procedure reads from filename and puts the data into a vector.
//    The file must have the number of columns specified by col.
//    The vector returned have elements double[col]. col must be >= 1.
//
// ********************************************************************************

import java.io.*;
import java.util.*;

public class FileIO {

    // ********************************************************************************
    // P U B L I C   M E T H O D S
    // ********************************************************************************

    // Constructor
    // --------------------------------------------------------------------------
    public FileIO() {
    }

    // writeFileWithColumns
    // --------------------------------------------------------------------------
    public void writeFileWithColumns(String filename, Vector v, int col) throws Exception
{
        try {
            OutputStream out = new BufferedOutputStream(new FileOutputStream(filename));

            for (int i=0; i<v.size(); i++) {
                double[] d = (double[])v.elementAt(i);
                for (int j=0; j<col; j++) {
                    writeString(out, Double.toString(d[j]));
                    writeString(out, " ");
                }
                writeString(out, "\n");
            }
            out.flush(); out.close();

        } catch (Exception e) {
            throw new Exception("Error in FileIO.writeFile()");
        }
    }
    public void writeFileWithColumns(String filename, double[][] d, int dim1, int dim2)
throws Exception {
        try {
            OutputStream out = new BufferedOutputStream(new FileOutputStream(filename));

            for (int i=0; i<dim1; i++) {
                for (int j=0; j<dim2; j++) {
```

```
                writeString(out, Double.toString(d[i][j]));
                writeString(out, " ");
            }
            writeString(out, "\n");
        }
        out.flush(); out.close();

    } catch (Exception e) {
        throw new Exception("Error in FileIO.writeFile()");
    }
}

void writeString(OutputStream out, String str) throws IOException
{
   int len = str.length();
   byte[] buf = new byte[len];
   str.getBytes( 0, len, buf, 0 );
   out.write(buf);
}

// readFileWithColumns
// ---------------------------------------------------------------------------
public Vector readFileWithColumns (String filename, int col) throws Exception {

    FileInputStream instream;
    StreamTokenizer token;
    Vector angleDataV = new Vector(0,1);

    try {
        // Open File and Tokenizer
        instream = new FileInputStream (filename);
        token = new StreamTokenizer(instream);

        int tok;
        while ((tok = token.nextToken()) != token.TT_EOF) {

            // record first column
            double[] datapt = new double[col];

            // record the rest of the columns
            datapt[0] = token.nval;
            for (int c=0; c<col-1; c++) {
                token.nextToken();
                datapt[c+1] = token.nval;
            }

            angleDataV.addElement(datapt);
        }
    } catch (Exception e) {
        System.out.println(e);
        throw new Exception("Error in FileIO.readAngleData()");
    }

    return angleDataV;
}

// ***************************************************************************
// P R I V A T E   M E T H O D S
// ***************************************************************************

// debug util procedures
// ---------------------------------------------------------------------------
public void debug(String s) {
    System.out.println(s);
}
public void debug(double d) {
    System.out.println(d);
}

}
```

11.4 CLASS FINDPOINT

```
/*
// ***************************************************************************
/*
// FindPoint                                          Wandy Sae-Tan Spring 1998
//
// Finds the next point in the grid using x,y coordinates
//
// ***************************************************************************

public class FindPoint {

    public final int   UP    = 0, DOWN = 1,
               RIGHT = 2, LEFT = 3;

    final double rightx  = Math.cos(Math.PI*2/6),
                 leftx   = Math.cos(Math.PI*4/6),
                 topy    = Math.sin(Math.PI*1/6),
                 bottomy = Math.sin(Math.PI*11/6);

    int[][] array;
    int X, Y, arraysize;

    public FindPoint(int[][] array, int arraysize, int xcol, int ycol) {
        this.array = array;
        this.arraysize = arraysize;
        this.X = xcol;
        this.Y = ycol;
    }

    // ***************************************************************************
    // P U B L I C   M E T H O D S
    // ***************************************************************************

    // FIND NEXT GRID POINT
    // ---------------------------------------------------------------------------
    // find the next Grid point in the direction specified from the nowpt.
    // new point's x,y are placed in int[] nextpt. If there are no more
    // points (i.e. at the border), the procedure returns false.
    public int findNextPoint(int direction, int x, int y)
    {
        int nextpt = -1;
        int mindist = Integer.MAX_VALUE;

        for (int i=0; i<arraysize; i++) {
            if (inCorrectHalf(array, direction, i, x, y) &&
                inCorrectCone(array, direction, i, x, y)) {
                if (sqrdist(array, i, x, y) < mindist) {
                    nextpt = i;
                    mindist = sqrdist(array, i, x, y);
                }
            }
        }
        return nextpt;
    }

    // ***************************************************************************
    // P R I V A T E   M E T H O D S
    // ***************************************************************************

    // sqrdist
    // ---------------------------------------------------------------------------
    // An utility funciton that calculates the distance between
    // two points in the specified array
    private int sqrdist (int[][] array, int a, int x, int y)
    {
        int xlen = array[a][X]-x;
```

```
        int ylen = array[a][Y]-y;
        return (xlen*xlen + ylen*ylen);
    }

    // inCorrectHalf
    // ------------------------------------------------------------------------
    // An utility function that checks if the test point is in the desired
    // "half" of the cartesian plane.
    private boolean inCorrectHalf(int[][] array, int direction, int test, int x, int y)
    {
        switch (direction) {
        case UP:
            return array[test][Y] > y;
        case DOWN:
            return array[test][Y] < y;
        case LEFT:
            return array[test][X] < x;
        case RIGHT:
            return array[test][X] > x;
        default:
            // Exception ("Invalid direction in inCorrectHalfGrid");
            return false;
        }
    }

    // inCorrectCone
    // ------------------------------------------------------------------------
    // An utility function that checks if the test point is in a +/- 30 degree
    // "cone" space of the cartesian plane.
    private boolean inCorrectCone(int[][] array, int direction, int test, int intx, int
inty)
    {
        // NOTE: this function assumes that the point is in the correct half!
        double x  = (double)(array[test][X]-intx);
        double y  = (double)(array[test][Y]-inty);
        double ptlen = Math.sqrt(x*x + y*y);
        double unitx = x/ptlen;
        double unity = y/ptlen;

        switch (direction) {
        case UP:
        case DOWN:
            return (unitx < rightx && unitx > leftx);
        case LEFT:
        case RIGHT:
            return (unity < topy && unity > bottomy);
        default:
            // Exception ("Invalid direction in inCorrectConeGrid");
            return false;
        }
    }


}
```

11.5 CLASS GRIDPOINTPROCESSOR

```
// ************************************************************************
// GridPointProcessor                              Wandy Sae-Tan Spring 1998
//
// Function: Locates the clusters in the image and selects the lower-right-
//           most pixel from the cluster to be the representing pixel of that
//           cluster.
//
// Input:    Image of grid points (in grayscale)
//
// Output:   A vector of int[2] containing the x,y coordinates of the
//           intersection points.
//
// ************************************************************************

import java.awt.*;
import java.awt.image.*;
import java.util.*;

public class GridPointProcessor {

    int w, h;
    int[] pixels;
    boolean[] dirty;
    Vector pts = new Vector(0,1);
    int threshold = 20;

    // ************************************************************************
    // P U B L I C   M E T H O D S
    // ************************************************************************

    // processGridPointImage
    //------------------------------------------------------------------------
    public Vector processGridPointImage (Image img, int w, int h) throws Exception {
        this.w = w;
        this.h = h;

        try {
            // Grab Pixels
            pixels = new int[w*h];
            PixelGrabber grabber = new PixelGrabber(img, 0, 0, w, h, pixels, 0, w);
            grabber.grabPixels();
        }
        catch (Exception e) {
            throw new Exception ("Error grabbing pixels from image in
GridPointProcessor.processGridPointImage");
        }

        // Copy only single component
        for (int c=0; c<w*h; c++) {
            pixels[c] = pixels[c] & 0x000000ff;
        }

        // Scan the picture, pick out the clusters, and choose one pixel
        // from each cluster
        dirty = new boolean[w*h];
        for (int i=0; i<w*h; i++) {

            // Find the next pixel that is below the threshold, but has not
            // process before. i.e. does not belong to any previously
            // seen clusters.
            if (pixels[i] < threshold && !dirty[i]) {

                // if pixel is not dirty, locate the cluster, choose
                // a pixel, and add an entry to the Vector.
                int p = processCluster(i);
                int[] pt = new int[2];
```

```
            pt[0] = p - (p/w)*w;
            pt[1] = p/w;
            pts.addElement(pt);
        }
        dirty[i]=true;
    }

    return pts;
}

// ************************************************************************
// P R I V A T E   M E T H O D S
// ************************************************************************

// processCluster
//------------------------------------------------------------------------
private int processCluster(int i) {

    int starty   = i/w,
        currenty = starty,
        currentx = i-(i/w)*w,
        minx = currentx,
        maxx = currentx,
        realmaxx = maxx;

    dirty[i] = true;

    // locate the cluster by scanning right, left, and down
    boolean atleastone = true;
    while (atleastone) {

        // Look to the left
        while (minx>0 && pixels[currenty*w+minx] < threshold) {
            minx--;
            dirty[currenty*w+minx] = true;
        }
        // Look to the right
        while (maxx<w && pixels[currenty*w+maxx] < threshold) {
            maxx++;
            dirty[currenty*w+maxx] = true;
        }
        realmaxx=maxx-1;
        if (minx>0) { minx--; dirty[currenty*w+minx] = true; }
        if (maxx<w) { maxx++; dirty[currenty*w+maxx] = true; }

        // Now we look down a row. If there are no good pixels in
        // the next row, then we have found the bottom boundary of
        // this cluster
        atleastone=false;
        if (currenty < h) {
            currenty++;
            int oldminx = minx, oldmaxx = maxx;
            for (int x=oldminx; x<=oldmaxx; x++) {
                dirty[currenty*w+x] = true;
                if (pixels[currenty*w+x]<30) {
                    atleastone = true;
                    if (minx == oldminx) minx=x; // we only want to set minx once
                    maxx = x;                    // keep on incrementing maxx
                    realmaxx = x;
                }
            }
        }
    }

    // Pick a pixel from the cluster
    int chosen;
    if (realmaxx < w) {
        boolean noupperright = true;
        for (int j=starty; j<currenty; j++) {
            if (pixels[j*w+realmaxx+1] < threshold) noupperright = false;
        }
```

```
            if (noupperright) chosen = currenty*w+realmaxx;
            else chosen =  currenty*w+realmaxx+1;
        }
        else
            chosen = currenty*w+realmaxx;

        // Diagnostic graphics for debugging purposes
        /*
        Graphics g = offscreen.getGraphics();
        g.setColor(Color.red);
        g.drawRect(minx-1,starty-1,maxx-minx+2,currenty-starty+2);
        g.setColor(Color.green);
        g.drawRect(chosen-chosen/w*w,chosen/w,1,1);
        */

        return chosen;
    }
}
```

11.6 CLASS IMAGEPROCESSOR

```
//*************************************************************************
// ImageProcessor                  Wandy Sae-Tan Spring 1998
//
// public Image loadImageFile (String file) throws Exception
//
//    This procedure reads the image from the specify file and returns
//    it as an Image object.
//
// public Image maskImageFile (String base, String mask) throws Exception
//
//    This procedure takes the mask image file and puts it over the base image.
//    The base image pixels would only "show through" if the same pixels in
//    mask image are white. (i.e. white is set to be the transparent color).
//
//    output: a grayscale image that shows the base image in the "cut-out"
//            shape of the mask image
//
//*************************************************************************

import java.awt.*;
import java.awt.image.*;

public class ImageProcessor {

    Button com = new Button(); // dummy component serves as image observer

    // THRESHOLD CONSTANT: used in imageSubtract to increase contrast of the
    // resultant difference image
    public int threshold = 75;

    //*************************************************************************
    // P U B L I C   M E T H O D S
    //*************************************************************************

    // Constructor
    // ------------------------------------------------------------------------
    public ImageProcessor() {}

    // loadImage
    // ------------------------------------------------------------------------
    public Image loadImageFile (String file) throws Exception {
        try {
            Image img = Toolkit.getDefaultToolkit().getImage(file);
            waitForImage(com, img);
            debug ("Loaded image file " + file);
            return img;
        } catch (Exception e) {
            throw new Exception ("Error in ImageProcessor.loadImageFile");
        }
    }

    // intoArray
    // ------------------------------------------------------------------------
    public int[] intoArray (Image img) throws Exception {
        try {
            int w = img.getWidth(com);
            int h = img.getHeight(com);
            int[] pix = new int[w*h];
            PixelGrabber grabber = new PixelGrabber(img, 0, 0, w, h, pix, 0, w);
            grabber.grabPixels();
            return pix;
        } catch (Exception e) {
            throw new Exception ("Error in ImageProcessor.intoArray");
        }
    }
```

```
    // imageFileCropSubtract
    // ------------------------------------------------------------------------
    public Image imageFileCropSubtract (String f1, String f2, int w, int h) throws
Exception{

        try {
            Image img1 = cropImage(loadImageFile(f1), w, h);
            Image img2 = cropImage(loadImageFile(f2), w, h);
            return imageSubtract(img1, img2);
        } catch (Exception e) {
            debug ("Problem in imageFileCropSubtract!");
            throw e;
        }
    }

    // cropImage
    // ------------------------------------------------------------------------
    public Image cropImage (Image img, int neww, int newh) throws Exception {

        try {
            int w = img.getWidth(com);
            int h = img.getHeight(com);
            int[] pix = new int[w*h];
            PixelGrabber grabber = new PixelGrabber(img, 0, 0, w, h, pix, 0, w);
            grabber.grabPixels();

            int[] newpix = new int[neww*newh];
            for (int i=0; i<neww; i++) for (int j=0; j<newh; j++)
newpix[j*neww+i]=pix[j*w+i];

            return com.createImage(new MemoryImageSource(neww, newh,
ColorModel.getRGBdefault(), newpix, 0, neww));

        } catch (Exception e) {
            debug("Problem in cropImage!");
            throw e;
        }
    }

    // imageMask
    // ------------------------------------------------------------------------
    // the images are assumed to be grayscale
    public Image maskImageFile (String base, String mask) throws Exception {
        Image img1, img2;
        try {
             img1 = loadImageFile(base);
            img2 = loadImageFile(mask);
            return maskImage (img1, img2);

        } catch (Exception e) {
            throw e;
        }
    }
    public Image maskImage (Image img1, Image img2) throws Exception {

        // image 1: base
        // image 2: mask

        int[] img1pix, img2pix;
        int w, h;
        PixelGrabber grabber;

        try {
            // Get width and height
            w = img1.getWidth(com);
            h = img1.getHeight(com);

            img1pix = new int[w*h];
            img2pix = new int[w*h];

            // Grab Pixels
```

```
            grabber = new PixelGrabber(img1, 0, 0, w, h, img1pix, 0, w);
            grabber.grabPixels();
           grabber = new PixelGrabber(img2, 0, 0, w, h, img2pix, 0, w);
           grabber.grabPixels();

           // Perform masking
           for (int i=0; i<w*h; i++) {
               int b  = img2pix[i] & 0x000000ff;
               if (b > 200) img1pix[i] = 0xffffffff;
           }

           debug("Masked image");
           return com.createImage(new MemoryImageSource(w, h, ColorModel.getRGBdefault(),
img1pix, 0, w));

       } catch (Exception e){
           throw e;
       }
   }

   // imageFileSubtract
   // ------------------------------------------------------------------------
   // substracts the two color images and returns a grayscale image that is
   // the difference of the two input images. f1 and f2 are assumed to have the
   // same dimensions.
   //
   // we need to implement auto level for the input images.
   // then invert the difference. and then adjust the brightness/contrast.
   // problem is -- haven't figured out an automatic way to judge brightness/
   // contrast tuning.
   public Image imageFileSubtract (String f1, String f2) throws Exception {

       try {
           Image img1 = loadImageFile(f1);
           Image img2 = loadImageFile(f2);
           return imageSubtract (img1, img2);

       } catch (Exception e) {
           throw e;
       }
   }
   public Image imageSubtract (Image img1, Image img2) throws Exception {

       int[] img1pix, img2pix;
       int w, h;
       PixelGrabber grabber;

       try {
           // Get width and height
           w = img1.getWidth(com);
           h = img1.getHeight(com);
            debug("w="+w+" h="+h);

            // Grab Pixels
            img1pix = new int[w*h];
            grabber = new PixelGrabber(img1, 0, 0, w, h, img1pix, 0, w);
            grabber.grabPixels();
           img2pix = new int[w*h];
           grabber = new PixelGrabber(img2, 0, 0, w, h, img2pix, 0, w);
           grabber.grabPixels();
            debug("grabbed pixels");

           // Process images
           for (int i=0; i<w*h; i++) {
               int a  = img1pix[i] & 0x00ff0000;
               int b  = img1pix[i] & 0x0000ff00;
               int c  = img1pix[i] & 0x000000ff;
               int a2 = img2pix[i] & 0x00ff0000;
               int b2 = img2pix[i] & 0x0000ff00;
               int c2 = img2pix[i] & 0x000000ff;
```

```
                // perform substraction
                if (a>=a2) a=a-a2; else a=a2-a;
                if (b>=b2) b=b-b2; else b=b2-b;
                if (c>=c2) c=c-c2; else c=c2-c;

                // turn results into grayscale
                int bri = find_brightness(a, b, c);
                if (bri < threshold) bri = 255; else bri = 0;
                img1pix[i] = 0xff000000 | bri*65536 | bri*256 | bri;
            }
            debug("Subtracted images");
            return com.createImage(new MemoryImageSource(w, h, ColorModel.getRGBdefault(),
img1pix, 0, w));

        } catch (Exception e) {
            throw e;
        }
    }

    //************************************************************************
    // P R I V A T E   M E T H O D S
    //************************************************************************

    // find_brightness
    // ------------------------------------------------------------------------
    private int find_brightness (int ar, int ag, int ab) {
        int r = ar / 65536;
        int g = ag / 256;
        int b = ab;
        if (b >= g &&  b >= r) return b;
        else if (g >= r) return g;
        else return r;
    }

    // waitForImage
    // ------------------------------------------------------------------------
    public void waitForImage(Component component, Image image) {
      MediaTracker tracker = new MediaTracker(component);
      try {
         tracker.addImage(image, 0);
         tracker.waitForID(0);
      } catch(Exception e) {
        errorhandle(e);
      }
    }
    public void debug(Object thing) {
        System.out.println("APP MSG: "+ thing);
    }
    public void debug(String s, int i) {
        System.out.println(s + " " + i);
    }
    public void debug(String s, byte b) {
        System.out.println(s + " " + b);
    }
    public void errorhandle (Exception e) {
        System.out.println("APP ERROR: "+ e); //System.exit(1);
    }
}
```

11.7 CLASS MAPPOINTS

```
// ********************************************************************************
~
// MapPoints                                              Wandy Sae-Tan Spring 1998
~
//
~
// Takes the intersection points from the grid image and maps them to their
~
// original locations in reference image.
//
// Input: grid and ref vectors containing the intersection points
//        from the respective images
//
// Output: an array of int[4] containing gridx, gridy, and their corresponding
//         refx, refy.
//
// Implementation Notes:
//
// The synchronization begins with a starting point, then the
// procedure doMapping() processes the two vectors and produce an
// array of numbers.
//
// This implementation uses the indices to navigate around the
// two arrays: grid and ref. The arrays are never sorted, we just
// want to give every pair of x,y in array grid a reference x,y from
// array ref.
//
// The "cone" is set to 30 degrees to either side of the axis.
//
// ********************************************************************************

import java.awt.image.*;
import java.util.*;
//import SyncTest;

public class MapPoints {

    Vector gridV, refV;
    int[][] ref;    // nx2 contains x,y of ref intersection points
    int[][] grid;   // nx4 contains x,y of grid and will contain
                    // x,y of corresponding ref.
    int gridsize, refsize;
    final int  UP    = 0, DOWN = 1,
               RIGHT = 2, LEFT = 3;

    final int  X = 0, Y = 1,
               REFX  = 2, REFY  = 3;

    final double rightx  = Math.cos(Math.PI*2/6),
                 leftx   = Math.cos(Math.PI*4/6),
                 topy    = Math.sin(Math.PI*1/6),
                 bottomy = Math.sin(Math.PI*11/6);

    // ********************************************************************
    // P U B L I C   M E T H O D S
    // ********************************************************************

    // CONSTRUCTOR
    // --------------------------------------------------------------------
    public MapPoints(Vector gridV, Vector refV) //, SyncTest app)
    {
        //this.app = app;
        this.gridV = gridV;
        this.refV  = refV;
        this.gridsize = gridV.size();
        this.refsize  = refV.size();
```

```
    grid = new int[gridsize+1][4];
    ref  = new int[refsize+1][2];

    // Copy vectors into array for speedy access
    for (int i=0; i<gridsize; i++) {
        int[] pt = (int[])gridV.elementAt(i);
        grid[i][X] = pt[0];
        grid[i][Y] = pt[1];
    }
    for (int i=0; i<refsize; i++) {
        int[] pt = (int[])refV.elementAt(i);
        ref[i][X] = pt[0];
        ref[i][Y] = pt[1];
    }
}

// SYNCRONIZE THE POINTS
// -----------------------------------------------------------------------
public int[][] doMapping(int[] gridSyncPt, int[] refSyncPt) {

    // Copy the sync points into the very last entry of the arrays
    // This is a bit of a hack -- but making the sync point part of the
    // arrays means that we can access the point via array index and thus
    // allowing us to navigate the arrays using indices.
    grid[gridsize][X] = gridSyncPt[0];
    grid[gridsize][Y] = gridSyncPt[1];
    ref[refsize][X] = refSyncPt[0];
    ref[refsize][Y] = refSyncPt[1];

    int startpt, newpt, lastpt;

    // find start point in grid and ref from given x,y
     startpt = mapNextGridPoint(UP, gridsize);

    // Loop from start point to the right
    newpt = startpt;
    lastpt = startpt;
    while (newpt != -1) {

        mapGridPoints(UP, newpt);
        lastpt = mapGridPoints(DOWN, newpt);

        newpt = mapNextGridPoint(RIGHT, newpt);
    }

    // Map grid points from bottom to top along the last
    // vertical line to the right
    newpt = lastpt;
    while (newpt != -1) {

        mapGridPoints(RIGHT, newpt);
        newpt = findNextPoint(grid, UP, newpt);
    }

    // look from start point to the left
    newpt = mapNextGridPoint(LEFT, startpt);
    while (newpt != -1) {

        mapGridPoints(UP, newpt);
        lastpt = mapGridPoints(DOWN, newpt);

        newpt = mapNextGridPoint(LEFT, newpt);
    }

    // Map grid points from bottom to top along the last
    // vertical line to the right
    newpt = lastpt;
    while (newpt != -1) {

        mapGridPoints(LEFT, newpt);
```

```
        newpt = findNextPoint(grid, UP, newpt);
    }
    return grid;
}

// ************************************************************************
// P R I V A T E   M E T H O D S
// ************************************************************************

// MAP GRID POINTS
// ------------------------------------------------------------------------
// map Grid points along a single direction until we come to
// the border. for every grid point mapped, we assign the ref
// x,y to it. we return the index of the last Grid point processed.
public int mapGridPoints (int direction, int pt)
{
    int newpt = pt,
        lastpt = pt;

    while (newpt != -1) {
        lastpt = newpt;
        newpt  = mapNextGridPoint(direction, lastpt);
    }
    return lastpt;
}

// MAP NEXT GRID POINT
// ------------------------------------------------------------------------
public int mapNextGridPoint(int direction, int pt)
{
    int newpt, newrefpt;

    newrefpt = findNextPoint(ref, direction, pt);
    newpt    = findNextPoint(grid, direction, pt);

    if (newpt != -1) {
        assignPoint(newpt, newrefpt);
        //draw(newpt);
        return newpt;
    } else {
        return -1;
    }
}

// FIND NEXT GRID POINT
// ------------------------------------------------------------------------
// find the next Grid point in the direction specified from the nowpt.
// new point's x,y are placed in int[] nextpt. If there are no more
// points (i.e. at the border), the procedure returns false.
public int findNextPoint(int[][] array, int direction, int nowpt)
{
    int nextpt = -1;
    int mindist = Integer.MAX_VALUE;

    for (int i=0; i<gridsize; i++) {
        if (inCorrectHalf(array, direction, i, nowpt) &&
            inCorrectCone(array, direction, i, nowpt)) {
            if (sqrdist(array, i, nowpt) < mindist) {
                nextpt = i;
                mindist = sqrdist(array, i,nowpt);
            }
        }
    }
    return nextpt;
}

// sqrdist
// ------------------------------------------------------------------------
// An utility funciton that calculates the distance between
// two points in the specified array
private int sqrdist (int[][] array, int a, int b)
```

```
    {
        int xlen = array[a][X]-array[b][X];
        int ylen = array[a][Y]-array[b][Y];
        return (xlen*xlen + ylen*ylen);
    }

    // inCorrectHalf
    // ------------------------------------------------------------------------
    // An utility function that checks if the test point is in the desired
    // "half" of the cartesian plane.
    private boolean inCorrectHalf(int[][] array, int direction, int test, int pt)
    {
        switch (direction) {
        case UP:
            return array[test][Y] > array[pt][Y];
        case DOWN:
            return array[test][Y] < array[pt][Y];
        case LEFT:
            return array[test][X] < array[pt][X];
        case RIGHT:
            return array[test][X] > array[pt][X];
        default:
            // Exception ("Invalid direction in inCorrectHalfGrid");
            return false;
        }
    }

    // inCorrectCone
    // ------------------------------------------------------------------------
    // An utility function that checks if the test point is in a +/- 30 degree
    // "cone" space of the cartesian plane.
    private boolean inCorrectCone(int[][] array, int direction, int test, int pt)
    {
        // NOTE: this function assumes that the point is in the correct half!
        double x  = (double)(array[test][X]-array[pt][X]);
        double y  = (double)(array[test][Y]-array[pt][Y]);
        double ptlen = Math.sqrt(x*x + y*y);
        double unitx = x/ptlen;
        double unity = y/ptlen;

        switch (direction) {
        case UP:
        case DOWN:
            return (unitx < rightx && unitx > leftx);
        case LEFT:
        case RIGHT:
            return (unity < topy && unity > bottomy);
        default:
            // Exception ("Invalid direction in inCorrectConeGrid");
            return false;
        }
    }

    // assignPoint
    // ------------------------------------------------------------------------
    // An utility function that assigns refpt x,y to gridpt x,y
    private void assignPoint(int gridpt, int refpt)
    {
        grid[gridpt][REFX] = ref[refpt][0];
        grid[gridpt][REFY] = ref[refpt][1];
    }

    // FOR TESTING WITH SYNCTEST
    // ------------------------------------------------------------------------
    /* SyncTest app;
    int count = 0;
    private void draw(int p)
    {
        app.offscreen.getGraphics().drawString(Integer.toString(count), grid[p][X],
grid[p][Y]);
        app.repaint();
```

```
        count++;
    }*/
}
```

11.8 CLASS OPTICCALCULATOR

```
// ************************************************************************
~
// OpticCalculator                                          Wandy Sae-Tan Spring 98
~
//
~
//
// This is a "calculator" which basically models the optical
// mathematics detailed in Chapter 5. The constructor sets
// all the required parameters to build the model. Then the
// findAngle method is called for each grid line to calibrate
// the projection "angle". Only then, can you use the findDepth
// method to find actual depths of the points from horizontal
// displacement of the pixels.
//
// ************************************************************************

import FindPoint;
import java.util.*;

public class OpticCalculator {

    // Constants for referencing mappedGridPts
    final int GRIDX = 0, GRIDY = 1, REFX = 2, REFY = 3;
    // Constants for referencing angleData
    final int ANGLEX = 0, GR_LEN = 1, ANGLE = 2;
    // Names for angle and xyz data


    // ************************************************************************
    // P U B L I C   M E T H O D S
    // ************************************************************************

    // Constructor
    // ------------------------------------------------------------------------
    public OpticCalculator (double PO, double CP, double CR, double gridwidth,
                            int picRx, int picRy, double picRpicC) {

        this.picRx = picRx;
        this.picRy = picRy;
        PO_len = PO;
        gridwidth_len  = gridwidth;
        picRpicC_len   = picRpicC;
        CP_len = CP;
        CR_len = CR;

        RO_len         = Math.sqrt((CP_len - CR_len)*(CP_len-CR_len) + PO_len*PO_len);
        PCO_ang        = Math.atan(PO_len / CP_len);
        RcroC_len      = CR_len * Math.sin(PCO_ang);
        real2pic_ratio = RcroC_len / picRpicC_len;
        ROC_ang        = Math.PI/2 - PCO_ang;
    }

    // findAngleData
    //-------------------------------------------------------------------------
    public Vector findAngleData (int[][] mappedGridPts, int mappedGridPtsSize, int depth)
{

        Vector pts = new Vector(0,1);
        double GR_len = 0, picRpicT_len;
        double[] p;

        // Find starting point that is closest to ref pont
        FindPoint finder = new FindPoint(mappedGridPts, mappedGridPtsSize, REFX, REFY);
        int startpt = finder.findNextPoint(finder.DOWN, picRx, picRy);
```

```
        // Find points to the left
        int nowpt = startpt;
        int count = 0;
        while (nowpt != -1) {

            picRpicT_len = (double)(picRx - mappedGridPts[nowpt][GRIDY]);

            p = new double[3];
            p[ANGLEX] = (double)mappedGridPts[nowpt][REFX];
            p[GR_LEN] = gridwidth_len*count;
            p[ANGLE]  = findAngle(depth, GR_len, picRpicT_len);

            pts.addElement(p);
            count++;

            nowpt = finder.findNextPoint(finder.LEFT,
                                         mappedGridPts[nowpt][REFX],
                                         mappedGridPts[nowpt][REFY]);
        }

        // Find points to the right
        nowpt = finder.findNextPoint(finder.RIGHT,
                                     mappedGridPts[startpt][REFX],
                                     mappedGridPts[startpt][REFY]);
        count = 0;
        while (nowpt != -1) {

            picRpicT_len = (double)(picRx - mappedGridPts[nowpt][GRIDY]);

            p = new double[3];
            p[ANGLEX] = (double)mappedGridPts[nowpt][REFX];
            p[GR_LEN] = gridwidth_len*count;
            p[ANGLE] = findAngle(depth, GR_len, picRpicT_len);

            pts.addElement(p);
            count--;

            nowpt = finder.findNextPoint(finder.RIGHT,
                                         mappedGridPts[nowpt][REFX],
                                         mappedGridPts[nowpt][REFY]);
        }

        return pts;
    }

    // findDepthData
    //---------------------------------------------------------------------------
    public int[][] findDepthData (int[][] mappedGridPts, int mappedGridPtsSize, Vector v)
{

        int[][] xyz = new int[mappedGridPtsSize][3];

        // copy angle data vector v into double[][] angleData
        int angleDataSize = v.size();
        double[][] angleData = new double[angleDataSize][3];
        for (int i=0; i<v.size(); i++) {
               double[] d = (double[])v.elementAt(i);
               for (int j=0; j<3; j++) angleData[i][j] = d[j];
        }

        // find depth for each point in mappedGridPts
        for (int i=0; i<mappedGridPtsSize; i++) {
            int p = closestAngleDataPoint(mappedGridPts[i][REFX], angleData,
angleDataSize);
            double picRpicT_len = (double)(picRx - mappedGridPts[i][GRIDX]);
            findDepth (angleData[p][ANGLE],
                       angleData[p][GR_LEN],
                       picRpicT_len);
        }
        return xyz;
    }
```

```
// ************************************************************************
// P R I V A T E   M E T H O D S
// ************************************************************************

//-------------------------------------------------------------------------
// Given the depth of the object, calculate the distortion
// angle of the projected light beam.  This method is used
// for calibrating the system.
private double findAngle(double GT_len, double GR_len, double picRpicT_len) {

    croTcroC_len = (picRpicC_len - picRpicT_len)*real2pic_ratio;
    croCO_len    = Math.sqrt(RO_len*RO_len - RcroC_len*RcroC_len);
    TOC_ang      = Math.atan(croTcroC_len / croCO_len);

    GP_len  = CP_len - CR_len + GR_len;
    PGO_ang = Math.atan(PO_len/GP_len);
    POG_ang = Math.PI/2 - PGO_ang;

    GOT_ang = ROC_ang - POG_ang - TOC_ang;

    GO_len  = Math.sqrt(GP_len*GP_len + PO_len*PO_len);
    GTO_ang = Math.PI - Math.asin(Math.sin(GOT_ang) * GO_len / GT_len);
    OGT_ang = Math.PI - GTO_ang - GOT_ang;
    RGT_ang = OGT_ang + PGO_ang;

    return RGT_ang;
}

//-------------------------------------------------------------------------
// Given the angle and horizontal displacement information,
// calculate depth of point on object
private double findDepth (double RGT_ang, double GR_len, double picRpicT_len) {

    croTcroC_len = (picRpicC_len - picRpicT_len)*real2pic_ratio;
    croCO_len    = Math.sqrt(RO_len*RO_len - RcroC_len*RcroC_len);
    TOC_ang      = Math.atan(croTcroC_len / croCO_len);

    GP_len  = CP_len - CR_len + GR_len;
    PGO_ang = Math.atan(PO_len/GP_len);
    POG_ang = Math.PI/2 - PGO_ang;

    GOT_ang = ROC_ang - POG_ang - TOC_ang;

    GTO_ang = Math.PI - OGT_ang - GOT_ang;
    GO_len  = Math.sqrt(GP_len*GP_len + PO_len*PO_len);
    OGT_ang = RGT_ang - PGO_ang;
    GT_len = GO_len * Math.sin(GOT_ang) / Math.sin(GTO_ang);
    return GT_len;
}

//-------------------------------------------------------------------------
private int closestAngleDataPoint(int refx, double[][] angleData, int angleDataSize) {

    double x = (double)refx;
    int minpt = 0;
    double mindist = Math.abs(angleData[0][ANGLEX]);

    for (int i=0; i<angleDataSize; i++) {
        if (Math.abs(angleData[i][ANGLEX] - x) > mindist) {
            mindist = angleData[i][ANGLEX] - x;
            minpt = i;
        }
    }
    return minpt;
}

// Utilities
//-------------------------------------------------------------------------
private void print (Object obj, double d) {
```

```
        System.out.println(obj + " " + d);
    }
    private double deg (double d) {
        return d/Math.PI*180;
    }
    private void testprint () {
        print ("PCO_ang"  , deg(PCO_ang));
        print ("RcroC_len", RcroC_len);
        print ("real2pic" , real2pic_ratio);

        print ("croTcroC_len", croTcroC_len);
        print ("croCO_len"   , croCO_len);
        print ("TOC_ang"     , deg(TOC_ang));
        print ("GP_len"      , GP_len);

        print ("PGO_ang"     , deg(PGO_ang));
        print ("POG_ang"     , deg(POG_ang));
        print ("ROC_ang"     , deg(ROC_ang));
        print ("GOT_ang"     , deg(GOT_ang));
        print ("GO_len"      , GO_len);
        print ("GTO_ang"     , deg(GTO_ang));
        print ("OGT_ang"     , deg(OGT_ang));
        print ("RGT_ang"     , deg(RGT_ang));
    }

    // ************************************************************************
    // V A R I A B L E S
    // ************************************************************************

    protected int picRx,
                  picRy;
    protected double PO_len,
                     RO_len,
                     gridwidth_len,

                     picRpicC_len,
                     CP_len,
                     CR_len,

                     PCO_ang,
                     RcroC_len,
                     real2pic_ratio,

                     croTcroC_len,
                     croCO_len,

                     GP_len,
                     PGO_ang,
                     POG_ang,

                     ROC_ang,
                     TOC_ang,
                     GOT_ang,

                     GO_len,

                     GT_len,
                     GTO_ang,
                     OGT_ang,

                     RGT_ang;
}
```

11.9 CLASS PATTERN

```
// ***************************************************************************
//
// Pattern Object                                          Wandy Sae-Tan Spring 1998
// ***************************************************************************


import java.util.*;

public class Pattern {

    // Imlementation Dependent Variables
    private boolean changable = true;
    private Vector vect;
    private int i=0;
    private int[] data;

    // Publically accessible parameters
    public int row, col,
               targetcol, targetrow,
               threshold;

    // ***************************************************************************
    // P U B L I C   M E T H O D S
    // ***************************************************************************

    // Constructor
    public Pattern(int row, int col, int targetcol, int targetrow, int threshold) {
        vect = new Vector(row*col*3);
        this.row = row;
        this.col = col;
        this.targetrow = targetrow;
        this.targetcol = targetcol;
        this.threshold = threshold;
    }

    // Add a pattern entry
    public void addEntry(int num, int xoff, int yoff) throws Exception {
        if (changable) {
            vect.insertElementAt(new Integer(num), i); i++;
            vect.insertElementAt(new Integer(xoff), i); i++;
            vect.insertElementAt(new Integer(yoff), i); i++;
        } else
            throw new Exception ("Pattern has been frozen");
    }

    // Once the pattern is frozen, no more entries can be added
    public void freeze () {
        vect.trimToSize();
        data = new int[i];
        for (int c=0; c<i; c++)
            data[c]=((Integer)vect.elementAt(c)).intValue();
        changable = false;
    }

    // Public observers used to access patterns
    public int getNum(int index) {
        return data[index*3];
    }
    public int getXoff(int index) {
        return data[index*3+1];
    }
    public int getYoff (int index) {
        return data[index*3+2];
    }

    // Public observser gives size of pattern
```

```
    public int size() {
        return vect.size()/3;
    }

    // A useful test routine to check correctness of implementation
    public void print () {
        for (int t=0; t<vect.size(); t=t+3)
            System.out.println(data[t] + " " +
                               data[t+1] + " " +
                               data[t+2]);
        System.out.println("Size = " + size());
    }

    // dummy constructor
    public Pattern() {
    }

}
```

11.10 CLASS PATTERNFILTER

```
// ************************************************************************
// PatternFilter.java                        Wandy Sae-Tan Spring 98
//
//
//
// Images to be filtered are assumed to be GRAY SCALED
// ************************************************************************

import java.awt.*;
import java.util.*;
import java.io.*;
import java.awt.image.*;

public class PatternFilter {

    Pattern pattern;
    int w,h,size;
    int val, xtest, ytest;
    int[] data;
    int[] pixels, newpixels;
    Button b = new Button();

    // ************************************************************************
    // P U B L I C   M E T H O D S
    // ************************************************************************

    // CONSTRUCTOR
    //------------------------------------------------------------------------
    public PatternFilter (File patternfilename)
    {
        // Read Pattern File and Create Pattern
        try {
            PatternParser parser = new PatternParser();
            pattern = parser.parseFile(patternfilename);
           size = pattern.size();
        } catch (Exception e) {
        }
    }

    // FILTER IMAGE
    //------------------------------------------------------------------------
    public Image filterImage (Image img, int w, int h)
    {
        pixels = new int[w*h];
        newpixels = new int[w*h];

        try {
            // Grab Pixels from img
           pixels = new int[w*h]; newpixels = new int[w*h];
            PixelGrabber grabber = new PixelGrabber(img, 0, 0, w, h, pixels, 0, w);
            grabber.grabPixels();

        } catch (Exception e) {
        }

        // Run Filter
        filterPixels(pixels, newpixels, w, h);

        // Draw to new image
        Image newimg = b.createImage(w,h);
        Graphics g = newimg.getGraphics();
        for (int i=0; i<w; i++) {
            for (int j=0; j<h; j++) {
                if ((newpixels[j*w+i] & 0x000000ff) < 10)
                    g.fillRect(i,j,1,1);
            }
```

```
        }
        return newimg;
    }

    // ************************************************************************
    // P R I V A T E   M E T H O D S
    // ************************************************************************

    // filterPixels
    //------------------------------------------------------------------------
    private void filterPixels (int[] pixels, int[] newpixels, int w, int h)
    {
        int x,y;

        // Calcuate and set boundaries
        int xmin = pattern.targetcol - 1;
        int xmax = w - (pattern.col - pattern.targetcol);
        int ymin = pattern.targetrow - 1;
        int ymax = h - (pattern.row - pattern.targetrow);

        // We do not process the pixels at the borders because there isn't
        // enough area to match the pattern. Therefore, we them all to white.
        for (x=0;x<w;x++) for (y=0;y<ymin;y++) newpixels[y*w+x]=255;
        for (x=0;x<w;x++) for (y=ymax;y<h;y++) newpixels[y*w+x]=255;
        for (x=0;x<xmin;x++) for (y=0;y<h;y++) newpixels[y*w+x]=255;
        for (x=xmax;x<w;x++) for (y=0;y<h;y++) newpixels[y*w+x]=255;

        // Loop to filter
        for (x=xmin; x<=xmax; x++) {
            for (y=ymin; y<ymax; y++) {

                int c,
                    s=0,
                    threshold = pattern.threshold;

                for (int i=0; i<size; i++) {
                    int xtest = x + pattern.getXoff(i);
                    int ytest = y + pattern.getYoff(i);
                    int val   = pattern.getNum(i);

                    if ((pixels[ytest*w+xtest] & 0x000000ff) < threshold) s++;
                }
                if (s == size) c = 0; else c = 255;

                //newpixels[y*w+x] = c;
                newpixels[y*w+x] = 0xff000000 | c * 65536 | c * 256 | c ;
            }

        }
    }
}
```

11.11 CLASS PATTERNPARSER

```
// ************************************************************************
// Pattern File Parser                                    Wandy Sae-Tan Spring 1998
//
// A pattern file describes the filter. For example:

// row 5
// col 5
// targetrow 3
// targetcol 3   note: target is the pixel to be filtered
// 0 0 2 0 0
// 0 0 0 0 0
// 2 0 0 0 2
// 0 0 0 0 0
// 0 0 2 0 0

// The parser reads the file and turns it into a
// list of values and "offset from target" to expedite
// image filtering. For the above file we would get:

// (  2, (-2, 0)
//    2, ( 0,-2)
//    2, ( 0, 2)
//    2, ( 2, 0)  )

// File format MUST rigidly follow the example above.
// i.e. row, col, targetrow, targetcol, pattern matrix
// Do not insert empty lines.
// ************************************************************************

import java.applet.*;
import java.awt.*;
import java.util.*;
import java.awt.image.*;
import java.io.*;
import Pattern;

public class PatternParser {

    // ************************************************************************
    // P U B L I C   M E T H O D S
    // ************************************************************************

    // Constructor
    // ---------------------------------
    public PatternParser () {}


    // Parser
    // ---------------------------------
    public Pattern parseFile (File f) throws Exception {

        // Open File and Tokenizer
        FileInputStream instream = new FileInputStream (f);
        StreamTokenizer token = new StreamTokenizer(instream);


        // Read the first four arguments
        int tok, num, row, col, targetrow, targetcol, threshold;
        int i=0, j=0, offcol, offrow;

        try {
            row = readArg(new String("row"), token);
            col = readArg(new String("col"), token);
            targetrow = readArg(new String("targetrow"), token);
            targetcol = readArg(new String("targetcol"), token);
            threshold = readArg(new String("threshold"), token);
```

```
        }
        catch (Exception e) {
            throw e;
        }

        // Read the rest of file
        Pattern pattern = new Pattern(row, col, targetrow, targetcol, threshold);
        token.eolIsSignificant(true);

        while ((tok = token.nextToken()) != token.TT_EOF) {

            if (tok == token.TT_EOL) {
                i = 1; j++;
            } else {
                num = (int)token.nval;
                if (num > 0) {
                    offcol = i - targetcol;
                    offrow = j - targetrow;
                    pattern.addEntry(num, offcol, offrow);
                }
                i++;
            }
        }

        pattern.freeze();
        pattern.print();
        return pattern;
    }

    // ************************************************************************
    // P R I V A T E   M E T H O D S
    // ************************************************************************

    private int readArg(String s, StreamTokenizer token) throws Exception {

        if ((token.TT_WORD == token.nextToken()) &&
            (token.sval.equalsIgnoreCase(s))) {

            token.nextToken();
            return (int)token.nval;

        } else
            throw new Exception("APP ERROR! error reading number of " + s);
    }

}
```

11.12 CLASS GRIDGENERATOR

```
// ********************************************************************************
// GridGenerator                                          Wandy Sae-Tan Spring 1998
//
// This program draws grid on the screen.
//
// Grid is adjusted by the following parameters:
// int m_Width        Grid Width
// int m_Thickness    Grid Thickness
// int screenheight   Screen Height
// int screenwidth    Screen Width
//
// Program responds to the following keystrokes:
// 'g'  grid on / grid off
// 's'  sync on / sync off
// '1'  make squares wider
// '2'  make squares narrower
// '9'  make grid lines thicker
// '0'  make grid lines thinner
// mouse click - add sync / remove sync
//
// Can be run as both applet and application (see private boolean m_fStandAlone)
// ********************************************************************************

import java.applet.*;
import java.awt.*;
import java.util.*;
import GridGeneratorFrame;
import Square;

//================================================================================
// Main Class for applet GridGenerator
//
//================================================================================

public class GridGenerator extends Applet
{

    // PUBLIC VARIABLES
    //--------------------------------------------------------------------------
    public boolean grid_on = true,
                   sync_on = false;

    public int  m_Width = 20,
                m_Thickness = 3;

    // PRIVATE VARIABLES
    //--------------------------------------------------------------------------
    public int screenheight = 768,
               screenwidth  = 1024;

    Vector squares = new Vector(0,1);

    // PARAMETER SUPPORT:
    //--------------------------------------------------------------------------
    // Parameter names.  To change a name of a parameter, you need only make
    // a single change.  Simply modify the value of the parameter string below.
    final String PARAM_Width = "Width";
    final String PARAM_Thickness = "Thickness";


    // GridGenerator Paint Handler
    //--------------------------------------------------------------------------
    public void paint(Graphics g)
    {
        // draw the grid if grid_on is set to true
        // grid_on is controlled by user pressing the 'g' key
```

```
        if (grid_on) {

            for (int x=0; x<screenwidth; x=x+m_Width)
                g.fillRect(x,0,m_Thickness, screenheight);
            for (int y=0; y<screenheight; y=y+m_Width)
                g.fillRect(0,y,screenwidth, m_Thickness);
        }

        // draw the sync squares if sync_on in set to true
        // sync on is controlled byuser pressing the 's' key
        if (sync_on) {
            for (int i=0; i<squares.size(); i++) {
                Square s = (Square) squares.elementAt(i);
                s.drawSelf(g, m_Width, m_Thickness);

            }
        }
    }

    // Add Sync Square
    //---------------------------------------------------------------------------
    public void doSquare (int x, int y) {
        Square newsqr = new Square (x, y);

        // make sure this isn't an exiting square
        int i=0, exists = -1;
        while (i<squares.size() && (exists == -1)) {
            if (((Square)squares.elementAt(i)).sameAs(newsqr, m_Width))
                exists = i;
            i++;
        }

        if (exists > -1) // remove square
            squares.removeElementAt(exists);
        else            // add the square
            squares.addElement(newsqr);
    }


    // ************************************************************************
    // Support Code
    // ************************************************************************

    // STANDALONE APPLICATION SUPPORT:
    //      m_fStandAlone will be set to true if applet is run standalone
    //------------------------------------------------------------------------
    private boolean m_fStandAlone = false;

    // STANDALONE APPLICATION SUPPORT
    //  The GetParameter() method is a replacement for the getParameter() method
    // defined by Applet. This method returns the value of the specified parameter;
    // unlike the original getParameter() method, this method works when the applet
    // is run as a standalone application, as well as when run within an HTML page.
    // This method is called by GetParameters().
    //------------------------------------------------------------------------

    String GetParameter(String strName, String args[])
    {

        if (args == null)
"
        {
"
            // Running within an HTML page, so call original getParameter().
"
            return getParameter(strName);
"
        }
"

"
```

```
    // Running as standalone application, so parameter values are obtained from
    // the command line. The user specifies them as follows:
    //
    //  JView GridGenerator param1=<val> param2=<"val with spaces"> ...
    //-------------------------------------------------------------------------
    int    i;
    String strArg  = strName + "=";
    String strValue = null;
    int    nLength  = strArg.length();

    try
    {
        for (i = 0; i < args.length; i++)
        {
            String strParam = args[i].substring(0, nLength);

            if (strArg.equalsIgnoreCase(strParam))
            {
                // Found matching parameter on command line, so extract its value.
                // If in double quotes, remove the quotes.
                //-----------------------------------------------------------------
                strValue = args[i].substring(nLength);
                if (strValue.startsWith("\""))
                {
                    strValue = strValue.substring(1);
                    if (strValue.endsWith("\""))
                        strValue = strValue.substring(0, strValue.length() - 1);
                }
                break;
            }
        }
    }
    catch (Exception e){}

    return strValue;
}

// STANDALONE APPLICATION SUPPORT
//  The GetParameters() method retrieves the values of each of the applet's
// parameters and stores them in variables. This method works both when the
// applet is run as a standalone application and when it's run within an HTML
// page.  When the applet is run as a standalone application, this method is
// called by the main() method, which passes it the command-line arguments.
// When the applet is run within an HTML page, this method is called by the
// init() method with args == null.
//--------------------------------------------------------------------------
void GetParameters(String args[])
{
    // Query values of all Parameters
    //---------------------------------------------------------------
    String param;

    // Width: Parameter description
    //---------------------------------------------------------------
    param = GetParameter(PARAM_Width, args);
    if (param != null)
        m_Width = Integer.parseInt(param);

    // Thickness: Parameter description
    //---------------------------------------------------------------
    param = GetParameter(PARAM_Thickness, args);
    if (param != null)
        m_Thickness = Integer.parseInt(param);

}

// STANDALONE APPLICATION SUPPORT
//  The main() method acts as the applet's entry point when it is run
//  as a standalone application. It is ignored if the applet is run from
//  within an HTML page.
//--------------------------------------------------------------------------
```

```
    public static void main(String args[])
    {

        // The following code starts the applet running within the frame window.
        // It also calls GetParameters() to retrieve parameter values from the
        // command line, and sets m_fStandAlone to true to prevent init() from
        // trying to get them from the HTML page.
        //--------------------------------------------------------------------
        GridGenerator applet_GridGenerator = new GridGenerator();

        // Create Toplevel Window to contain applet GridGenerator
        //--------------------------------------------------------------------
        GridGeneratorFrame frame = new GridGeneratorFrame("GridGenerator",
applet_GridGenerator);

        // Must show Frame before we size it so insets() will return valid values
        //--------------------------------------------------------------------
        frame.show();
        frame.hide();
        frame.resize(frame.insets().left + frame.insets().right  + 320,
                     frame.insets().top  + frame.insets().bottom + 240);

        frame.add("Center", applet_GridGenerator);
        applet_GridGenerator.m_fStandAlone = true;
        applet_GridGenerator.GetParameters(args);
        applet_GridGenerator.init();
        applet_GridGenerator.start();
        frame.show();
    }

    // APPLET INFO SUPPORT:
    //--------------------------------------------------------------------
    public String getAppletInfo()
    {
        return "Name: GridGenerator\r\n" +
               "Author: Wandy Sae-Tan\r\n" +
               "Created with Microsoft Visual J++ Version 1.1";
    }

    // PARAMETER SUPPORT
    //--------------------------------------------------------------------
    public String[][] getParameterInfo()
    {
        String[][] info =
        {
            { PARAM_Width, "int", "Parameter description" },
            { PARAM_Thickness, "int", "Parameter description" },
        };
        return info;
    }

    // INITIALIZATOIN
    //--------------------------------------------------------------------
    public void init()
    {
        if (!m_fStandAlone)
            GetParameters(null);
        requestFocus();
        resize(320, 240);
    }

    // CONSTRUCTOR
    //--------------------------------------------------------------------
    public GridGenerator () {
    }


}
```

11.13 CLASS GRIDGENERATORFRAME

```
// ****************************************************************************
// GridGeneratorFrame                                    Wandy Sae-Tan Spring 1998
// ****************************************************************************


import java.awt.*;

//============================================================================
// STANDALONE APPLICATION SUPPORT
//  This frame class acts as a top-level window in which the applet appears
//  when it's run as a standalone application.
//============================================================================

class GridGeneratorFrame extends Frame
{
    protected boolean grid_on = true;
    protected GridGenerator generator;

    // GridGeneratorFrame constructor
    //--------------------------------------------------------------------------
    public GridGeneratorFrame(String str, GridGenerator g)
    {
        super (str);
        generator = g;
    }

    // The handleEvent() method receives all events generated within the frame
    // window. It allows the user to interactively control the generator
    //--------------------------------------------------------------------------
    public boolean handleEvent(Event evt)
    {
        switch (evt.id)
        {

            case Event.MOUSE_DOWN:
                generator.doSquare(evt.x, evt.y);
                generator.sync_on = true;
                generator.repaint();
                return true;

            case Event.KEY_PRESS:

                switch (evt.key)
                {

                    // Turn on or Turn off Grid
                    // ---------------------------------------------------------
                    case 'g':
                        if (generator.grid_on)
                            generator.grid_on = false;
                        else generator.grid_on = true;
                        generator.repaint();
                        return true;

                    // Turn on or Turn off Sync Squares
                    // ---------------------------------------------------------
                    case 's':
                        if (generator.sync_on)
                            generator.sync_on = false;
                        else generator.sync_on = true;
                        generator.repaint();
                        return true;

                    // Make Grid Wider
                    // ---------------------------------------------------------
                    case '1':
                        if (generator.m_Width < generator.screenwidth &&
```

```
                        generator.m_Width < generator.screenheight)
                        generator.m_Width++;
                    generator.repaint();
                    println("Width = " + generator.m_Width);
                    return true;

                // Make Grid Narrower
                // ------------------------------------------------------
                case '2':
                    if (generator.m_Width > 1)
                        generator.m_Width--;
                    generator.repaint();
                    println("Width = " + generator.m_Width);
                    return true;

                // Make Line Fater
                // ------------------------------------------------------
                case '9':
                    if (generator.m_Thickness < generator.m_Width)
                        generator.m_Thickness++;
                    generator.repaint();
                    println("Thickness = " + generator.m_Thickness);
                    return true;

                // Exit Line Thinner
                // ------------------------------------------------------
                case '0':
                    if (generator.m_Thickness > 1)
                        generator.m_Thickness--;
                    generator.repaint();
                    println("Thickness = " + generator.m_Thickness);
                    return true;

                // Exit Program
                // ------------------------------------------------------
                case 'x':
                    dispose();
                    System.exit(0);
                    return true;
            }

        // Application shutdown (e.g. user chooses Close from the system menu).
        //----------------------------------------------------------------
        case Event.WINDOW_DESTROY:
            dispose();
            System.exit(0);
            return true;

        default:
            return super.handleEvent(evt);
    }
}

// SYSTEM PRINT UTIL
// ---------------------------------------------------------------------------
public void println(Object o) {
    System.out.println(o);
}
}
```

11.14 CLASS SQUARE

```
// ********************************************************************************
"// Square                                          Wandy Sae-Tan Spring 1998
"//
"// A very simple class for representing synchronization squares
"//
"// ********************************************************************************
"
"import java.awt.*;
"
"public class Square {
"
"
"   // x and y coordinate of the square
"   //--------------------------------------------------------------------------
"   public int x=-1,
"              y=-1;
"
"
"   // Check if another square is equal to itself
"   //--------------------------------------------------------------------------
"   public Square (int x, int y) {
"       this.x = x;
"       this.y = y;
"   }
"
"   // Check if another square is equal to itself
"   //--------------------------------------------------------------------------
"   public boolean sameAs(Square s, int width) {
"       return (s.x/width == x/width && s.y/width == y/width);
"   }
"
"   // Draw itself
"   //--------------------------------------------------------------------------
"   public void drawSelf(Graphics g, int width, int thickness) {
"        g.fillRect((x/width)*width+thickness+1, (y/width-1)*width+thickness+1,
"                   width-thickness-2, width-thickness-2);
"   }
"
"}
"
"
"
```