Obstacle Detection for Robot Navigation

Using Structured Light

by

Deborah L. Tran

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 23, 2001

Author _________

Department of Electrical Engineering and Computer Science
May 23, 2001

Certified by ___

Kenneth M. Houston
Principal Member of Technical Staff, Charles Stark Draper Laboratory
Thesis Supervisor

Certified by

Leslie Pack Kaelbling
Professor of Computer Science and Engineering, Massachusetts Institute of Technology
Thesis Supervisor

Accepted by_________

Arthur C. Smith
Chairman, Department Committee on Graduate Theses

[This page intentionally left blank]

Obstacle Detection for Robot Navigation Using Structured Light

by
Deborah L. Tran

Submitted to the
Department of Electrical Engineering and Computer Science

May 23, 2001

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

ABSTRACT

Obstacle detection is an essential function for autonomous mobile robots. Current autonomous systems are generally heavy, high-powered devices. Additionally, obstacle detection systems often employ active ranging sensors that exhibit poor angular resolution, thereby preventing vehicles from navigating through narrow spaces. Other systems require considerable processing to resolve objects. The work in this thesis attempts to design a small, lightweight, low-cost, and low-power system to detect obstacles in the direct field of view of a small robotic vehicle. In particular, the range to the obstacle as well as shape estimation is key data that would aid in robot navigation as well as data gathering. Rather than using conventional active or passive ranging techniques, this thesis examines projections of structured light to determine range to the obstacle and surface information of the obstruction. A physical model was developed and tested through simulation, and verified in hardware.

Thesis Supervisor: Kenneth M. Houston
Title: Principal Member of Technical Staff

Thesis Advisor: Leslie Pack Kaelbling
Title: Professor of Computer Science and Engineering

[This page intentionally left blank]

ACKNOWLEDGMENTS

This thesis was prepared at The Charles Stark Draper Laboratory, Inc., under Independent Research and Development Project No. 3045.

Publication of this thesis does not constitute approval by Draper or the sponsoring agency of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

(author's signature)

I would like to thank Robert Larsen and Kenneth Houston for their ideas, guidance, and support without which this project would not be possible.

I would also like to express my appreciation to Prof. Leslie Pack Kaelbling for her expertise, insights, and efforts towards my research.

In addition, I wish to acknowledge Fred Boelitz for his help on the environmental simulation aspects of my project as well as Mark Little for his advice concerning hardware implementation details.

Finally, I would like to thank my parents for all of their support and encouragement.

[This page intentionally left blank]

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

[This page intentionally left blank]

1 Introduction

Autonomous mobile robotic vehicles are often used to gather data over a wide range of environments for a variety of purposes. An essential function of these vehicles is the ability to detect obstacles as well as estimate range to potential obstructions in order to prevent a possible collision.

In order to identify potential obstacles, a variety of sensors and ranging techniques can be employed. However, the sensors currently used by existing ranging systems possess some or all of the following negative properties: large size, high power consumption, and low performance in specific conditions. The two types of traditional ranging techniques, active and passive sensing, either consume a large amount of power, or involve computationally intensive processing.

These characteristics are especially problematic for small robotic systems, whose size, weight, and power constraints are very limited. There are few practical ranging systems that currently exist that can achieve high accuracy, low power, and a wide, dynamic range within a small footprint.

Therefore, we would like to develop a system that is primarily small in size. A key part in the solution is to decrease power consumption, since power generators are generally the heaviest and largest part of an autonomous robot. Additionally, to reduce the power consumption of the processing unit, we would like to have simple functionality. Third, as with any product, we would like to have the robotic system be low cost. Finally, because this robot will experience non-ideal and rugged environments, the number of mechanical components should be minimal, so that the chance of failure is decreased.

Furthermore, the shape of an obstacle is important for smaller robots, which approach objects at closer distances than larger vehicles. Therefore, it would be advantageous if our technique could estimate the contour of potential obstacles.

In the following sections, we will discuss potential methods for sensing obstacles as well as gathering data from objects, and the types of sensors that are available. We will also propose a solution for a small mobile robotic range sensor.

1.1 Sensors

In order for an autonomous vehicle to self-navigate, it needs to be aware of its environment. A range of sensors can be used to inform the vehicle of its surroundings. Current methods of obstacle detection entail a variety of sensors, including ultrasonic sonar, ultra-wideband, infrared waves, millimeter waves, and lasers.

1.1.1 Ultrasonic Sonar

Ultrasonic waves are vibrations whose frequencies exceed the human auditory limit, which falls around 20 kHz. To detect obstacles, autonomous robots often transmit ultrasonic waves towards a surface and record its return time. This round-trip time, along with knowledge of the wave's velocity, is used to calculate the distance between the object and the robot. Ultrasonic sonar sensors are prevalent in many robots because of its low cost, ease of use, and relatively small footprint. At close ranges it can provide very reliable range information.

However, at long distances, the ultrasonic signal attenuates significantly. Additionally, the speed at which the wave travels is relatively slow at roughly 330 m/s, therefore systems using ultrasonic sensors are delayed by the reflected response. Another disadvantage of ultrasonic sonar is the wide-angle beam of sonar devices. This creates ambiguity as to the precise location of the object in the field of view. The use of the ultrasonic sonar is limited to certain areas due to the reflectivity and absorption of surfaces since certain surfaces absorb ultrasonic waves, while others reflect ultrasonic waves to a high degree. Due to the low frequency and long wavelength of ultrasonic waves, some smooth surfaces appear as a mirror to ultrasonic waves, and therefore appear invisible to detectors. Ultrasonic waves are also affected by changes in temperature.

Often, multiple responses can result from a lone transmitted signal. This behavior is due to echoes from the original signal that strike the obstacle at different locations. Ultrasonic time-of-flight sensors generally respond to the first reflected response from an object. This introduces a disadvantage, in that any other echoes from its sent signal is ignored. These echoes contain valuable information such as shape and motion. Therefore, one cannot extract additional data from the obstacle being detected.

A common ultrasonic sensor used for mobile robot navigation is the Polaroid 6500 series transducer. The specifications are outlined in Table 1.

Table 1 Specifications for the Polaroid 6500 Series Transducer

Cost	$50
Range	6" – 35 ft
Accuracy	± 1%
Response Time	2.38 ms
Power	0.45 - 9 W
Dimensions	1.69" (dia) x 0.328" (deep)

1.1.2 Infrared

The method of detecting objects using non-collimated infrared waves is similar to ultrasonic methods. The infrared range is usually divided into three regions: near infrared (nearest the visible spectrum), with wavelengths 0.78 to about 2.5 μm; middle infrared, with wavelengths 2.5 to about 50 μm; and far infrared, with wavelengths 50 to 1,000 μm. To calculate the proximity of an object, infrared waves are directed towards the object, and the robot computes range based on the elapsed time between the transmission and reflection of the beams.

The advantages of using infrared are also similar to ultrasonic, in that they are affordable and small. Infrared range finders are also impervious to the colors of obstacles since their waves fall outside the visible electromagnetic spectrum. Infrared signals also travel faster than ultrasonic waves, thereby enabling more measurements within a specific time frame. Infrared rangefingers tend to be more reliable than sonar as well. The disadvantages of infrared waves are its susceptibility to atmospheric conditions such as fog, as well as smoke and dust. Also, the range of infrared rangefinders tend to be shorter than ultrasonic sonar.

Sharp, Inc. offers a rangefinding module that uses an infrared diode and a position sensitive detector. Its features are provided in Table 2.

Table 2 Specifications for the Sharp GP2D12

Cost	$20
Range	10 cm – 80 cm
Resolution	± 0.5 cm
Response Time	38 ms
Power	96.8 mW
Dimensions	44.5 mm x 18.9mm x 13.5 mm

1.1.3 Millimeter-Wave Radar

Millimeter waves can be used in time-of-flight techniques as well. Millimeter waves are not as sensitive to atmospheric conditions as infrared waves. Their wavelengths range from 500 micrometers to 1 cm. Since these wavelengths are shorter than ultrasound waves, the response times of systems utilizing millimeter waves are faster than ultrasonic sonar.

Fujitsu Ten, Ltd. develops millimeter-wave collision warning systems for vehicles. The properties of such a system is listed in Table 3.

Table 3 Specifications for Fujitsu 76Ghz Millimeter-Wave Radar

Cost	$10,000 (pre-production)
Range	5-120m
Resolution	± 5%
Response Time	100ms
Power	14.4 W
Dimensions	106mm x 88mm x 90mm

1.1.4 Ultra-Wideband Radar

Another method for sensing objects is ultra-wideband (UWB) radar. Whereas most radar systems emit sine-wave bursts at a specific frequency, the signals from UWB radar systems are complex and span a wide range of frequencies. For UWB systems, this range of frequencies is defined to cover at least a 25% bandwidth. The duration of each pulse generally lasts less than a billionth of a second.

Due to this short lifetime, power consumption is greatly reduced. Since the frequency span of UWB radar is large, the resulting resolution from UWB echoes is greatly improved. Also, due to the variety of frequencies being used, more types of surfaces may be detected, resulting in a higher probability of sensing obstacles. A unique property of UWB is its ability to penetrate foliage and below ground surface. Additionally, the probability of UWB waves being detected is lower than with narrowband signals.

However, the increase in bandwidth results in greater processing demands as well as an increased chance of false alarms. Wide signal bandwidths also result in high noise levels. The physical size of UWB sensors is also relatively large compared to ultrasonic sensors. The complexity of UWB signals makes it more difficult to use than ultrasonic sonar. Furthermore, UWB radar systems are less readily available than ultrasonic and radar systems due to FCC regulations.

The specifications for an ultra-wideband radar developed for the Naval Surface War Center by Multispectral Solutions, Inc. is given in Table 4.

Table 4 Specifications for a UWB radar by Multispectral

Cost	\$1000-\$2000
Range	Up to several hundred feet
Resolution	Less than one foot
Response Time	10 kHz
Power	25 μW(average)
Dimensions	150 cubic inches

1.1.5 Lasers

Lasers can be used as range finding devices similar to ultrasonic rangefinders. Since the beam from a laser is extremely focused, it does not exhibit the wide-beam dispersion of ultrasonic sonar, therefore, the readings tend to be superior in range determination and angular resolution. In addition, because the energy from the laser does not dissipate as much as ultrasonic at further distances, laser sensing can function over a much greater range. Third, because laser pulses travel at the speed of light, the range readings return much faster, therefore, the responsiveness of laser range finding devices is quicker.

However, the fact that laser pulses travel so fast is also a disadvantage. Using time-of-flight ranging, one can measure the time between sending and receiving a signal. However, because of the high velocity of laser pulses, the processing speed on the receiver side needs to also be fast at near distances. Therefore, laser range finding does not work very well at close ranges. Also, laser rangefinders available today often consume high power and are physically large, especially those that incorporate a scanning head that rotates in order to sweep a beam across large areas. In addition, some lasers are not eye-safe, and thus must be limited to areas of low human occupancy. Furthermore, laser waves are more affected by atmospheric conditions than ultrasonic waves.

Sick Optic-Electronic, Inc. designs various laser range finding modules that are often used for robotics. The properties of the smallest of its indoors laser range finding devices are presented in Table 5.

Table 5 Specifications for the LMS-200

Cost	$5750
Range	Up to 30 m
Resolution	10 mm
Error	±15 mm within 1 to 8m ±4 cm between 8 to 20 m
Response Time	56/26/13ms based on angular resolution
Power	17.5 W (maximum)
Dimensions	155 mm x 185 mm x 156 mm

1.2 Techniques for Sensing

The techniques used to detect obstacles can be divided into two categories: active sensing and passive sensing. Active sensing generally involves a transmission of energy, such as acoustic or electromagnetic waves which returns reflections from an obstacle to a sensor, while passive sensing generally uses the already illuminated scene to detect depth.

Active sensing techniques include time-of-flight ranging and use of structured light while passive sensing uses image-processing techniques that analyze texture, linear perspective, motion, focus, occlusion and similar aspects.

Because passive sensing procedures generally require much computation and processing, passive sensing cannot perform well in real-time environments. The power and equipment needed for such processing would also contribute significantly to the load of the robotic system. Therefore, an active sensing scheme would be more favorable.

One common active ranging method is time-of-flight (TOF) ranging. Sensors employing TOF techniques calculate distance by determining the elapsed time between transmitted signals and received signals, and multiplying this time by the velocity of the signal. However, using TOF at very close ranges with fast signals yield poor accuracy, since the speed at which the signal can be processed is limited by the processor speed.

Another active method is structured light. Structured light is a technique for recovering shape by extrapolating three-dimensional meshes from rays of projected light and a camera. The typical set-up for structured light range finding is shown in Figure 1.

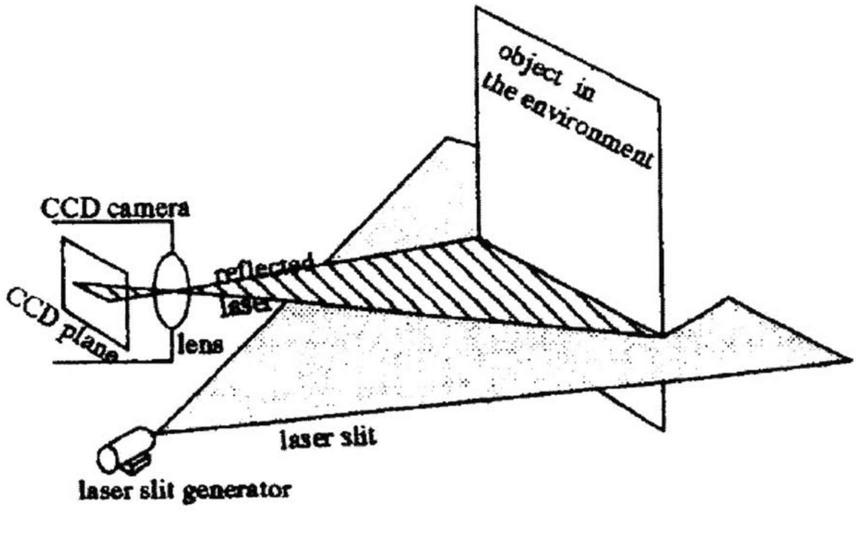


Figure 1 Setup for Structured Light Sensing (Figure reproduced from [15] by permission, © 1991 IEEE.)

A camera is oriented down towards the object, while a laser is directed such that its beam intersects the object. To determine the distance to the object, triangulation is used between the camera, the object, and the laser. Patterns of projected light can be achieved by sending the laser beam through a slit or grid filter.

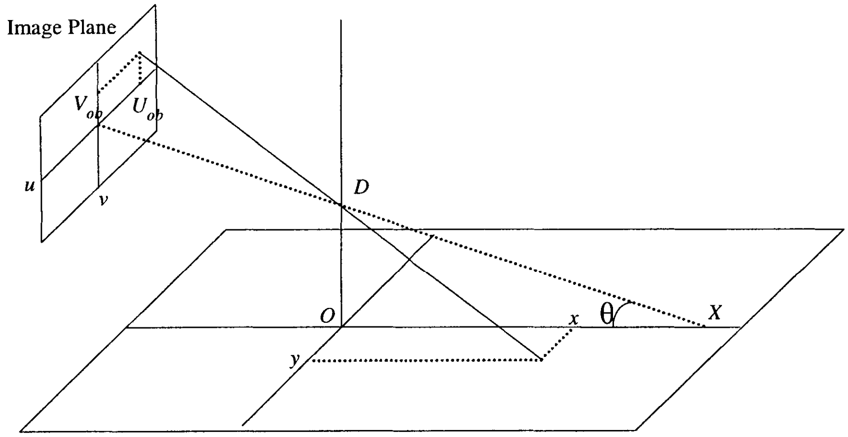


Figure 2 Diagram for triangulation calculations. The camera lens is modeled as a pinhole. y is the range to the obstacle. (Figure adapted from De la Escalera, Moreno, Salichs, and Armingol, 1996.)

Few components are required, reducing overall costs. In addition, high accuracy at close ranges can be achieved. Therefore, the use of structured light is a potentially viable solution.

1.3 Previous work in Structured Light

Structured light is often used for object recognition, 3-D reconstruction, as well as obstacle detection. In the next section, we will discuss past work performed in this field that provide valuable information to solve our problem.

1.3.1 PRIME

PRIME (PRofile Imaging ModulE) is a structured light sensor developed by DePiero and Trivedi (1996) for three dimensional computer vision. Using a downward pointing camera, and a laser fitted with a slit filter, one can capture the image of the beam striking the object. From this image, one can determine range and shape for the slice of the object that gets intercepted by the beam. To recreate the 3D shape of the object, the object is incrementally moved via a conveyor belt so that the beam can ultimately sweep all planes of the object's surface and generate a 3D model of the object.

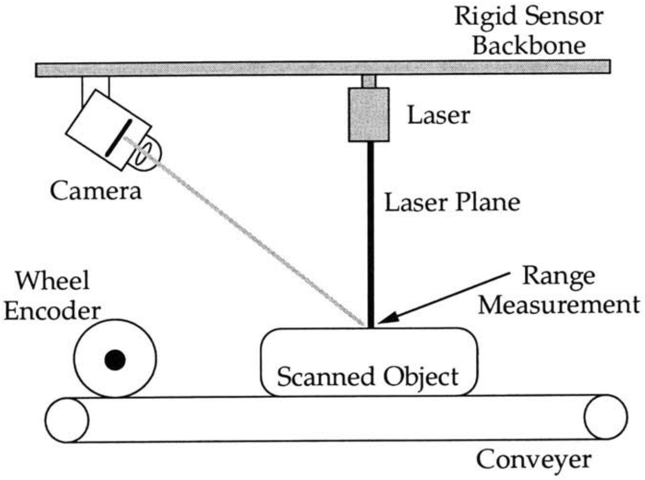


Figure 3 PRIME Setup. (Figure reproduced from [5] by permission of the authors.)

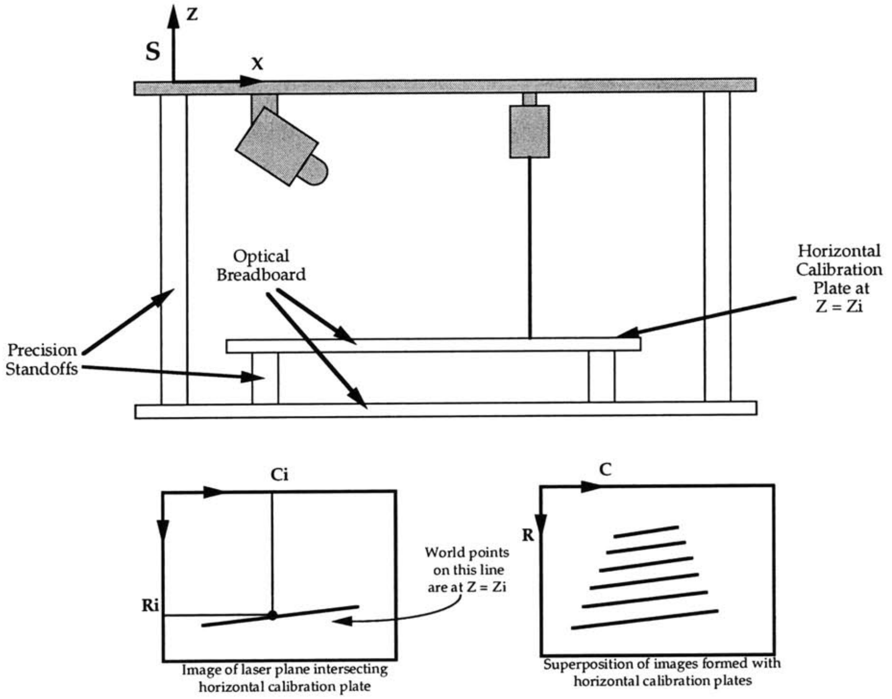

Z
S
X
Optical Breadboard
Horizontal Calibration Plate at Z = Zi
Precision Standoffs
Ci
Ri
World points on this line are at Z = Zi
C
R
Image of laser plane intersecting horizontal calibration plate
Superposition of images formed with horizontal calibration plates

Figure 4 Calibration Setup for PRIME. (Figure reproduced from [5] by permission of the authors.)

Correct calibration, therefore, is vital to accurately reproducing the object's model. Each movement of the object needs to be known precisely so that calculations use the correct increment for range determination. PRIME uses complex calibration models for reliable measurements.

1.3.2 Multiplexing Light Sources

Since the object in the previous project needs to be incremented deliberately, the amount of time required to scan an object can become long. Baba and Konishi (1999) attempt to reduce this time by using a multiplexed structured light source.

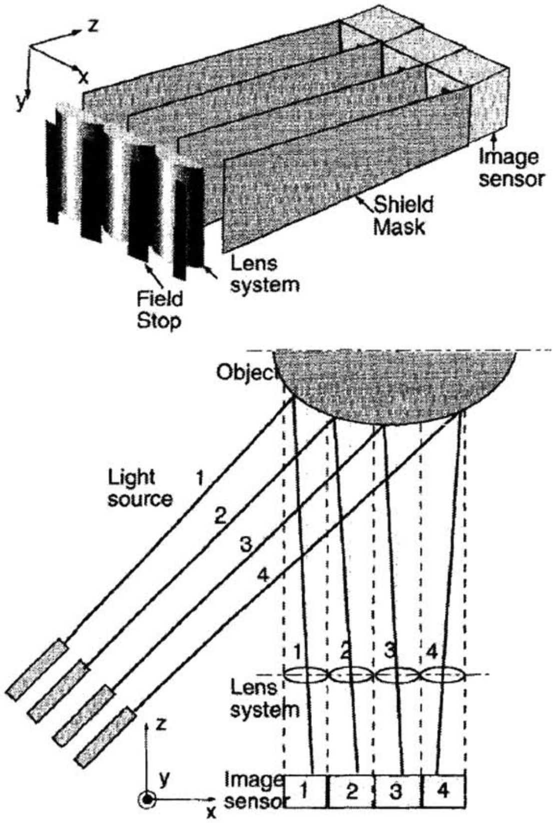


Figure 5 Multiplexed Light Source Setup. (Figure reproduced from [1] by permission of the authors.)

Previous work using multiple light stripes were complicated because each light stripe needed to be tagged in order to determine which stripe corresponded to which source. The conventional method encodes each stripe with a gray code, and then sequentially projects each gray-coded pattern. However, this requires several projections per sample.

Baba and Konishi (1999) instead use a system of lenses that direct each light stripe to a specific image sensor. Therefore, identifying the stripe is easy and only one projection needs to be performed per sample. The result is a faster imaging system. Calibration, however, is still necessary.

1.3.3 Visually Located, Structured Light Source

A structured-light range finder developed by Fisher, Ashbrook, Robertson, and Werghi (1999) attempts to remove the difficulties of calibration by using a system that does not require prior knowledge of the distance between the light source and the light sensor (camera) or the distance between the camera and the surface. Instead of projecting light rays onto an object, it uses an obstruction to cast a shadow on part of the object. The line of difference between the lighted section and shadowed section of the object is similar to the line created when a laser beam intercepts the object. The obstruction also contains marks that, when captured by the camera, can provide range information.

The obstruction used is a long, marked opaque triangular prism. One face of the prism is painted black and is pointed away from the camera. Another face of the prism is painted white, and this is presented toward the camera. Also on this face are two dark marks. These marks are two parallel bands whose thickness and separation are known. The image captured partially includes both faces of the prism as well as the shadow it casts on the object. The amount of the white face showing versus the amount of the black face showing determines the angle that the wand is rotated towards the camera.

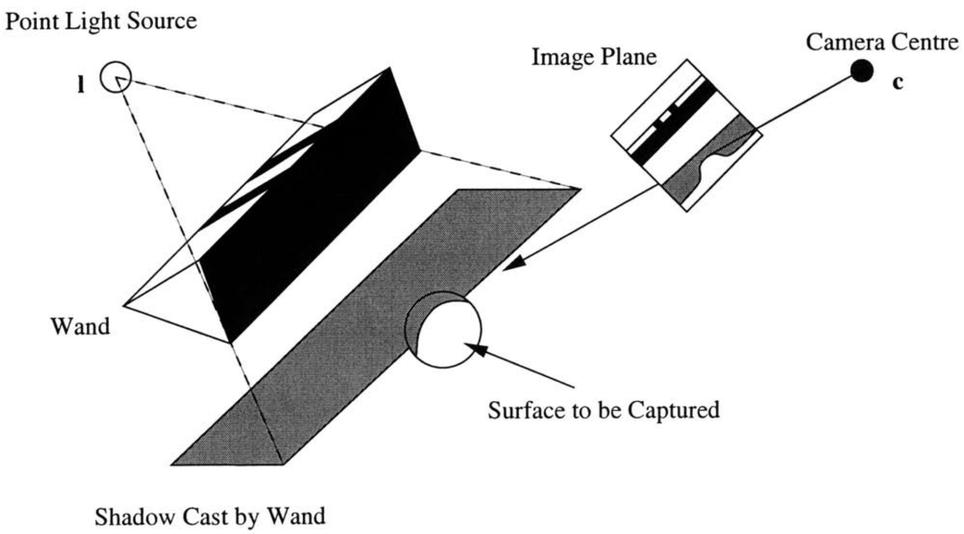


Figure 6 Visually Located, Structured Light Source. (Figure reproduced from [7] by permission of the authors.)

Additionally, since the distance between the bands is known, one can calculate how far away the prism is from the camera by the distance between the bands in the image. From this, we derive the position of the prism. Finally, depth can be determined from the position of the object in the image, the pose of the prism, and the position of the light source.

1.3.4 Mars Rover

Structured light has also been used for obstacle detection during robot navigation. Matthies, Balch, and Wilcox (1997) of the Jet Propulsion Laboratory developed such a system for planetary rovers. Their basic arrangement consists of a camera angled towards the ground and a laser beam that is also directed downwards. The laser beam is diffused into 15 co-planar beams using a diffraction grating. The resulting image captured by the camera contains a line of 15 laser spots. If one or more of the beams strikes an object, their position in the image along the line would thus be shifted, and the shifted spots would therefore supply information about the obstacle. The distance to the obstacle can be computed by using the expected position of a spot and its offset.

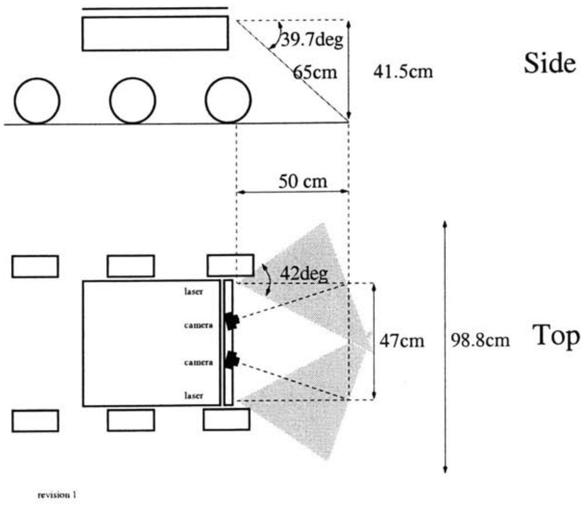


Figure 7 Multiple Spot Laser Setup. (Figure reproduced from [10] by permission of the authors.)

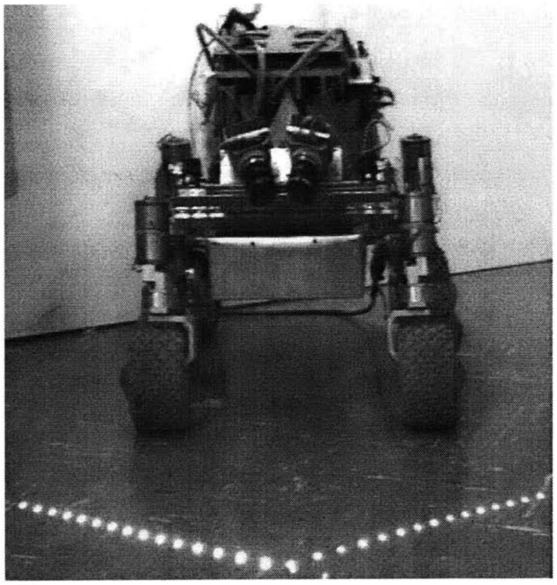


Figure 8 Front View of Rover. (Figure reproduced from [10] by permission of the authors.)

In order to differentiate the 15 spots from the scene, Matthies et al. (1997) align the camera and the laser such that the line of spots that appears in the camera's image will coincide with the optical centers of the camera. Therefore, the line of spots will always lie on the vertical center line of the image. The 15 spots would always be co-planar since spots will only shift left or right upon contact with an obstacle. In case the line of spots moves, the system need only find the scanline on which the spots lie. Since the spots will only appear on a specific scanline, image processing becomes greatly reduced.

1.4 Sensor Comparison

As discussed in Section 1.1, ultrasonic sensors yield poor angular resolution. Therefore, when using such sensors for robot navigation, the location of specific points on the object cannot be determined accurately. A potential consequence is the misdirection of the robot. For example, consider the case of a small robot approaching a chair. Because of the size of the robot, it may proceed through the legs of the chair.

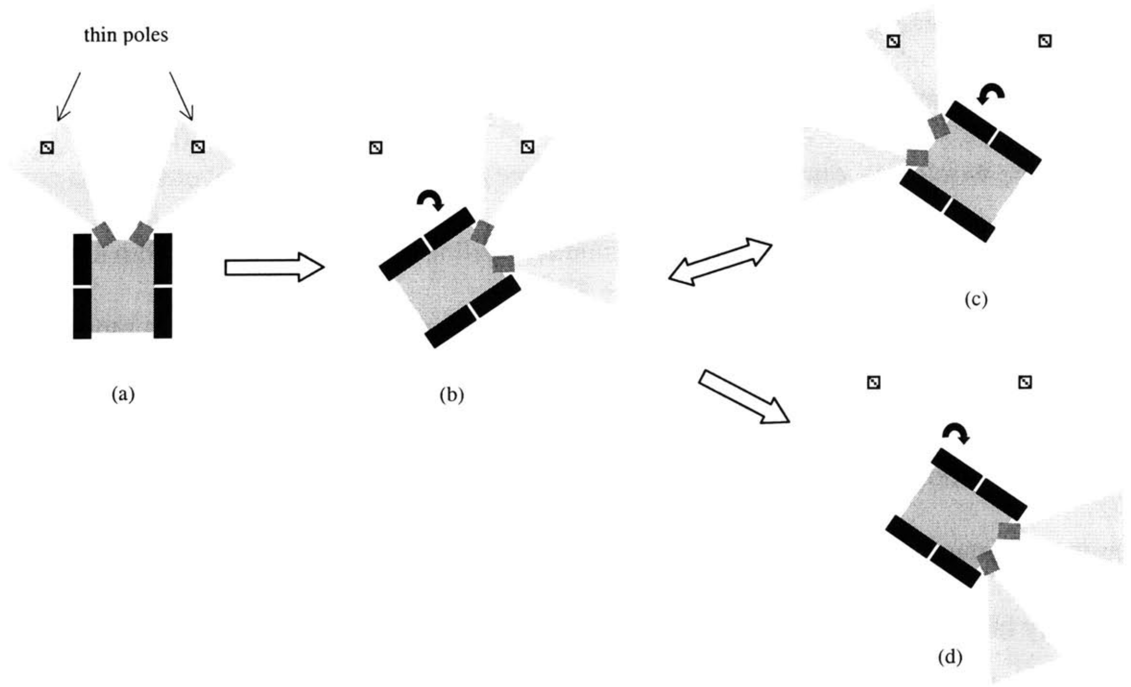


Figure 9 Thin-Pole Obstacles. (a) Original starting position (b) Rotation after sensing object (c) One possible result after sensing object. (d) Second possible outcome from sensing.

However, if a wide-beam sensor detects a leg of the chair, it cannot ascertain the precise size and location of this obstacle. Therefore, it will advise the robot to maneuver away from the chair. In the worst case, the robot may position itself such that the sensor encounters the other leg of the chair. Depending on the obstacle avoidance algorithm used, two interesting cases may result. It may reorient itself to advance in an opposite direction to the original path, which is an undesired result. In the second case, it may end up rotating back to the previous orientation, thereby sensing the first leg again. If this occurs, it will persistently alter between facing the first and the second leg, and would thus never progress from its current position.

If the robot was equipped with a sensor that exhibited a narrower beam, the robot could proceed through the thin poles without having to deviate from the original course.

Also, accurately determining the shape of an obstacle is difficult with wide-beam systems, such as ultrasonic and UWB. Distinct points on a surface cannot be distinguished since such sensors will only detect the closest point of all points being swept by the beam. The set of points covered is large due to the width of the beam.

We would like our robot to take advantage of its size and maneuver through small passages. In addition, since our robot is small, it will approach obstacles within shorter distances than larger robots. Consequently, the shape of an obstacle becomes more important. Therefore, ultrasonic and UWB sensors are not practical as they cannot detect shape and cannot approach obstacles in close proximity. A laser based system, which emits a narrow beam, would then be the best solution for our purposes. Because a laser will only hit points that it will ultimately contact, our robot would not be falsely warned of potential obstacles. Since a laser's beam divergence is very small, even points at further distances can be targeted precisely. The beam's intensity would not attenuate significantly as well. Control of the laser is also simple – lasers can be switched on and off with low or high TTL levels.

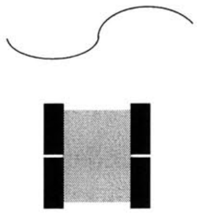


Figure 10 Importance of Shape

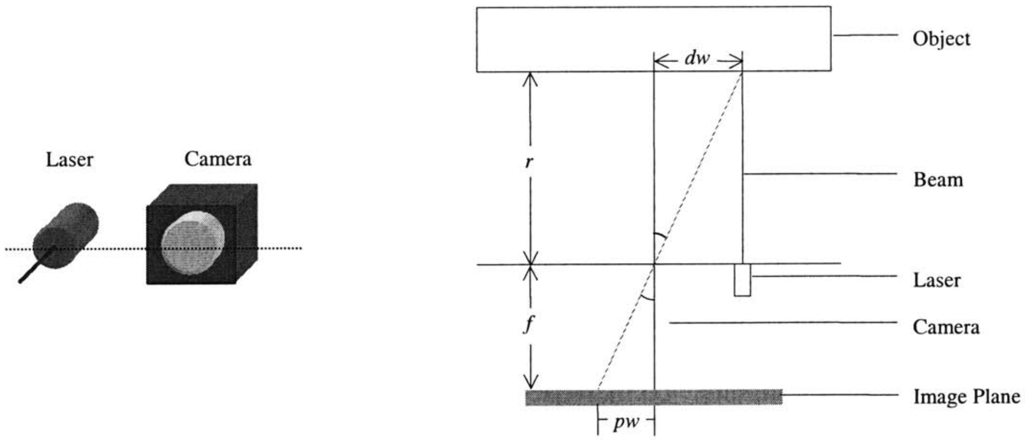
1.5 Technique Comparison

In the structured light methods discussed in Section 1.3, the camera is always positioned above the object being observed. However, because our robot is small, the top of obstacles will generally be above the plane of view of the camera. The rough distance between the object and the camera also changes, unlike the fixed structure systems in traditional methods. Calculations used in these conventional methods may also become complex. Therefore, a different method needs to be used.

1.6 Coplanar Multiple Source Structured Light Ranging

We propose to reduce complexity in the calculations by functioning in only one plane, rather than 3-dimensional space as done in previous work. Therefore, the camera will be placed on the same axis as the laser. The distance between the laser module and the camera in real space will ideally be related to the distance between the laser spot and the horizontal center in the image plane. The laser spot offset from the center is also correlated to the distance from the laser to the obstacle. Through knowledge of the camera's focal point and sensor array dimensions, and the distance between the camera and the laser, one can use triangulation to calculate the distance to the obstacle.

$$\frac{dw}{r} = \frac{pw}{f}$$



a) Front View

b) Top View

Figure 11 Proposed Structure

The triangulation methods in Section 1.3 detect shape by "striping" or projecting a pattern of rays onto an object. Because these pattern generation techniques disperse a laser beam across wide angles, one loses information about the distance *dw*. In our proposed method, this information is crucial to estimating the distance to the obstacle. Therefore, in order for us to detect shape, we must not alter the original laser beam. Since multiple points are required to determine shape, we may either use mirrors or motors to scan the beam, or use multiple lasers. To avoid movable mechanical components, which add a source of complexity as well as potential mechanical failure to our design, we choose the multiple laser option. As stated previously, available laser modules are inexpensive, low power, and lightweight. Thus, the idea of employing multiple lasers is very realistic. Each reflection from each laser will give distance estimation to a specific target on the obstacle. We can map these points to a function that describes the contour of the object's surface.

Because of the relationship between the range of the obstacle and the distances between the laser(s) and the camera, we can actually control the specifications of our system, specifically the minimum distance required for correct functionality.

$$\textit{minimum distance} = \frac{dw \cdot f}{\tfrac{1}{2}\ \textit{maximum image plane width}}$$

With all other systems previously mentioned, complex, interior physical changes are required by the sensors in order to adjust their distance requirements. With the proposed system, one need only position the lasers closer to the camera for reduced minimum distance requirements. To increase the field of view, one may simply slide the lasers away from the camera or add a lens to the camera.

1.7 Goals

With this design in mind, and considering specifications that other sensors have achieved, we wish to satisfy the following requirements for our system: range from a minimum of 10 cm to a maximum of 5 m, distance acquisition times of 30Hz, total power consumption less than 1 W, weight of at most 1 lb, an accuracy of 1%, and a cost of less than $1000.

2 Design Issues

There arises a multitude of issues with our proposed design. This section will address them. In the next section, we will attempt to solve these problems.

2.1 Textures of Surfaces

The first potential problem with this system involves types of surfaces. If a surface is highly reflective or absorbent, the resulting laser spot cannot be discerned by the camera, and thus, range finding cannot be performed. Surfaces that are highly reflective include mirrored objects. If the target surface is near 100% reflective, then a spot will not appear on the surface due to the specular nature of the reflection. If the reflected beam does strike another object, and appears on the target's surface, this point will appear in the camera's field of view. The resulting point on the image would yield false distance measurements.

If a surface is highly absorbent, then no visible light from the laser will be reflected. Therefore, no spot will show up on the camera's image. Distance cannot be determined in this case as well.

2.2 Color of Surfaces

Another concern with different surfaces is the color of a surface. If the color of a target surface is very close to the wavelength of the laser, then the camera might not be able to differentiate the laser spot from the object. Thus, no laser spot will appear, and distance measurement cannot occur.

A complication may arise when the laser hits an object that alters the color of the reflecting beam. If the system was expecting a certain color for the spot, then the system may overlook spots that reflect a different color. Distance measurements would also be affected in such cases.

2.3 Visibility

If the intensity of the ambient light is greater than that of the laser light, a "wash-out" may occur. Therefore, the camera may not detect the spot. Additionally, if other lasers are present in the environment, and are emitting a beam with a similar wavelength, our system may confuse those spots with our own. Atmospheric conditions such as fog, rain, and smoke, will also affect visibility.

The spot size of the laser beam may affect distance measurements. The further away an object is, the larger the spot size will be. Therefore, we will need to accurately determine the center of the spot in order to accurately determine range.

Another issue involves image quality. Small cameras generally have a sensor array of dimensions 4.8 mm x 3.6 mm. The pixel array consists of 320 to 640 horizontal pixels by 240 to 480 vertical pixels. Therefore, there is much image degradation when small cameras capture the scene. Efforts need to be made to preserve as much information about the scene as possible.

2.4 Arrangement

A different design issue is the number of lasers we choose to use in our system. Only one point is needed to determine distance to the object. However, since one point will provide only a limited field of view, we will need multiple points to ensure that our vehicle does not collide with an object. Therefore, we need to determine a minimal, yet reasonable, number of lasers to locate obstacles a majority of the time. At least two lasers will give us incline information about the surface if the surface is angled. At least three lasers may give us curvature information about the target. Also, it would be wise to include extra lasers in case other lasers fail or cannot detect an obstacle in their limited range. In addition, whether to use an odd or even number of lasers should be considered. An even number of lasers may aid in simplifying calculations due to symmetry.

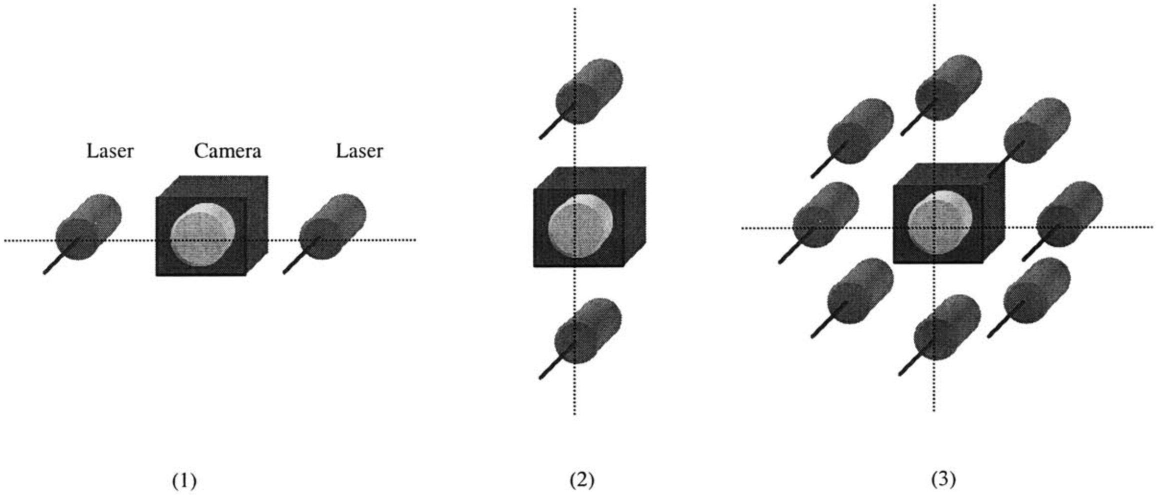


Figure 12 Possible Laser Configurations

Furthermore, we must consider how to arrange the lasers around the camera. Potential configurations included orienting the lasers horizontally or vertically in the same plane as the camera, or even diagonally or in a circle around the camera. In addition, the spacing between lasers should be considered. Lasers can be spaced evenly, or in some manner that maximizes accuracy. Some configurations may yield better results than other configurations.

2.5 Detection

We should consider the case where all lasers miss. If we encounter thin poles that none of the lasers detect, we may end up hitting them. For example in the case of a chair, the vehicle may be absolutely centered and approaching the leg head on. Since the lasers will not detect this leg, a collision will result. We may need to add some auxiliary hardware to prevent such a case.

There also arises the situation where some lasers miss, and some lasers hit. A problem arises as to which laser spot corresponds to which laser, since calculations are dependent on this information. A system for resolving such ambiguities needs to be developed.

2.6 Obstacle Shape

As stated previously, since our robot is small, the shape of an obstacle is important. If the vehicle is approaching an incline, then calculations as to the closest point of the incline could be made if we know the angle of approach. If the vehicle is headed towards a curved object, we can derive the closest point of that object if we know the radius of curvature. With such knowledge, a laser beam is not required to hit the closest point of the object. Therefore, we need to produce algorithms that not only provide shape, but also the polynomial that fits the points of the obstacle.

In some environments, our robot may encounter negotiable obstacles – objects that are permeable or navigable. A vehicle may easily traverse grass and similar lightweight objects. However, such objects will appear as obstacles to the system. As a vehicle approaches an inclined ramp, it will also detect an obstruction. However, a vehicle can easily climb a ramp. Since most surfaces are not completely flat, this problem needs to be addressed.

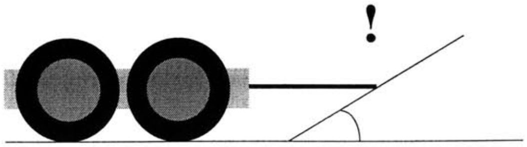


Figure 13 Incline Perceived as an Obstacle

3 Design

In this section, we will propose a model for obstacle detection that satisfies our criteria as defined in Section 1.7 as well as resolves the issues addressed in the previous section.

3.1 Laser Configuration

As stated in Section 1.6, the lasers that we are using have to be positioned on the same plane as the camera. The laser spots will therefore fall along a line that will intersect the center of the image. To reduce image processing requirements, we can predetermine which line the spots would appear on. Therefore, we need only scan this line for points. The easiest way to scan a line would be either horizontally or vertically. The only advantage to positioning the lasers diagonally would be the fact that the diagonal axis provides a larger field of view than either the horizontal or vertical axis. However, in this case, the extra field of view does not warrant the extra amount of processing.

Since the image array consists of 640 pixels horizontally and 480 pixels vertically, image resolution would be improved if we placed the lasers horizontally. However, this orientation does not allow us to detect vertical inclines such as ramps. Therefore, it would be useful to include both horizontal and vertical lasers. We only need a minimum of two lasers vertically to determine potential inclines. If the vehicle is expected to encounter only flat terrain, then we may remove the lasers along the vertical axis.

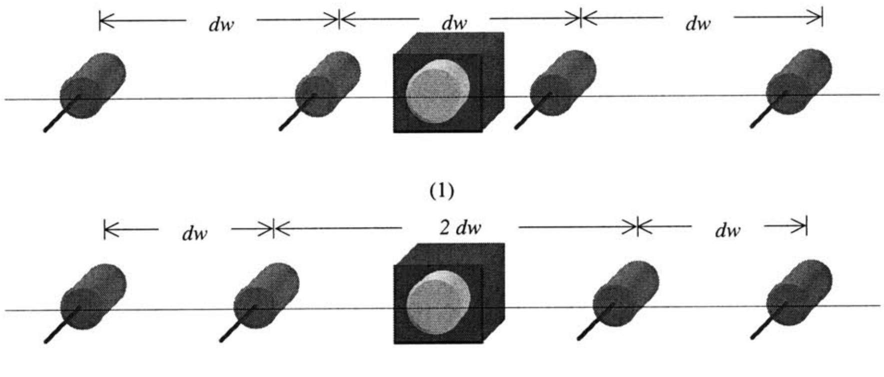
One problem that arises with including both horizontal and vertical lasers involves points at a far distance. These points will translate close to the horizontal and vertical center of the image. However, the scanline may shift left or right for the vertical lasers, or up and down for the horizontal lasers due to mechanical inconsistencies. Points that are close to the center cannot be distinguished as belonging to the vertical line of lasers or the horizontal line of lasers, as they could be shifted from either orientation. These points generally correspond to objects that are at a considerable distance away, and can be deemed non-threatening. Therefore, we may ignore such obstacles because they cannot harm the robot. If we happen to get closer to such an object, the system should be able to distinguish the resulting points, as they should be far from the center of the image.

3.2 Number of Lasers

Next, we need to determine the number of lasers. As stated before, a minimum of three lasers is needed to detect a curve described by a second-degree polynomial. However, calculations would be greatly simplified if we used an even number of lasers since we can exploit symmetrical properties. Also, an extra laser would reinforce the current system should points be missing or a laser fail. Therefore, a total of four lasers in the horizontal plane would be reasonable. Since our robot is small, it does not seem likely that it would encounter a shape that has greater than four important points of curvature.

3.3 Laser Spacing

The next issue concerns the spacing between lasers. We would like to space lasers symmetrically since this would simplify our distance-finding equations. One option would be to equally space all lasers across the front of the vehicle. Another option would be to equally space the lasers on each side of the camera by a gap *dw*, resulting in a separation of 2·*dw* between the two center lasers.



(1)

(2)

Figure 14 Possible Laser Spacings

In terms of accuracy, the latter option is the better choice. This is due to the fact that the farther away a laser is from the camera, the more accurate the distance reading will be. Let *dw* represent the distances between lasers, *dp* represent the distance between pixels, and *dr* be the difference between two ranges. As seen in Figure 15, as *dw* increases, *dp* increases. Since the distance between pixels is increased, the pixel resolution is increased, and therefore, more accurate measurements result. In addition, positioning two lasers closer to the camera will appear to overprotect the center. Since collisions with the center of the vehicle are less likely than collisions with outer points, having more sensors toward the sides should prove more beneficial. In the chance that such an event would occur, we can safeguard the vehicle by placing a physical bumper below the camera.

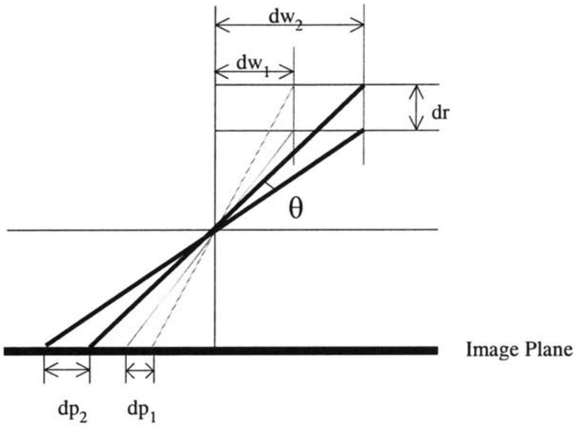


Figure 15 Relationship Between Accuracy and Distance Between Lasers

3.4 Spot Differentiation

In order to avoid problems with interference from the scene or additional light sources in the environment, we use image subtraction to differentiate spots from the background. Therefore, two images are acquired for each distance calculation, one with the lasers enabled and one with all lasers disabled. After the second image is subtracted from the first, only the laser spots will remain. In cases where the spot hits a similarly colored object, the intensity of the spots with the lasers switched on will differentiate the spot from the object. We assume that the speed of pulsing is sufficiently high as to avoid any major changes between scenes as the laser is pulsed on and off.

This method of laser pulsing will also reduce power consumption. The lasers will be only triggered half of the time, and thus will only consume half of the power of lasers that are always enabled. The tradeoff is more complexity with control of the lasers as well as the image subtraction process. Image subtraction will effectively double the response time. However, the power savings generated with this scheme is more than enough to compensate for the power resulting from the added complexity.

The laser spot will appear as a circle whose intensity is strongest in the center, and gradually fades as you move outward. Therefore, the first step in extracting the absolute center of a laser spot would be thresholding. Any pixel value from the subtracted image that is less than a certain value should be disregarded as a potential center. If no pixels remain after thresholding, then our initial threshold value is too high and should be decremented until a logical number of pixels appear. If it seems that too many pixels are apparent, i.e. more clusters of pixels exist than the number of lasers, then our estimated threshold value may be too low and background noise may be included. The current threshold value should thus be incremented. The end result of thresholding should result in four circles. By knowing the diameter of the circle, we can calculate the exact center of the circle. This value can be used to calculate distance.

Additionally, if we cannot determine which one pixel of several pixels should be chosen, we can give higher priority to points that seem to lie along the scan line of the other points that have appeared.

We assume that most of the objects encountered do not alter the color of the laser beam as few objects in the environment we expect to encounter display this characteristic. Therefore, to increase the accuracy of detecting our own spots versus other light sources, as well as to aid the thresholding process, we will apply a narrow band pass interference filter that will only allow certain frequencies to reach the camera. The wavelength of these signals must fall within some tolerance of the wavelength of the lasers we intend to use. The color filter will also partially solve the problems associated with high ambient light intensity, as most light waves from this source will be filtered out. However, some of the wavelengths contained in ambient light may match that of our laser. We will need to ensure that the power of our laser is high enough for the distances and environments we expect to encounter.

3.5 Missing Points

If we successfully capture four points, we can associate each point with a laser. The leftmost point is associated with the rightmost laser, and so on. However, we expect that oftentimes we will not be able to recover all points from the lasers. Therefore, there is ambiguity as to which point is associated with which laser. In such a case, we may execute the following procedure. We first fire the leftmost laser only. From the resulting spot, we can determine that the resulting point belongs to it, and we may calculate the distance of the point on the object as well. Next, we pulse the next adjacent laser, and complete the same steps. After incrementing through each of the lasers, we should be able to establish which distances correspond to which laser. From this list of distances, we may determine the shape of the obstacle.

It is possible that we erroneously include points that did not originate from one of our lasers. If we use four lasers, after thresholding we may find four center points that we assume are correct. However, if one of these points is an external point, then this may significantly alter the real distance measurement. We assume that we will be taking many measurements in a small time period, and that this external point will not be present in all measurements. Therefore, one or two atypical measurements may be ignored by the system.

3.6 Reflective and Absorbent Surfaces

Finally, we consider the case of highly reflective and absorbent objects. For the purposes of our project, we will assume that such objects are rare and thus will not be encountered. If a robot encounters an object from which it cannot extract any points, it may advance in small increments in case forward positions may provide more information. If it still cannot observe any spots, the robot should alter its orientation until it is in a position to safely detect obstacles.

3.7 Plan

For this project, we will create a simulation of the environment, create a simulation of what appears on the image plane, and test the functionality and accuracy of our system using this simulation. After debugging is completed, and the system works reliably, we shall implement the hardware version and verify the precision of our system.

4 Calculations

With the design in place, we will now derive equations that we will use to calculate distance as well as determine shape.

4.1 Distance Computation

The model that we use for the camera's lens equations is a pinhole model. Since we are not measuring distances on the order of the lens thickness, we can approximate the camera's lens to a pinhole. Therefore, the resulting image on the sensor array will be an inverted and horizontally mirrored view of the scene. For example, any objects in the top, left portion of the scene will appear mirrored on the bottom, right side of the image. Additionally, because we are not operating on the scale of the lens, we can ignore any spherical aberrations that are associated with rays that hit the edge of the lens.

The image plane lies behind the lens. The distance between the lens (pinhole) and the image plane is the camera's focal length, *f*. In the real world, the image plane is an array of charge-coupled sensors. This array of sensors maps to an array of pixels in simulation. We will denote the width of the sensor array as *sw* and the height of the sensor array as *sh*. The pixel array width will be symbolized by *pw* and the pixel array height will be symbolized by *ph*. The range to an object will be abbreviated as *r*, and *dw* will refer to the distance between lasers. Each measurement associated with laser *i* will be subscripted with the number *i*.

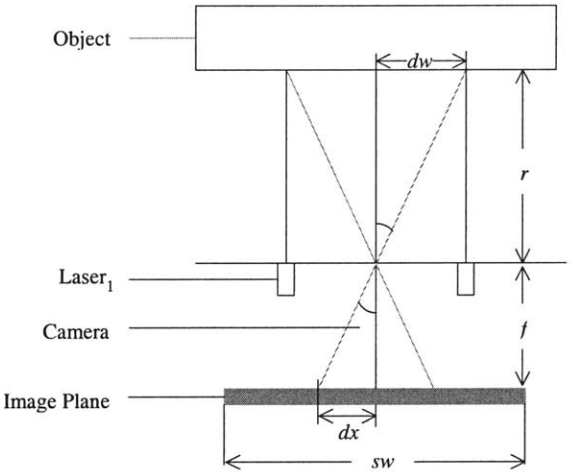


Figure 16 Calculation Diagram

To simulate the system, we will need to determine where points will fall on our sensor array. To calculate the point on the sensor array that corresponds to a laser spot on an object, we use triangulation.

$$\frac{dw}{r} = \frac{dx}{f}$$

$$dx = \frac{dw \cdot f}{r} \qquad (4.1)$$

The next step would be to convert this real distance value into a pixel value. For horizontal pixels, this value would translate to:

$$\frac{dp}{pw} = \frac{dx}{sw}$$

$$dp = \frac{dx \cdot pw}{sw} \qquad (4.2)$$

If the laser was positioned to the right of the camera, then the point will fall to the left of the image center in the sensor array. Therefore, the final pixel position, *p* will be at:

$$p = \frac{pw}{2} - dp$$

Similarly, if the laser was positioned to the left of the camera:

$$p = \frac{pw}{2} + dp$$

Therefore, if given the pixel position of the point on the pixel array, we can easily calculate the range of the object:

$$dp = \frac{pw}{2} - p \qquad \text{for } p < \frac{pw}{2} \qquad (4.3)$$

$$dp = p - \frac{pw}{2} \qquad \text{for } p > \frac{pw}{2}$$

$$dx = \frac{dp \cdot sw}{pw}$$

$$r = \frac{dw \cdot f}{dx}$$

For a vertical orientation, substitute *ph* for *pw*, and *sh* for *sw*.

Rounding error may occur in this case since pixels are integer values. We can determine the error in distance by calculating the distance if the pixel was incremented by one-half and decremented by one-half.

$$r_{\min} = \frac{dw \cdot f \cdot pw}{(dp + 0.5) \cdot sw}$$

$$r_{\max} = \frac{dw \cdot f \cdot pw}{(dp - 0.5) \cdot sw}$$

$$\text{error} = \frac{(r - r_{\min}) + (r_{\max} - r)}{2} \qquad (4.4)$$

4.2 Laser Spot Size and Intensity

The next step is to determine the spot size of a laser spot at a distance *r*. Let θ be the full-angle beam divergence, *di* be the initial diameter of the beam (at distance 0). The beam diameter is defined to be the distance across the center of the beam for which the irradiance equals $1/e^2$ of the maximum irradiance. The spot size is then half this distance. This spot size, ω, is calculated as:

$$d' = r \cdot \theta + di$$

$$\omega = \frac{d'}{2} = \frac{r \cdot \theta + di}{2} \qquad (4.5)$$

We approximate the spot size to a circle and in order to simplify calculations, we do not consider elliptical spots that will occur when approaching a surface at an angle.

The intensity of a point a distance *r* from the center of a laser spot is defined by Equation 4.6.

$$I(r) = I_0 \cdot e^{\frac{-2r^2}{\omega_o^2}} \qquad \omega_o = \text{spot size} \qquad (4.6)$$

The maximum intensity level, I_0 can be determined empirically.

The brightness of a pixel inside a simulated laser spot is proportional to the intensity of the corresponding point within the real laser spot. To determine the value, *ip*, of a pixel inside a simulated laser spot:

$$\frac{ip}{ip_{\max}} = \frac{I(r)}{I_0} = I_{norm}$$

$$I_{norm} = e^{\frac{-2r^2}{\omega_o^2}}$$

$$ip = \mathrm{I}_{\mathrm{norm}} \cdot ip_{\max} \qquad (4.7)$$

To find the center of a laser spot, we can use the centroid calculation. As given above, the center of a laser spot is more intense than the edges. Therefore, pixels with higher values are more likely to be towards the center of a laser spot. To determine the center, then, one needs to weight each pixel position by its intensity, and compute the weighted average.

$$\text{Center} = \frac{\sum_i x_i \cdot ip_i}{\sum_i ip_i} \qquad (4.8)$$

4.3 Shape Determination

We now address the problem of determining the shape of the object. We assume that the shape of the object can be described by a polynomial, $y(x) = a_1 + a_2x + a_3x^2 + \ldots + a_{m+1}x^m$. This polynomial is a linear combination of equations:

$$\mathrm{f(x)} = \sum_{k=1}^{m} a_k \cdot x^k$$

To determine the shape of our object, we need to find the parameters a_j for $j = 1$ to $m + 1$. However, we cannot expect that our ranges will fit the polynomial given by the parameters a_j perfectly. Therefore, we need to define a figure-of-merit function. We can use the least squares method as a measure of "fit" between the range, *y(x)*, and the positions of the lasers, *x*. Let σ_i be the variance of the value y_i For simulation purposes, σ is set to the round off errors calculated in Equation 4.4.

$$X^2 = \sum_{i=1}^{n} \left(\frac{y_i - \sum_{k=1}^{m} a_k \cdot x^k}{\sigma_i} \right)^2 \qquad (4.9)$$

The best fit, therefore, will occur when the function is minimized, i.e. when the derivative of X^2 equals zero.

$$0 = \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \left(y_i - \sum_{j=1}^{m} a_j x^j \right) \cdot x^k \qquad \text{for } k = 1 \ldots m \quad (4.10)$$

This equation is commonly referred to as the Normal equation of the least-squares problem [12]. We can use the Normal equations to find the parameters of our polynomial. We choose the Gauss-Jordan elimination method to solve this set of linear equations. The measure of confidence in our derived polynomial can be determined using the chi-square function. For the chi-square test, the number of degrees of freedom equal the number of lasers –1.

5 Implementation

Our implementation consists of four major components: software control, image processing, software simulation, and hardware implementation. A software simulation was developed to simulate the image captured by the camera, as well as the spots created on an obstacles by the lasers. Therefore, this simulation can be used to verify correct functionality of range determination without the problems associated with hardware inconsistencies. A software simulation can also be used to test various hardware configurations. Thus, the configuration that yields the best results can be implemented in hardware.

5.1 Equipment

The development environment chosen was Microsoft Visual C++, since it provides a software development kit for video input devices as well as various functions to paint to the screen. The computer used was a 500 MHz Pentium II processor, with 128 MB RAM. The operating system was Windows 2000.

5.2 Software Implementation

The software implementation is divided into three different categories: main control, distance calculation, and image processing.

The main control is the central commander that initiates and directs image capture and distance calculation.

The distance calculation unit is responsible for calculating spot size, calculating threshold, pixel-to-real world value conversion, determining pixel positions given distance, calculating range given pixel positions, and determining shape.

The image processing unit is responsible for capturing an image to a buffer, displaying images to the screen, image subtraction, point extraction, pixel filtration, and drawing simulated points to the screen.

5.2.1 Main Control

The main control of our system is responsible for initialization, determining when to enable and disable lasers, selecting which lasers to pulse, deciding when to capture an image, detection of missing points, and handling of any errors in the subsystems. In simulation mode, the main control is also responsible for reading user input, such as specifications, and setting simulation variables.

5.2.1.1 Initialization

Initialization in the main control consists of registering and creating the image capture window. This requires creating a double buffer to store the image of a scene with lasers enabled and a scene with lasers disabled. For non-simulation operation, we must also connect to the frame grabber board, handshake with the card, and retrieve properties of the card, link the output buffer of the card to our double buffer, as well as define settings for the mode of operation of the card, e.g. image resolution, frame rate, and color.

In addition, for stand-alone simulation (where the user defines distances and shape rather than a simulated environment) the user dialog needs to be created and default settings for input fields should be defined.

5.2.1.2 Main Calling Block

Next is the main calling block of the program. The flowchart for our main function is shown in Figure 17.

We first assume that all lasers are finding target points on the obstacle at readable distances. Therefore, Main Control enables all lasers and captures the scene into the primary buffer. Main Control then directs the image processing unit to display this primary buffer. The next step requires disabling all lasers and capturing this scene into the secondary buffer. Again, we call the image processor for displaying the second scene. With the two buffers defined, the image processor is then called to perform image subtraction, pixel extraction, and pixel filtration. After the final pixel positions are determined, the image processor calls the distance calculator to estimate range and shape.

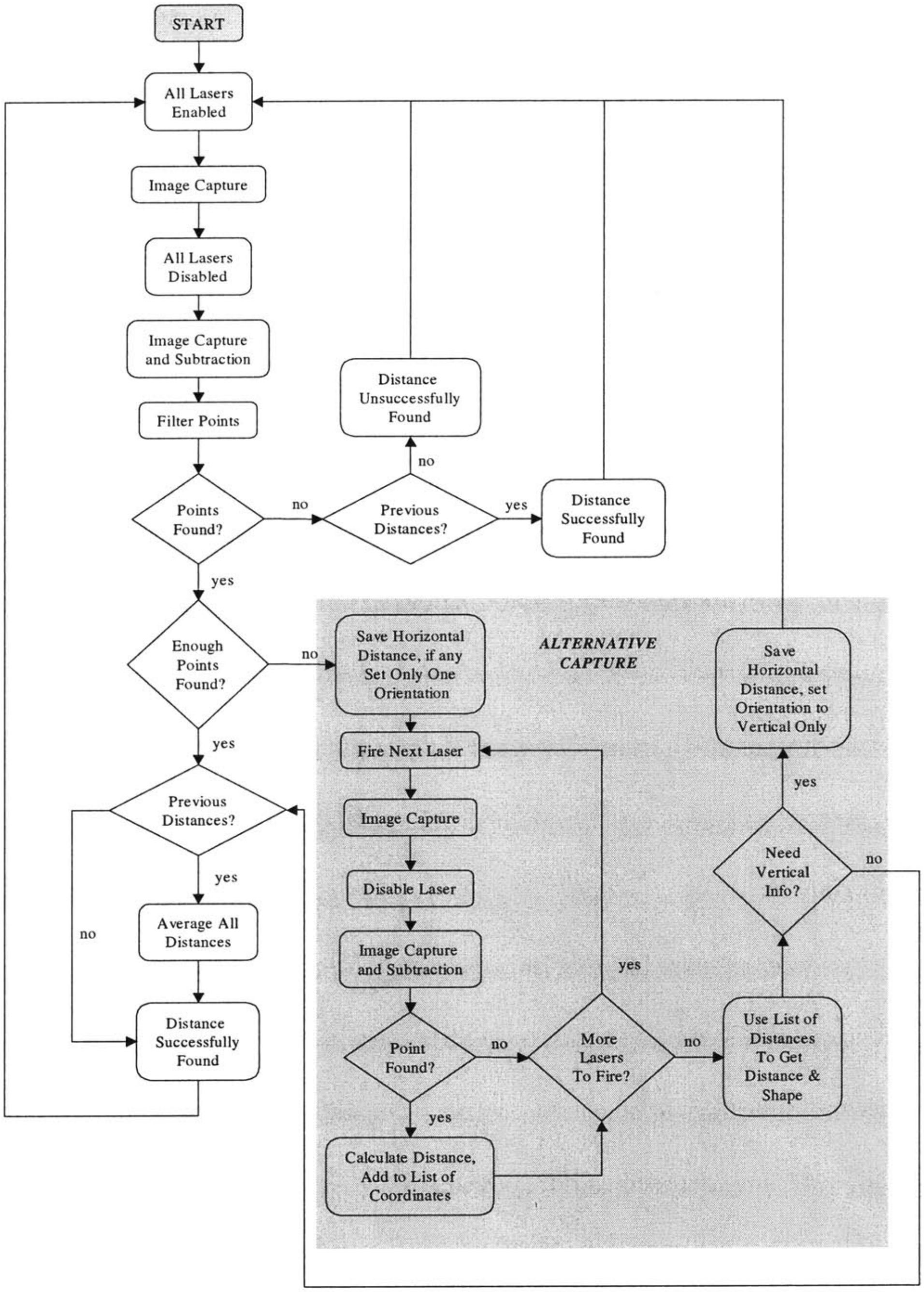

START
All Lasers Enabled
Image Capture
All Lasers Disabled
Image Capture and Subtraction
Filter Points
Points Found?
no
Previous Distances?
no
Distance Unsuccessfully Found
yes
Distance Successfully Found
yes
Enough Points Found?
no
yes
Previous Distances?
yes
no
Average All Distances
Distance Successfully Found
ALTERNATIVE CAPTURE
Save Horizontal Distance, if any Set Only One Orientation
Fire Next Laser
Image Capture
Disable Laser
Image Capture and Subtraction
Point Found?
no
yes
Calculate Distance, Add to List of Coordinates
More Lasers To Fire?
yes
no
Use List of Distances To Get Distance & Shape
Need Vertical Info?
yes
no
Save Horizontal Distance, set Orientation to Vertical Only

Figure 17 Flowchart for Main Control

If both horizontal and vertical orientations are chosen, then the distances corresponding to the horizontal axis and the vertical axis are computed separately, and then weighted by their corresponding resolutions, and averaged. The shape of the object horizontally is determined independently of the vertical shape, and both pieces of information are provided. These values are returned to the main control with a status message. If the status message indicates successful distance determination, the distance and shape calculated is displayed, and the process is repeated. If the distance calculator fails to determine distance, it will immediately exit with an error and return the last orientation considered as well as any distances successfully measured. This range is stored into a variable *h_dist*, which has been initialized to zero at the start of every cycle of distance determination. If no points were collected, *h_dist* is set to -1. If only a few points were resolved, Main control must then enter Alternative Capture mode. In this mode, each laser is fired sequentially rather than simultaneously, and distance measurements are also computed individually.

After each laser is fired, only one spot should appear after image subtraction, point extraction, and pixel filtration. After the distance calculator determines the distance corresponding to this point, main control stores the pair (the position of the current enabled laser and distance calculated) in a list.

Assume that the system failed to extract enough points and that the last orientation considered by the distance calculator is horizontal. During the sweeping of the horizontal lasers, a list of points resulting from each laser is updated. Upon completion, the distance calculator is called again with this new set of coordinates. The distance calculator can determine the shape from these points, as well as extrapolate the closest point of the object to the robot. This value is stored into *h_dist*.

If the vertical orientation is not chosen, then the distance *h_dist* is final. If *h_dist* is less than zero, then an error message will be returned. Since the next set of points may all be successfully acquired, we reset the system to the original configuration with all lasers being enabled and exit the Alternative Capture mode. The process is then repeated.

If the vertical orientation is chosen, we sweep the vertical lasers, and compute distance and shape in a similar manner to the horizontal-only configuration. If no points from the vertical lasers are recovered, *h_dist* can be used as the final range estimation. If *h_dist* is undefined, then the system failed to retrieve distance and shape, and an error message is returned. If range calculation for the vertical orientation was successful, then both distance measurements can be averaged to better estimate distance.

One can conclude that the worst case scenario of range calculation in terms of time would be a horizontal and vertical enabled orientation, where only some points from the horizontal lasers and some points from the vertical lasers can be extracted. The best case scenario in terms of time would be a one-orientation system where no points are extracted. The best case scenario in terms of results would be a two-orientation system where all points are successfully extracted.

5.2.2 Distance Calculation

The next unit is the distance calculator. As stated before, the distance calculator is responsible for calculating spot size, calculating threshold, conversion, computing pixel positions given distance, determining distance given pixel positions, and approximating shape.

5.2.2.1 Distance to Pixel Conversion

One function of the distance calculator is to compute pixel values corresponding to given distance values. Given the current laser being considered, and the number of lasers in the system, the distance calculator uses Equation 4.2 to determine the center of the simulated laser spot in the image array.

To perform the opposite function, converting distances to pixels, an analogous method is followed.

5.2.2.2 Spot Size Calculation

The calculation of spot size is straightforward. Given distance, Equation 4.5 is used to compute spot size. Equation 4.6 yields spot intensity given spot size and distance from the center of the spot.

5.2.2.3 Shape Estimation

After the corresponding range for each pixel point is computed, they should be delivered to the shape estimator. The shape estimator attempts to determine whether the object has one of three different surfaces: flat perpendicular, flat angled, or round. Based on these shapes, the shape estimator will also return what it deems to be the closest distance to the obstacle. The first step it performs is to send the distance values, and the corresponding laser positions to the best-fit function. The best fit function requires, among other things, the estimated degree of the polynomial. In order to first determine whether the object is flat, it will test with a degree of one. (The polynomial will thus be a linear equation, $y(x) = a_1 + a_2 x$). The best-fit function will return with a list of coefficients for the polynomial that best describes the object. If the chi-square that it returns is reasonable, then the object surface is determined to be flat. The coefficient a_2

corresponds to the incline of the object. If it is zero or close to zero, then the object is orthogonal to the vehicle's direction. The range is stored in the coefficient a_1.

If a_2 is not close to zero, then the object has an inclined surface. The slope of the incline is stored in a_2. Determining the minimum distance is relatively easy. Basically, the closest point of the obstacle will be near one of the edges (left or right for horizontal, top or bottom for vertical). Therefore, applying the function for the outermost laser positions will yield the farthest and closest distances to the target.

If the chi-square is not reasonable, the estimator will attempt to fit the object to a curve. The maximum degree of this curve is the number of lasers. Naturally, the best fit curve corresponds to a polynomial with the highest number of degrees. Therefore, our function will always return a polynomial with the maximum degree allowed. To ensure that the best-fit function checks lower-degreed curves first, we upper bound the polynomial to two degrees, and then increment this limit until a satisfactory fit is returned. To determine the closest point of the resulting curve, we use the polynomial returned by the best-fit function to calculate the distance to the object from each point across the front of the vehicle. We then return the minimum distance in this set as our closest range to the obstacle.

minimum distance = min (f(x) for x=0, 0.1, 0.2 , ... , *width of vehicle)*

Because the vehicle is small, the number of points to consider is few. Therefore, using brute-force should be faster than attempting to calculate the derivative of the function.

Theoretically, we should always achieve a perfect fit with a polynomial whose degree is equal to the number of lasers. However, it is possible that the closest point returned by the polynomial is "behind" the vehicle, i.e., the minimum of the function is less than zero. In that case, we may conclude that the shape of the object cannot be determined. Therefore, the minimum value of the distances given is returned, without a determined shape.

5.2.3 Image Processing

The image processing subsystem is responsible for capturing an image to a buffer, displaying images to the screen, image subtraction, point extraction, pixel filtration, and drawing simulated points to the screen.

5.2.3.1 Information Storage

The image processing unit (IPU) contains the storage unit for image points. Therefore, main control calls the IPU rather than directly calling the distance calculator to determine distance and shape. This prevents unneeded structure and data passing.

5.2.3.2 Image Display

Image capture will occur when main control notifies the image processing unit to display an image. If a simulation is running, the IPU will produce a scene illuminated by lasers, or draw a scene with lasers switched off. Otherwise, the image captured by the camera will be displayed.

5.2.3.2.1 Image Simulation

If a simulation is running, then a line of spots must be created. The distance values that these spots map to can be defined in two ways. First, an external independent simulation may provide these distances. Second, the user may input the desired type of surface and minimum distance to the target. In the latter case, the IPU must call a procedure to create a list of distance values based on the surface type and minimum range.

Once these distances are defined, the distance calculator is called to convert them to a list of pixel coordinates. These coordinates represent the centers of the laser spots to be drawn.

To create a simulated laser spot centered at pixel (x, y), the IPU first retrieves the simulated spot diameter, d, from the distance calculator. The simulated spot can be described by a d x d grid of pixels. The top, left corner of this square lies at (x-d/2, y-d/2) and the bottom, right coordinate of this square is (x+d/2, y+d/2). Therefore, to paint the spot, the IPU increments through each pixel in this square, and computes the distance from this pixel to the center (x, y). This will yield the radius r at which the pixel lies. Given r, the resulting value of the current pixel can be computed using Equation 4.7. Any pixel with r greater than $d/2$ will be ignored.

5.2.3.2.2 Video Capture

If a simulation is not occurring, we will receive images from the camera. A callback function is executed each time a message is sent from the frame grabber board indicating that a full image has been captured by the video camera. During each callback, we copy the image in the video buffer to a storage buffer. Main control manages the rate of video capture such that each image received alternates between a scene illuminated by a laser(s) and a non-illuminated scene. Image subtraction between the current video buffer and the storage buffer can be performed to isolate the laser spots.

5.2.3.3 Point Extraction

After image subtraction is completed, the next step is point extraction. A pointer is incremented through the result buffer. At each step, the data addressed by the pointer is examined. If it is above the noise threshold (calculated by the distance calculator), the coordinates of the pixel are added to a list. If not, the data byte is ignored. Image extraction then returns this list of points.

5.2.3.4 Point Filtration

The list of pixels acquired needs to be further filtered to retrieve the centers of the laser spots. These center coordinates will be the points used to compute distances. Points corresponding to the horizontal and vertical orientations will be stored in their separate, respective lists. Filtering requires detecting pixels along the scanlines, and if there are groups of pixels, to determine the centers of these spots. The filtering function examines each point's coordinates. Any pixel extracted will be pigeonholed into the horizontal or vertical set of points, based on its x,y coordinates. More specifically, if the *x* or *y* value of the pixel lies along the vertical or horizontal scanline, respectively, these points should be included as potential points in laser spots. Before including these points, the filter must confirm that this point lies outside the ambiguous zone in the center of the image (see Section 3.1). If a point is judged to be a neighbor of another point, this point should be removed from the list of final points and placed in a temporary list of points that corresponds to that laser spot. Once all points corresponding to this laser spot is extracted, the centroid of these points will be calculated, and then included in the final list of pixels from which distance will be determined. (If only the horizontal orientation was chosen, all points which passed filtration will be put on the horizontal list. An analogous procedure is performed if only the vertical orientation was chosen). With this list of pixels, distance computation and shape determination is ready to be completed.

5.2.3.5 Distance Computation

We first set our initial distance measurement to 0. If no points exist in the list of filtered points, a status message of *NO_POINTS* is returned. If the horizontal orientation is not selected, we proceed to considering the vertical case. If the horizontal orientation is selected, we check whether there are any points in the horizontal list of distances.

If there are fewer horizontal points than horizontal lasers, we set the last considered orientation to horizontal and return with a status message of *NOT_ENOUGH_POINTS*. If there are enough horizontal points, we call the distance calculator with a structure containing the horizontal pixel points, as well as other necessary data. We save the resulting range. Next we consider the vertical axis.

If the vertical orientation is selected, we determine its corresponding distance in a similar manner.

If both vertical and horizontal distance was successfully found, then we weight-average these values. Specifically, we multiply the horizontal value with its resolution and repeat the same with our vertical value. Then we sum these two values, and divide the result by the sum of the horizontal and vertical resolutions. We then return the result with a status message of *DONE.*

Horizontal and vertical shape information is actually stored in the distance calculator. Since main control initialized an instance of the image processing unit, and the image processing unit has an instance of the distance calculator, the main control has access to the distance calculator's public members, and thus has access to the horizontal and vertical shape results.

Each time main control disables a laser, a range calculation needs to be completed. Therefore, the IPU finds the distance measurement for the current laser, and adds it to the list of calculated distances. The order of addition is important. Since our model of the camera is based on a pinhole camera, points corresponding to lasers on the left side will appear on the image on the right side, as a pinhole camera mirrors the image horizontally. Therefore, additions occur at the head of the list rather than at the tail, as our distance calculator expects points corresponding from the left laser first to the right laser last.

If the current laser is the last laser being evaluated for the current orientation, the image processing unit returns a *DONE* status message to main control. At this point, main control notifies the IPU to retrieve shape and distance. The IPU then calls the distance calculator which returns with a closest distance and shape estimation.

5.3 Environmental Simulation

The last component of our software simulation system is the environmental simulation. This model creates a three-dimensional simulation of the environment in which our robot will operate. Since it is important that the simulation development environment be compatible with our previous work, WorldUp, a software package that allows three-dimensional interactive and real-time animation, was chosen. WorldUp also provides extensive support for integrating Visual C++ and MATLAB source files with simulations.

The most important output of the WorldUp simulation is the list of distances between the lasers and the impending obstacle. Main control also notifies the simulation package when to power the lasers. As an added option, the distance and shape estimations may be used to direct the autonomous vehicle when to proceed.

5.3.1 Communication

In order to link Main Control with WorldUp, the code is compiled into a dynamically linked library. From this, WorldUp gains access to all functions. However, we need to command WorldUp to call our main control.

In order to accomplish this task, we have WorldUp create a new custom object. If we add this object to our simulation, each time a frame is rendered, the object will execute its callback procedure. In this callback function we can plant calls to our routines. This object may also be an intermediary for messages. Specifically, it can hold the values for distance variables. After the simulation finds distances to the target, it can store the values in the custom object. Because the custom object has access to the main control, it can pass this data to any routine.

Also, because the object has access to the simulation environment, it has access to the laser models. Therefore, it can enable and disable the lasers pursuant to the output from Main Control.

5.4 Hardware

Our hardware components were selected to be as inexpensive and as widely available as possible to make for a very realistic and cost-effective solution. The main components of our hardware, besides the computer, are the following: laser diode modules, CCD digital camera, narrow-band pass interference filter, frame grabber video card, and digital I/O board.

5.4.1 Lasers

The laser diode modules selected are developed by Quarton, Inc. model number 650-002LPA/LPT. These modules were selected based on their size, power requirements, power output, cost, and availability. In addition, these sensors were selected to be of similar or better value than the sensors discussed in Section 1.1. These modules feature a variable resistor for output power adjustment. The power output of the laser is less than 3 mW, therefore classifying the laser as a class IIIa type laser, which may cause eye injury under specific conditions. As the output power of the laser can be adjusted, this laser can be operated under 1mW. This would classify the laser as a class II type laser, which is not considered to be an optically dangerous device unless the observer's eye is subjected to an extended period of time in direct line of the beam. The voltage requirement of this laser is between 2.7 to 5V, the operating current is about 40 mA, the wavelength of the laser is between 645 and 665 nm, and the beam divergence is 1.6mrad. The diameter of the housing module is 10.52 mm, with a length of 27.6 mm discounting the leads. The modules cost about $35 each.

5.4.2 Camera

The digital camera selected is a Ganz EMH200 Series 1/3" monochrome camera. We selected a monochrome camera rather than a color camera since we will be using a color filter to reject any other wavelengths. A monochrome camera would be more sensitive to grayscale and thus more accurate for our application. Monochrome cameras tend to provide greater contrast and a better signal-to-noise ratio than similarly priced color cameras. The digital camera has an imaging area of 4.8 mm by 3.6 mm and a pixel array of 768 pixels wide and 492 pixels tall. The focal length of the camera is 3.8 mm and the field of view is 71.5° (horizontal). The scanning frequency is 30 frames per second, and the signal to noise ratio is 45 dB. The minimum illumination is 0.6 lux . The electronic shutter speed is 1/60 to 1/80,000. It requires a 9 V power source that provides 20 mA of current. The cost of the camera is about $200.

5.4.3 Filter

The Edmund Optics narrow-band pass color filter used has a central wavelength of 650.0 nm and a 0.950" diameter. The full width-half maximum is 11.4 nm. The cost of the filter is about $65. Filters with smaller diameter are less expensive but were not available at this time.

5.4.4 Frame Grabber

The frame grabber board chosen was Hauppage's WinTV-dbx Model 401. This board is actually designed for receiving broadcast TV, but also features video, image, and frame capture as well. We chose this board because of its low cost and ease of use. Its card format is a PCI card. Video can be captured at 30 frames per second when using a resolution of 320x240. It can capture a 24-bit image using 4:2:2 digitizing . It can support an image size of up to 640x480 for NTSC video sources and 758x576 for PAL. The cost of this card was $85.

5.4.5 Input/Output

We decided to use a digital I/O board because it was much simpler than constructing a circuit to convert the output from our system (via a RS232 port) to TTL levels. The digital I/O board also takes care of noise reduction. The digital I/O board is the PC-DIO-96 manufactured by National Instruments. This board fits a PCI slot and features 96 lines (5 V/TTL) static I/O in 8-bit ports. The board allows both unidirectional and bi-directional communication. The connection runs through an industry standard 50-pin connection. The cost is about $300.

6 Testing

Our testing phase was divided into two portions, software and hardware. The software testing phase included both testing of functionality, as well as testing the efficiency and accuracy of our system under simulation. Since the camera has a finite resolution, the accuracy of our system cannot reach 100%. Therefore, our simulation, which functions in ideal conditions, can determine the maximum accuracy with which our system can estimate range and shape. The hardware testing measured the precision and accuracy of our system in real-world situations not accounted for by our simulation, primarily time delays and environmental effects.

6.1 Software

6.1.1 Stand-Alone Simulation

In the stand-alone simulation, the user defines the environment in which the vehicle operates. The options available are: minimum distance to obstacle, distance between lasers, frame rate, number of lasers, and shape. For shape, the possibilities were flat perpendicular, angled, and round. If an angled incline was chosen, the desired angle must be specified. If the round option was chosen, the degree of the polynomial and the coefficients of the polynomial must be selected. One may also select one or two orientations. If two orientations are selected, the distance between lasers, number of lasers, and shape of one orientation may be independent of the second orientation. The minimum distance to the target is not independent.

If we are not running a stand-alone simulation, simulated distance values will be retrieved from a virtual environment.

6.1.2 Virtual Environment

Our three-dimensional simulation environment consisted of a small vehicle equipped with four lasers and a camera if only one orientation was used, or eight lasers and a camera if two orientations were selected. The vehicle was sized to be very small - the dimensions of the base of the vehicle were 240 mm long, 120 mm wide, and 30 mm deep. The centers of the wheels coincided with the centers of the side of the base. The wheels had a radius of 60 mm and a width of 30 mm. Therefore, the total dimensions of our vehicle was 240 mm long, 180 mm wide, and 120 mm deep.

Two simulation environments were created. One environment consisted of a flat terrain with four walls, and three obstacles. The obstacles consisted of a large cube, an angled rectangular prism, and a cylinder. The cube measured 200 mm x 200 mm x 200mm. The angled rectangular prism is 1000 mm x 200 mm x 200 mm and tilted at an angle of 30° to the path of the vehicle. The cylinder had a radius of 200 mm and a height of 200 mm. The centers of each obstacle was placed at a distance of 6000 mm from the object. For vertically oriented lasers, an extra block was added at a vertical incline of 30°. This environment tested the vehicle's ability to recognize shape as well as its ability to determine distance and shape when hitting the edges of these objects. The second environment simulated a real-world office space and contained everyday obstacles such as chairs, tables, shelves and walls. The chairs and tables contain legs which test the vehicle's ability to maneuver in a thin-pole scenario. The office environment was also used for a simulation of a vehicle using ultrasonic sensors, so comparisons can be made between the two as well.

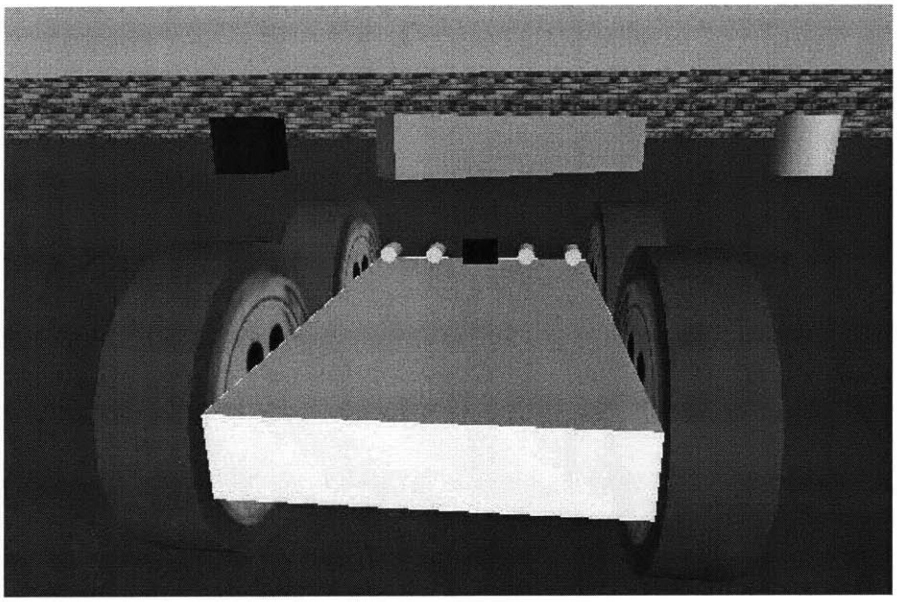


Figure 18 Simulation – Simple Environment with three obstacles

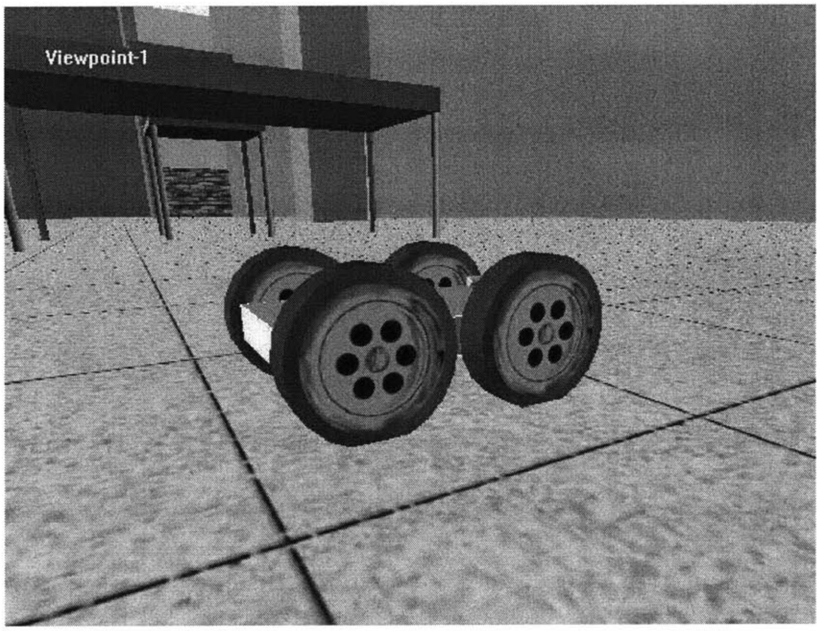


Figure 19 Simulation - Lab Environment

To detect the minimum distance between a laser module and an obstacle, a motion vector was created originating from the base of the laser module, and rotated to the current rotation of the vehicle. The distance between the origin of this vector and the closest intersection of this vector with any other polygon, if any, is returned. If no obstacle is detected, the distance is set to the maximum length of the environment. The reason behind originating this vector at the base rather than the center of the module is due to the fact that this space is being occupied by the laser beam itself, which is modeled as a long thin cylinder. If we were to position the vector in the center, the first obstacle it would detect would be the laser beam, which would result in a distance of zero.

This distance detection scheme is repeated for each module and stored in the custom object described in Section 5.3.1. A script was written and attached to our vehicle such that this cycle is repeated at each rendering. In addition, the values of these distances are displayed in the simulation's development window, as well as written to a file for accuracy analysis later.

6.1.3 Tests To Perform

For each orientation, nine tests need to be executed. For each shape (flat perpendicular, flat inclined, and round) the following conditions need to be tested: no points observed, some points observed, and all points observed. In addition, testing needs to be performed at the minimum detectable distance to the maximum detectable distance, and at angles ranging from 0° to 90°. Two-degree, three-degree, and four-degree curves also need to be tested, as well as combinations of both horizontal and vertical orientations.

6.2 Hardware

In terms of hardware, the most questionable element of our system is the intensity of ambient light in the environment. It is difficult to simulate the effects of shadows as we approach obstacles or pass by objects in the scene, as well as the effect that ambient light will have on the visibility of the laser spot. Therefore, hardware testing will be of utmost value in this respect.

In addition, it is difficult to estimate in simulation where a light beam will hit the sensor array, as photons of one beam may strike two adjacent pixels.

An interesting case related to the laser spot size is that at longer distances, the calculated spot size is less than one pixel. The beam diameter is described by Equation 4.5. In our case, the initial diameter of our laser is 1.5 mm. The divergence is 1.6 mrad. To determine the distance at which the spot's diameter falls less than a pixel:

$$\frac{1\ pixel}{640\ pixels} = \frac{x}{4.8}$$

$$x = 0.0075$$

Plugging in these values to equation 4.2 and 4.5:

$$\frac{0.0075}{3.8} = \frac{1.5 + 1.6 \cdot 10^{-3} \cdot d}{d}$$

$$d = 3166.67 \text{ mm}$$

Therefore, we should closely examine spot size at distances over 3100 mm. It is possible that these points may not even register on the sensor array.

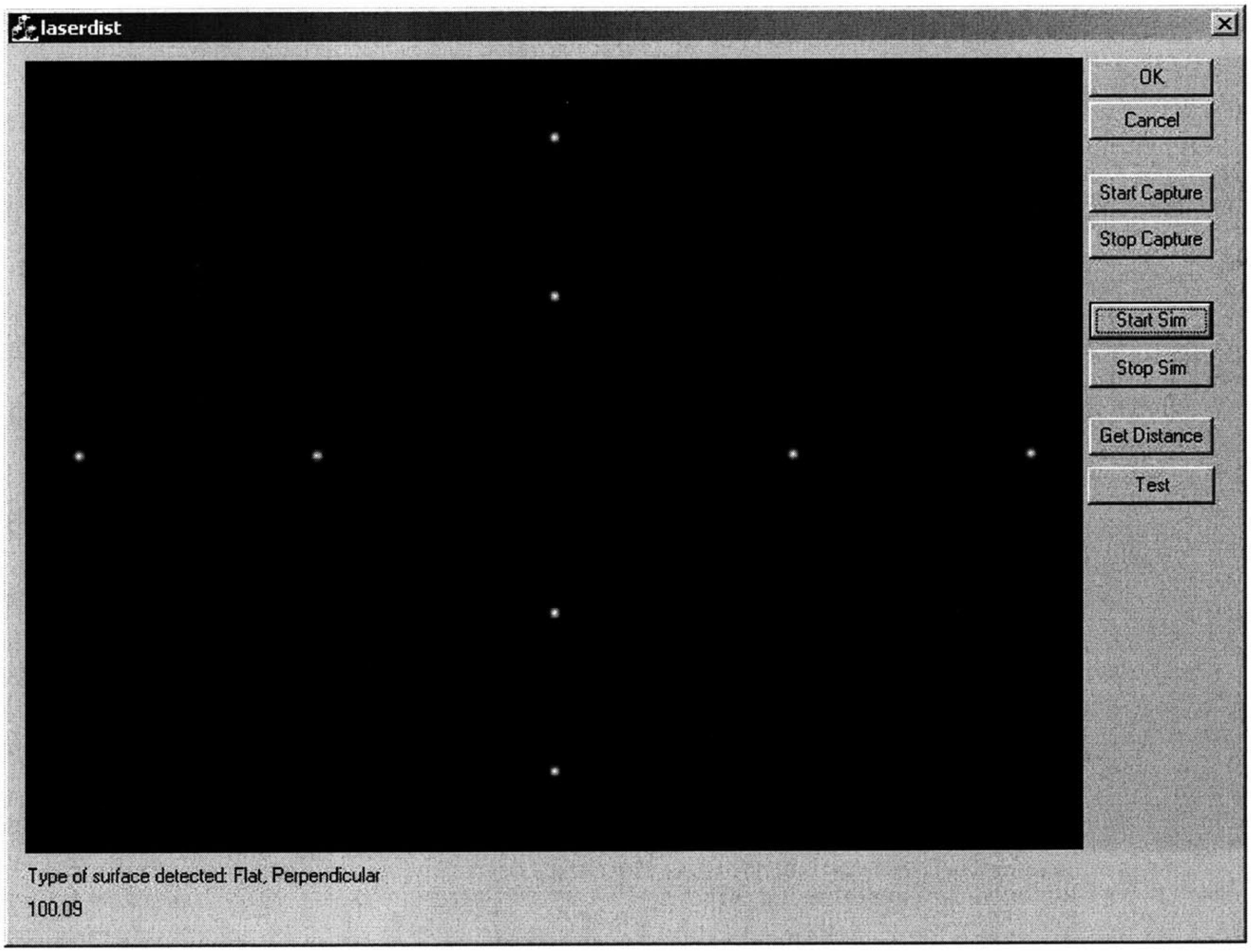
Also, we would like to test the signal-to-noise ratio of our system, and especially measure the improvements when using a color filter. In addition, we would like to identify any loss of important information that is associated with the color filter.

Finally, we would like to analyze the ability of our system to accurately detect shape as well as the minimum distance required to the target to detect shape.

7 Results

7.1 Simulation Testing

Figure 20 presents a screen capture of the image plane simulation during laser illumination of the scene. The system used both the horizontal set and vertical set of four lasers each. The spacing between the lasers were 30 mm, symmetric about the camera.



```
Type of surface detected: Flat, Perpendicular
100.09
```

Figure 20 Screen Snapshot of Camera View Simulation

The first test performed tested the accuracy of distance measurement on approach to a flat, orthogonal surface using horizontally oriented lasers.

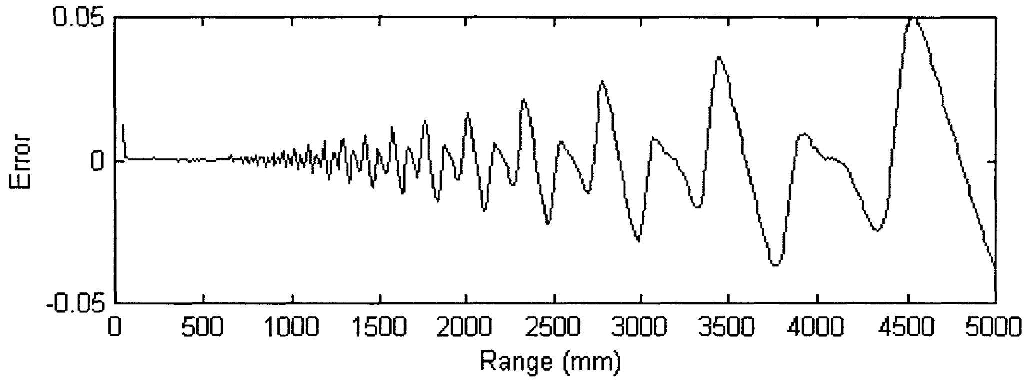


Figure 21 Distance Measurement Error for a Flat, Orthogonal Surface – Horizontal

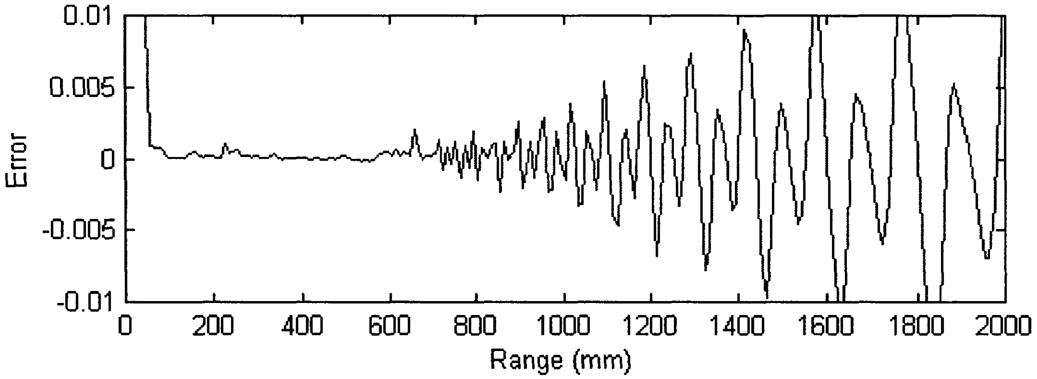


Figure 22 Distance Measurement Error for a Flat, Orthogonal Surface – Close Range

The maximum distance that the system can resolve is 12.6 m. However, the accuracy at that range is very poor. We analyzed distances between 45 mm, which is the range at which at least one point can be detected, to 5 m. The accuracy of the system at 5 m is within ± 0.05. At closer ranges, under 1.5 m, the accuracy is within ± 0.01. Under 1 m, the accuracy is within ±0.005.

The reason for the oscillations evident in the error plot is due to the fact that a certain range of distances will fall into one pixel. However, when calculating the distance using that pixel value, only one distance (say, *dist_final*) will be returned. Therefore, the entire range of distance will map into *dist_final*. If the vehicle happens to be *dist_final* away from the target, then the error in measurement will be zero. If the vehicle happens to be at the extreme end of the range of distances that will correspond to that pixel, a high error will be returned.

The relatively high error at 45 mm can be explained by the fact that the vehicle is so close to the object that only one point is detected. The lack of information contributes to the decline in accuracy.

The same test was computed for vertically oriented lasers. The error is slightly higher, due to the decrease in pixel resolution in the vertical axis (480 pixels versus 640 pixels for horizontal).

The flat, orthogonal test was also applied to a system using both horizontally and vertically oriented lasers. The result is an accuracy that falls between the vertical-only results and horizontal-only results. The horizontally oriented lasers improve the accuracy rate of vertical distance measurements, however, the inclusion of the vertical distance measurements degrade the overall weighted-average distance calculation.

The next test conducted was applied to flat, inclined surfaces. The range tested started at 45 mm and ended at 5 m, and tested angles of 30° and 60°.

As seen in Figure 23, the distance error measurements for an angled surface are slightly higher than those for a flat, orthogonal surface. However, this is expected, since angled surfaces include points at farther distances as well as close distances, and resolving these ranges is more complicated than with flat surfaces. Therefore, surfaces that are more angled yield less accurate results.

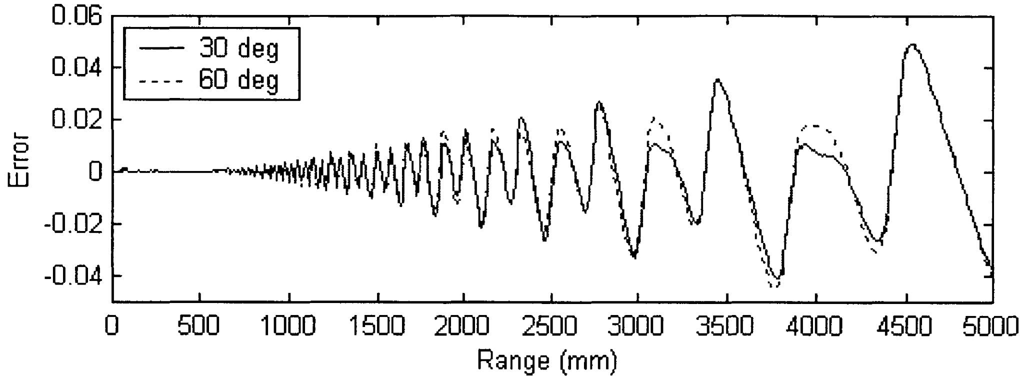


Figure 23 Distance Measurement Error for a Flat, Inclined Surface – Horizontal

The angle measurement error was also calculated.

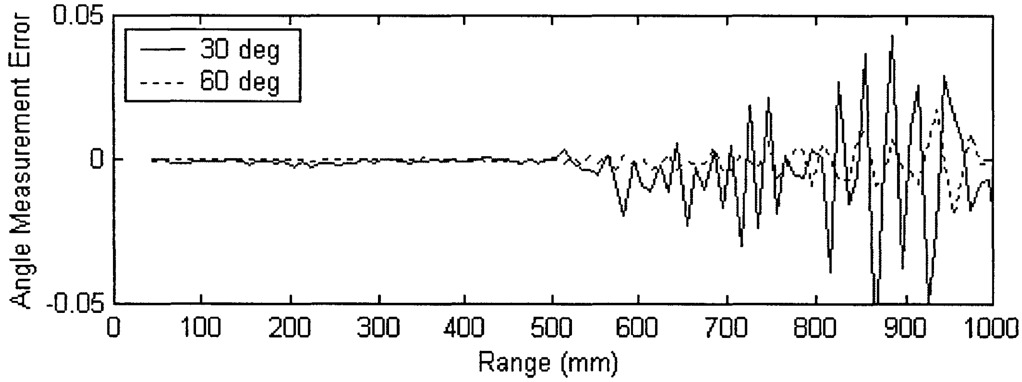


Figure 24 Angle Measurement Error for a Flat, Inclined Surface – Horizontal

As seen in Figure 24, measuring higher degree angles yields more accurate results. This is due to the fact that larger angles are easier to detect than flatter surfaces.

Correct detection of angle and shape starts to decline around 1500 mm for 30° angled surfaces. Angle measurement error is about ± 0.05 under 1 m. As expected, at far distances, the system will perceive targets as flat orthogonal surfaces. For 60° angled surfaces, shape and angle estimation degrades at around 2500 mm.

An identical set of angled tests were performed for vertically oriented lasers. As expected, the error measurements are slightly higher due to the decreased resolution in the vertical plane. However, accuracy is still high at close ranges.

Next, tests for round surfaces were run.

We first tested an obstacle described by a 2nd degree polynomial. The equation for this polynomial was $y(x) = 0.02\,x^2$.

We tested the system at horizontal shifts of 0 mm and 60 mm.

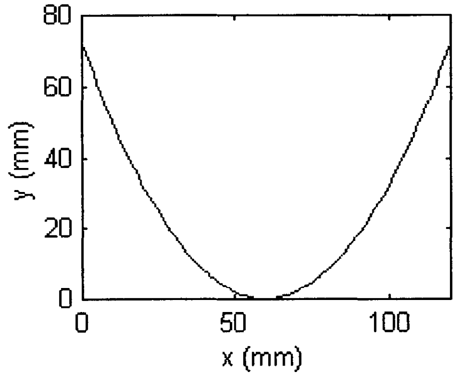


Figure 25 Shape of 2-degree Round Obstacle (shifted by 60 mm)

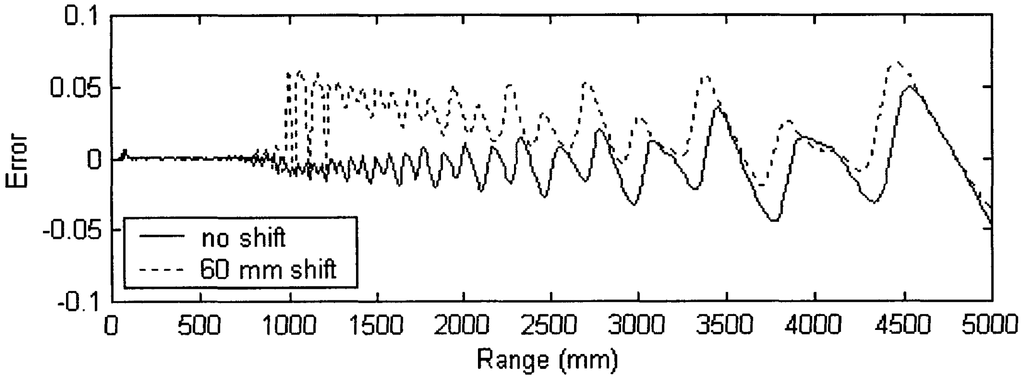


Figure 26 Distance Measurement Error for a Round Surface 2 degree

The maximum error for a non-shifted object was ±0.05. Between 100 and 900 mm, where all points were detected, the maximum error was 0.002, which is quite low for a round surface. Shape estimation declined after 900 mm. Between 45 mm and 100 mm, only some points were extracted, therefore, the degree of the polynomial was incorrectly determined.

The polynomial for a 3rd degree curved obstacle was $y(x) = 0.0003\,x^3$. The obstacle was also shifted by 60 mm as well.

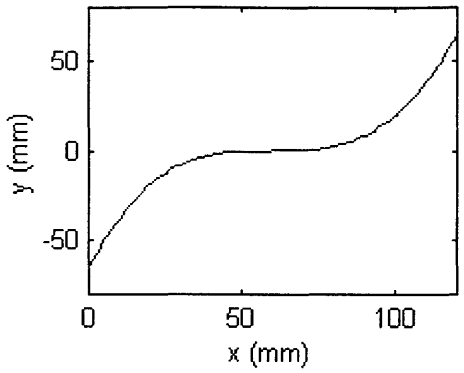


Figure 27 Shape of 3rd Degree Round Obstacle (shifted by 60 mm)

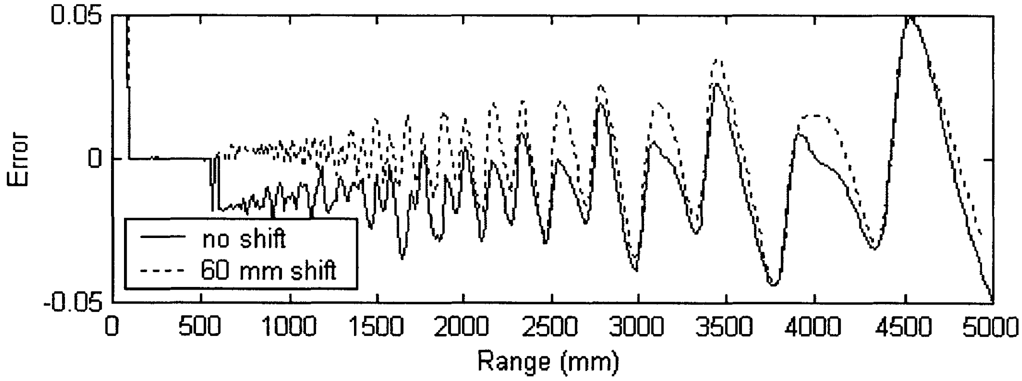


Figure 28 Distance Measurement Error for Round Surface 3 degree

The maximum error rate between 100 and 5000 mm is around ±0.05. Again, the high error between 40 and 100 mm is due to missing points that fall outside the field of view of the camera. The degree of the curve was successfully found at ranges up to 600 mm. At ranges from 600 mm to 1500 mm, the system found a curve of degree 2.

The last curve tested was a 4th degree polynomial. The equation describing this polynomial is $y(x) = -16.4\,x + 0.403\,x^2 - 0.003186\,x^3 + 0.0000073\,x^4$.

The measurement error for this curve is less than ± 0.015 under 1500 mm. The error increases significantly after 1500 mm. This is due to the fact that shape cannot be estimated well at longer distances, especially for a shape that is this complicated.

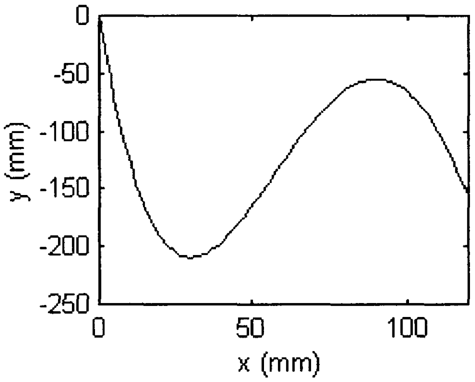


Figure 29 Shape of a 4th Degree Round Obstacle

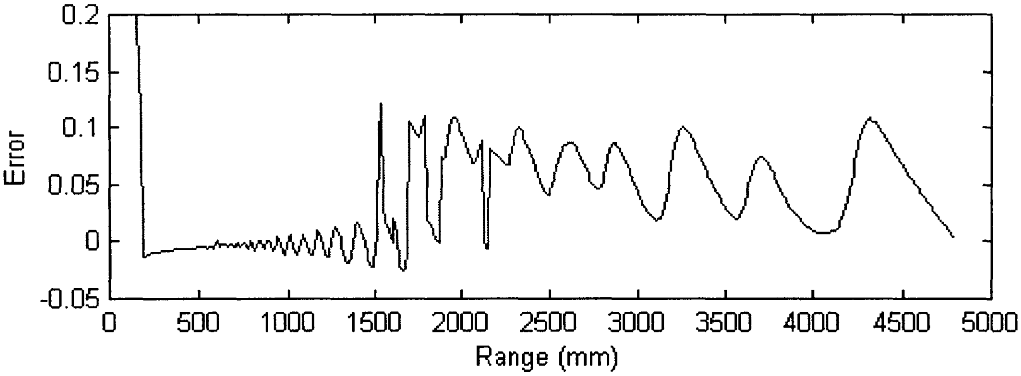


Figure 30 Distance Measurement Error for Round Surface 4 degree

The error at greater distances is higher than the lower-degree curves. This is partially due to the fact that the complexity of the shape makes fitting more difficult.

The error rates for scenarios including missing points are also similar. The maximum error is ± 0.05 from 45 mm to 5m.

7.2 Hardware Testing

7.2.1 Setup

The setup of the lasers and the camera is shown in Figures 31 and 32. Each laser is spaced 30 mm apart (with the exception of the two center lasers, which are spaced 60 mm apart). The plane from which the beams from the lasers originate is referred to as the base.

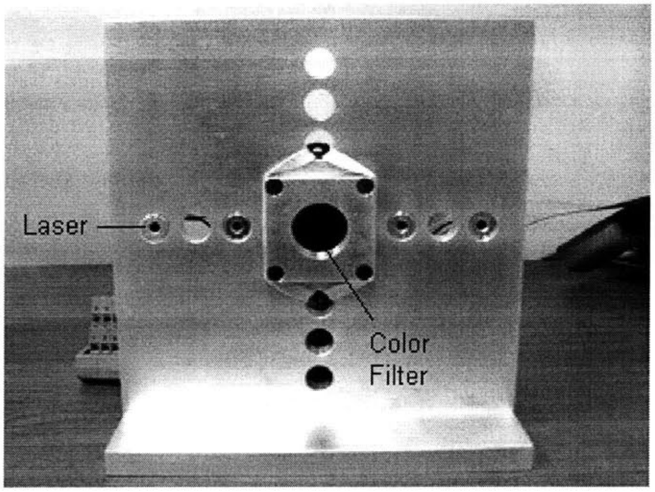


Figure 31 Setup of Lasers and Camera (Front View)

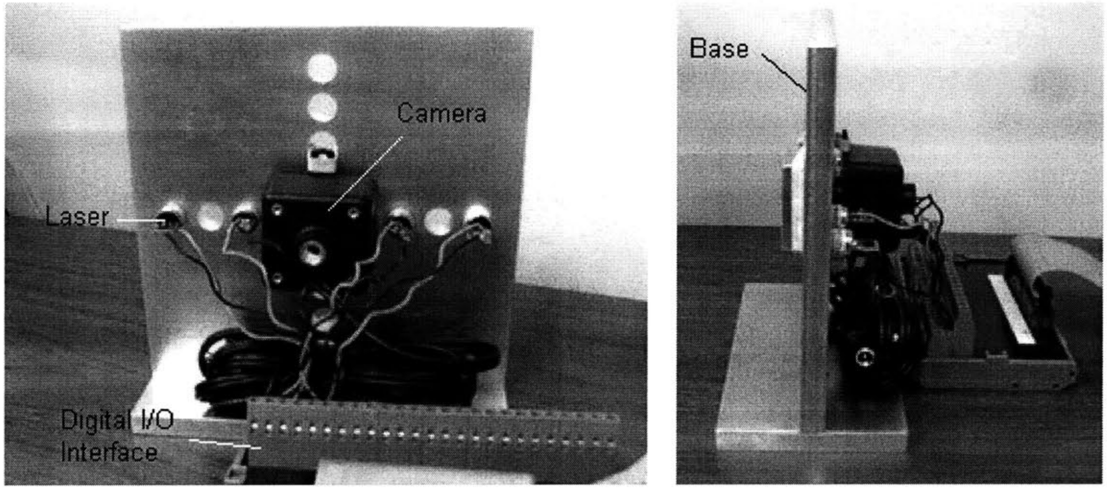


Figure 32 Setup of Lasers and Camera (Back View and Side View)

Before testing was conducted, two configuration steps were necessary. First, the four lasers were aligned such that their beams are collinear with each other as well as normal to the surface of the base. Second, the exact focal length of the camera was calculated.

To calculate focal length, the base was set 500 mm away from an orthogonal surface. An image was captured, and the resulting pixel values were examined. Using these pixel values and Equations 4.1 and 4.2, the average focal length can be calculated.

One issue that arose with image capture was the offset between the optical center of the camera and the center of the image. Specifically, the image was shifted slightly downwards and to the right. The horizontal offset was accounted for by adjusting Equation 4.3:

$$dp' = dp - x\_offset$$

The vertical offset was corrected by specifying the scanline as $ph/2 + y\_offset$, rather than $ph/2$.

An interesting result from initial tests was the effectiveness of the color filter. In indoor environments, the color filter isolates the laser spots sufficiently to eliminate the need for image subtraction. Therefore, image subtraction was not performed during testing. In addition, a "visor" was placed above the lens to further block out ambient light from overhead.

Another result from initial tests is the fact that our system was strict in determining shape. The system often found a flat surface to be a third-degree polynomial because our initial tolerances were too low. Therefore, we increased our tolerances.

We also experienced a problem with the lens of the camera. There was a bit of distortion with the lens of the camera at very near distances. The results of this distortion will be discussed in the next section.

The obstacles were located at ranges between 125 mm to 4500 mm. Due to the horizontal offset of the image plane, we could not test at the minimum range of 45 mm that we had predicted was possible. The maximum range was set to 4500 mm due to the dimensions of the test room. The texture for all obstacles was plain white paper. Five measurements were recorded for each range and shape.

7.2.2 Tests

The first test performed measured distances to a flat perpendicular surface. The results of this test is shown in Figure 33. (The absolute value of the error measurements are plotted).

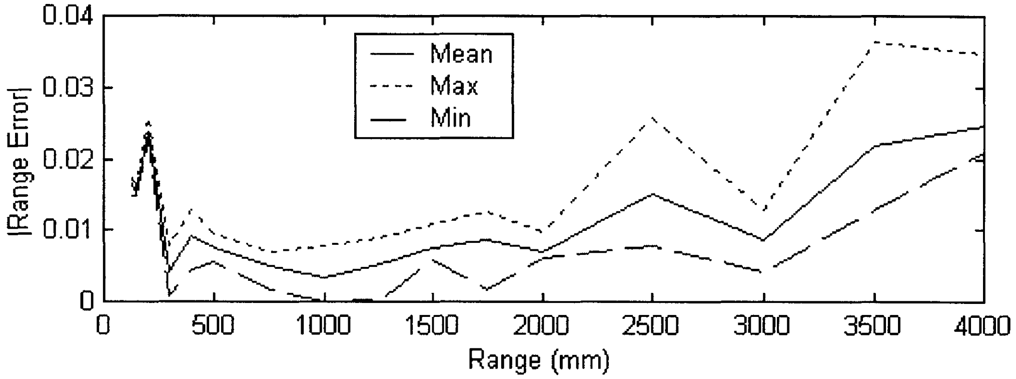


Figure 33 Actual Distance Measurement Error for a Flat, Orthogonal Surface

The relatively high error at ranges less than 200 mm is likely due to the distortion of the lens. However, the average range error is less than 2.5%, and is therefore still fairly accurate. There are numerous methods that correct for lens distortion, which may be implemented in future work.

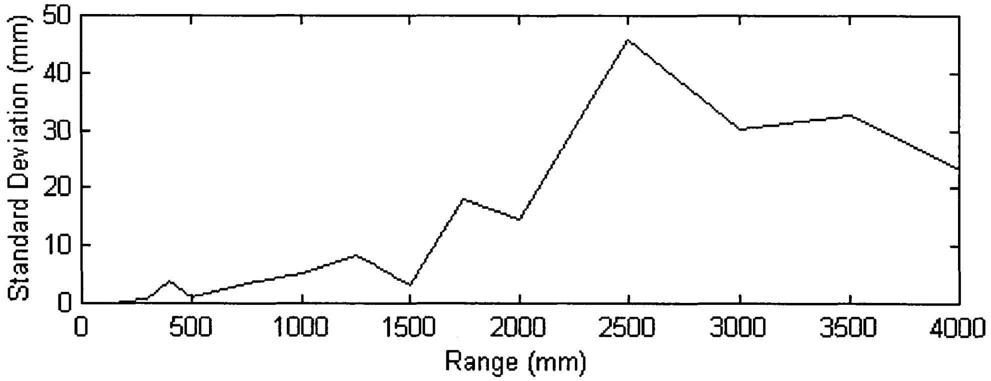


Figure 34 Standard Deviation of Distance Measurements for a Flat, Orthogonal Surface

The precision of our system is rather low, as evidenced in Figure 34. A likely reason for this outcome is the noise inherent in the camera. If noise corrupts the image, especially near or at the laser spots, the accuracy of the centroid calculation will be decreased, and thus distance estimations will be affected. The maximum error rate is reasonable, however, and thus precision does not appear to be a major issue. If needed, a camera with a higher signal-to-noise ratio should solve this problem.

Interestingly, at 4500 mm, the distance between laser spots could not be resolved by the camera. The system needed to individually pulse each laser to isolate each spot to determine distance. This successfully resulted in a range estimate, with an error of about 0.035.

Situations where points were missing were also tested. The average error rates resulting from removing one laser spot and three laser spots were less than 0.06.

Overall, the results of the flat, orthogonal test verified the results of our simulation.

The next test performed involved inclined surfaces. The absolute error rates for range determination and angle measurement is plotted in Figures 35-38.

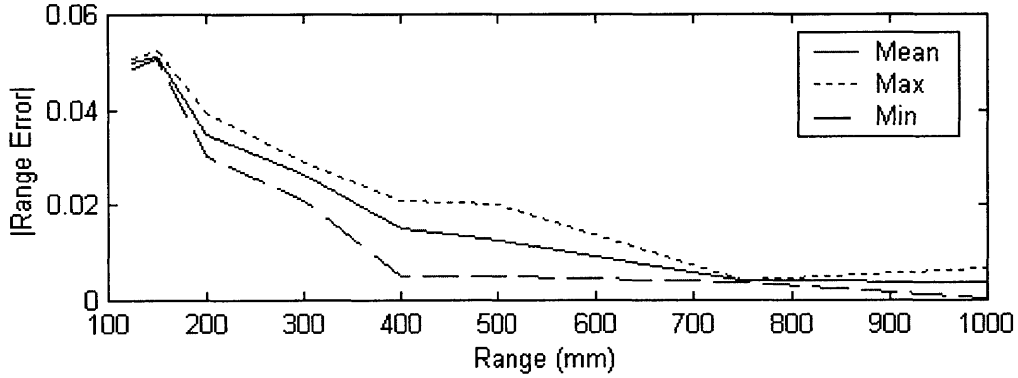


Figure 35 Actual Distance Measurement Error for a Flat, Inclined Surface at 30°

Relatively high error rates occur at 125 mm. The distance error is most likely due to the angle measurement error, as the distance to an object depends on the angle returned by the shape estimator. Again, lens distortion may have caused the inaccuracy in angle calculation. Since each of the four lasers provide range information that are used to determine angle, one inaccurate measurement would significantly affect the angle returned.

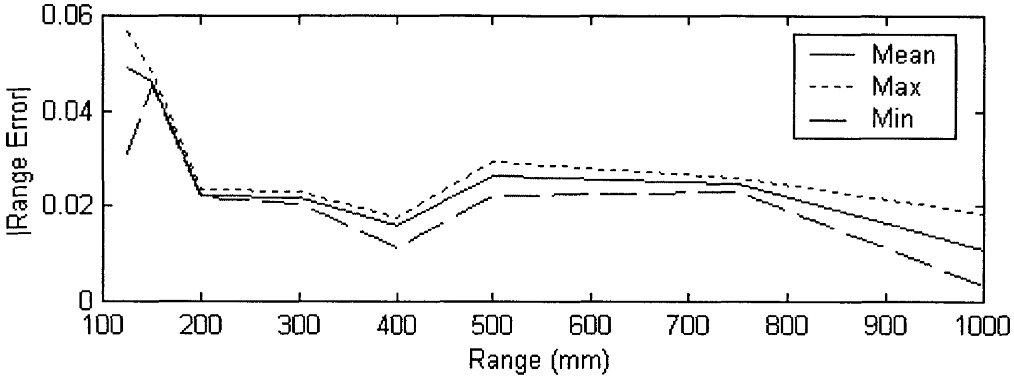


Figure 36 Actual Distance Measurement Error for a Flat, Inclined Surface at 60°

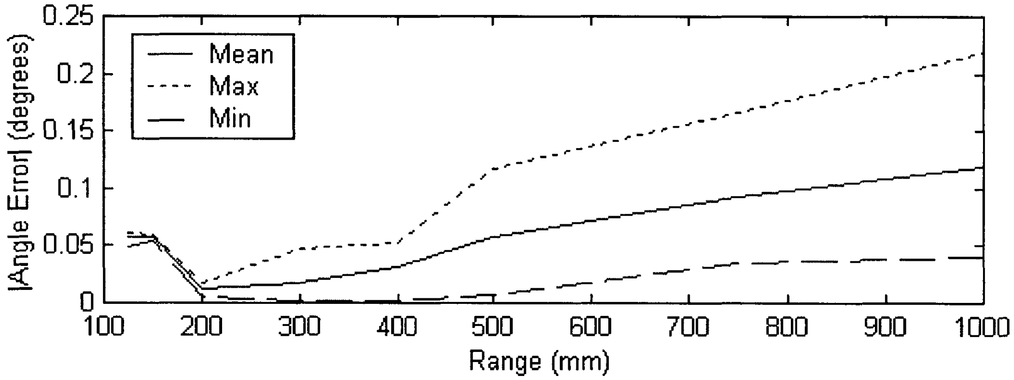


Figure 37 Actual Angle Measurement Error for a Flat, Inclined Surface at 30°

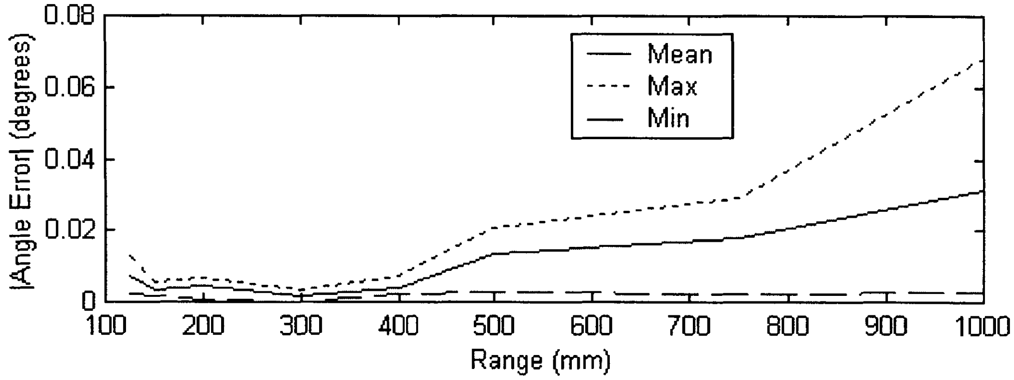


Figure 38 Actual Angle Measurement Error for a Flat, Inclined Surface at 60°

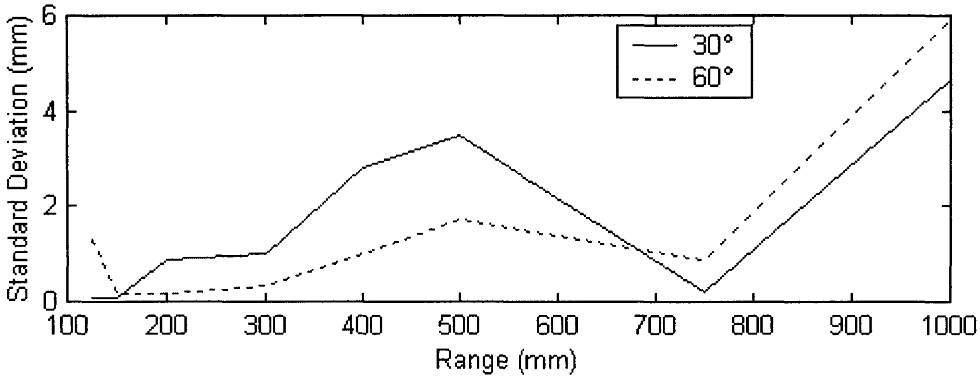


Figure 39 Standard Deviation of Distance Measurements for a Flat, Inclined Surface

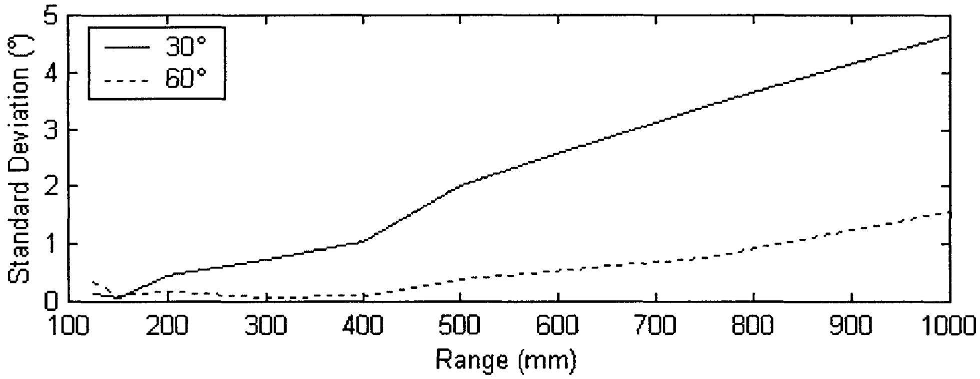


Figure 40 Standard Deviation of Angle Measurements for a Flat, Inclined Surface

The error rates under 500 mm are higher than the rates our simulation yields. However, this is expected, since simulation computations did not take into account the elliptical spots that would result from directing a laser beam towards an inclined surface.

The last test performed involved a round surface. A cylinder with a diameter of 178 mm was used and was centered along the same axis as the camera. The results are shown in Figure 41. (Absolute values are graphed).

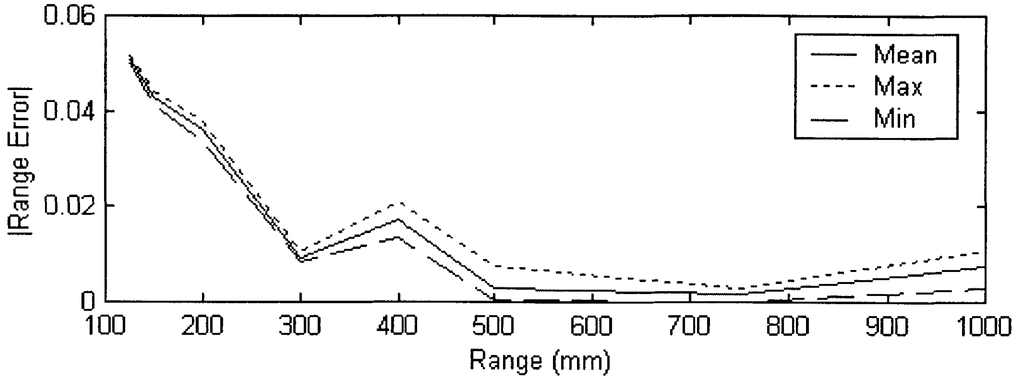


Figure 41 Actual Distance Measurement Error for a Round Surface

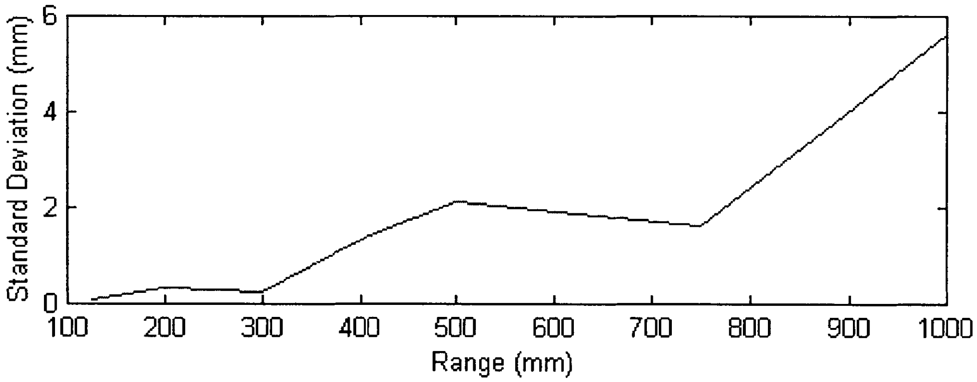


Figure 42 Standard Deviation of Distance Measurements for a Round Surface

The system performs reasonably well in the case of a round surface. The average error rate does not exceed 0.05.

7.3 Discussion

The accuracy of the system is quite respectable. At long distances, the accuracy is slightly worse than accuracies of other sensors discussed in Section 1.1. However, our error rate is better than that of these sensors at close range, and this is most important for a small vehicle. In addition, the minimum distance ranges for our system are closer than can be achieved with comparable sensors. The accuracy of our system can be improved with a higher resolution camera.

However, the speed of our system is not as fast as other sensors, since we are limited by the camera's capture speed. For this system, capture rates are about 30 frames per second.

8 Future Work

There are several improvements as well as interesting additions that can be added to this project.

First, the system can be improved by incorporating a diagonal line of lasers to improve resolution and field of view for the same camera. The increase in computing power, however, should be examined.

Also, we can make this system more portable. This project required the use of a 500 MHz Pentium II processor since we were also simulating our environment and camera output. We can remove the simulation code as well as reduce the size of the resulting code in order to fit the software onto a microprocessor. In addition, we may use an embedded frame grabber instead of the PCI card used for this project. The result of these changes would allow an onboard processing unit that is lightweight and requires little power and space.

We may also consider using a laser of a different color. Cameras tend to have higher sensitivity readings for wavelengths in the 575 nm to 605 nm region. This would increase our spot visibility. However, lasers generating wavelengths in this range tend to be more expensive.

In addition, rather than firing each laser sequentially to identify missing points, we can tag laser spots easily by using different color lasers. However, this would require developing a filter that would accept all of the various wavelengths of the lasers while blocking out other wavelengths. It is possible that image subtraction alone may yield reliable results in certain environments, eliminating the need for a specialized filter. Using multi-colored lasers would greatly reduce the response time of our system.

Also, to better detect shape, we can employ a scanning-head laser system. Distance can be calculated if the angle of the laser is known.

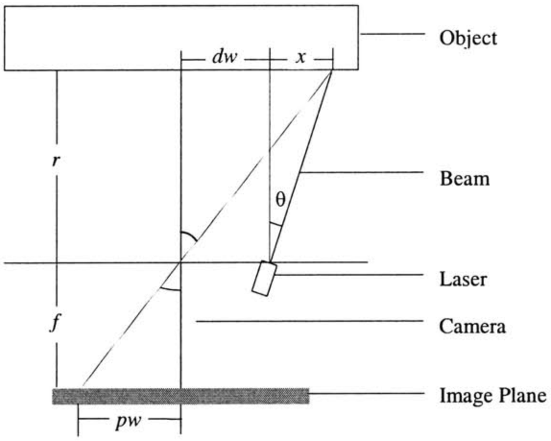


Figure 43 Diagram for a Scanning Laser

$$\frac{dw+x}{r} = \frac{pw}{f}$$

$$\frac{dw + r \tan\theta}{r} = \frac{pw}{f}$$

$$r = \frac{dw}{\frac{pw}{f} - \tan\theta}$$

However, more shape information comes at the cost of mechanical stability, as well as higher complexity in timing issues and motor control.

9 Conclusion

Overall, we have developed a small, low-cost, low power, and lightweight system to perform obstacle detection for robot navigation using structured light. Our system's total dimensions are 13cm x 8cm x 7cm and weighs less than 500 g. The accuracy of the system is greater that 95% under 5 m and can achieve an ideal minimum range of 45 mm. The cost of the entire system (discounting the computer) is around $600. The time to detect distance and shape in the best-case scenario is 33 ms, and in the worst case scenario 0.13s. Our power consumption is estimated to be less than 1.85 W for a four laser, one camera system.

In addition, the setup of our system is highly flexible and can be adjusted to the specifications of the user. Because only one laser is needed to determine range, and we incorporate four lasers in our design, our system is more resilient to errors. Also, our system offers an advantage over laser rangefinders because of its high accuracy at very close ranges. Furthermore, our system can estimate shape without the use of moving components, which allows for a highly mechanically stable system.

Therefore, the features of our system makes it a favorable system for obstacle detection for small autonomous vehicles.

10 References

[1] Baba, M., and T. Konishi. "Range imaging system with multiplexed structured light by direct space encoding." *Proceedings of the 16th IEEE Instrumentation and Measurement Technology Conference*, Vol. 3 , 1999, pp. 1437 –1442.

[2] Bhatnagar, Deepak, Pujari, Arun K, and P. Seetharamulu. "Static scene analysis using structured light." *Image and vision computing,* Vol. 9 No. 2, April 1991, pp. 82-87.

[3] Clark, Robert. "Optical Distance Measurement Sensor." *Medical Electronics,* Vol. 25 No. 1, Feb. 1994, pp.74-78.

[4] De la Escalera, A., Moreno, L., Salichs, M.A, and J.M. Armingol. "Continuous mobile robot localization by using structured light and a geometric map." *International Journal of Systems Science*, Vol. 27 No. 8, 1996, pp.771-782.

[5] DePiero, F., and M. Trivedi. "3-D Computer Vision Using Structured Light: Design, Calibration, and Implementation Issues." *Advances in Computers*, Vol. 48, 1996, pp. 243-277.

[6] Everett, H.R. Sensors for Mobile Robots: Theory and Application. Wellesley, MA: A. K. Peters, Ltd., 1995.

[7] Fisher, R.B., Ashbrook, A.P., Robertson, C., and N. Werghi. "A low-cost range finder using a visually located, structured light source." *Proceedings of the Second International Conference on 3-D Digital Imaging and Modeling*, 1999, pp. 24 –33.

[8] Fontana, R. J., Larrick, J. F., Cade, J. E. and E. P. Rivers, Jr. "An Ultra Wideband Synthetic Vision Sensor for Airborne Wire Detection." *SPIE Proceedings*, Vol. 3364, 1998.

[9] Fowler, C., Entzminger, J., and J. Corum. "Assessment of ultra-wideband (UWB) technology." *IEEE Aerospace and Electronics Systems Magazine*, Vol. 5 No.11 , Nov. 1990, pp.45-49.

[10] Matthies, L., Balch, T., and B. Wilcox. "Fast Optical Hazard Detection for Planetary Rovers Using Multiple Spot Laser Triangulation." *Proceedings of the 1997 IEEE International Conference on Robotics and Automation*, Vol. 1, 1997, pp. 859 –866.

[11] Phan, Long. "Collision Avoidance Via Laser Rangefinding." Masters of Science Thesis, Massachusetts Institute of Technology, Cambridge, MA, 1999.

[12] Press, William H. Numerical Recipes in C: The Art of Scientific Computing. Cambridge [Cambridgeshire]; New York: Cambridge University Press, 1988.

[13] Taylor, James D. Introduction to Ultra-Wideband Radar Systems. Boca Raton: CRC Press, 1995.

[14] Verdeyen, Joseph. Laser Electronics, 3rd Edition. Englewood Cliffs, N.J.: Prentice Hall, 1995.

[15] Yuta, S., Suzuki, S., Saito, Y., and S. Iida. "Implementation of an Active Optical Range Sensor Using Laser Slit for an In-Door Intelligent Mobile Robot." *International Workshop on Intelligent Robots and Systems IROS*, Nov. 3-5, 1991, pp. 415-419.

APPENDIX A

Main Control

```
// laserdistDlg.cpp : implementation file
// Main control
// D. Tran 2001
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "laserdist.h"
#include "laserdistDlg.h"
#include "nidaqex.h" // for digital/ttl i/o

// for hardware testing
#define DO_HORZ 1
#define DO_VERT 0
#define NUM_LASERS 4

////////////////////////////////////////////////////////////////////////
/////
// CLaserdistDlg message handlers

BOOL CLaserdistDlg::OnInitDialog()
{
   CDialog::OnInitDialog();

   // create capture window
   m_hCapWnd = thisProcess.Create(this->GetSafeHwnd(), 10,10);

   // Initialize Lasers
   InitLasers();

   m_status.SetWindowText("Status: Idle");
   m_bOn = TRUE;
   m_bAlt = FALSE;
   m_bTest = FALSE;
   return TRUE;  // return TRUE  unless you set the focus to a control
}



////////////////////////////////////////////////////////////////////////
/////
// Non-simulation functions

void CLaserdistDlg::OnSingleCapture()
{
   // for hardware testing - single measurement
   float dist = 0.0;
   char distTxt[10];
   m_simSettings.ManualSetSettings(DO_HORZ, DO_VERT, 0,
                 0, NUM_LASERS, 0.0, 30.0, 0, 0, "",
                 0, NUM_LASERS, 0.0, 30.0, 0, 0, "");
   OnStartSimTest(DO_HORZ, DO_VERT);

   capOverlay(m_hCapWnd, FALSE);
   thisProcess.SetLaserMode(FALSE);
   OnTimerNoSubtract();
   EnableLasers(0,NUM_LASERS,0);
}

void CLaserdistDlg::OnTimerNoSubtract()
{
   int status;
   float dist = 0.0;
   char distTxt[10];
   int recall = 1;
   if (m_bAlt) {
      EnableLasers(m_iLaserNum, NUM_LASERS, 1);
      thisProcess.AltCapture(m_iCurrOrientation, m_iLaserNum);
   } else
      EnableLasers(0, NUM_LASERS, 1);

   // capture image
   Sleep(100);
   BOOL fResult = capGrabFrame(m_hCapWnd);
   // display image
   thisProcess.TestImageSubtract();

   if (m_bAlt) {  // calculate distance laser by laser
      status = thisProcess.AltUpdateReadings(m_iCurrOrientation,
                                             m_iLaserNum++);
      if  (status == DONE) {
         dist = thisProcess.AltFindDistance(m_iCurrOrientation);
         if ((m_iCurrOrientation == HORZ) && DO_VERT){
            // then try vertical
            m_iCurrOrientation = VERT;
            m_rHDist = dist;
            m_iLaserNum = 1;
            m_iPaintHorz = 0;
            m_iPaintVert = 1;
            m_bAlt = FALSE;
         } else { // currently vertical
            if (dist < 0) {
               if (m_rHDist > 0) {
                  // already got a distance from horizontal, use that
                  sprintf (distTxt, "%.2f", m_rHDist);
                  m_DistanceText.SetWindowText((CString) distTxt);
                  m_status.SetWindowText("Type of surface detected: " +
                                    thisProcess.m_distCalc.m_sShape);
                  // reset to initial settings
                  m_iLaserNum = 1;
                  recall = 0;
               } else {
                  // tried every possible way to determine distance
                  AfxMessageBox("Cannot determine distance", MB_OK, 0);
                  recall = 0;
               }
            } else { // found distance (less points) fine
               // average with m_rHDist
               if (m_rHDist > 0)
                  dist = (dist*CAPHEIGHT +
                          m_rHDist*CAPWIDTH)/(CAPHEIGHT+CAPWIDTH);
               sprintf (distTxt, "%.2f", dist);
               m_iLaserNum = 1;
               m_DistanceText.SetWindowText((CString) distTxt);
               m_status.SetWindowText("Type of surface detected: " +
                                    thisProcess.m_distCalc.m_sShape);
               recall = 0;
            }
         }
      }
   } else { // enough points found
      status = thisProcess.simFindDistance(&m_iCurrOrientation, &dist);
      if (status == NO_POINTS) {
         if (m_rHDist > 0) {
            sprintf (distTxt, "%.2f", m_rHDist);
            m_DistanceText.SetWindowText((CString) distTxt);
            m_status.SetWindowText("Type of surface detected: " +
                                   thisProcess.m_distCalc.m_sShape);
         } else {
            m_DistanceText.SetWindowText("No Distance Available.");
            m_status.SetWindowText("No Points Detected.");
         }
         recall = 0;
      }
      else
      if (status == NOT_ENOUGH_POINTS) {
         if (!(m_rHDist > 0)) // might have previous horz dist
            m_rHDist = dist;
         // if m_rHDist > 0, means that we have horz dist, so maintain
         // current m_rHDist
         m_bAlt = TRUE;
         m_iLaserNum = 1;
      } else {
         if (m_rHDist > 0)
            dist = (dist*CAPHEIGHT +
                    m_rHDist*CAPWIDTH)/(CAPHEIGHT+CAPWIDTH);
         sprintf (distTxt, "%.2f", dist);
         m_DistanceText.SetWindowText((CString) distTxt);
         m_status.SetWindowText("Type of surface detected: " +
                                thisProcess.m_distCalc.m_sShape);
         recall = 0;
      }
   }

   if (recall)
      OnTimerNoSubtract();

}

////////////////////////////////////////////////////////////////////////
/////
// Simulation functions

void CLaserdistDlg::OnStartSim()
{
   // Start Timer
   // repaints capture window every _FRAMERATE_ seconds
   m_simSettings.DoModal();
   m_status.SetWindowText("Status: Simulation Running");
   int framerate = (int) 1000/m_simSettings.m_iSimFrameRate;

   // pass on horizontal laser info to CImageProcess
   if (m_simSettings.m_radioCheckedH)
      thisProcess.setSettings(&m_simSettings.m_pH);

   // pass on vertical laser info to CImageProcess
   if (m_simSettings.m_radioCheckedV)
      thisProcess.setSettings(&m_simSettings.m_pV);
   m_iPaintHorz = m_simSettings.m_radioCheckedH;
   m_iPaintVert = m_simSettings.m_radioCheckedV;
   m_iCurrOrientation = m_simSettings.m_radioCheckedH;
   m_bOn = TRUE;
   m_bAlt = FALSE;
   m_rHDist = 0.0;
   SetTimer (REFRESH_ID, framerate, NULL);

}

void CLaserdistDlg::OnTimer(UINT nIDEvent)
{
   // For software simulation
   int status;
   float dist = 0.0;
   char distTxt[10];
   if (m_bOn) {   // lasers on
      if (m_bAlt) {
//       EnableLasers(m_iLaserNum, NUM_LASERS, 1);
         thisProcess.AltCapture(m_iCurrOrientation, m_iLaserNum);
         m_iPaintHorz = m_iCurrOrientation;
         m_iPaintVert = !m_iCurrOrientation;
      } else{
//       EnableLasers(0, NUM_LASERS, 1);
      }
   thisProcess.PaintSimulatedCaptureWindow(m_iPaintHorz,m_iPaintVert);
      m_bOn = FALSE;
   } else { // lasers off
//    EnableLasers(0,NUM_LASERS,0);
      thisProcess.PaintOff();
      if (m_bAlt) {  // calculate distance laser by laser
         status = thisProcess.AltUpdateReadings(m_iCurrOrientation,
                                          m_iLaserNum++);
         if  (status == DONE) {
            dist = thisProcess.AltFindDistance(m_iCurrOrientation);
            if ((m_iCurrOrientation == HORZ) &&
                (m_simSettings.m_radioCheckedV)){
               // then try vertical
               m_iCurrOrientation = VERT;
               m_rHDist = dist;
               m_iLaserNum = 1;
               m_iPaintHorz = 0;
               m_iPaintVert = 1;
               m_bAlt = FALSE;
            } else { // currently vertical
               if (dist < 0) {
                  if (m_rHDist > 0) {
                     // already got a distance from horizontal, use
                     // that
                     sprintf (distTxt, "%.2f", m_rHDist);
                     m_DistanceText.SetWindowText((CString) distTxt);
                     m_status.SetWindowText("Type of surface
                      detected: " + thisProcess.m_distCalc.m_sShape);
                     if (m_bTest) {
                        fprintf (m_pStream, "%f\t", m_rHDist);
                        fprintf (m_pStream, "%% %s\n",
                                 thisProcess.m_distCalc.m_sShape);
                        m_bTestDone = TRUE;
                     }
                     // reset to initial settings
                     m_iLaserNum = 1;
               m_iPaintHorz = m_simSettings.m_radioCheckedH;
               m_iPaintVert = m_simSettings.m_radioCheckedV;
               m_bAlt = FALSE;
            } else {
               // tried every possible way to determine
               // distance
               AfxMessageBox("Cannot determine distance",
                             MB_OK, 0);
               OnStopSim();
            }
         } else { // found distance (less points) fine
            // average with m_rHDist
            if (m_rHDist > 0)
               dist = (dist*CAPHEIGHT +
                       m_rHDist*CAPWIDTH)/(CAPHEIGHT+CAPWIDTH);
            sprintf (distTxt, "%.2f", dist);
            m_iLaserNum = 1;
            m_DistanceText.SetWindowText((CString) distTxt);
            m_status.SetWindowText("Type of surface detected: " +
                               thisProcess.m_distCalc.m_sShape);
            // IF TESTING
            if (m_bTest) {
               fprintf (m_pStream, "%f\t", dist);
               fprintf (m_pStream, "%% %s\n",
                        thisProcess.m_distCalc.m_sShape);
               m_bTestDone = TRUE;
            }
            m_iPaintHorz = m_simSettings.m_radioCheckedH;
            m_iPaintVert = m_simSettings.m_radioCheckedV;
            m_bAlt = FALSE;
         }
      }
   }
} else { // enough points found
   status = thisProcess.simFindDistance(&m_iCurrOrientation,
                                        &dist);
   if (status == NO_POINTS) {
      if (m_rHDist > 0) {
         sprintf (distTxt, "%.2f", m_rHDist);
         m_DistanceText.SetWindowText((CString) distTxt);
         m_status.SetWindowText("Type of surface detected: " +
                              thisProcess.m_distCalc.m_sShape);
         // IF TESTING
         if (m_bTest) {
            fprintf (m_pStream, "%f\t", m_rHDist);
            fprintf (m_pStream, "%% %s\n",
                     thisProcess.m_distCalc.m_sShape);
            m_bTestDone = TRUE;
         }
      } else {
         m_DistanceText.SetWindowText("No Distance Available.");
         m_status.SetWindowText("No Points Detected.");
         // IF TESTING
         if (m_bTest) {
            fprintf (m_pStream, "%f\t", -1.0);
            fprintf (m_pStream, "%% %s\n", "N/A");
            m_bTestDone = TRUE;
         }
      }
      m_iPaintHorz = m_simSettings.m_radioCheckedH;
      m_iPaintVert = m_simSettings.m_radioCheckedV;
```

```
         }
         else
         if (status == NOT_ENOUGH_POINTS) {
            if (!(m_rHDist > 0))    // might have previous horz dist
               m_rHDist = dist;
            // if m_rHDist > 0, means that we have horz dist, so
            // maintain current m_rHDist
            m_bAlt = TRUE;
            m_iLaserNum = 1;
         } else {
            if (m_rHDist > 0)
               dist = (dist*CAPHEIGHT +
                       m_rHDist*CAPWIDTH)/(CAPHEIGHT+CAPWIDTH);
            sprintf (distTxt, "%.2f", dist);
            m_DistanceText.SetWindowText((CString) distTxt);
            m_status.SetWindowText("Type of surface detected: " +
                                    thisProcess.m_distCalc.m_sShape);
            // IF TESTING
            if (m_bTest) {
               fprintf (m_pStream, "%f\t", dist);
               fprintf (m_pStream, "%% %s\n",
                        thisProcess.m_distCalc.m_sShape);
               m_bTestDone = TRUE;
            }
            m_iPaintHorz = m_simSettings.m_radioCheckedH;
            m_iPaintVert = m_simSettings.m_radioCheckedV;
         }
      }
      m_bOn = TRUE;
   }
   if (!m_bTest)
   CDialog::OnTimer(nIDEvent);
}

void CLaserdistDlg::OnStopSim()
{
   KillTimer(REFRESH_ID);
   m_status.SetWindowText("Status: Simulation Stopped");
}


/////////////////////////////////////////////////////////////////////////
/////
// Testing functions

void CLaserdistDlg::OnStartSimTest(int do_horz, int do_vert) {
      // pass on horizontal laser info to CImageProcess
      if (do_horz)
         thisProcess.setSettings(&m_simSettings.m_pH);
      // pass on vertical laser info to CImageProcess
      if (do_vert)
         thisProcess.setSettings(&m_simSettings.m_pV);
      m_iPaintHorz = do_horz;
      m_iPaintVert = do_vert;
      m_iCurrOrientation = do_horz;
      m_bOn = TRUE;
      m_bAlt = FALSE;
      m_rHDist = 0.0;
      m_bTestDone = FALSE;
}
```

```
////////////////////////////////////////////////////////////////////////
/////
// Laser control functions

void CLaserdistDlg::InitLasers()
{
    i16 iStatus = 0;
    i16 iRetVal = 0;
    i16 iDevice = 1;
    i16 iPort = 0;
    i16 iMode = 0;
    i16 iDir = 1;
    i16 iIgnoreWarning = 0;

   /* Configure port as output. */
   iStatus = DIG_Prt_Config(iDevice, iPort, iMode, iDir);
   iRetVal = NIDAQErrorHandler(iStatus, "DIG_Prt_Config",
                               iIgnoreWarning);
}

void CLaserdistDlg::EnableLasers(int laser_num, int max, bool state)
{
   i16 iStatus = 0;
   i16 iRetVal = 0;
   i16 iDevice = 1;
   i16 iPort = 0;
   i16 iLine = 0;
   i16 iIgnoreWarning = 0;
   i32 iPattern = 0;

   if (state && (laser_num == 0)) {
      for (int i = 0; i < max; i++)
         iPattern += pow(2,i);
   } else {
      if (state && (laser_num != 0))
         iPattern += pow(2, laser_num-1);
   }

   iStatus = DIG_Out_Prt(iDevice, iPort, iPattern);
   iRetVal = NIDAQErrorHandler(iStatus, "DIG_Out_Prt", iIgnoreWarning);
}


void CLaserdistDlg::calibrate()
{
   float dist = 0.0, total=0.0;
   char distTxt[10];
   int status;
   int lnum;
   m_simSettings.ManualSetSettings(DO_HORZ, DO_VERT, 0,
                 0, NUM_LASERS, 0.0, 30.0, 0, 0, "",
                 0, NUM_LASERS, 0.0, 30.0, 0, 0, "");
   OnStartSimTest(DO_HORZ, DO_VERT);
   thisProcess.SetLaserMode(FALSE);
   for (lnum=1; lnum<NUM_LASERS+1; lnum++) {
      EnableLasers(lnum,1,1);
      Sleep(50);
      m_iCurrOrientation = HORZ;
      for (int i = 0; i<10; i++) {
         BOOL fResult = capGrabFrame(m_hCapWnd);
         // display image
         thisProcess.TestImageSubtract();
         thisProcess.AltCapture (m_iCurrOrientation, 1);
         thisProcess.AltCapture (m_iCurrOrientation, lnum);
         status = thisProcess.AltUpdateReadings(m_iCurrOrientation,
                                                lnum);
         dist = thisProcess.AltFindDistance(m_iCurrOrientation);
         if (dist < 0) {
            // tried every possible way to determine distance
            AfxMessageBox("Cannot determine distance", MB_OK, 0);
         } else { // found distance (less points) fine
            // average with m_rHDist
            sprintf (distTxt, "%.2f", dist);
            m_DistanceText.SetWindowText((CString) distTxt);
            m_status.SetWindowText("Type of surface detected: " +
                                    thisProcess.m_distCalc.m_sShape);
            // IF TESTING
            if (m_bTest) {
               fprintf (m_pStream, "%f\t", dist);
               fprintf (m_pStream, "%% %s\n",
                        thisProcess.m_distCalc.m_sShape);
               m_bTestDone = TRUE;
            }
         }
         Sleep(10);
      }
      EnableLasers(0,1,0);
   }
}
```

CBestFit Class

```
// BestFit.cpp: implementation of the CBestFit class.
// Functions to fit data to a polynomial
// D. Tran 2001
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "laserdist.h"
#include "BestFit.h"
#include <math.h>

//////////////////////////////////////////////////////////////////////
// matrix functions

// from Numerical Recipes in C
float ** CBestFit::matrix(int nrl, int nrh, int ncl, int nch)
{
   // Allocates a float matrix with range[nrl..nrh][ncl..nch]
   int i;
   float **m;

   // Allocate pointers to rows
   m=(float **) malloc((unsigned) (nrh-nrl+1)*sizeof(float*));
   if (!m) fprintf(stderr,"allocation failure in matrix()");
   m -= nrl;

   // Allocate rows and set pointers to them
   for (i=nrl; i<=nrh; i++) {
      m[i]=(float *)malloc ((unsigned) (nch-ncl+1)*sizeof(float));
      if (!m[i]) fprintf(stderr,"allocation failure 2 in matrix()");
      m[i] -= ncl;
   }

   // Return pointer to array of pointers to rows
   return m;
}

// from Numerical Recipes in C
float * CBestFit::vector(int nl, int nh)
{
   // Allocates a float vector with range [nl..nh]
   float *v;

   v=(float *)malloc((unsigned) (nh-nl+1)*sizeof(float));
   if (!v) fprintf(stderr,"allocation failure in vector()");
   return v-nl;
}

// from Numerical Recipes in C
int * CBestFit::ivector(int nl, int nh)
{
   //Allocates an int vector with range [n1..nh]
   int *v;
   v = (int *) malloc((unsigned) (nh-nl+1)*sizeof(int));
   if (!v) fprintf (stderr, "allocation failure in ivector()");
   return v-nl;
}

// from Numerical Recipes in C
void CBestFit::free_matrix(float **m, int nrl, int nrh, int ncl,
                           int nch)
{

   // Frees a matrix allocated with matrix
   int i;

   for(i=nrh;i>=nrl;i--) free((char *) (m[i]+ncl));
   free((char*) (m+nrl));
}

// from Numerical Recipes in C
void CBestFit::free_vector(float *v, int nl, int nh)
{
   // Frees a float vector allocated by vector()
   free((char *) (v+nl));
}

void CBestFit::free_ivector(int *v, int nl, int nh)
{
   // Frees an int vector allocated by ivector()
   free((char *) (v+nl));
}

//////////////////////////////////////////////////////////////////////
// best fit functions

// from Numerical Recipes in C
void fpoly(float x, float p[], int np)
{
   int j;
   p[1]=1.0;
   for (j=2; j<=np; j++) p[j] = p[j-1]*x;
}

// from Numerical Recipes in C
void CBestFit::lfit(float x[], float y[], float sig[], int ndata,
                    float a[], int ma, int lista[], int mfit,
                    float **covar, float *chisq,
                    void (*funcs)(float, float[], int))
{

   int k, kk, j, ihit, i;
   float ym, wt, sum, sig2i, **beta, *afunc;
   beta = matrix(1,ma,1,1);
   afunc = vector (1,ma);
   kk = mfit+1;
   for (j=1; j<=ma; j++) {
      ihit = 0;
      for (k=1; k<=mfit; k++)
         if (lista[k]==j) ihit++;
      if (ihit == 0)
         lista[kk++] = j;
      else
         if (ihit > 1) fprintf (stderr,"Bad LISTA permutation in
                          LFIT-1");
   }
   if (kk != (ma+1)) fprintf(stderr, "Bad LISTA permutation in LFIT-2");
   for (j=1; j<= mfit; j++) {
      for (k=1; k<=mfit; k++) covar[j][k] = 0.0;
      beta[j][1]=0.0;
   }
   for (i=1;i<=ndata;i++) {
      (*funcs) (x[i], afunc, ma);
      ym = y[i];
      if (mfit < ma)
         for (j=(mfit+1); j<=ma;j++)
            ym -= a[lista[j]]*afunc[lista[j]];
      sig2i=1.0/square(sig[i]);
      for (j=1; j<=mfit; j++) {
         wt = afunc[lista[j]]*sig2i;
         for (k=1; k<=j; k++)
            covar[j][k] += wt*afunc[lista[k]];
         beta[j][1] += ym*wt;
      }
   }
   if (mfit > 1)
      for (j=2; j<=mfit; j++)
         for (k=1; k<=j-1; k++)
            covar[k][j] = covar[j][k];
   gaussj(covar, mfit, beta, 1);
   for (j=1; j<=mfit; j++) a[lista[j]] = beta[j][1];
   *chisq=0.0;
   for (i=1; i<=ndata; i++) {
      (*funcs) (x[i], afunc, ma);
      for (sum=0.0, j=1; j<=ma; j++) sum += a[j]*afunc[j];
      *chisq += square((y[i]-sum)/sig[i]);
   }
   covsrt(covar, ma, lista, mfit);
   free_vector(afunc, 1, ma);
   free_matrix(beta,1,ma,1,1);
```

```
}

// from Numerical Recipes in C
void CBestFit::covsrt(float **covar, int ma, int lista[], int mfit)
{
   int i,j;
   float swap;

   for (j=1; j<ma; j++)
      for (i=j+1; i<=ma; i++) covar[i][j]=0.0;
   for (i=1; i<mfit; i++)
      for (j=i+1; j<=mfit; j++) {
         if (lista[j] > lista[i])
            covar[lista[j]][lista[i]]=covar[i][j];
         else
            covar[lista[i]][lista[j]]=covar[i][j];
      }
   swap = covar[1][1];
   for (j=1; j<=ma; j++) {
      covar[1][j] = covar[j][j];
      covar[j][j] = 0.0;
   }
   covar[lista[1]][lista[1]]=swap;
   for (j=2;j<=mfit;j++) covar[lista[j]][lista[j]]=covar[1][j];
   for (j=2;j<=ma;j++)
      for (i=1; i<=j-1; i++) covar[i][j]=covar[j][i];

}

#define SWAP(a,b) {float temp=(a); (a)=(b); (b)=temp;}

// from Numerical Recipes in C
void CBestFit::gaussj(float **a, int n, float **b, int m)
{
   int *indxc, *indxr, *ipiv;

   int i, icol, irow, j, k, l, ll;
   float big, dum, pivinv;

   indxc= ivector (1,n);
   indxr= ivector (1,n);
   ipiv= ivector (1,n);
   for(j=1;j<=n;j++) ipiv[j]=0;
   for (i=1; i<=n; i++) {
      big=0.0;
      for(j=1; j<=n; j++)
         if (ipiv[j] != 1)
            for (k=1; k<=n; k++) {
               if (ipiv[k] == 0) {
                  if (fabs(a[j][k]) >= big) {
                     big=fabs(a[j][k]);
                     irow=j;
                     icol=k;
                  }
               } else if (ipiv[k] > 1) fprintf (stderr,"GAUSSJ: Singular
                                                Matrix-1");
            }
      ++(ipiv[icol]);

      if(irow != icol) {
         for (l=1; l<=n; l++) SWAP(a[irow][l], a[icol][l]);
         for (l=1; l<=m; l++) SWAP(b[irow][l], b[icol][l]);
      }
      indxr[i]=irow;
      indxc[i]=icol;
      if (a[icol][icol] == 0.0) fprintf (stderr,"GAUSSJ: Singular
                                         Matrix-2");
      pivinv=1.0/a[icol][icol];
      a[icol][icol]=1.0;
      for (l=1;l<=n;l++) a[icol][l] *= pivinv;
      for (l=1;l<=m;l++) b[icol][l] *= pivinv;
      for (ll=1; ll<=n; ll++)
         if (ll != icol) {
            dum=a[ll][icol];
            a[ll][icol]=0.0;
            for (l=1; l<=n; l++) a[ll][l] -= a[icol][l]*dum;
            for (l=1; l<=m; l++) b[ll][l] -= b[icol][l]*dum;
         }
   }
   for (l=n; l>=1;l--) {
      if (indxr[l] != indxc[l])
         for (k=1; k<=n; k++)
            SWAP(a[k][indxr[l]],a[k][indxc[l]]);
   }
   free_ivector(ipiv,1,n);
   free_ivector(indxr,1,n);
   free_ivector(indxc,1,n);

}

//////////////////////////////////////////////////////////////////////
// calling functions

void CBestFit::getParams (int degree, int* fit_params)
{
   fit_params[0] = 0;
   for (int i=1; i<degree+2; i++) {
      fit_params[i] = i;
   }

}

void CBestFit::getCoeffs (int degree, float* coeffs)
{
   coeffs[0] = 0;

   for (int i=2; i<=degree; i++)
      coeffs[i] = 1;

   coeffs[1] = 1; // change init val?
   coeffs[degree+1] = 1;

}

float CBestFit::findBestFit (CList <PointInfo, PointInfo> *points, int
degree, float *coeffs)
{
   int numPoints = points->GetCount();
   POSITION pos = points->GetHeadPosition();
   FLOATCOORD tempCoord;

   float *x_data, *y_data;
   float **covar, chisq, *sig;
   int i;
   // degree of polynomial shouldn't be more than the number of points
   if (degree > numPoints)
      degree = numPoints;
   // only have the form a*x^degree + b
   int *lista = (int *) malloc ((degree+2)*sizeof(int));
   getParams(degree, lista);
   float *a= (float *) malloc ((degree+2)*sizeof(float));
   getCoeffs(degree, a);
   int mfit = degree+1;
   // fill in xdata and ydata
   // x_data and y_data are expected to be: x_data[1..numPoints];
   x_data = (float *) malloc ((numPoints+1)*sizeof(float));
   y_data = (float *) malloc ((numPoints+1)*sizeof(float));
   sig = (float *) malloc ((numPoints+1)*sizeof(float));
   x_data[0] = 0; y_data[0] = 0; sig[0]=1;
   for (i=1; i<numPoints+1; i++) {
      tempCoord = points->GetAt(pos).c;
      x_data[i] = tempCoord.X;
      y_data[i] = tempCoord.Y;
      // variance
      sig[i] = ((points->GetAt(pos).err_max-y_data[i]) +
                (y_data[i] - points->GetAt(pos).err_min))/2; //
   sig[i]=1;      // if uncertainity is unknown, sig[i]=1
      points->GetNext(pos);
   }
   int ma = degree+1;
   covar = (float **) malloc((ma+1)*sizeof(float *));
   for(i = 0; i < ma+1; i++)
         covar[i] = (float *)malloc((ma+1) * sizeof(float));
   lfit(x_data, y_data, sig, numPoints, a, ma, lista, mfit, covar,
&chisq,
        fpoly);

   // copy resulting coeffs
   if (coeffs)
      for (i=0; i<degree+1; i++) {
         coeffs[i] = a[i+1];
      }

   for(i = 0; i < ma+1; i++)
         free(covar[i]);
   free(sig);
   free(covar);
   free(a);
   free(lista);
   free(x_data);
   free(y_data);

   return chisq;
}
```

CDistCalc Class

```
// DistCalc.cpp: implementation of the CDistCalc class.
// Functions to calculate distance
// D. Tran 2001
//////////////////////////////////////////////////////////////////////
```

```
#include "stdafx.h"
#include <math.h>

#include "laserdist.h"
#include "DistCalc.h"
#include "bestfit.h"

float square (float x) {
   return x*x;
}

////////////////////////////////////////////////////////////////////////
// functions to create lists of distances from user specifications

void CDistCalc::getSimPointsGiven(CLaserSettings *ls, CList <float,
float> *distances)
{
   int numPoints = ls->m_iNumLasers;
   float dist;
   POSITION pos;
   int i;
   CList <float, float> distances2;

   pos = distances->GetHeadPosition();
   for (i=1; i<ls->m_iLaserNum;i++) {
      distances->GetNext(pos);
   }
   for (i=0; i<numPoints;i++) {
      dist = distances->GetAt(pos);
      distances2.AddTail(dist);
      distances->GetNext(pos);
   }

   convertDistToPixels(&distances2, ls);
}

void CDistCalc::getSimPointsStraight(CLaserSettings *ls)
{
   float dist = ls->m_rDist;
   int numPoints = ls->m_iNumLasers;
   int i;
   CList <float, float> distances;

   for (i=0; i<numPoints;i++) {
      distances.AddTail(dist);
   }

   convertDistToPixels(&distances, ls);
}


void CDistCalc::getSimPointsIncline(CLaserSettings *ls)
{
   float dist = ls->m_rDist;
   int numPoints = ls->m_iInitNumLasers;
   float angle = ls->m_rAngle;
   float dw = ls->m_rDW;


   float real_dist;
   int i;
   CList <float, float> distances;

   for (i=ls->m_iLaserNum; i<ls->m_iLaserNum+ls->m_iNumLasers; i++) {
      if (i < numPoints/2+1) {
         // get real distances
         real_dist = dist + (i-1)*dw*tan(angle);
         distances.AddTail(real_dist);
      }
      else {
         real_dist = dist + i*dw*tan(angle);
         distances.AddTail(real_dist);
      }
   }
   convertDistToPixels(&distances, ls);
}

void CDistCalc::getSimPointsRound(CLaserSettings *ls)
{
   // fills in distances in *points from right to left
   float dist = ls->m_rDist;
   int numPoints = ls->m_iInitNumLasers;
   float *coeffs = ls->m_pCoeffs;
   int degree = ls->m_iDegree;
   float shift = ls->m_rShift;
   float dw = ls->m_rDW;

   float real_dist, x;
   int i,j;
   CList <float, float> distances;

   for (i=ls->m_iLaserNum; i<ls->m_iLaserNum+ls->m_iNumLasers; i++) {
      if (i < numPoints/2+1) {
         // get real distances
         x = (i-1)*dw;
         real_dist = 0.0;
         // y = coeffs[0]*x^0 + coeffs[1]*x^1+...coeffs[degree]*x^degree
         for (j=0; j <= degree; j++)
            real_dist = real_dist + coeffs[j]*powf(x-shift,j);
         distances.AddTail(real_dist);
      }

      else {
         x = i*dw;
         real_dist = 0.0;
         // y = coeffs[0]*x^0 + coeffs[1]*x^1+...coeffs[degree]*x^degree
         for (j=0; j <= degree; j++)
            real_dist = real_dist + coeffs[j]*powf(x-shift,j);
         distances.AddTail(real_dist);
      }
   }

   convertDistToPixels(&distances, ls);

}

//////////////////////////////////////////////////////////////////////
// conversion functions

void CDistCalc::convertDistToPixels(CList <float, float> *distances,
CLaserSettings *ls)
{
   // expects distances from left laser to right laser - therefore will
   // be calculating pixels from right pixel to left pixel
   float dist, d, p;
   int i;
   FLOATCOORD tempCoord;
   CList <FLOATCOORD, FLOATCOORD> *pixels = &ls->points;
   int numPoints = ls->m_iInitNumLasers;
   int bHorizontal = ls->m_bHorizontal;
   float dw = ls->m_rDW;
   float real_dia;

   POSITION pos = distances->GetHeadPosition();

   for (i=ls->m_iLaserNum; i<ls->m_iLaserNum+distances->GetCount(); i++)
{
      if (i < numPoints/2+1) {
         dist = distances->GetAt(pos);
         // find the distance of this spot from the center of the array
         d = FL*(numPoints/2-i+1)*dw/dist;
         // now convert mm to pixels
         if (bHorizontal) {
            p = CAPWIDTH/2 - d*CAPWIDTH/SENSE_WIDTH;  // + if reversed
         // add to points list
            tempCoord.X = p;
            tempCoord.Y = CAPHEIGHT/2;
            }
         else {
            p = CAPHEIGHT/2 - d*CAPHEIGHT/SENSE_HEIGHT;// + if reversed
         // add to points list
            tempCoord.X = CAPWIDTH/2;
            tempCoord.Y = p;
            }
         pixels->AddTail(tempCoord);
         // find spot size
         real_dia = findSpotSize(dist);
         // transform to image distance
         real_dia = real_dia*FL/dist;
         // convert to pixels
         ls->m_pSpotDiams.AddTail(real_dia*ls->m_iMaxDP/ls->m_rMaxDX);

         distances->GetNext(pos);
      } else {
         dist = distances->GetAt(pos);
         // find the distance of this spot from the center of the array
         d = FL*(i-numPoints/2)*dw/dist;
         // now convert mm to pixels
         if (bHorizontal) {
            p = CAPWIDTH/2 + d*CAPWIDTH/SENSE_WIDTH;  // - if reversed
         // add to points list
            tempCoord.X = p;
            tempCoord.Y = CAPHEIGHT/2;
         } else {
            p = CAPHEIGHT/2 + d*CAPHEIGHT/SENSE_HEIGHT;// - if reversed
         // add to points list
            tempCoord.X = CAPWIDTH/2;
            tempCoord.Y = p;
         }
         pixels->AddTail(tempCoord);
         // find spot size
         real_dia = findSpotSize(dist);
         // transform to image distance
         real_dia = real_dia*FL/dist;
         // convert to pixels
         ls->m_pSpotDiams.AddTail(real_dia*ls->m_iMaxDP/ls->m_rMaxDX);

         distances->GetNext(pos);
      }
   }

}

void CDistCalc::convertPixelToDist (float *x, float *y, int numPoints,
float p_pos, float dw, int laserNum, int maxDP,
                    float maxDX, float err)
{
   float x_pos;

/* // for reversed
   if (laserNum < numPoints/2+1) {
      // transform pixel points to distance on CCD array
      x_pos = (p_pos - maxDP/2 + err) * maxDX/maxDP;
      // extrapolate real distance, assume evenly spaced lasers from
center
      *y = FL*(numPoints/2 - laserNum+1)*dw/x_pos;
      *x = (laserNum-1)*dw;
   }
   else {
      // transform pixel points to distance on CCD array
      x_pos = (maxDP/2 - p_pos + err) * maxDX/maxDP;
      // extrapolate real distance, assume evenly spaced lasers from
center
      *y = FL*(laserNum-numPoints/2)*dw/x_pos;
      *x = laserNum*dw;
   }
*/
   // for non-reversed
   if (laserNum < numPoints/2+1) {
      // transform pixel points to distance on CCD array
      x_pos = (maxDP/2 - p_pos + err) * maxDX/maxDP;
      // extrapolate real distance, assume evenly spaced lasers from
      // center
      *y = FL*(numPoints/2 - laserNum+1)*dw/x_pos;
      *x = (laserNum-1)*dw;
   }
   else {
      // transform pixel points to distance on CCD array
      x_pos = (p_pos - maxDP/2 + err) * maxDX/maxDP;
      // extrapolate real distance, assume evenly spaced lasers from
      // center
      *y = FL*(laserNum-numPoints/2)*dw/x_pos;
      *x = laserNum*dw;
   }


}

//////////////////////////////////////////////////////////////////////
// distance and shape estimation functions

float CDistCalc::findSpotSize(float dist)
{
   float spot_diam = dist*1e-3*BEAM_DIVERGENCE + INI_BEAM_DIAM;
   return spot_diam;

}

float CDistCalc::getBestEstimate(CList<PointInfo, PointInfo> *points,
int degree, float dw, int numLasers)
{

   float dist;
   char angle[20] = "";
   char sDegree[5] = "";
   float *coeffs = (float *) malloc ((degree+1)*sizeof (float));
   if (isIncline (points, coeffs)) {
      if (fabs(coeffs[1]) <0.0001) {
         m_sShape = "Flat, Perpendicular";
         dist = coeffs[0];
         free (coeffs);
         return (dist);
      }
      // if incline, return closest distance
      else {
         m_sShape = " Incline";
         sprintf(angle, "%.4f", atan(coeffs[1])*180/PI);
         m_sShape = angle + m_sShape;
         dist = getMinimumInclineDist(coeffs, (numLasers+1)*dw);
         free (coeffs);
         return dist;
      }
   }
   int final_degree;
   if (isRound(points, degree, coeffs, &final_degree)) {
      m_sShape = "Round";
      sprintf(sDegree, " %d", final_degree);
      m_sShape = m_sShape + sDegree;
      dist = getRoundDist(points, final_degree,coeffs, dw);
      free (coeffs);
      return dist;
   }
   free (coeffs);
   m_sShape = "Unknown";
   return getMinimumDist(points);
}

float CDistCalc::getDist(CLaserSettings *ls)
{
   CList <FLOATCOORD, FLOATCOORD> *points = &ls->points;
   CList <PointInfo, PointInfo> realPoints;
   float p_pos;
   float val;
   int i, numPoints;
   PointInfo pi;

   if (points->IsEmpty()) {
      AfxMessageBox ("Error, no points to calculate distance", MB_OK,
                      0);
      return 0;
   }

   // points in list are from order of left pixel to right pixel,
   // therefore the laser's results are from right laser to left laser
   numPoints = points->GetCount();
   POSITION pos = points->GetHeadPosition( );

   // transform pixel points to distances
   // realPoints -> Y values are distance, X values are positions of
   // lasers
// for (i=numPoints; i>0; i--)      // (for reversed)
   for (i=1; i<numPoints+1; i++)    // i corresponds to the laser number
   {
      p_pos = (points->GetAt(pos)).X;
      convertPixelToDist(&pi.c.X, &pi.c.Y, numPoints,p_pos,ls->m_rDW, i,
                         ls->m_iMaxDP, ls->m_rMaxDX, 0);
      // provide error information for each pixel
      convertPixelToDist(&val, &pi.err_max, numPoints,p_pos,ls->m_rDW,i,
                         ls->m_iMaxDP, ls->m_rMaxDX, -0.5);//
      convertPixelToDist(&val, &pi.err_min, numPoints,p_pos,ls->m_rDW,i,
                         ls->m_iMaxDP, ls->m_rMaxDX, 0.5);//
      realPoints.AddTail(pi);
      points->GetNext(pos);
   }

   return getBestEstimate(&realPoints, ls->m_iNumLasers, ls->m_rDW,
                          ls->m_iInitNumLasers);
}


bool CDistCalc::isIncline(CList <PointInfo, PointInfo> *points,
                          float *coeffs)
{

   CBestFit bf;
   float chisq;

   int numPoints = points->GetCount();
   chisq = bf.findBestFit(points,1, coeffs);
   if (chisq < numPoints-1) // chisq should be less than #of bins-1
// if (chisq < 181)      // for less strict cases, e.g. inclines
      return TRUE;
   else
      return FALSE;

}

bool CDistCalc::isRound(CList <PointInfo, PointInfo> *points,
                        int degree, float *coeffs, int *final_degree)
{

   CBestFit bf;
   float fit;

   int numPoints = points->GetCount();
   for (int i=2; i<= degree; i++) {
      fit = bf.findBestFit(points, i, coeffs);
      if (fit < numPoints-1) { // chisq should be less than #of bins-1
         *final_degree = i;
         return TRUE;
      }
   }
   return FALSE;
}
float CDistCalc::getAverageDist (CList<PointInfo, PointInfo> *points) {
   int i;
   float total = 0.0;
   int numPoints = points->GetCount();
   POSITION pos = points->GetHeadPosition( );

   for (i=0; i< numPoints; i++)
   {
      total = total + points->GetAt(pos).c.Y;
      points->GetNext(pos);
   }

   return (total/numPoints);
}

float CDistCalc::getMinimumDist (CList<PointInfo, PointInfo> *points) {
   int i;
   int numPoints = points->GetCount();
   POSITION pos = points->GetHeadPosition( );

   float min = points->GetAt(pos).c.Y;
   points->GetNext(pos);
   for (i=1; i<numPoints; i++) {
      if (points->GetAt(pos).c.Y < min)
         min = points->GetAt(pos).c.Y;
      points->GetNext(pos);
   }
   return min;
}

float CDistCalc::getMinimumInclineDist (float *coeffs, float max_x) {

   float min_dist = coeffs[0]+coeffs[1]*max_x;
   if (coeffs[0] < min_dist)
      return coeffs[0];
   else return min_dist;
}

float CDistCalc::getRoundDist (CList<PointInfo, PointInfo> *points,
                               int degree, float *coeffs, float dw)
{

   // use brute force to determine closest point (probably more faster
   // than using some algorithm to determine point of concavity)

   float real_dist, min_dist;
   int x, j;

   min_dist = 0.0;
   x=0;
   for (j=0; j <= degree; j++)
      min_dist = min_dist + coeffs[j]*powf(x,j);
   int spaces = points->GetCount();
   for (x=1; x<spaces*dw; x++) {
      // calculate distance using coeffs
      real_dist = 0.0;
      // y = coeffs[0]*x^0 + coeffs[1]*x^1+...coeffs[degree]*x^degree
      for (j=0; j <= degree; j++)
         real_dist = real_dist + coeffs[j]*powf(x,j);
      if (real_dist < min_dist)
         min_dist = real_dist;
```

```
    }
    return min_dist;

}
```

CImageProcess Class

```
// ImageProcess.cpp : implementation file
// Functions to analyze captured data and display images
// D. Tran 2001
//////////////////////////////////////////////////////////////////////


#include "stdafx.h"
#include "laserdist.h"
#include "ImageProcess.h"
#include "distcalc.h"
#include <math.h>


LRESULT PASCAL vidCallbackProc(HWND hWnd, LPVIDEOHDR lpVHdr);
LRESULT PASCAL frameCallbackProc(HWND hWnd, LPVIDEOHDR lpVHdr);
LPVIDEOHDR ptrToOnBuf;
LPVIDEOHDR ptrToOffBuf;
bool gOn;

/////////////////////////////////////////////////////////////////////////
/////
// CImageProcess

CImageProcess::CImageProcess()
{

    // Initialize BITMAP for Simulation

    // Fill in the BITMAPINFOHEADER
    m_bmpInfo = (LPBITMAPINFO) new BYTE[sizeof(BITMAPINFOHEADER) +
                                        sizeof(RGBQUAD)];
    m_bmpInfo->bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
    m_bmpInfo->bmiHeader.biWidth = CAPWIDTH;
    m_bmpInfo->bmiHeader.biHeight = CAPHEIGHT;
    m_bmpInfo->bmiHeader.biPlanes = 1;
    m_bmpInfo->bmiHeader.biBitCount = 24;
    m_bmpInfo->bmiHeader.biCompression = 0;
    m_bmpInfo->bmiHeader.biSizeImage = 3*CAPWIDTH*CAPHEIGHT;
    m_bmpInfo->bmiHeader.biXPelsPerMeter = 0;
    m_bmpInfo->bmiHeader.biYPelsPerMeter = 0;
    m_bmpInfo->bmiHeader.biClrUsed = 0;
    m_bmpInfo->bmiHeader.biClrImportant = 0;

    // Fill in color information for the bitmap
    m_bmpInfo->bmiColors[0].rgbBlue = 255;
    m_bmpInfo->bmiColors[0].rgbGreen = 255;
    m_bmpInfo->bmiColors[0].rgbRed = 255;
    m_bmpInfo->bmiColors[0].rgbReserved = 0;

    m_lpVHdr = (videohdr_tag *) malloc (sizeof (videohdr_tag));
    m_lpVHdr->lpData = (unsigned char *) malloc
                    (3*CAPWIDTH*CAPHEIGHT*sizeof (unsigned char));
```

```
   m_lpOnBuffer = (videohdr_tag *) malloc (sizeof (videohdr_tag));
   m_lpOnBuffer->lpData = (unsigned char *) malloc
                         (3*CAPWIDTH*CAPHEIGHT*sizeof (unsigned char));

   m_lpOffBuffer = (videohdr_tag *) malloc (sizeof (videohdr_tag));
   m_lpOffBuffer->lpData = (unsigned char *) malloc
                        (3*CAPWIDTH*CAPHEIGHT*sizeof (unsigned char));

   // set globals
   ptrToOnBuf = m_lpOnBuffer;
   ptrToOffBuf = m_lpOffBuffer;
   gOn = TRUE;

}

CImageProcess::~CImageProcess()
{
   delete m_bmpInfo;
   free (m_lpVHdr->lpData);
   free (m_lpVHdr);
   free (m_lpOnBuffer->lpData);
   free (m_lpOnBuffer);
   free (m_lpOffBuffer->lpData);
   free (m_lpOffBuffer);
}


///////////////////////////////////////////////////////////////////////
/////
// CImageProcess message handlers
// create unvisible child window at (x,y)
HWND CImageProcess::Create(
    HWND hwndParent,
    int x,
    int y
){

    static BOOL bInitDone = FALSE;
   // Create a capture window
    m_hWndCap = capCreateCaptureWindow(
                    NULL,
                    WS_CHILD | WS_VISIBLE,
                    x, y, 640, 480,
                    hwndParent,         // parent window
                    1                   // child window id
              );


    if (m_hWndCap == NULL) {
        return(NULL);
   }
   BOOL bError = capDriverConnect(m_hWndCap, 0);

   if (!bError)
      AfxMessageBox ("Can't connect to capture driver", MB_OK, 0);

   CAPDRIVERCAPS gCapDriverCaps;
   CAPSTATUS gCapStatus;

   // Get the capabilities of the capture driver
   capDriverGetCaps(m_hWndCap, &gCapDriverCaps, sizeof(CAPDRIVERCAPS));

   // Get the settings for the capture window
   capGetStatus(m_hWndCap, &gCapStatus , sizeof(gCapStatus));

   // initialize capture
   if (!capDlgVideoFormat(m_hWndCap))
      AfxMessageBox("Can't Set Video Format", MB_OK, 0);

   capSetCallbackOnFrame(m_hWndCap, frameCallbackProc);

    return(m_hWndCap);
}

LRESULT PASCAL frameCallbackProc(HWND hWnd, LPVIDEOHDR lpVHdr)
{

   // copy to buffer
   if (gOn)
      memcpy (ptrToOnBuf->lpData, lpVHdr->lpData, CAPWIDTH*CAPHEIGHT*3);
   else {
      memcpy (ptrToOffBuf->lpData, lpVHdr->lpData,
              CAPWIDTH*CAPHEIGHT*3);
   }

   return 0;
}

int CImageProcess::Round (float val)  // also included in distcalc.cpp
{
   if (val < 0)
      return ((int) (val-0.5));
   else return (int) (val+0.5);
}

float CImageProcess::square (float x) {
   return x*x;
}


///////////////////////////////////////////////////////////////////////////
/////
// structure functions

void CImageProcess::setSettings(CLaserSettings *ls)
{
   if (ls->m_bHorizontal) {
      m_pHorzDraw.Copy(ls);
      m_pHorz.CopyBasics(ls);
   }
   if (!ls->m_bHorizontal) {
      m_pVertDraw.Copy(ls);
      m_pVert.CopyBasics(ls);
   }
}

void CImageProcess::SetLaserMode(bool on)
{
   gOn = on;
}
```

```
void CImageProcess::AltCapture (int orientation, int laserNum)
{
   CLaserSettings *ls, *ls2;
   if (orientation == HORZ) {
      ls = &m_pHorzDraw;
      ls2 = &m_pHorz;
   }
   else {
      ls = &m_pVertDraw;
      ls2 = &m_pVert;
   }

   if (laserNum == 1) { // initialize alternate capturing
      ls->m_iInitNumLasers =ls->m_iNumLasers;
      ls->m_iNumLasers = 1;
      if (!ls2->realPoints.IsEmpty())
         ls2->realPoints.RemoveAll();
   }
   if (ls->points.GetCount() > 0)
      ls->points.RemoveAll();
   if (ls->m_pSpotDiams.GetCount() > 0)
      ls->m_pSpotDiams.RemoveAll();

   ls->m_iLaserNum = laserNum;
}

int CImageProcess::AltUpdateReadings(int orientation, int laserNum) {
   CLaserSettings *ls, *lsDraw;
   int status;
   float val, p_pos;
   PointInfo pi;

   if (orientation == HORZ) {
      ls = &m_pHorz;
      lsDraw = &m_pHorzDraw;
   } else {
      ls = &m_pVert;
      lsDraw = &m_pVertDraw;
   }
   // for calibration recording
   FILE *stream = fopen("fl_calib.dat", "a+");
   extractPoints();
   if (filterPoints() != -1) {
      // similar to CDistCalc::getDist
      // add realPoints to ls
      // add new point
      POSITION pos = ls->points.GetHeadPosition();
      p_pos = ls->points.GetAt(pos).X;
       fprintf(stream, "%f\t", p_pos);    // for calibration recording
       fclose(stream);                    // for calibration recording
      m_distCalc.convertPixelToDist(&pi.c.X, &pi.c.Y, ls->m_iNumLasers,
                                    p_pos,ls->m_rDW,laserNum,
                                    ls->m_iMaxDP, ls->m_rMaxDX, 0);
      // provide error information for each pixel
      m_distCalc.convertPixelToDist(&val, &pi.err_max, ls->m_iNumLasers,
                                    p_pos,ls->m_rDW,laserNum,
                                    ls->m_iMaxDP, ls->m_rMaxDX, -0.5);
      m_distCalc.convertPixelToDist(&val, &pi.err_min, ls->m_iNumLasers,
                                    p_pos,ls->m_rDW,laserNum,
                                    ls->m_iMaxDP, ls->m_rMaxDX, 0.5);
      ls->realPoints.AddHead(pi);
      ls->points.RemoveAll();
      status = 0;
   } else
      status = -1;

   if (laserNum == ls->m_iNumLasers) {
      ls->m_iLaserNum = 1;    // reset to initial value
      ls->m_iNumLasers = ls->m_iInitNumLasers;
      lsDraw->m_iLaserNum = 1;
      lsDraw->m_iNumLasers = ls->m_iInitNumLasers;
      status = DONE;
   }

   return status;
}

float CImageProcess::AltFindDistance(int orientation) {
   CLaserSettings *ls;
   if (orientation == HORZ)
      ls = &m_pHorz;
   else
      ls = &m_pVert;
   if (ls->realPoints.IsEmpty())
      return -1;
   return m_distCalc.getBestEstimate (&ls->realPoints,
         ls->realPoints.GetCount(), ls->m_rDW, ls->m_iInitNumLasers);
}

/////////////////////////////////////////////////////////////////////////
/////
// paint to window functions

void CImageProcess::PaintWindow(HWND hwnd, LPVIDEOHDR lpVHdr)
{

   ASSERT(m_bmpInfo);
   if(lpVHdr)
   {
      HDC dc = ::GetDC(m_hWndCap);
      StretchDIBits(dc,
               0,
               0,
               CAPWIDTH,
               CAPHEIGHT,
               0,
               0,
               ((LPBITMAPINFOHEADER)m_bmpInfo)->biWidth,
               ((LPBITMAPINFOHEADER)m_bmpInfo)->biHeight,
               (CONST VOID *)lpVHdr->lpData,
               m_bmpInfo,
               DIB_RGB_COLORS,
               SRCCOPY );

      ::ReleaseDC(m_hWndCap,  (HDC) dc);
      ::ValidateRect(m_hWndCap, NULL);
   }

}

void CImageProcess::PaintSimulatedCaptureWindow(int doHorz, int doVert)
{
   unsigned char *ptr;
   ptr = m_lpVHdr->lpData;

   // make background black
   for (int i=0; i < CAPWIDTH*CAPHEIGHT; i++) {
      *ptr = 0;
      ptr++;
      *ptr = 0;
      ptr++;
      *ptr = 0;
      ptr++;
   }

   // do calculations
   if (doHorz) {
      switch (m_pHorzDraw.m_iSurfaceMode) {
         case 0:
            m_distCalc.getSimPointsStraight(&m_pHorzDraw);
            break;
         case 1:
            m_distCalc.getSimPointsIncline(&m_pHorzDraw);
            break;
         case 2:
            m_distCalc.getSimPointsRound(&m_pHorzDraw);
            break;
         default:
            m_distCalc.getSimPointsStraight(&m_pHorzDraw);
            break;
      }

      // draw spots
      putSpots(&m_pHorzDraw.points, &m_pHorzDraw.m_pSpotDiams);

   }

   if (doVert) {
      switch (m_pVertDraw.m_iSurfaceMode) {
         case 0:
            m_distCalc.getSimPointsStraight(&m_pVertDraw);
            break;
         case 1:
            m_distCalc.getSimPointsIncline(&m_pVertDraw);
            break;
         case 2:
            m_distCalc.getSimPointsRound(&m_pVertDraw);
            break;
         default:
            m_distCalc.getSimPointsStraight(&m_pVertDraw);
            break;
      }
      putSpots(&m_pVertDraw.points, &m_pVertDraw.m_pSpotDiams);

   }

   memcpy (m_lpOnBuffer->lpData, m_lpVHdr->lpData,
           CAPWIDTH*CAPHEIGHT*3);
   // paint window
   PaintWindow(m_hWndCap, m_lpVHdr);
}
```

```
void CImageProcess::PaintOff()
{

   unsigned char *ptr;
   int i;
   ptr = m_lpVHdr->lpData;

   // make background black
   for (i=0; i < CAPWIDTH*CAPHEIGHT; i++) {
      *ptr = 0;
      ptr++;
      *ptr = 0;
      ptr++;
      *ptr = 0;
      ptr++;
   }

   memcpy (m_lpOffBuffer->lpData, m_lpVHdr->lpData,
           CAPWIDTH*CAPHEIGHT*3);

   ImageSubtract();

   PaintWindow(m_hWndCap, m_lpVHdr);

}

void CImageProcess::putSpots(CList <FLOATCOORD, FLOATCOORD> *points,
CList <float, float> *diams)
{

   FLOATCOORD tempCoord;
   POSITION pos, pos2;
   unsigned char *ptr;
   int i;
   float diam;

   ptr = m_lpVHdr->lpData;
   pos = points->GetHeadPosition( );
   pos2 = diams->GetHeadPosition( );
   // add laser spots
   for (i=0; i<points->GetCount(); i++) {
      tempCoord = points->GetAt(pos);
      diam = diams->GetAt(pos2);
      PaintSpot(tempCoord, diam, ptr);
      points->GetNext(pos);
      diams->GetNext(pos2);
   }
}

void CImageProcess::PaintSpot(FLOATCOORD center, float diam, unsigned
                              char *ptr) {
   // ptr points to the head of the buffer
   unsigned char *tempPtr;
   float r_sq, beam_rad_sq, i_norm;
   beam_rad_sq = (diam/2)*(diam/2);
   int pixel_val;

   for (int y=center.Y-ceil(diam/2); y < center.Y+ceil(diam/2); y++) {
         for (int x=center.X-ceil(diam/2); x < center.X+ceil(diam/2);
            x++) {
            r_sq = square(x-center.X) + square(y-center.Y);
            i_norm = exp(-2*r_sq/beam_rad_sq);
            pixel_val = Round(i_norm*MAX_INTENS_PIX_VAL);
            if ((x > -1) && (x < CAPWIDTH) &&
               (y > -1) && (y < CAPHEIGHT)) {
               tempPtr = ptr + y*CAPWIDTH*3 + x*3;
               *tempPtr = pixel_val; tempPtr++;
               *tempPtr = pixel_val; tempPtr++;
               *tempPtr = pixel_val; tempPtr++;
            }
         }
      }

}

void CImageProcess::TestImageSubtract()
{
   PaintWindow(m_hWndCap, m_lpOffBuffer);
}

///////////////////////////////////////////////////////////////////////
/////
// image processing

float CImageProcess::getThreshold()
{
   return 50;
}

void CImageProcess::ImageSubtract()
{
   // perform image subtraction
   for (int i=0; i < CAPWIDTH*CAPHEIGHT*3; i++) {
      m_lpOffBuffer->lpData[i] = m_lpOnBuffer->lpData[i] -
                                 m_lpOffBuffer->lpData[i];
   }

}

void CImageProcess::extractPoints()
{

   int i,j;

   // get points that exceed the threshold
   unsigned char *ptr;
   float threshold;
   unsigned char b_val, g_val, r_val;
   COORD tempCoord;

   threshold = getThreshold();
   ptr = m_lpOffBuffer->lpData;
   if (!m_extractedPoints.IsEmpty())
      m_extractedPoints.RemoveAll();
   if (!m_extractedVals.IsEmpty())
      m_extractedVals.RemoveAll();
   for (i=0; i<CAPHEIGHT; i++) {
      for (j=0; j<CAPWIDTH; j++) {
         b_val = *ptr;        ptr++;
         g_val = *ptr;        ptr++;
         r_val = *ptr;        ptr++;
         if ((b_val > threshold) && (g_val > threshold) &&
             (r_val > threshold)) {
            // add i, j to list
            tempCoord.X = j-X_OFFSET;
            tempCoord.Y = i;
            m_extractedPoints.AddTail(tempCoord);
            m_extractedVals.AddTail ((b_val+g_val+r_val)/3);
         }
      }
   }
}

int CImageProcess::filterPoints()
{
   COORD tempCoord, lastCoordH, lastCoordV;
   int numPoints, i, totalpoints;
   totalpoints = 0;
   FLOATCOORD tempFloatCoord;
   // for centroid calculation
   float horz_centroid_sum = 0.0;
   int horz_centroid_points = 0;
   float vert_centroid_sum = 0.0;
   int vert_centroid_points = 0;
   float tempVal, lastVal;

   if (m_extractedPoints.IsEmpty()) {
      return -1;
   }
   numPoints = m_extractedPoints.GetCount();
   POSITION pos = m_extractedPoints.GetHeadPosition( );
   POSITION pos2 = m_extractedVals.GetHeadPosition( );
   lastCoordH.X = 0;
   lastCoordH.Y = 0;
   lastCoordV.X = 0;
   lastCoordV.Y = 0;

   if (!m_pVert.points.IsEmpty())
      m_pVert.points.RemoveAll();
   if (!m_pHorz.points.IsEmpty())
      m_pHorz.points.RemoveAll();
   for (i=0; i< m_extractedPoints.GetCount(); i++) {
      // enter some filtering function here
      // try to find the next set of points that are sequential, and
      // find the center of it
      tempCoord = m_extractedPoints.GetAt(pos);
      tempVal = m_extractedVals.GetAt(pos2);
      // horizontal point?
      if (((tempCoord.X > CENTER_X+2) || (tempCoord.X < CENTER_X-2)) &&
         (tempCoord.Y == CENTER_Y) &&
         (tempCoord.X > 0)) {
         // if point is next to a previous point, delete it from the
         // list of points and add it to tempPoints, the centroid of
         // which will later be added
         if ((tempCoord.X - lastCoordH.X == 1) &&
             (tempCoord.Y == lastCoordH.Y)) {
            // if lastpoint was added to the final list of points,
            // delete it
            // add it to the running sum
            if (!m_pHorz.points.IsEmpty()) {
               if (m_pHorz.points.GetTail().X == lastCoordH.X) {
                  m_pHorz.points.RemoveTail();
                  totalpoints--;
            horz_centroid_sum += lastCoordH.X*lastVal;
            horz_centroid_points+= lastVal;
         }
      }
      // add current to the running sum
      horz_centroid_sum += tempCoord.X*tempVal;
      horz_centroid_points+=tempVal;
   }
   else {
      // if there was a set of points previously collected,
      // process it
      if (horz_centroid_points > 0) {
         tempFloatCoord.X =
                         horz_centroid_sum/horz_centroid_points;
         tempFloatCoord.Y = lastCoordH.Y;
         m_pHorz.points.AddTail(tempFloatCoord);
         totalpoints++;
         horz_centroid_sum = 0.0;
         horz_centroid_points = 0;
      }
      // add current point as well
      tempFloatCoord.X = tempCoord.X;
      tempFloatCoord.Y = tempCoord.Y;
      m_pHorz.points.AddTail(tempFloatCoord);
      totalpoints++;
   }
   lastCoordH = tempCoord;
}

// vertical point?
if (((tempCoord.Y > CENTER_Y+2) || (tempCoord.Y < CENTER_Y-2)) &&
      (tempCoord.X == CENTER_X)) {
   // if point is next to a previous point, delete it from the
   // list of points and add it to tempPoints, the centroid of
   // which will later be added
   if ((tempCoord.Y - lastCoordV.Y == 1) &&
       (tempCoord.X == lastCoordV.X)) {
      // if lastpoint was added to the final list of points,
      // delete it
      // add it to the running sum
      if (!m_pVert.points.IsEmpty()) {
         if (m_pVert.points.GetTail().X == lastCoordV.Y) {
            // get .X rather than .Y cuz points are swapped in
            // m_pVert.points
            m_pVert.points.RemoveTail();
            totalpoints--;
            vert_centroid_sum += lastCoordV.Y*lastVal;
            vert_centroid_points+=lastVal;
         }
      }
      // add current to the running sum
      vert_centroid_sum += tempCoord.Y*tempVal;
      vert_centroid_points+=tempVal;
   }
   else {
      // if there was a set of points previously collected,
      // process it
      if (vert_centroid_points > 0) {
         // Since distance finding functions expects points in the
         // .X field, swap X and Y for vertical points
         tempFloatCoord.X =
                               vert_centroid_sum/vert_centroid_points;
               tempFloatCoord.Y = lastCoordV.X;
               m_pVert.points.AddTail(tempFloatCoord);
               totalpoints++;
               vert_centroid_sum = 0.0;
               vert_centroid_points = 0;
            }
            // add current point as well
            // Since distance finding functions expects points in the .X
            // field, swap X and Y for vertical points
            tempFloatCoord.X = tempCoord.Y;
            tempFloatCoord.Y = tempCoord.X;
            m_pVert.points.AddTail(tempFloatCoord);
            totalpoints++;
         }
         lastCoordV = tempCoord;
      }

      m_extractedPoints.GetNext(pos);
      m_extractedVals.GetNext(pos2);
      lastVal = tempVal;
   }

   if (horz_centroid_points > 0) {
      tempFloatCoord.X = horz_centroid_sum/horz_centroid_points;
      tempFloatCoord.Y = lastCoordH.Y;
      m_pHorz.points.AddTail(tempFloatCoord);
      totalpoints++;
   }

   if (vert_centroid_points > 0) {
      tempFloatCoord.X = vert_centroid_sum/vert_centroid_points;
      tempFloatCoord.Y = lastCoordV.X;
      m_pVert.points.AddTail(tempFloatCoord);
      totalpoints++;
   }
   if (totalpoints == 0)
      return -1;
   return totalpoints;
}


int CImageProcess::simFindDistance(int *currOrientation, float *dist)
{
   *dist = 0.0;
   float total = 0;

   extractPoints();
   if (filterPoints() == -1)
      return NO_POINTS;
   if (m_pHorz.points.GetCount() > 0) {
      *currOrientation = HORZ;
      if ((m_pHorz.points.GetCount() < m_pHorz.m_iNumLasers) ||
         (m_pHorz.points.GetCount() > m_pHorz.m_iNumLasers))      {
         return NOT_ENOUGH_POINTS;
      }
      else {
         // weight by resolution
         *dist = m_distCalc.getDist(&m_pHorz)*CAPWIDTH;
         total = total + CAPWIDTH;
      }
   }
   if (m_pVert.points.GetCount() > 0) {
      *currOrientation = VERT;
      if ((m_pVert.points.GetCount() < m_pVert.m_iNumLasers) ||
         (m_pVert.points.GetCount() > m_pVert.m_iNumLasers)) {
         *dist = *dist/total;
         return NOT_ENOUGH_POINTS;
      }
      else {
         float vertDist = m_distCalc.getDist(&m_pVert);
         if (vertDist < 0) return (-1);
         // weight by resolution
         else *dist += vertDist*CAPHEIGHT;
         total = total + CAPHEIGHT;
      }
   }

   *dist = *dist/total;
   return 0;

}
```

CLaserSettings Class

```
// LaserSettings.h: interface for the CLaserSettings class.
//
//////////////////////////////////////////////////////////////////////

#include <Afxtempl.h>   // for CList

#define HORZ 1
#define VERT 0

class CLaserSettings
{
public:
   void Copy (CLaserSettings *ls);
   void CopyBasics(CLaserSettings *ls);

   float m_rDist;       // mm
   int m_iNumLasers;
   float m_rDW;
   float m_rAngle;            // radians
   int m_iDegree;
   float *m_pCoeffs;
   float m_rShift;            // mm
   int m_iSurfaceMode;
   float m_rMaxDX;                  // mm
   int m_iMaxDP;              // pixels
   int m_bHorizontal;
   CList <FLOATCOORD, FLOATCOORD> points;
   CList <float, float> m_pSpotDiams;                      // pixels
   int m_iInitNumLasers;
   int m_iLaserNum;     // current Laser Firing
   CList <PointInfo, PointInfo> realPoints;

   CLaserSettings();
   virtual ~CLaserSettings();

};
```

```
// LaserSettings.cpp: implementation of the CLaserSettings class.
// Data Structure
// D. Tran 2001
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "laserdist.h"
#include "LaserSettings.h"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

CLaserSettings::CLaserSettings()
{
   m_rDist = 0.0;
   m_iNumLasers = 4;
   m_rDW = 30.0;
   m_rAngle = 0;
   m_iDegree = 0;
   m_pCoeffs = NULL;
   m_rShift = 0;
   m_iSurfaceMode = 0;
   m_rMaxDX = SENSE_WIDTH;
   m_iMaxDP = CAPWIDTH;
   m_bHorizontal = TRUE;

}

CLaserSettings::~CLaserSettings()
{
   if (m_pCoeffs)
      free (m_pCoeffs);
}

void CLaserSettings::Copy(CLaserSettings *ls)
{
   int i;
   FLOATCOORD temp;
   m_bHorizontal = ls->m_bHorizontal;
   m_iDegree = ls->m_iDegree;
   m_iMaxDP = ls->m_iMaxDP;
   m_rMaxDX = ls->m_rMaxDX;
   m_iNumLasers = ls->m_iNumLasers;
   m_iInitNumLasers = ls->m_iInitNumLasers;
   m_rAngle = ls->m_rAngle;
   m_rDist = ls->m_rDist;
   m_rDW = ls->m_rDW;
   m_rShift = ls->m_rShift;
   m_iSurfaceMode = ls->m_iSurfaceMode;
   m_iLaserNum = ls->m_iLaserNum;
   if (ls->m_pCoeffs) {
      if (m_pCoeffs)
         free (m_pCoeffs);
      m_pCoeffs = (float *) malloc ((m_iDegree+1)*sizeof(float));
      for (i=0; i< (m_iDegree+1); i++)
         m_pCoeffs[i] = ls->m_pCoeffs[i];
   }
   points.RemoveAll();
   m_pSpotDiams.RemoveAll();
   POSITION pos = ls->points.GetHeadPosition();
   for (i=0; i<ls->points.GetCount(); i++) {
      temp = ls->points.GetAt(pos);
      points.AddTail(temp);
      ls->points.GetNext(pos);
   }
}

void CLaserSettings::CopyBasics(CLaserSettings *ls)
{
   // copy non-simulation data
   m_bHorizontal = ls->m_bHorizontal;
   m_iMaxDP = ls->m_iMaxDP;
   m_rMaxDX = ls->m_rMaxDX;
   m_iNumLasers = ls->m_iNumLasers;
   m_rDW = ls->m_rDW;
   m_iLaserNum = ls->m_iLaserNum;
   m_iInitNumLasers = ls->m_iInitNumLasers;
}
```

CSimSettings Class

```
// SimSettingsDlg.cpp : implementation file
// Simulation parameters
// D. Tran 2001
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "laserdist.h"
#include "SimSettingsDlg.h"
#include <math.h>

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////////////////////////////////////////////////////////
/////
// CSimSettingsDlg dialog


CSimSettingsDlg::CSimSettingsDlg(CWnd* pParent /*=NULL*/)
   : CDialog(CSimSettingsDlg::IDD, pParent)
{
   //{{AFX_DATA_INIT(CSimSettingsDlg)
   m_iNumLasersH = 4;
   m_iNumLasersV = 4;
   m_rDistBtwH = 30.0f;
   m_rDistBtwV = 20.0f;
   m_rAngleH = 0.0f;
   m_rAngleV = 0.0f;
   m_iDegreeH = 0;
   m_iDegreeV = 0;
   m_sCoeffsH = _T("");
   m_sCoeffsV = _T("");
   m_rShiftH = 0.0f;
   m_rShiftV = 0.0f;
   m_iSimFrameRate = 15;
   m_rSimDistance = 90.0f;
   //}}AFX_DATA_INIT
}


////////////////////////////////////////////////////////////////////////
/////
// CSimSettingsDlg message handlers

void CSimSettingsDlg::OnOK()
{
   CDialog::OnOK();
}

void CSimSettingsDlg::OnApply()
{
   char *token;
   LPTSTR str;
   float min_dist;

   UpdateData(TRUE);

   m_radioCheckedH = m_radioH.GetCheck();
   m_radioCheckedV = m_radioV.GetCheck();

   if (!m_radioCheckedH && !m_radioCheckedV)
      AfxMessageBox ("At least one orientation has to be chosen", MB_OK,
                     0);

   if (m_radioCheckedH) {
      m_pH.m_bHorizontal = TRUE;
      m_pH.m_iDegree = m_iDegreeH;
      m_pH.m_iMaxDP = CAPWIDTH;
      m_pH.m_rMaxDX = SENSE_WIDTH;
      m_pH.m_iNumLasers = m_iNumLasersH;
      m_pH.m_iInitNumLasers = m_iNumLasersH;
      m_pH.m_rAngle = m_rAngleH*PI/180;         // convert to radians
      m_pH.m_rDist = m_rSimDistance;
      m_pH.m_rDW = m_rDistBtwH;
      m_pH.m_iLaserNum = 1;
      m_pH.m_rShift = m_rShiftH;
      if (m_radioFlatH.GetCheck())
         m_pH.m_iSurfaceMode = 0;
      else if (m_radioAngledH.GetCheck())
         m_pH.m_iSurfaceMode = 1;
      else if (m_radioRoundH.GetCheck()) {
         m_pH.m_iSurfaceMode = 2;
         // for y=a0x^0+a1x^1... coeffs[0]=a0, coeffs[1]=a1...
         if (m_pH.m_pCoeffs) free (m_pH.m_pCoeffs);
         m_pH.m_pCoeffs = (float *) malloc ((m_pH.m_iDegree +1) *
                           sizeof (float));
         str = m_sCoeffsH.GetBuffer(m_sCoeffsH.GetLength());
         m_sCoeffsH.ReleaseBuffer();
         token = strtok( str, " ");
         for (int i=0; i<(m_pH.m_iDegree+1); i++) {
            if (token == NULL) {
               AfxMessageBox("Error, not enough coefficients!", MB_OK,
                              0);
            }
            m_pH.m_pCoeffs[i] = atof(token);
            token = strtok( NULL, " ");
         }
      }

      // check that the angle given is valid
      min_dist = m_rSimDistance + (m_pH.m_iNumLasers+1)*m_pH.m_rDW*
                 tan(m_pH.m_rAngle*PI/180);
      if (min_dist <= 0)
         AfxMessageBox ("Horizontal Angle and Simulation Distance
                         Combination Invalid", MB_OK, 0);
   }

   if (m_radioCheckedV) {
      m_pV.m_bHorizontal = FALSE;
      m_pV.m_iDegree = m_iDegreeV;
      m_pV.m_iMaxDP = CAPHEIGHT;
      m_pV.m_rMaxDX = SENSE_HEIGHT;
      m_pV.m_iNumLasers = m_iNumLasersV;
      m_pV.m_iInitNumLasers = m_iNumLasersV;
      m_pV.m_rAngle = m_rAngleV*PI/180;         // convert to radians
      m_pV.m_rDist = m_rSimDistance;
      m_pV.m_rDW = m_rDistBtwV;
      m_pV.m_iLaserNum = 1;
      m_pV.m_rShift = m_rShiftV;
      if (m_radioFlatV.GetCheck())
         m_pV.m_iSurfaceMode = 0;
      else if (m_radioAngledV.GetCheck())
         m_pV.m_iSurfaceMode = 1;
      else if (m_radioRoundV.GetCheck()) {
         m_pV.m_iSurfaceMode = 2;
         // for y=a0x^0+a1x^1... coeffs[0]=a0, coeffs[1]=a1...
         if (m_pV.m_pCoeffs) free (m_pV.m_pCoeffs);
         m_pV.m_pCoeffs = (float *) malloc ((m_pV.m_iDegree +1) *
                          sizeof (float));
         str = m_sCoeffsV.GetBuffer(m_sCoeffsV.GetLength());
         m_sCoeffsV.ReleaseBuffer();
         token = strtok( str, " ");
         for (int i=0; i<(m_pV.m_iDegree+1); i++) {
            if (token == NULL) {
               AfxMessageBox("Error, not enough coefficients!", MB_OK,
                              0);
            }
            m_pV.m_pCoeffs[i] = atof(token);
            token = strtok( NULL, " ");
         }
      }

      // check that the angle given is valid
      min_dist = m_rSimDistance + (m_pV.m_iNumLasers+1)*m_pV.m_rDW*
                 tan(m_pV.m_rAngle*PI/180);
      if (min_dist <= 0)
         AfxMessageBox ("Vertical Angle and Simulation Distance
                         Combination Invalid", MB_OK, 0);
   }
}
```

```
BOOL CSimSettingsDlg::OnInitDialog()
{
   CDialog::OnInitDialog();

   m_radioFlatH.SetCheck(1);
   m_radioFlatV.SetCheck(1);
   return TRUE;  // return TRUE unless you set the focus to a control
                 // EXCEPTION: OCX Property Pages should return FALSE
}


void CSimSettingsDlg::ManualSetSettings(int do_horz,int do_vert,
                         float rSimDistance,
                         int iDegreeH,int iNumLasersH,float rAngleH,
                         float rDistBtwH,float rShiftH,
                         int iSurfaceModeH, CString sCoeffsH,
                         int iDegreeV, int iNumLasersV, float
                         rAngleV,float rDistBtwV,float rShiftV,
                         int iSurfaceModeV,CString sCoeffsV)
{
   char *token;
   LPTSTR str;

   m_radioCheckedH = do_horz;
   m_radioCheckedV = do_vert;

   if (m_radioCheckedH) {
      m_pH.m_bHorizontal = TRUE;
      m_pH.m_iDegree = iDegreeH;
      m_pH.m_iMaxDP = CAPWIDTH;
      m_pH.m_rMaxDX = SENSE_WIDTH;
      m_pH.m_iNumLasers = iNumLasersH;
      m_pH.m_iInitNumLasers = iNumLasersH;
      m_pH.m_rAngle = rAngleH*PI/180;         // convert to radians
      m_pH.m_rDist = rSimDistance;
      m_pH.m_rDW = rDistBtwH;
      m_pH.m_iLaserNum = 1;
      m_pH.m_rShift = rShiftH;
      m_pH.m_iSurfaceMode = iSurfaceModeH;
      if (iSurfaceModeH == 2) {
         // for y=a0x^0+a1x^1... coeffs[0]=a0, coeffs[1]=a1...
         if (m_pH.m_pCoeffs) free (m_pH.m_pCoeffs);
         m_pH.m_pCoeffs = (float *) malloc ((m_pH.m_iDegree +1) *
                          sizeof (float));
         str = sCoeffsH.GetBuffer(sCoeffsH.GetLength());
         sCoeffsH.ReleaseBuffer();
         token = strtok( str, " ");
         for (int i=0; i<(m_pH.m_iDegree+1); i++) {
            if (token == NULL) {
               AfxMessageBox("Error, not enough coefficients!", MB_OK,
                             0);
            }
            m_pH.m_pCoeffs[i] = atof(token);
            token = strtok( NULL, " ");
         }
      }
   }

   if (m_radioCheckedV) {
      m_pV.m_bHorizontal = FALSE;
      m_pV.m_iDegree = iDegreeV;
      m_pV.m_iMaxDP = CAPHEIGHT;
      m_pV.m_rMaxDX = SENSE_HEIGHT;
      m_pV.m_iNumLasers = iNumLasersV;
      m_pV.m_iInitNumLasers = iNumLasersV;
      m_pV.m_rAngle = rAngleV*PI/180;           // convert to radians
      m_pV.m_rDist = rSimDistance;
      m_pV.m_rDW = rDistBtwV;
      m_pV.m_iLaserNum = 1;
      m_pV.m_rShift = rShiftV;
      m_pV.m_iSurfaceMode = iSurfaceModeV;
      if (iSurfaceModeV == 2) {
         // for y=a0x^0+a1x^1... coeffs[0]=a0, coeffs[1]=a1...
         if (m_pV.m_pCoeffs) free (m_pV.m_pCoeffs);
         m_pV.m_pCoeffs = (float *) malloc ((m_pV.m_iDegree +1) *
                           sizeof (float));
         str = sCoeffsV.GetBuffer(sCoeffsV.GetLength());
         sCoeffsV.ReleaseBuffer();
         token = strtok( str, " ");
         for (int i=0; i<(m_pV.m_iDegree+1); i++) {
            if (token == NULL) {
               AfxMessageBox("Error, not enough coefficients!", MB_OK,
                              0);
            }
            m_pV.m_pCoeffs[i] = atof(token);
            token = strtok( NULL, " ");
         }
      }
   }
}

// stdafx.h : include file for standard system include files,
//  or project specific include files that are used frequently, but
//      are changed infrequently
//


#define VC_EXTRALEAN    // Exclude rarely-used stuff from Windows
                        // headers

#include <afxwin.h>         // MFC core and standard components
#include <afxext.h>         // MFC extensions
#include <afxdisp.h>        // MFC Automation classes

typedef struct {
   float X;
   float Y;
} FLOATCOORD;

typedef struct {
   FLOATCOORD c;
   float err_max;
   float err_min;
} PointInfo;

#define CAPWIDTH 640
#define CAPHEIGHT 480
#define PI 3.14159265358979323846264338327950
#define REFRESH_ID 1
#define SENSE_WIDTH 4.8       // mm
#define SENSE_HEIGHT 3.6      // mm
#define INI_BEAM_DIAM 1.5     // mm
```

```
#define BEAM_DIVERGENCE 1.6
#define MAX_INTENS_PIX_VAL 255
#define DONE 1
#define NO_POINTS -2
#define NOT_ENOUGH_POINTS -1

// For Non-Simulation
#define CENTER_X 358.7
#define CENTER_Y 235
#define X_OFFSET 38.7
#define FL 4.06
#define THRESHOLD 50

/*
// For Simulation
#define CENTER_X 320
#define CENTER_Y 240
#define X_OFFSET 0
#define FL 3.6
#define THRESHOLD 0
*/
```

Appendix B

Environmental Simulation Files

```
// WUPLaserDist.cpp : Defines the initialization routines for the DLL.
//

#include "stdafx.h"
#include "WUPLaserDist.h"
#include "LaserdistDlg.h"
#include <windows.h>
#include <stdio.h>
#include <math.h>

#include "wupplugin.h"          // This is the World Up API and must be
                                // included in all plugins.

// Type names defined as constants

char szAuthor[256] = "D. Tran";
char szVersion[10] = "5.0";
const char *TN_LASERDIST = "LaserDistGroup";
FILE *stream;

// Custom type prototypes
void UnInitLaserDistGroup(HINSTANCE hDLL);
void InitLaserDistGroup(HINSTANCE hDLL);
WUPENTRYPOINT void AddedLaserDistGroup(WUPobject * pObject,
                                       void * pData );
WUPENTRYPOINT void RemovedLaserDistGroup(WUPobject * pObject,
                                         void * pData );
WUPENTRYPOINT void DoLaserDistGroup(WUPobject * pObject, void * pData );

CLaserdistDlg lDlg;

////////////////////////////////////////////////////////////////////////
/////
// The one and only CWUPLaserDistApp object

CWUPLaserDistApp theApp;

/*
 * WUPPluginInitialize() is called automatically by World Up upon
 * discovering this DLL in
 * the .\plugins or project (.up file ) directory.
 * myWUPModule : the handle to WorldUp module (worldup.exe or wup*.ocx)
 * hDLL : the handle to this DLL
 */
WUPENTRYPOINT void WUPPluginInitialize( HINSTANCE myWUPModule,
                                        HINSTANCE hDLL )
{
   // Register the World Up API function addresses
   BOOL success = InitializeWorldUpAPI(myWUPModule);
   if (!success) {
      MessageBox(NULL,"Support for some requested functionality was not
                 found, Exiting...", "Error", 0);
      return;  //we don't try to proceed unless all our functionality is
               // available.
   }
   // Create our types
   InitLaserDistGroup(hDLL);

}

// WUPPluginShutDown() is called by World Up on exiting the application
//(or project if this DLL was project specific ( loaded with .up file)).
WUPENTRYPOINT void WUPPluginShutDown( HINSTANCE hDLL )
{
   UnInitLaserDistGroup(hDLL);

}

/***********************************************************************
/* LaserDistGroup
/*
/*  LaserDistGroup subtypes the World Up Group type.  It has all the
/*  functionality of the World Up group type, with the additional
/*  functionality specified in the DoLaserDistGroup callback below.
*/
void UnInitLaserDistGroup(HINSTANCE hDLL)
{
   wuUnregisterCustomType(TN_LASERDIST, hDLL);
   fclose(stream);
}

void InitLaserDistGroup(HINSTANCE hDLL)
{

   // Retrieve the World Up type we want to derive our custom type from.
   WUPtype * pType = wuGetType( "Group" ) ;

   WUPtype * pLaserDistType = wuRegisterCustomType(
      hDLL,                     // Handle to this DLL
      TN_LASERDIST,             // Name of your custom type as it will
                                // appear in the type browser
      pType,                    // Base Type ( This must be some valid
                                // World Up/ Type )
      "AddedLaserDistGroup",    // OnCreate callback function name
      "RemovedLaserDistGroup",// OnDelete callback function name
      "DoLaserDistGroup",       // OnRun callback
      (void *)0 );              // This is an optional void pointer in the
                                // event that you need to pass a data
                                // structure into a callback ( pased in as
                                // pData ).

   if ( pLaserDistType ) {
      wuTypeAddProperty(pLaserDistType, "Rate", WUPTYPE_SINGLE, NULL,
                        kIsImportant);
      wuTypeAddProperty(pLaserDistType, "Distance1", WUPTYPE_SINGLE,
                        NULL, kIsImportant);
      wuTypeAddProperty(pLaserDistType, "Distance2", WUPTYPE_SINGLE,
                        NULL, kIsImportant);
      wuTypeAddProperty(pLaserDistType, "Distance3", WUPTYPE_SINGLE,
                        NULL, kIsImportant);
      wuTypeAddProperty(pLaserDistType, "Distance4", WUPTYPE_SINGLE,
                        NULL, kIsImportant);
      wuTypeAddProperty(pLaserDistType, "Move", WUPTYPE_BOOLEAN, NULL,
                        kIsImportant);
      wuTypeAddProperty(pLaserDistType, "Reference Time",
                      WUPTYPE_SINGLE, NULL, NULL);
      wuTypeSetPropertySingle(pLaserDistType, "Distance1", 1.0f);
      wuTypeSetPropertySingle(pLaserDistType, "Distance2", 1.0f);
      wuTypeSetPropertySingle(pLaserDistType, "Distance3", 1.0f);
      wuTypeSetPropertySingle(pLaserDistType, "Distance4", 1.0f);
      wuTypeSetPropertyBoolean(pLaserDistType, "Move", 1);
      wuTypeSetPropertySingle(pLaserDistType, "Rate", 15.0f);
   }

   stream  = fopen( "data", "w++" );

}

WUPENTRYPOINT void AddedLaserDistGroup( WUPobject * pObject, void *
pData )
{
   float reftime;

   wuStatusMessage( "Added LaserDistGroup:" ) ;
   wuStatusMessage(wuObjectGetName( pObject ) );
   // Initialize our times.  Simtime is seconds elapsed since Sim->Run.
   reftime = 0;
   wuObjectSetPropertySingle(pObject, "Reference Time", reftime);
   AFX_MANAGE_STATE(AfxGetStaticModuleState());
   lDlg.Create(CLaserdistDlg::IDD,NULL);
   lDlg.ShowWindow(SW_SHOW);

}

WUPENTRYPOINT void RemovedLaserDistGroup( WUPobject * pObject, void *
pData )
{
   wuStatusMessage( "Removed LaserDistGroup:" ) ;
   wuStatusMessage(wuObjectGetName( pObject ) );
}

WUPENTRYPOINT void DoLaserDistGroup( WUPobject * pObject, void * pData )
{
   float simtime, reftime, rate, dist, intervaltime, final_dist;
   CList <float, float> distances;
   int i, len, go;
   WUPobject *lasers[NUM_LASERS];
   CString prefix, distlist="";
   int enabled[NUM_LASERS];
   char distText[10];

   float min_dist = 100000;

   simtime = wuGetSimulationTime();
   rate = wuObjectGetPropertySingle(pObject, "Rate");
   reftime = wuObjectGetPropertySingle(pObject, "Reference Time");

   prefix = "Distance";
   len = prefix.GetLength();
   if (!distances.IsEmpty())
      distances.RemoveAll();
   for (i=0; i<NUM_LASERS; i++) {
      prefix.Insert(len, i+48+1);
      dist = wuObjectGetPropertySingle(pObject, prefix);
      distances.AddTail(dist);
      sprintf (distText, "%.5f\t", dist);
      distlist += distText;
      if (dist < min_dist)
         min_dist = dist;
      prefix.Remove(i+48+1);
   }

   if ( simtime < 1.0 ) {
      reftime = simtime;
      wuObjectSetPropertySingle(pObject, "Reference Time", reftime);
   }

   prefix = "Laser-";
   len = prefix.GetLength();
   for (i=0; i<NUM_LASERS; i++) {
      prefix.Insert(len, i+48+1);
      lasers[i] = wuGetObject(prefix);
      prefix.Remove(i+48+1);
   }

      lDlg.OnTimer2(&distances, enabled, &final_dist, &go);
      wuObjectSetPropertyBoolean(pObject, "Move", go);
      if (stream && go) {
         fprintf (stream, "%s", distlist);
         fprintf (stream, "%.5f\t%.5f\n", final_dist,
                  (min_dist-final_dist)/min_dist);
      }

      for (i=0; i<NUM_LASERS; i++) {
         if (lasers[i])
            wuObjectSetPropertyBoolean(lasers[i],"Enabled", enabled[i]);

      }
      reftime = simtime;
      wuObjectSetPropertySingle(pObject, "Reference Time", reftime);

}

// Functions used in WorldUp but not in LaserDist

void CImageProcess::PaintSimulatedCaptureWindow(int doHorz, int doVert,
                                           CList <float, float> *dists)
{

   unsigned char *ptr;
   ptr = m_lpVHdr->lpData;

   // make background black
   for (int i=0; i < CAPWIDTH*CAPHEIGHT; i++) {
      *ptr = 0;
      ptr++;
      *ptr = 0;
      ptr++;
      *ptr = 0;
      ptr++;
   }

   // do calculations
   if (doHorz) {
      m_distCalc.getSimPointsGiven(&m_pHorzDraw, dists);

      // draw spots
      putSpots(&m_pHorzDraw.points, &m_pHorzDraw.m_pSpotDiams);

   }

   if (doVert) {
      m_distCalc.getSimPointsGiven(&m_pVertDraw, dists);

      putSpots(&m_pVertDraw.points, &m_pVertDraw.m_pSpotDiams);

   }

   memcpy (m_lpOnBuffer->lpData, m_lpVHdr->lpData,
          CAPWIDTH*CAPHEIGHT*3);
   // paint window
   PaintWindow(m_hWndCap, m_lpVHdr);
}


// FOR WORLDUP SIMULATION
void CSimSettingsDlg::SimSetVals(int iHorz, int iVert,
                                 int m_iNumLasersH, int m_iNumLasersV,
                                 float m_rDistBtwH, float m_rDistBtwV)
{
   m_radioCheckedH = iHorz;
   m_radioCheckedV = iVert;

   if (!m_radioCheckedH && !m_radioCheckedV)
      AfxMessageBox ("At least one orientation has to be chosen", MB_OK,
                     0);

   if (m_radioCheckedH) {
      m_pH.m_bHorizontal = TRUE;
      m_pH.m_iDegree = m_iDegreeH;
      m_pH.m_iMaxDP = CAPWIDTH;
      m_pH.m_rMaxDX = SENSE_WIDTH;
      m_pH.m_iNumLasers = m_iNumLasersH;
      m_pH.m_iInitNumLasers = m_iNumLasersH;
      m_pH.m_rDW = m_rDistBtwH;
      m_pH.m_iLaserNum = 1;
      m_pH.m_iSurfaceMode = 0;
   }

   if (m_radioCheckedV) {
      m_pV.m_bHorizontal = FALSE;
      m_pV.m_iDegree = m_iDegreeV;
      m_pV.m_iMaxDP = CAPHEIGHT;
      m_pV.m_rMaxDX = SENSE_HEIGHT;
      m_pV.m_iNumLasers = m_iNumLasersV;
      m_pV.m_iInitNumLasers = m_iNumLasersV;
      m_pV.m_rDW = m_rDistBtwV;
      m_pV.m_iLaserNum = 1;
      m_pV.m_iSurfaceMode = 0;
   }
}
```

Script Files

```
// getdist.ebs
' Determine distance to nearest obstacle ahead
' For horizontally oriented lasers only
' Automatic drive
' D. Tran 2001

sub task( obj as Group )

   ' Add code here
   ' Commands in this routine will be
   ' executed every frame
   dim dir as vect3d
   dim trans as vect3d
   dim theroot as Node
   dim dist as Single
   dim shift as Single
   dim rot as Orientation

   dim mynode as LaserDistGroup
   set mynode = GetLaserDistGroup ("DistHolder")

   obj.GetTranslation trans
   if mynode.move = TRUE then
      if trans.z < 4700 then
         trans.z = trans.z + 10
      end if
      obj.SetTranslation trans
   end if

   obj.GetRotation rot
   OrientToDir rot, dir
   Vect3DNorm dir


   set theroot = getnode("Root-1")

   shift = -60.0
   dist = FindDist ("Laser-1", trans, shift, dir, theroot)
   if dist > 0 then
      mynode.distance1 = dist
   end if

   shift= -30.0
   dist = FindDist ("Laser-2", trans, shift, dir, theroot)
   if dist > 0 then
      mynode.distance2 = dist
   end if

   shift=30.0
   dist = FindDist ("Laser-3", trans, shift, dir, theroot)
   if dist > 0 then
      mynode.distance3 = dist
   end if

   shift=60.0
   dist = FindDist ("Laser-4", trans, shift, dir, theroot)
   if dist > 0 then
      mynode.distance4 = dist
   end if

end sub

Function FindDist (lasername as String, _
   transl as Vect3D, _
   shift as Single, _
   direct as Vect3D, _
   droot as Node) _
   as Single

   dim laser as Geometry
   dim lasertrans as Vect3D
   dim stret as Vect3D
   dim orig as Vect3D
   dim poly as Long
   dim geom as Geometry
   dim dist as Single
   dim ori as Orientation
   dim moduleAxis as Vect3D

   dist = -1
   set laser = GetGeometry(lasername)
   laser.GetTranslation lasertrans
   laser.GetStretch stret

   ' add 120 to take distance from front
   ' of vehicle rather than center
   orig.x = transl.x + 120*direct.x
   orig.y = transl.y + 120*direct.y
   orig.z = transl.z + 120*direct.z

   ' move detector up 75 (1/2 of wheel + 1/2 base)
   orig.y = orig.y - 75

   orientset ori, 0, 90, 0
   Vect3dRotate direct, ori, moduleAxis

   'shift by shift to appropriate module
   orig.x = orig.x + shift*moduleAxis.x
   orig.y = orig.y + shift*moduleAxis.y
   orig.z = orig.z + shift*moduleAxis.z

   poly = RayIntersect (droot,orig,direct,geom,dist)
   if poly <> 0 then
      message "distance" + str$(dist)
      lasertrans.z = dist/2 + 120
      stret.y = dist
      laser.SetTranslation lasertrans
      laser.SetStretch stret
   end if
   FindDist = dist
end Function

' display.ebs
' Display distance to nearest obstacle ahead
' For horizontally oriented lasers only
' D. Tran 2001

declare sub DisplayOffsetText( win as Window, _
     HorizontalPosition as Single, _
     VerticalPostion as Single, msgstring as String, _
     returnBottomOfText as Single )

sub task( obj as Window )

   dim message as String
   dim dist as Single
   dim vpos as Single
   dim temp as Single

   dim mynode as LaserDistGroup
   set mynode = GetLaserDistGroup("DistHolder")

   dist = mynode.Distance3
   message = "Distance 3:" + str$(dist)
   DisplayOffsetText obj, 0.2, 0, message, vpos

   dist = mynode.Distance4
   message = "Distance 4:" + str$(dist)
   DisplayOffsetText obj, 0.6, 0, message, vpos

   dist = mynode.Distance1
   message = "Distance 1:" + str$(dist)
   DisplayOffsetText obj, 0.2, vpos, message, temp

   dist = mynode.Distance2
   message = "Distance 2:" + str$(dist)
   DisplayOffsetText obj, 0.6, vpos, message, temp


end sub

sub DisplayOffsetText( win as Window, _
                       HorizontalPosition as Single, _
                       VerticalPosition as Single, _
                       msgstring as String, _
                       returnBottomOfText as Single )

   dim size as Vect2d
   win.TextExtent msgstring, size

   ' Draw the text centered, along the top
   'win.DrawText 0.5 - size.X / 2, VerticalPosition - size.Y, msgstring
   win.DrawText HorizontalPosition, VerticalPosition + size.Y, _
          msgstring


   returnBottomOfText = VerticalPosition + size.Y

end sub
```

5-13-11