Examining Laser Triangulation System Performance

Using a Software Simulation

A Thesis Presented to The Faculty of the

Fritz J. and Dolores H. Russ College of Engineering and Technology

Ohio University

In Partial Fulfillment

of the Requirement for the Degree

Master of Science

by

Jeff Collier

June, 1998

Acknowledgments

Thanks to God for all his blessings. Thanks to Heather, my wife, and my parents, the prayers of which I am convinced are the most significant factor in any success I have. Thanks to Dr. Nurre, my advisor, who has been an excellent guide and has given me many opportunities. Thanks to Erick Lewark, a fellow graduate student.

Table of Contents

List of Tables

List of Figures

I. Introduction

A. Introduction

The concept of triangulation is very well understood. Starting with the ancient Greeks [16], triangulation has been used to measure distances. The invention of the laser diode, the microcomputer and the CCD camera have opened entirely new areas in which laser triangulation measurement systems (known as scanners) can be used. Current applications now range from scanners for the insides of old pipes [11] to a vision tool for the blind [7][8]. As such, it is important that techniques be developed to minimize the error in laser triangulation measurement systems.

A powerful technique for examining the performance of equipment is computer simulation. A computer simulation is inexpensive, easy to maintain and highly flexible. Simulation software can be used to aid the design of scanning equipment and help make decisions about what triangulation geometry would be best to purchase for a given application. A simulator can also be used to determine the best orientation for scanning a particular subject.

The purpose of this work was to create a simulator for a laser triangulation system. In addition to presenting the simulator itself, this thesis describes and demonstrates the use of the simulator. A limitless number of laser triangulation systems can be modeled and most subjects represented in CAD files can be used in the computer simulation.

B. Overview of laser triangulation

A brief description of a laser triangulation system will be discussed so that the terminology used in this thesis can be understood. In a laser triangulation system the surface of the object being scanned is intersected with a beam of laser light. A position sensor detects the light and a location in space is calculated based on the location on the position sensor. Figure 1 shows a diagram of a laser triangulation system.

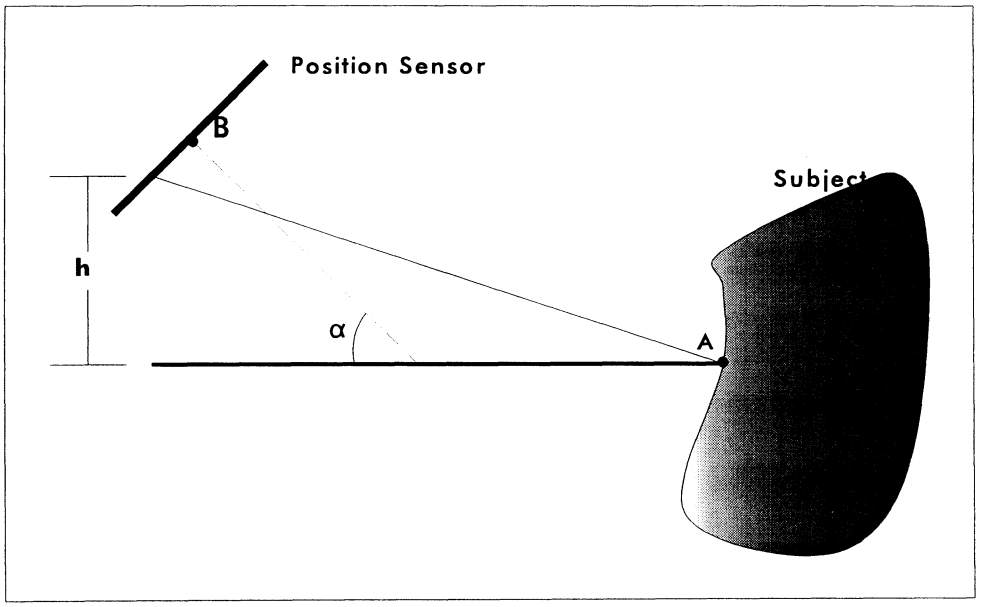

Figure 1: Laser triangulation diagram.

In Figure 1 the horizontal line represents the laser beam. Point A represents the intersection of the laser beam and the surface of the object being scanned. Point B is the position recorded by the position sensor. The distance h is between the center of a sensing element on the position sensor and the laser plane. The angle α is the angle between the optical axis of the position sensor and the laser beam.

The model described above can be extended by replacing the laser beam with a laser plane and making the position sensor two dimensional. An additional method of extending the system to three dimensions is to move the laser beam along a known path.

The following terms can be used to describe aspects of Figure 1 and will be used throughout this thesis. First, the distance between the laser plane and the position sensor will be called the baseline. The angle between the laser beam (or plane) and the optical axis of the position sensor will be called the camera angle because the position sensor will most often be a CCD camera. Finally, the object to be scanned will be called the subject.

C. Problems associated with current systems

There are two main problems associated with laser triangulation systems. The first problem occurs when the laser light is blocked somewhere between the laser and the position sensor. This problem will be called occlusion. There are many reasons for occlusion and it will be discussed at length in chapter two. The second problem is the quality of the measured point is dependent on the ratio of the baseline distance to the surface point distance.

If a CCD camera is used for a position sensor, the quality of the baseline distance will be limited by the resolution of the CCD camera. As with any triangulation system, the precision of the measurement is highest when the baseline distance is equal to the measured distance. For a laser triangulation system, however, an increase in the baseline

distance means an increase in the likelihood of having occlusion problems. Thus as the quality of the data increases, occlusion also increases.

D. Computer simulation

There are an increasing number of applications for which three-dimensional data would be useful. With the increase in applications comes an increase in the variety of subjects. The subject to be scanned is an essential part of how a laser triangulation system will perform. Therefore, a precise analysis of error within a system should include a model of the object to be scanned. An analysis of error within a laser triangulation system should be flexible enough to handle a large variety of subjects.

Due to the nature of a camera lens, points farther from the camera along a laser beam will be measured with less precision. In addition, the precision does not change linearly. The nonlinear nature of stereo vision systems is explained by Nurre [12] and will be discussed more in chapter two.

Due to the nonlinear nature of the problem and the fact that it depends on an ever changing and vast number of subjects, a computer simulation was written to examine the trade-offs between occlusion and data quality. A computer simulation allows for a large amount of flexibility by giving the user the ability to calculate the error for a given configuration without having to build and test it.

E. Organization of thesis

This thesis is organized into six chapters. Following this introduction is a discussion of recent research related to errors in a laser triangulation system. Next is a chapter that covers the specifics of occlusion and describes how they will be represented by the computer simulation. Chapter four is a discussion of the simulation code itself. Chapter five will present results generated by the simulator and demonstrate with a simple example the trade-off between occlusion and data quality. The final chapter is a discussion of the results of this research.

II. Research of errors in laser triangulation systems

A. Introduction

There are many factors that can generate error in a laser triangulation system. Elements such as the quality of the laser beam and variations in the intensity of light striking the position sensor can play a role in measurement accuracy [5]. The error generated due to these factors is reported to be on the order of micrometers. Other sources of error due to the geometry of the system can be on the order of millimeters. Because errors due to the geometry of the system can have a much greater impact on system performance, this work will be limited to the errors due to the geometry of the system.

Researchers have examined this problem from two angles. Some have looked strictly at the factors that could cause errors. Others have designed systems from which errors have been observed and analyzed.

The first part of this chapter contains a summary of research that involves the observation and analysis of error in actual triangulation systems. Next research that relates strictly to the analysis of error due to the geometry of the system will be given. This provides a foundation for creating a tool which can model a laser triangulation system.

B. Research relating to laser triangulation systems

Although the baseline is the main geometric parameter of a laser triangulation system to impact the quality of the data, there are a number of other factors that can have an effect. Many people have adjusted these factors in an attempt to come up with a better measurement system. Because the quality of the measurement depends heavily on the subject and because error within the system cannot be analyzed in a linear fashion [12], a system has not been created to work optimally for a large variety of different subjects.

One of the first laser triangulation systems described in the literature was created by Rioux [14]. It uses a rotating mirror to direct a laser beam and camera view across an object. Figure 2 is a diagram of the system.

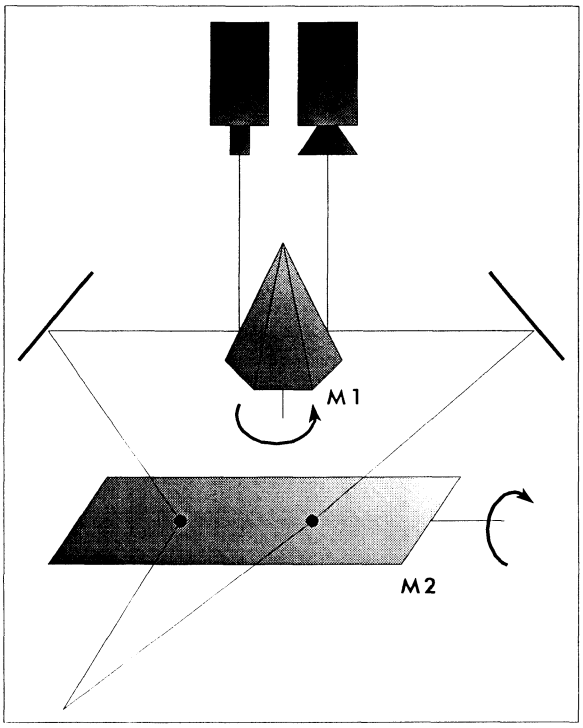

Figure 2: A diagram of the Rioux laser triangulation scanner.

In Figure 2 the pyramid mirror M1 rotates, and as it does, the laser and the view of the sensor scan across the subject. A second rotating mirror, M2, directs the laser beam and camera in another direction, thereby covering the surface of the object.

Rioux divides occlusion, which he calls shadow effect problems, into two categories. One category is occlusion due to the camera being blocked, and the other is due to the laser being blocked. In addition to categorizing the errors, the author suggests scanner configurations that could help reduce the occlusion.

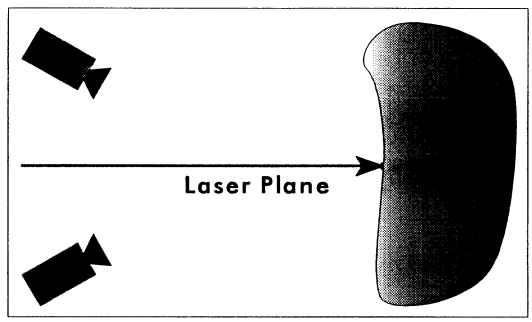

Figure 3: Laser triangulation system with two camera views.

The purpose of the research of Saint-Marc, et al. was to examine the capabilities of a low cost laser triangulation system [15]. Two cameras and a laser plane were used. The second camera was used to reduce the occlusion problem. Figure 3 diagrams the configuration of the laser plane and cameras. In Figure 3 the horizontal line represents the laser beam intersecting the subject at point P. Either or both cameras can record the position of this point.

The subject to be scanned was set on a rotating table. The system used a vibrating mirror to create a laser plane to intersect with the subject. The position of points along the intersection were then calculated based on the two camera views.

Some indication of the data quality was given by measuring the resolution of the scanner. Occlusion was discussed only to demonstrate that for a given object some of the occlusion was eliminated as a result of having two cameras. No analysis was made of the trade-offs between the data quality and the occlusion problem.

Dalgish et al. built a laser triangulation system that reduced the hardware burden of the host CPU [4]. The triangulation system built used a laser plane created by a cylindrical lens and a baseline of 720 mm. The resources that were devoted to the system by the controlling computer were drastically reduced. Because they chose a relatively long baseline, the system was sensitive to occlusion problems.

Two sources of occlusion were identified. The first was due to the laser light striking the subject's surface at a small angle. A small angle between the laser plane and the subject's surface will not result in enough diffuse light reflecting back to the camera. The second identified problem was called "the problem of hidden parts" by Dalgish et al. The problem of hidden parts is occlusion due to the camera or laser being blocked by an intervening surface on the subject.

A laser triangulation system was designed to track weld joints [20]. The system uses a laser plane generated by a cylindrical lens. The subjects for which the system was designed are relatively flat, so little mention was given to occlusion. The process of choosing the system geometry was not given in detail. A rough estimate of the resolution of the scanner was given.

The research by Zeng was based on a laser triangulation system that was unique in many ways [21]. A scanning laser beam was used instead of a plane of light. The beam

was moved by means of an acousto-optical device. A second laser beam was added to reduce the occlusion problem.

The improvement that results from having a second laser beam is similar to the improvement that results from having a second camera. This method is effective for reducing a particular type of occlusion which is due to the laser beam not being perpendicular to the subjects surface. A second beam at a large angle to the first increases the chances that a laser beam will intersect the subjects surface at an extreme angle. Because there is a higher likelihood that at least one laser will strike the object at an acceptable angle, the amount of occlusion is reduced. An in depth analysis of how the geometry of the system effected the amount of occlusion was not given.

Much of the current published research does not address the trade-offs between the scanning resolution and the amount of occlusion that occurs in a scan. In addition most of the work does not describe how the geometry of the system was decided upon. Some research discusses these issues as they relate to an experimental system that has already been implemented.

All of the systems discussed above would benefit from an analysis of the trade-off between occlusion and data quality. The weld bead scanner would benefit the least from this analysis because the subject is known to have a flat geometry with little chance for occlusion.

C. Research of error in a triangulation systems

One way to examine the trade-offs between occlusion and triangulation error is to build a scanner with an adjustable baseline. Such a scanner was designed and tested by Clark [3]. Planar elliptical mirrors allow for the scanner's baseline to be easily adjusted.

With a system that has an adjustable baseline, the geometry can be changed so that the optimal baseline may be chosen for the subject being scanned. One problem with the system is that it is very sensitive to calibration and the non-ideal elliptical mirrors can further aggravate the calibration process. Even so, the system admirably demonstrates the trade-off between resolution and occlusion.

For most applications, a scanner with an adjustable baseline would be impractical due to the added cost, complexity and sensitivity to calibration. However, Clark has shown that given an object, the scanning geometry can be optimized to meet the needs of the system designer. Clark has also demonstrated that it is important to examine the trade-offs between occlusion and scanner accuracy.

A analytical presentation of error in a stereo vision system was discussed by Nurre [12]. This discussion provides a good background for the analysis of error in a laser triangulation system. Based on a pinhole camera model, the article gives a precise description of the geometrical aspects of error present in a stereo vision system.

The situation for the laser triangulation system is very similar to that of the stereo vision system, but one of the cameras is replaced with a plane of laser light. This greatly reduces the complexity of the system. Consider the situation in which the laser plane of light is in the x-y plane, and the orientation of the camera can be described by a

transformation matrix [9]. Using the pinhole camera model, the following equation describes the transformation from a world point to an image plane coordinate.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} * \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \begin{bmatrix} w \cdot x_{ip} \\ w \cdot y_{ip} \\ w \cdot z_{ip} \\ w \end{bmatrix} \quad (1)$$

The point (X, Y, Z) is a point in world space. The vector [$w \cdot x_{ip}$, $w \cdot y_{ip}$, $w \cdot z_{ip}$, w] represents a homogeneous point on the image plane. The values a_{11} through a_{44} define the transformation matrix. The following equations can be written from equation one.

$$(a_{11} - a_{41}x_{ip})X + (a_{12} - a_{42}x_{ip})Y + (a_{13} - a_{43}x_{ip})Z = a_{44}x_{ip} - a_{14} \quad (2)$$

$$(a_{21} - a_{41}y_{ip})X + (a_{22} - a_{42}y_{ip})Y + (a_{23} - a_{43}y_{ip})Z = a_{44}y_{ip} - a_{24} \quad (3)$$

Because the laser plane is parallel to the x-y plane, Z is equal to some constant h. Equations two and three become:

$$(a_{21} - a_{41}y_{ip})X + (a_{22} - a_{42}y_{ip})Y = (a_{43}h + a_{44})y_{ip} - a_{24} - a_{23}h \quad (4)$$

$$(a_{11} - a_{41}x_{ip})X + (a_{12} - a_{42}x_{ip})Y = (a_{43}h + a_{44})x_{ip} - a_{14} - a_{13}h \quad (5)$$

If the following assignments are made,

$$A = (a_{11} - a_{41}x_{ip}) \tag{6}$$

$$B = (a_{12} - a_{42}x_{ip}) \tag{7}$$

$$C = (a_{43}h - a_{44})x_{ip} - a_{14} - a_{13}h \tag{8}$$

$$D = (a_{21} - a_{41}y_{ip}) \tag{9}$$

$$E = (a_{22} - a_{42}y_{ip}) \tag{10}$$

$$F = (a_{43}h - a_{44})y_{ip} - a_{24} - a_{23}h \tag{11}$$

equations four and five become a set of lines based on ipy and ipx and have the following form.

$$A \cdot X + B \cdot Y = C \tag{12}$$

$$D \cdot X + E \cdot Y = F \tag{13}$$

If x_{ip} and y_{ip} values are chosen that correspond to the boarders of pixels on the image plane of a CCD camera, equations twelve and thirteen represent two sets of lines in

world space that form a grid. This can be thought of as projection of the CCD array boundaries onto the laser plane. The areas within the projected grid represent all points that must be represented within the scanner as one discrete point. The area within each section of the grid represents the resolution of the scanner.

The following example will demonstrate the formation of the line sets. Consider the situation in which the laser plane is on the x-y plane. A camera is placed so that it has a view of the laser plane and the view can be described by the following transformation matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 04 \\ 0 & -0.5 & 0.866 & -0.693 \\ 0 & -0.866 & -0.5 & 0.4 \\ 0 & -17.321 & -10.0 & 9.0 \end{bmatrix} \quad (14)$$

Using this matrix in equation one, A, B, C, D, E and F (see equations 6-11) can be found in terms of x_{ip} and y_{ip}. If a set of eleven lines (based on equations twelve and thirteen) is created using values of x_{ip} between -0.025m and 0.025m and a set of fifteen lines is created using values of y_{ip} within the same range as x_{ip}, the grid in Figure 6 will be generated.

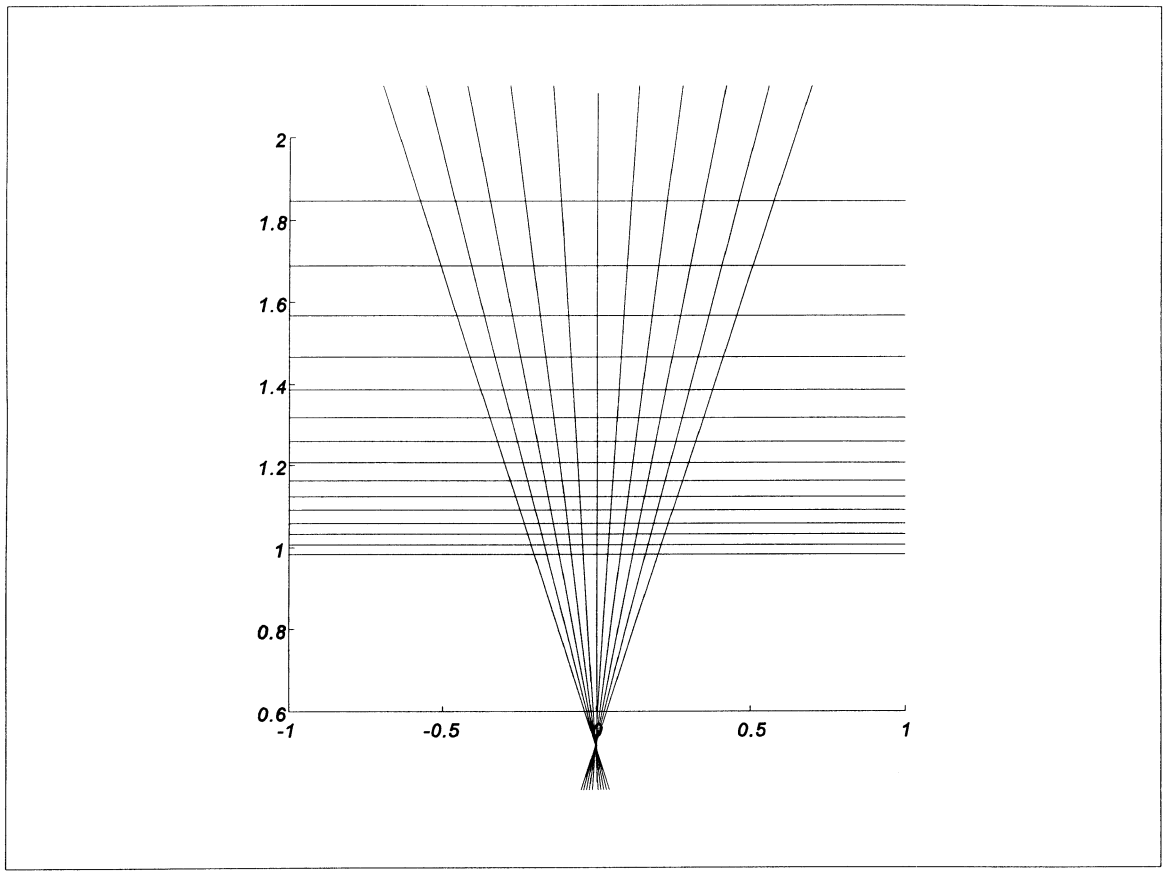

Figure 4: A grid representing projected pixel boundaries

In Figure 6 the horizontal lines represent the boundaries between the area of space seen by each column of pixels on an image plane. The slanted lines represent the boundaries for the columns of pixels. Thus the area in space measured by any given pixel can be determined by its corresponding area on the grid. Clearly the more distant areas represent a greater chance for error.

There are two problems in using the approach discussed above to examine the trade-offs between occlusion and scanner resolution. First, as described in reference [12] the equations for error are non-linear. The second problem is that the occlusion of the object is not taken into consideration. Both of these problems can be dealt with by using

a computer simulation. For a given laser triangulation system geometry and for a given object, the simulation program can decide, first, if a point is occluded and, second, what value would be generated within the given computer model for the system.

An excellent article by Tasi [16] describes a mathematical camera model. It is based on a pinhole camera model with some enhancements to make the model more accurate. One enhancement is the addition of lens distortion. Radial lens distortion in one direction can be described with the equation [13]:

$$X_d = \frac{X_u}{(1 + \kappa_1 \cdot r^2 + \kappa_2 \cdot r^4 \ldots)} \tag{15}$$

where the κ terms are based on the physical properties of the lens. X_d is the distorted point on the focal plane and X_u is the undistorted point. The variable r is the distance between the lens center (C_x, C_y) on the focal plane and the undistorted point:

$$r = \sqrt{(X_u - C_x)^2 + (Y_u - C_y)^2} \tag{16}$$

The distortion in the y direction can be found in a similar manor. Tsai states that based on his experience, a more in-depth lens distortion model would not help and would cause numerical instability.

Tsai's camera model accommodates a lens that is not centered on the focal plane. It also accommodates a camera with a different number of sensing elements than pixels.

It is clear that building systems and testing them to determine their performance in terms of occlusion and scanner accuracy is a costly and time consuming process. The work of Nurre, Tsai, and Clark can be incorporated into a computer simulation for a laser triangulation system. Such a simulation can provide a means of analysis that is quicker and flexible for future research.

III. Method of software development

A. Introduction

A software simulation of a laser triangulation system can be used as a tool to examine the trade-offs between occlusion and scanner accuracy. A simulation is particularly useful because factors such as the cost of the system, the type of object scanned and the purpose of the scan will make other prototype analysis difficult or impossible. The flexibility of the simulation software allows for an analysis that is specific to the needs of a particular system without losing the generality needed to analyze any system.

In particular the simulator was developed to reveal the presence of occlusion for a given scanning configuration and subject. There are many causes for occlusion. Priority should be given to the types of occlusion that are most common and have the greatest effect on the outcome. Therefore this section will start with a discussion of the types of occlusion and move to a description of the scan coverage approach. The last section of this chapter describes the method of obtaining results.

B. Types of occlusion

A laser triangulation system's performance will depend on the subject being scanned. Subjects with many deep crevasses or parts that could block a laser or camera will require a shorter baseline. A subject with no occlusion problems will generate more accurate results with a longer baseline. Because the subject scanned is a factor in the optimal scanner configuration, the subject must be considered in the analysis. The

geometry of the scanner and the subject are both necessary to consider the amount and quality of scan coverage. Separating occlusion into several categories will be helpful when describing the capabilities and limitations of the software.

Six separate cases are used to describe how data is collected or occluded. In some situations these cases may occur simultaneously. The types of occlusion can be grouped as follows:

1. The ideal case.
2. The camera occluded by the subject.
3. Laser plane occluded by the subject.
4. Poor vertical resolution.
5. Poor laser angle.
6. Poor laser diffusion.

1. The ideal case

The ideal case occurs when the laser plane intersects the subject and the intersection is clearly recorded by the camera. This is the most common situation and a profile of it is shown in Figure 5.

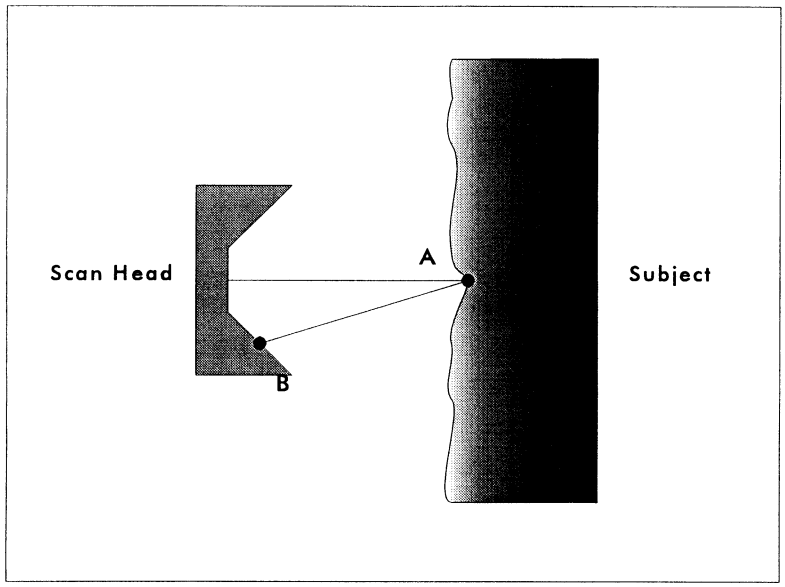

Figure 5: The ideal case in measuring a laser triangulated point.

In Figure 5 the horizontal line represents the laser plane and the jagged surface represents the surface of the subject. Point A is the point of intersection between the laser beam and the subject. Point B is the point that would be registered on the focal plane. Notice that neither the laser plane nor the line of sight of the camera (line AB) is occluded by the subject.

2. Camera occluded by the subject

When the subject has a very extreme indentation, the camera can become blocked which results in a void in the data. An example of this situation is a subject wearing wrinkled clothing. This situation is illustrated in Figure 6.

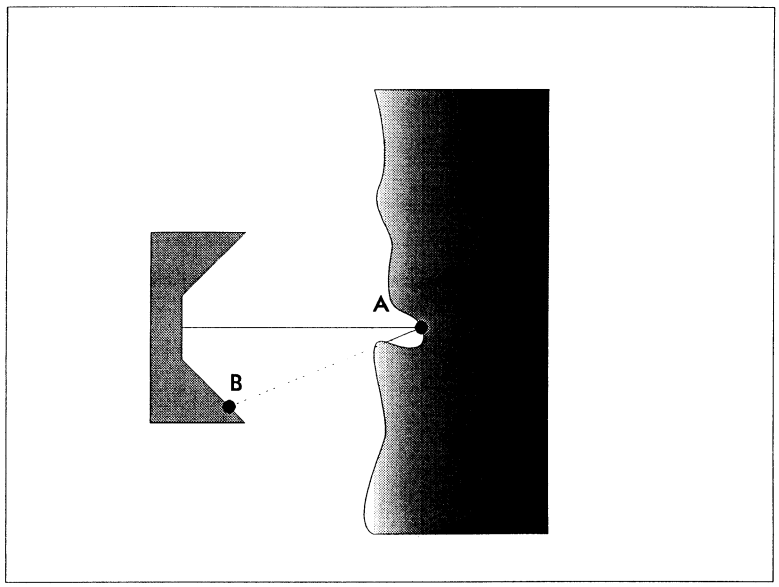

Figure 6: Occlusion caused by the subject blocking the camera.

In Figure 6 the laser intersects the subject at point A. The data point would have been collected by the camera's focal plane at point B, but the subject blocks the line of sight of the camera.

3. Laser plane occluded by the subject

When one part of the subject blocks another part of the subject from being intersected with the laser plane, data will be missed. This could occur in the armpit area of a human subject. The arm blocks the torso from receiving laser light. Figure 7 illustrates this type of occlusion.

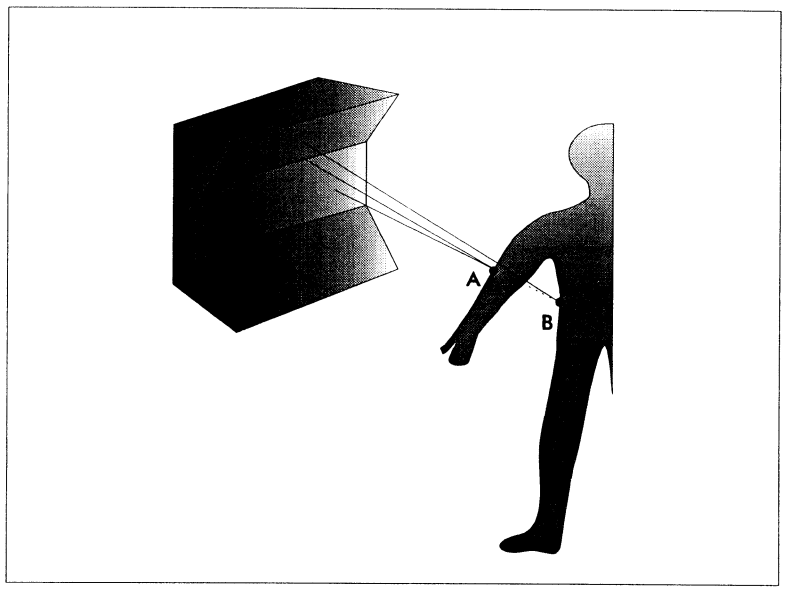

Figure 7: Laser plane occluded by the subject.

In Figure 7 one point on the subject (point A) is captured. Point B is within the line of sight of the camera. If the arm did not block the laser, point B would have been captured.

4. Poor horizontal surface coverage

When the surface of the subject is nearly parallel to the laser plane, a great deal of data can be lost between captured frames. The scanner captures data by taking a picture at a particular height. When the scan head moves to the next height, it may pass over much of the surface. This situation occurs in the shoulder area of the human subject. An example is shown in Figure 8.

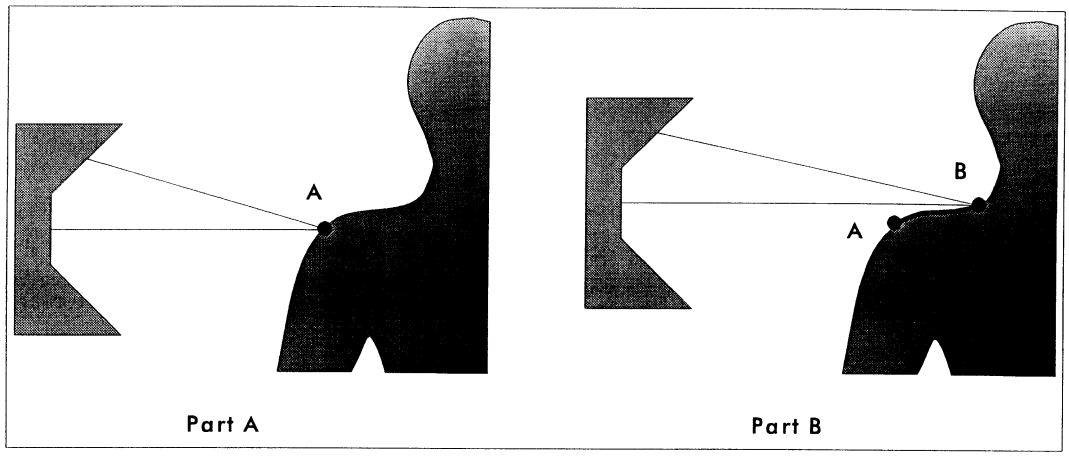

Figure 8: Missed surface due to a significantly sloping surface.

In Figure 8, Part A, the measurement of point A is an ideal case. The scanner moves in the vertical direction and takes its next measurement (Figure 8, Part B). Again the measurement taken fits into the category of the ideal case. Notice, however, the bold line between points A and B. This represents the surface area along the subject missed by the scanner.

5. Poor laser angle

Laser triangulation systems are always limited to subjects with a diffuse surface [6]. If the laser strikes at an angle nearly parallel to the subject's surface, less diffuse light can be seen by the camera. If too little light is observed by the camera, it will not register a point and that point becomes occluded.

6. Poor Laser diffusion

Situations exist in which a subject is made of a material that does not properly diffuse laser light. This can occur when the subject is too reflective, too transmissive or

too absorbtive. A mirror, for example, reflects the laser in such a way that the camera cannot detect the point at which the laser intersected the mirror. A black carbon object will absorb most of the laser light, and the camera will not be able to register a point. Darkly pigmented hair and skin can also cause occlusions in the data due to poor laser diffusion.

C. Scan coverage approach

This section will give the methodology used to create the simulation by first discussing the background issues. Next, an overview of the simulation concept will be given. Finally conceptual details of the simulation will be discussed.

1. Background

All of the types of error described can be improved or made worse by the configuration of the scanner (with the possible exception of "poor laser diffusion"). The individual aspects of the configuration affect each type of error differently. In the case for which error is due to the camera being occluded by the subject (see Figure 8), the most significant configuration parameter is the baseline distance. When the baseline is small, most crevasses in the subject will not be narrow enough to occlude the camera. However, making this distance small reduces the resolution of the scanner and increases the variance in the error of the captured data.

For the case in which error results due to the subject occluding the laser plane (see Figure 9), the most significant configuration parameter is the number of scan heads and

the position of the scan heads. Having multiple scan heads at a large variety of angles will reduce the chance that a point on the surface will be missed due to a blocked laser plane.

For the case in which error is due to poor horizontal surface coverage (see Figure 10), the most significant configuration parameter is the position of the scan heads and the path they follow. There is also a direct relationship between this type of error and the distance between each frame.

Although identifying the configuration parameters that affect the scan coverage is not difficult, it is difficult to decide what the optimal configuration parameters should be. Different configurations will be optimal for each subject. Therefore a tool that allows a designer to test designs based on the configuration of the system would be extremely helpful. This is the purpose of the software simulator.

A software simulator was created which incorporates the essential configuration parameters as discussed above. In addition a great deal of flexibility regarding the subject was made available. A few configuration parameters essential to scan coverage issues are:

- The position of the camera relative to the laser plane (baseline).
- The location of the scan heads.
- The path on which the scan heads travel.
- The number of scan heads.
- The distance covered between frames.

Many other configuration parameters can have an impact on the scan coverage (such as the camera model parameters). Control has been given to the user to adjust most of these parameters.

2. Simulation Overview

The issues and parameters discussed above were used to create a process that the simulation software would follow to generate results. The following is an explanation of that process. First, the intersection of the laser light and the subject's surface is calculated. When the intersection is found, the surface normal and the properties of the surface are also obtained. Next, a ray is formed starting from the intersection point and projecting back through the focal point of the camera. The ray is tested to see if it would intersect the subject at any point (thereby occluding the camera). If such an intersection exists, the process is stopped due to camera occlusion. If nothing is blocking the ray, the intersection point is captured by the camera according to the camera model.

Once the point is captured by the camera there are several techniques for generating the data point. Since many methods are available and because the methods themselves can be a source of error, the simulation is set up so that different techniques could be tied in with the current software. The methods for generating the resulting data points often rely on a particular scanner geometry. To keep the simulator as generic as possible, a default method was created. Using the center of the illuminated pixel, a data point in world space is generated based on the camera model.

The simulation method described allows for a variety of errors to be examined. A subject with crevasses or appendages that block the camera view will suffer from a lack of recorded points in this area. This type of subject also increased the chances that a laser ray will be blocked.

3. Simulation Details

The simulation process described above relies on having an intersection of the laser plane and the subject. An innovative approach is to model the laser plane as a set of coplanar rays. The rays start from a common origin and pass through a set of evenly spaced collinear control points. Figure 9 depicts the laser plane.

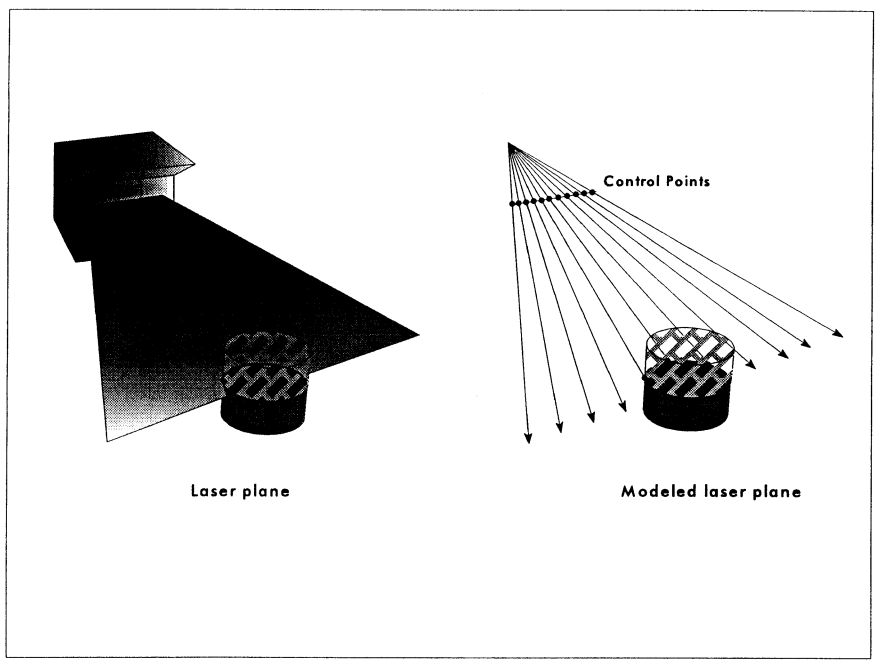

Figure 9: An actual laser plane and a plane of vectors

The left side of Figure 9 is a picture an actual scan head and laser plane. The right side shows how a set of rays starting at the origin and passing through a set of control points may simulate the plane. Using coplanar rays to represent the laser plane has many advantages. Finding the intersection of a ray with an object is less time consuming and generates more information than finding the intersection of a plane with the object. For each intersecting ray the following information is calculated: the intersection point, the

surface normal and the surface's material properties at the intersection. It is assumed that for each ray that intersects the subject, enough diffuse light will be generated to register a point on the camera focal plane.

For modeling the camera, the widely understood and accepted pin hole camera model was used. This involves multiplying a homogeneous point by a transformation matrix and a projection matrix to generate an image plane point. A correction for lens distortion was added according to the description given by Tsai [13][16].

A laser plane and one or more cameras can be put together to form a scan head. The data generated by one scan head is in two dimensions because the data must be entirely in the laser plane. By moving the scan head and thus the laser plane, a third dimension can be generated. The movement of the scan head is defined by tracks. Because the camera does not have to move relative to the laser plane, no re-calibration of the scan head needs to be done.

Like most real-world systems, the simulation software takes advantage of the linear nature of the tracks. In the simulation, however, the tracks are defined as NURBS. Therefore the simulation software could be easily adapted if future analysis should ever require a more complex path for the scan head.

Tracks can be nested so that a scan head moves along a track while the track moves along another track. This could be used to create a two-dimensional path for the scan head without the added computational complexity of moving the scan head along a NURBS.

D. Method of analyzing results

Occlusion is not easy to quantify. At first glance, finding the total number of occluded points seems straight forward because the simulator calculates every occluded point. However, because there are multiple camera views and often multiple scan heads, it is often possible that a point missed in one camera will be captured in another. The total number of captured points could be calculated and compared with the number of points captured by other configurations or some predetermined ideal. However, this has limited application because a system with many overlapping scan heads could collect more points and still contain more occlusion.

Qualitatively, however, occlusion is very easy to observe. Occlusion shows up as gaps in the data. A system suffering from occlusion will have parts of the subject with sparsely measured data or elements within the subject that are missing altogether. Many features for observing data are available in the simulator. In fact, software was written to enhance the ability to make observations of the data. This tool is called the Scan Data Analysis Tool (SDAT).

The approach used to analyze simulator occlusion is a combination of qualitative and quantitative measures. The simulator is set up to calculate the total number of points generated. Therefore, the amount of data generated can be compared between competing systems. The points are also displayed on the screen so that missing areas may be identified.

It is important to know the quality of the data generated as well as the amount of data occluded. Therefore, the simulator was created with the ability to generate statistics

relating to the quality of data. The statistics will be based on the error for each generated data point.

Two distinct ways to measure data can be identified. For discussion purposes they will be called min (minimum) error and actual error. Min error is the shortest distance between the value measured by the scanner and the surface of the subject. Actual error is the distance between the value measured by the scanner and the intersection of the laser plane. Figure 10 gives an example of these two types of error.

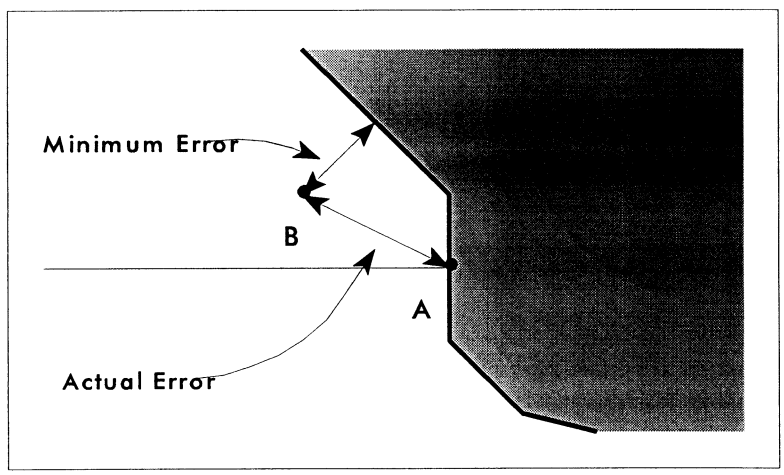

Figure 10: Examples of two ways to measure error.

The horizontal line in Figure 10 represents the laser beam. Point A represents the intersection with the subject and laser beam. Point B represents the value measured by the scanner. As seen in the figure, the min error and the actual error could be quite different if the subject has a rapidly sloping surface. Researchers evaluating a laser triangulation system usually attempt to calculate min error because the actual intersection point is unknown to them. When the surface of the subject is not sloping significantly, the actual error and the min error are the same.

Only actual error will be calculated and discussed. The reason that only actual error will be calculate is that the min error is dependent on the geometry of the subject while the actual error will be a truer reflection of the performance of the system. The simulator provides an advantage over an actual system because the intersection with the laser beam and the subject is known. Calculation of error is then just a matter of keeping a correlation between the intersection and the data point generated by the simulator.

E. Hooks for future work

Not only is the software a useful tool, but it was also written so it could be extended to be an even more precise model. Several hooks were intentionally left in the software and a short description will be provided for each of the following:

1. Consideration of subject's surface normals.
2. Consideration of subject's material properties.
3. Consideration of pixel intensity value.

1. Surface normals

When a laser plane strikes the surface of a subject, the amount of light that will be diffused will depend, in part, on the incident angle of the laser light. The simulation software was written in such a way that surface normals are available at the time the intersection is found. This surface normal could be used to calculate the incident angle and relate that to an amount of diffuse light to be seen by the camera.

2. Consideration of the surface materials

Open Inventor objects have a means of defining the object's material properties. At the time the laser ray intersection with the subject's surface is calculated, the material properties are available. By extending the software to use the material property information to determine the intensity of the diffuse light, the precision of the simulation could be increased.

3. Consideration of the pixel intensity

Currently if a laser ray is found to strike the image plane the pixel is considered turned on. Each laser ray diffused off the subject will represent an intensity. Therefore, consideration of this intensity would make the simulator more precise. To this end an intensity value is kept within the current data structure for each illuminated pixel. Extending the simulator for this capability would be a matter of putting the appropriate value in the intensity location and using the value to generate data points.

IV. Software implementation

A. Introduction

A simulator was implemented as an addition to a software package called CyScan, written and distributed by Cyberware Inc. CyScan is used to operate laser triangulation hardware and process three-dimensional data. The simulator code was mostly isolated from CyScan and only relied on CyScan to handle small infrastructure requirements. Simulated data can be imported into CyScan. In this way CyScan's three-dimensional data processing tools may be used on simulated data.

The simulation code is controlled by a fully functional menu-driven window that is independent from CyScan's menuing system. It allows for the creation, editing and saving of basic scanners and an analysis of their abilities. The following sections will describe the simulation code by first giving an overview of its structure. A brief explanation of Open Inventor is given to aid in the description of the simulation code objects and capabilities. Finally a functional description of the code is given.

B. Overview of the program structure

The code was written in C++ and a scripting language TCL/TK [17]. A tool kit, Open Inventor [2, 18, 19], was also used to interface with CyScan and simplify much of the code that relates to 3D objects. Open Inventor was chosen because there are converters from a large number of commercial CAD systems to this format. A simulator user can chose from a vast number of subjects already modeled with CAD software.

The program can be broken into two parts. The graphical user interface (GUI) and the simulator. The GUI was written with TCL/TK and the simulator was written with C++ and Open Inventor. The names of TCL/TK procedures begin with the letters OU (Ohio University) and the names of C++ classes begin with the letters OUs (Ohio University scanner).

The interaction between TCL/TK is set up in CyScan and works as follows. The user interacts with the GUI (written in TCL/TK). When the user requests an action that requires C++ code such as "scan the subject" the TCL/TK code calls a C++ global function and passes it a set of strings. The global function contains a pointer to an object that can operate the scanner simulator. Based on the strings passed to the global function the appropriate method within the simulator object is called.

The TCL/TK code is best described by giving a functional description of the GUI. OUsControler is the object that interfaces with the TCL/TK code (although often a method inherited from OUsSimulator is called directly). This would make the transition from CyScan to an independent GUI (such as one written in JAVA) simpler.

C. Explanation of open inventor

Because the simulator used Open Inventor extensively, it is important to have a basic understanding of this software package. Open Inventor is a tool kit for programming with 3D solid modeling objects. It organizes a solid model object in space onto graphs. These graphs define the object and its relationship with all other objects in the simulation world space. The graphs contain nodes that represent parts of the object.

The following are examples of nodes: point sets, primitive shapes, transformations and material properties. In the Open Inventor literature, different types of nodes are represented with icons. The icons can be arranged on a graph to depict how the nodes would be arranged in the code. Throughout this chapter, the icons used in the Open Inventor literature will be used for all scene graphs. A legend is given in the next figure.

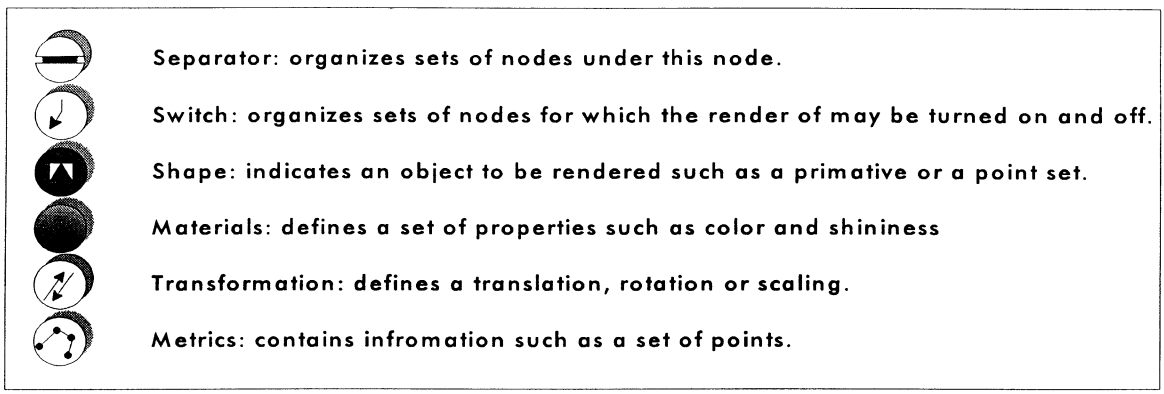

Figure 11: Open Inventor scene graph node legend.

Figure 11 contains a list of Open Inventor scene graph nodes and a description of each node. The set is not inclusive of all Open Inventor nodes but contains all that are used in this chapter. A complete list of the Open Inventor nodes can be found in the Open Inventor C++ reference manual [2].

D. Objects and their capabilities

The C++ code contains a set of classes. All classes ultimately inherit from OUsBase. The class OUsControler inherits from OUsSimulator which contains most of the other higher level classes. Figure 12 shows the class diagram. This section will highlight the most important elements from the C++ classes. Sometimes simple

explanations will suffice, but in other cases the underlying Open Inventor scene graph will be described.

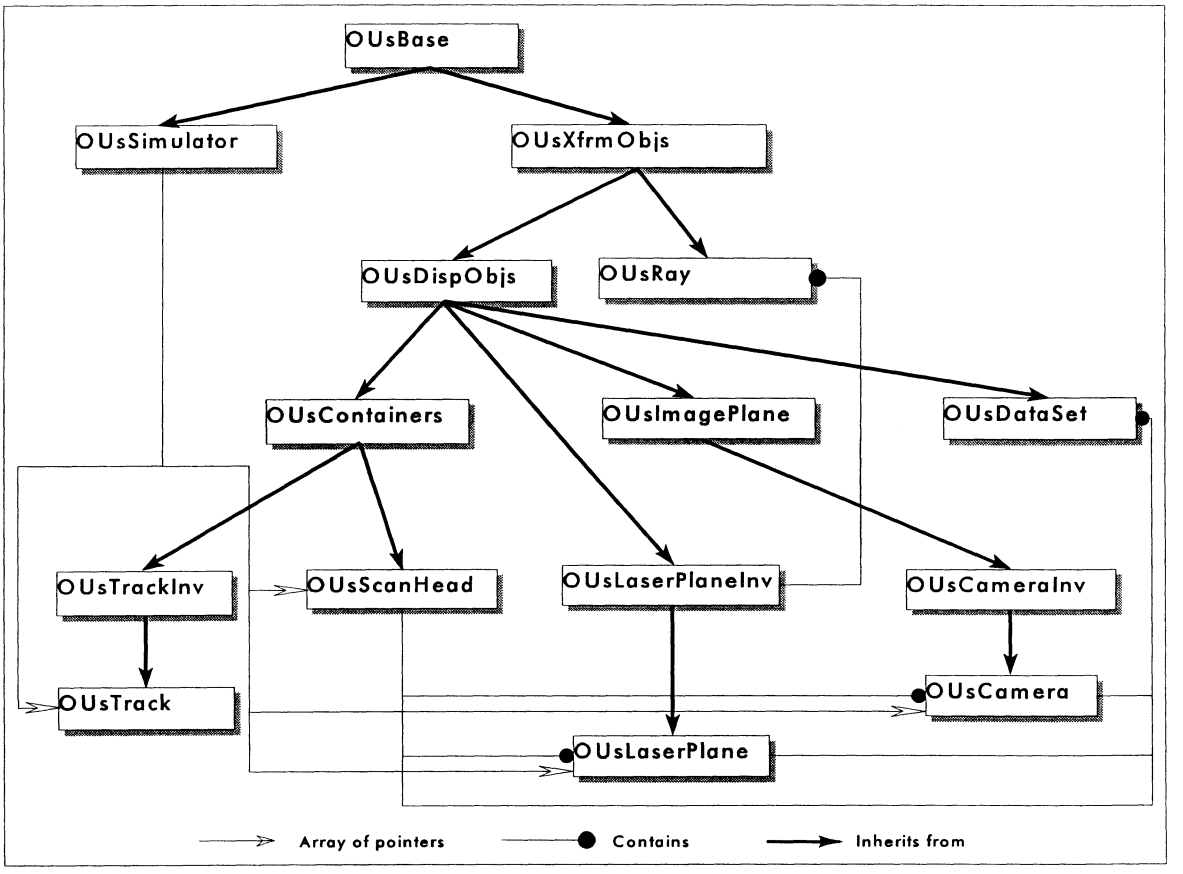

Figure 12: C++ class diagram for the simulator

In the simulator code, the higher level objects will have ties to Open Inventor. An attempt was made to isolate most of the Open Inventor functionality into specific classes designed to handle the Open Inventor aspect of the object. The lower level objects handle aspects more fundamental to the simulator and therefore do not need solid modeling capabilities. One exception to this is the foundational class OUsBase.

1. OUsBase

This is the base object for all other OUs classes. It contains static pointers to locations on an Open Inventor scene graph. It also contains a pointer to CyIvState, an object that is necessary for interacting with CyScan. OUsBase has a limited number of methods. OUsBase has the ability to do important book keeping tasks such as saving a scene graph to a file. OUsBase also sets up the scene graph to be used by the simulator the first time the OUsBase constructor is run. A diagram of the scene graph can be found in Figure 13.

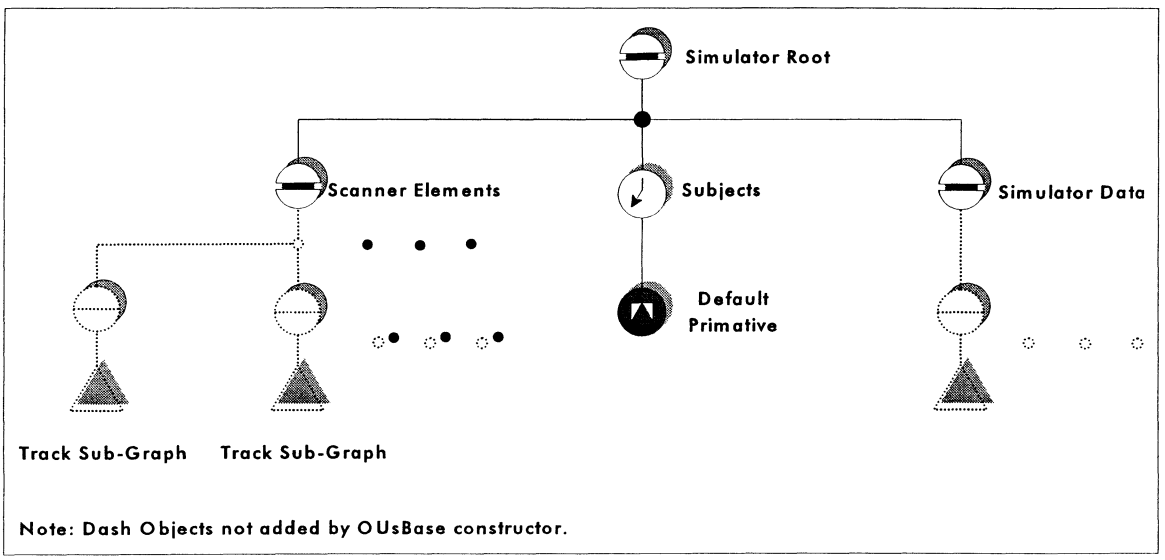

Figure 13: Base scene graph for the laser triangulation simulator.

In Figure 13 the scene graph has three main branches. The first branch is for the simulator and is shown under the "scanner elements" separator node. The second branch is for the subject and is shown under the "subjects" switch node. The last branch is for data and is shown under the "simulator data" separator node. The track sub-graph

represents the Open Inventor graph of a single track and scan head. The simulator starts with a default primitive object for the subject.

2. OUsXfrmObjs

For each object it was important to be able to transform points and objects from world space to object space. Therefore all classes except OUsSimulator and OUsControls inherit from OUsXfrmObjs.

OUsXfrmObjs contains the transformation matrices. OUsXfrmObjs does not contain the functionality needed to generate the values for these matrices. The reason for this is twofold. First the class OUsRay that inherits from OUsXfrmObjs does not need to determine the transformation matrices. It simply needs to link to another object's matrices. Second, OUsXfrmObjs does not have functionality relating to the Open Inventor scene graph (except what is inherited from OUsBase). To determine what the transform matrices should be, the scene graph interaction is essential. Therefore, establishing the values of the transformation matrices is done with the help of OUsXfrmObjs, OUsDispObjs and higher lever objects that inherit from OUsDispObjs. This functionality will be explained further below.

3. OUsRay

The class OUsRay provides a tool for the laser plane class and the camera class. To find a point of intersection between the laser plane and the subject, the laser plane uses OUsRay to project a ray into the subject's scene graph and find an intersection. Similarly, the camera class can use its OUsRay. The camera defines a ray from the

intersection point on the subject to the focal point of the camera. An intersection of this ray and the subject would indicate that the subject is occluding the camera.

4. OUsDispObjs

Objects that inherit from OUsDispObjs can display themselves on the screen and do all the things that objects represented in an Open Inventor scene graph can do. OUsDispObjs sets up a path through the scene graph by which the transformation matrices may be defined. However, finding the transformation matrices themselves requires that the inherited objects be created. Each object has a unique set of translations, rotations and scalings.

This is the lowest level class in which the methods setToScan and setToSlice are defined. These methods prepare all parts of the simulator to perform the simulation process. These functions are virtual functions that do nothing in OUsDispObjs. Each object that inherits from OUsDispObjs is expected to use these methods to perform actions prior to simulation execution.

5. OUsDataSet

There are many occasions when a set of points needs to be handled in the simulator; therefore, a class was written for this purpose. It creates a place on the scene graph and adds several Open Inventor nodes so that the points can be displayed in a variety of ways.

The OUsDataSet objects are linked together. Due to this linking, objects that contain a pointer to a specific set of points can search through all data sets. This is particularly useful for allowing the laser from one scan head to interact with other scan heads. The first laser plane object of a particular color adds a data set to the list. All other laser planes with the same color will find the first data set, and when they generate intersections, the intersections will be added to this set. The camera is simply looking for a set of intersections with the correct color.

Data set points are stored in a SoCoordinate3 node and the color is stored in a SoMaterials node. The materials node allows for several types of color: diffuse color, ambient color, emissive color, etc. The laser point intersections are the first set of points on the list with the specified emissive color.

6. OUsLaserPlaneInv and OUsLaserPlane

OUsLaserPlaneInv and OUsLaserPlane work together to implement a laser plane. OUsLaserPlaneInv handles most of the actions that relate to Open Inventor and OUsLaserPlane deals with the laser plane on a higher level. Because they are closely related, both are explained in this section. The Open Inventor graph shown in Figure 14 summarizes the structure of the laser plane.

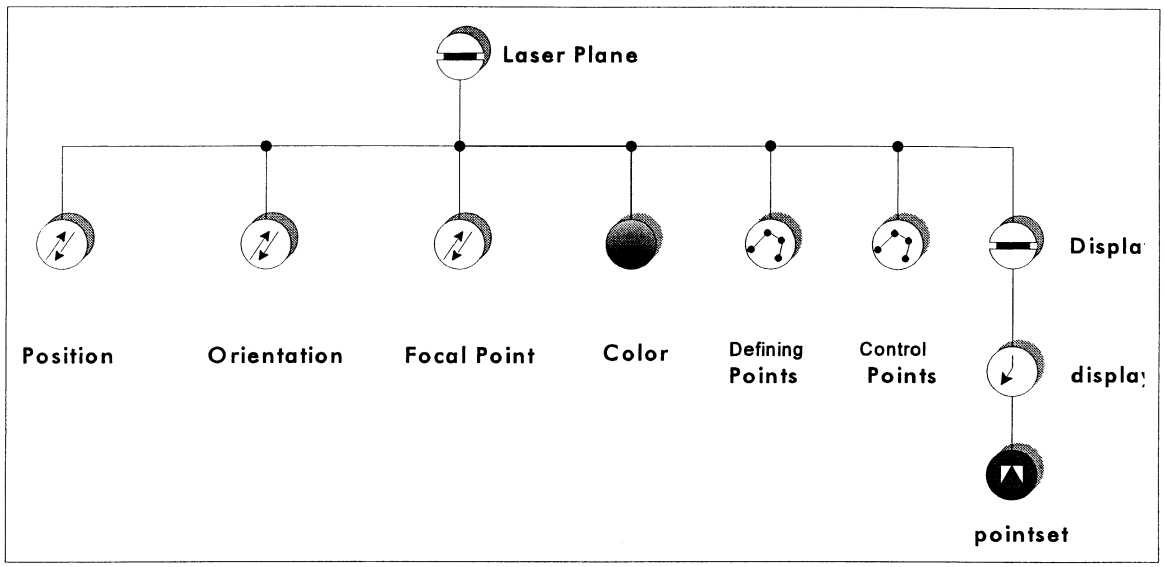

Figure 14: Open Inventor scene graph for a laser plane

The nodes "defining points" and "control points" contain the points needed to model the laser plane. The laser plane is set up in such away that the focal point is the origin of its object space. Therefore, the focal point is described as a translation relative to the object that contains the laser plane (usually a track). The Position and Orientation nodes describe any transformation relative to the containing object that the user might want. The color node is a SoMaterials node containing the emissive color that all other objects will look for to deal with a laser beam of this color. The control points define the laser plane as discussed in chapter three. The "defining points" are a set of four values stored in an SoCoordinate3 node. The first two values are the distance between adjacent control points and the distance between the first and last control point. The last two values are the pitch and tilt of the laser plane. These angles represent a misalignment of the laser plane and are set to zero for a perfectly true laser plane. The four defining points are used

to generate the control points. The display nodes and "pointset" node allow the laser plane to be rendered, but they are also used by OUsDispObjs to create a transformation path. The path will include the Position, Orientation and Focal Point nodes. When the transformation is calculated, the values in these nodes are combined to form one transformation. Once the transformation is calculated, the member OUsRay (which is not part of the scene graph) can use these transformations.

7. OUsImagePlane

The class OUsImagePlane uses the Tsai [16] camera model to represent an image plane. This class does not use the Open Inventor scene graph. It does however, provide some virtual functionality that will allow the information in this class to be stored on an Open Inventor scene graph. OUsImagePlane models an image plane as depicted in Figure 15.

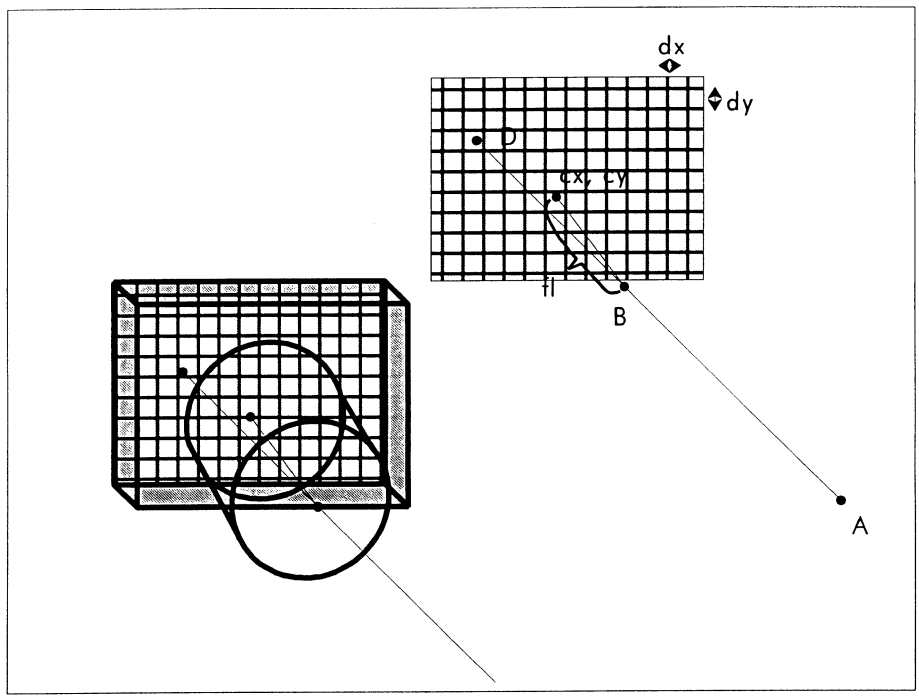

Figure 15: Image plane within camera and image plane parameters.

In Figure 15 a picture of a camera is depicted in the lower left and a diagram of its image plane is shown in the upper left. Some of the parameters used in the Tsai camera model have been labeled. Line AD is a line of sight from a world point (A) to a location on the image plane. Point B is the location of the focal point of the lens. The following is a list of camera parameters used by this class. The grid lines represent the distance from one sensing element to the next.

- Cx, Cy — The pixel values of the lens center.
- cx, cy — The lens center as given in actual coordinates.
- dx, dy — The distance from one sensing element to the next.
- Ncx — The number of sensing elements in a row.
- Nry — The number of rows of sensing elements
- Nfx — The number of pixels in a row.
- Sx — Parameter that accounts for timing discrepancies in the hardware.
- k — The first term of the radial lens distortion.
- fl — The focal length of the lens.

The definitions are sufficient explanation for most of the parameters. Sx is a term that Tsai introduced to take into account the fact that the timing that captured the pixels was slightly off from the timing that was running the sensing elements. While this is a problem for physical cameras, it is not a concern for a virtual camera. The parameter changes the ratio of the x and y values on the image plane just as if an anamorphic lens had been used. This parameter was left in, despite the fact that there would be no timing mismatches, so that an anamorphic lens could be used.

The OUsImagePlane class uses these parameters to find the pixel that would be turned on due to a light source at a point in space. The class also contains a set of points that represent the illuminated pixels. The points contain a row value, a pixel value along the row, an intensity value, and a fourth value used as an index that corresponds to the actual intersection point.

8. OUsCameraInv and OUsCamera

OUsCameraInv and OUsCamera work together to model a camera. OUsCameraInv handles most of the Open Inventor interaction while OUsCamera handles

most of the higher level functions. Both classes inherit methods and members of OUsImagePlane, but most of the interaction takes place in OUsCameraInv.

An OUsCameraInv object can be constructed from an Open Inventor scene graph. There is also a constructor that does not require a scene graph. In this case the constructor creates the scene graph. This graph contains enough information to completely define the camera and the image plane. Figure 16 depicts the Open Inventor scene graph of the camera.

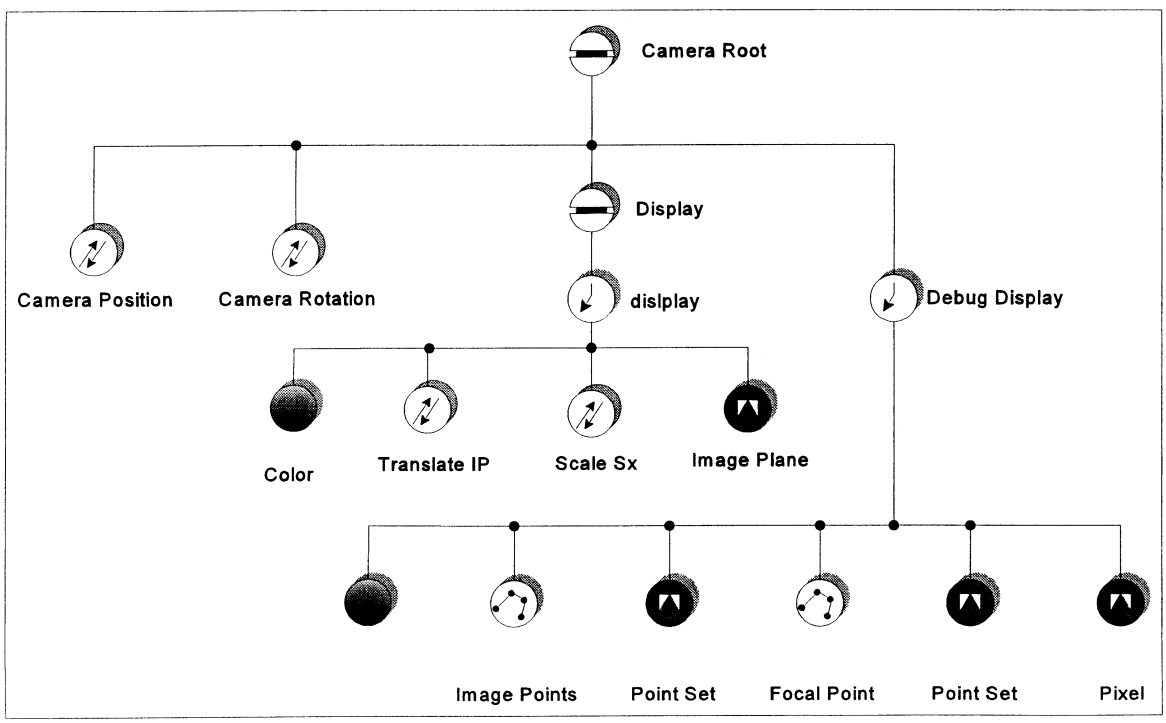

Figure 16: Open Inventor scene graph for the simulated camera.

On the graph in Figure 16 is a translation node (Camera Position) and a rotation node (Camera Rotation). In addition there are two branches that store information and help render the virtual camera on the screen. The first is the standard Display branch that is present in any OUsDispObjs. The second is Debug Display, so named because it can

display camera parameters that are not usually visible throughout the simulation process. When Debug Display is off, the sub-nodes still store the needed information, but it is not rendered to the screen.

OUsCamera provides the methods for modeling the way a point in space is captured by the image plane. This class also contains the functions for translating a point from the image plane back to the world space.

9. OUsContainers

The OUsContainers class was created so that any object derived from OUsContainers could contain any class derived from OUsDispObjs (including other OUsContainers objects). To contain another object means that the object is contained in the C++ class and within the Open Inventor scene graph. Examples of the use of OUsContainers are tracks that contain scan heads and scan heads that contain cameras and laser planes. An OUsContainers object can contain other OUsContainers, hence, a track can contain other tracks providing more complicated movement.

An OUsContainers class contains an array of pointers to OUsDispObjs. Most of the methods contained in OUsContainers class are methods that need to be executed by all contained classes. The OUsContainers class calls the method for each contained object (which usually results in a call to a virtual method) and then calls the method for itself.

10. OUsTrackInv and OUsTrack

OUsTrackInv and OUsTrack are used to model a track in a laser triangulation system. The functionality of the track is simple. Complexity arises in this object due to the structure it creates on the Open Inventor scene graph. As a result the OUsTrack class's main purpose is to provide the scene graph support needed to implement the track.

OUsTrackInv implements scene graph branches upon which the rest of the simulator can be placed. Figure 17 displays an Open Inventor scene graph for the track.

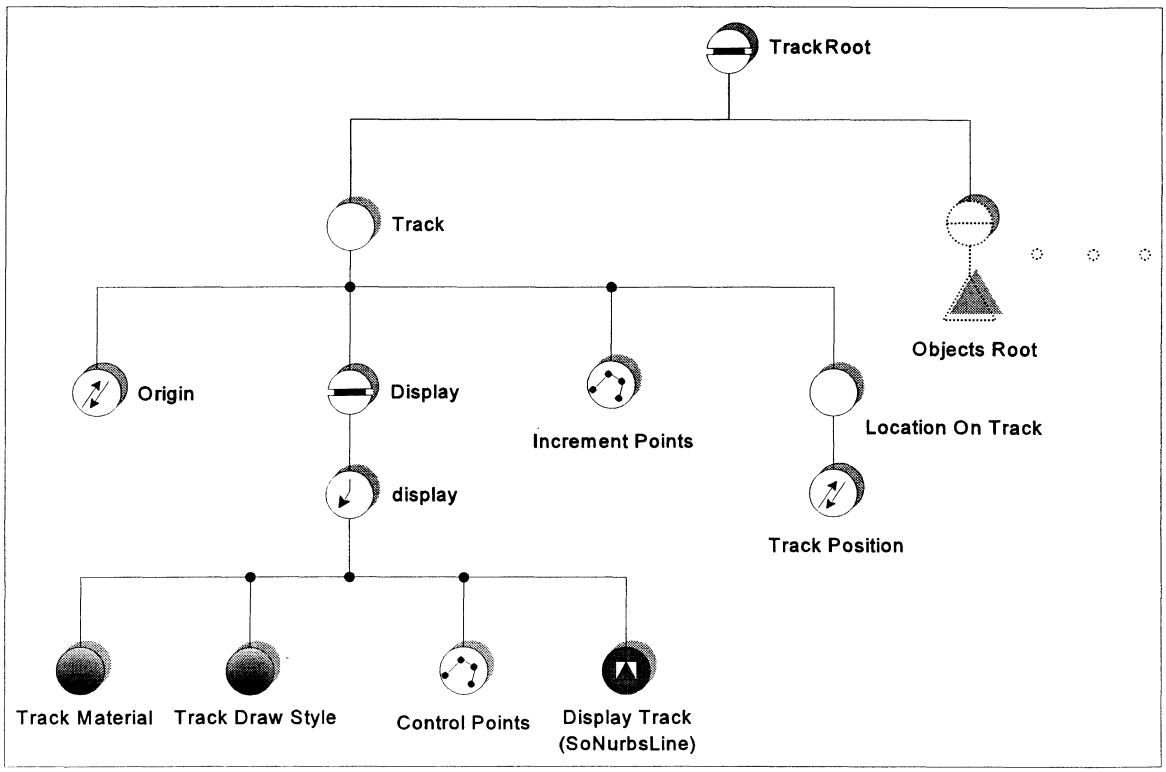

Figure 17: Open Inventor scene graph for a track

The first level of the track scene graph (shown in Figure 17) contains two nodes, one is named Track and the other is named Objects Root. Objects Root represents the

scene graph of all objects contained on the track. Because Track is a group node and not a separator node, the transformation defined for the track will also transform anything contained in the Objects Root sub-graph.

The origin is a transformation that positions the base of the track. Under the display branch is the node "Control Points." These are the NURBS control points that define the path of the track. Other nodes under the display switch are strictly for rendering the track. The node Increment Points contains information that lets the class OUsTrackInv know how much to move the objects. For speed, OUsTrackInv contains a copy of this information in a C++ variable. The primary reason to have the incriment amount on the scene graph is for saving and loading scanner files. The Track Position node is a transformation that represents how much the object has moved from the origin of the track.

11. OUsScanHead

The purpose of the OUsScanHead class is to simplify the simulation process by collecting a laser plane and two cameras into a unit that can be used to simulate the scanning process. Although OUsScanHead has a substantial amount of functionality, its scene graph is quite limited. The scene graph for OUsScanHead is implemented by the OUsContainer class.

OUsScanHead orchestrates the camera and laser plane methods in the correct sequence to generate a set of simulated points. It also contains two OUsDataSet objects: 1. the intersections of the laser plane and subject; 2. the corresponding points collected

by the simulator. These points represent the points generated by one camera frame and are referred to as a slice. There is a method for returning the points from each slice so that the simulator can assemble all the slices.

12. OUsSimulator

The class OUsSimulator contains tracks, scan heads, cameras and laser planes. It uses OUsScanHead to generate a slice. It uses OUsDataSet objects to keep statistics. The simulator only scans a single slice, but keeps data for all the slices it has scanned. A TCL/TK script handles the looping through the slices so that the GUI may update the progress indicator during the scan.

Besides the methods of performing the simulation, OUsSimulator provides methods for creating and editing the scanner.

13. OUsControls

The class OUsControls inherits from OUsSimulator and has additional functionality so that it may be used with TCL/TK. OUsControls main purpose is to exchange information between TCL/TK and C++ objects. Because the interaction with the GUI is encapsulated in this one class, the task of creating a stand alone program is simplified by this design.

E. Functional description of the simulator

The implementation of the software tool, described in the previous section, was created within the CyScan framework. It is called the Laser Scanner Simulator or SimScan for short. A description of the GUI will be given followed by an example of how SimScan may be used.

1. Description of GUI

The SimScan Control window has three main parts: the main menu, the user interface window (UIW) and the action buttons. Figure 18 displays the three main parts.

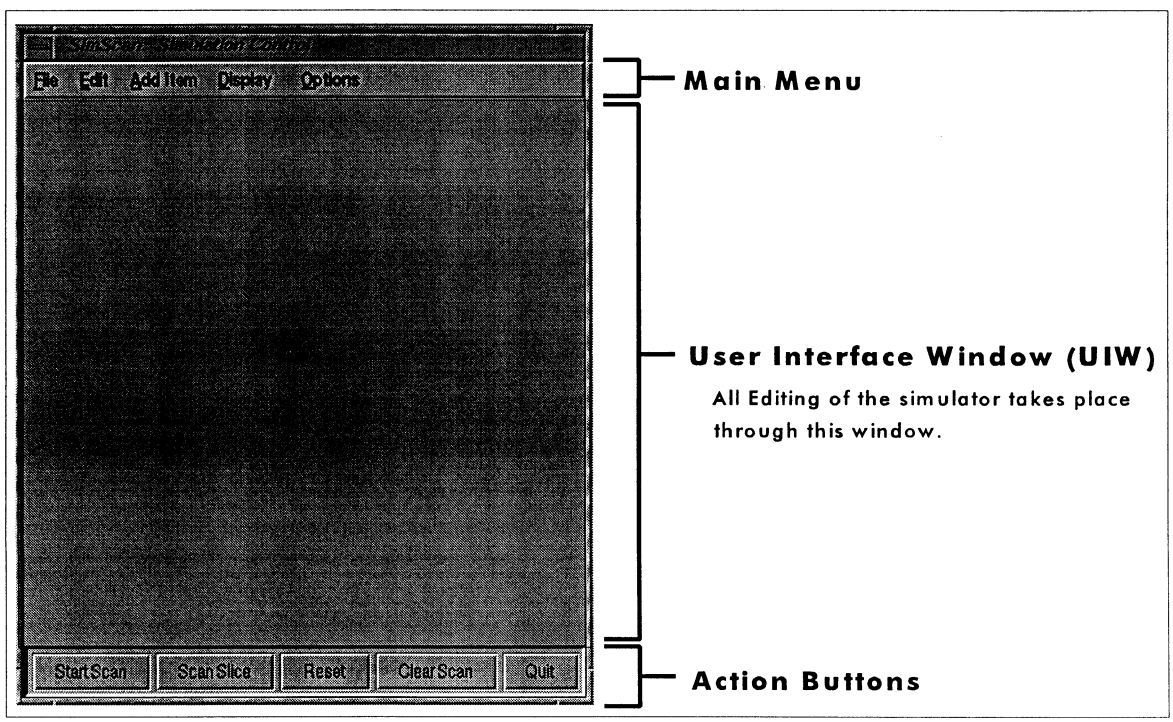

Figure 18: SimScan window for managing the simulated scanner.

Each part of the "SimScan: Simulation Control" window, shown in Figure 18, will be described in a section of this chapter. The UIW changes to accommodate the interaction needs of the user.

2. SimScan main menus

The menu shown in Figure 18 follows a traditional format. It contains a file, edit, add item, display and options menu. A picture of each menu and a description of its functionality is shown in Figure 19. The pull down menus file, edit, add, display are shown in the figure. The menu options is for future expansion and is not displayed in the figure.

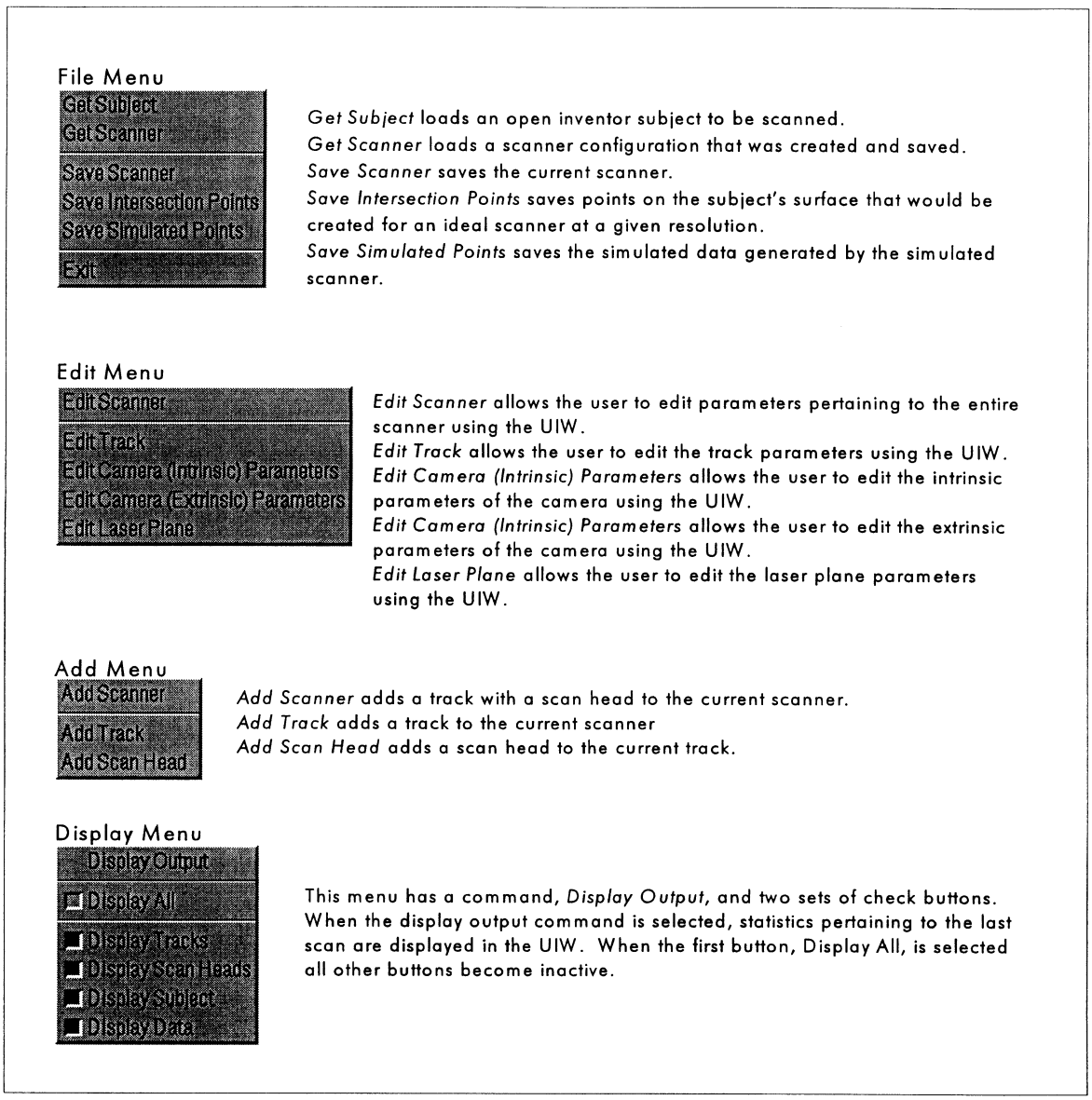

Figure 19: Menus and a description of their functionality

3. SimScan UIW (User interface window)

The UIW is used to edit a scanner or display results. The UIW reduces clutter on the desktop by using the main window area of the "SimScan: Simulation Control" (see Figure 20) for all user interface needs. The UIW has six main interface modes. The six

modes consist of five editing modes and a mode for output. Each of the editing modes alters the UIW by placing its interaction controls in the window. To summarize, the interaction modes are:

a. Edit scanner mode (Figure 20)
b. Edit track mode (Figure 21)
c. Edit camera (intrinsic and extrinsic) mode (Figure 24)
d. Edit laser plane (Figure 27)
e. Output mode

a. ***Edit scanner mode***

When the "SimScan: Simulation Control" window is in the edit scanner mode, the controls for editing the scanner are placed in the UIW shown in Figure 20.

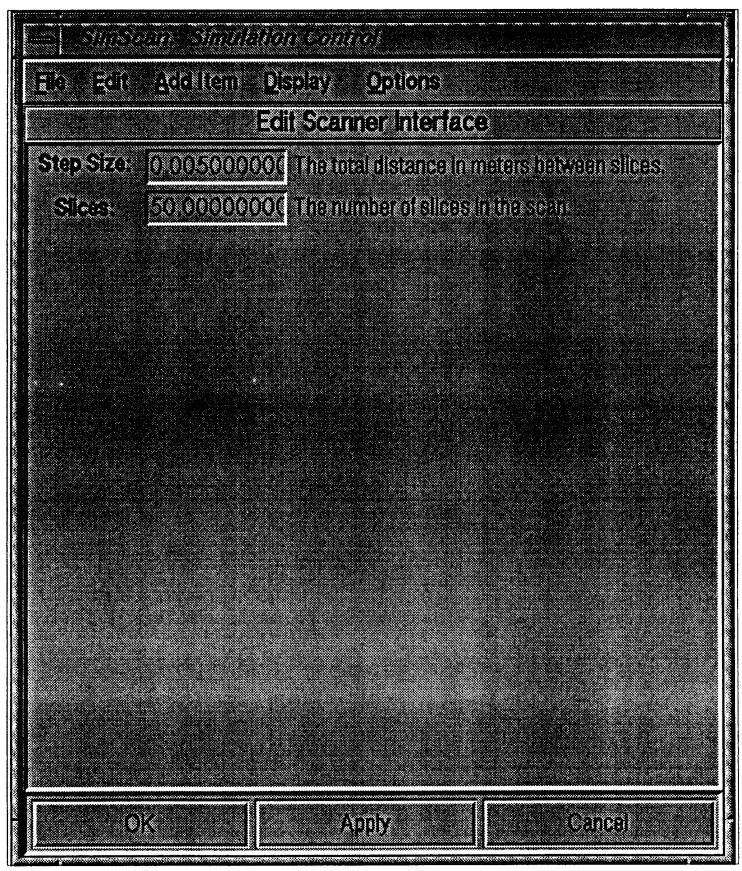

Figure 20: UIW in the edit scanner interface mode.

The simulator collects data into sets that are based on one frame of a CCD camera. Each frame of data is called a slice. In this mode, the user may control the number of slices to be scanned and the distance between each slice. These controls will impact the time it would take for a real scanner to operate, and the amount of scan coverage. It does not affect the physical geometry of the scanner itself.

b. ***Edit track mode***

When the "SimScan: Simulation Control" window is in the edit track mode, the controls for editing the track are placed in the UIW as shown in Figure 21.

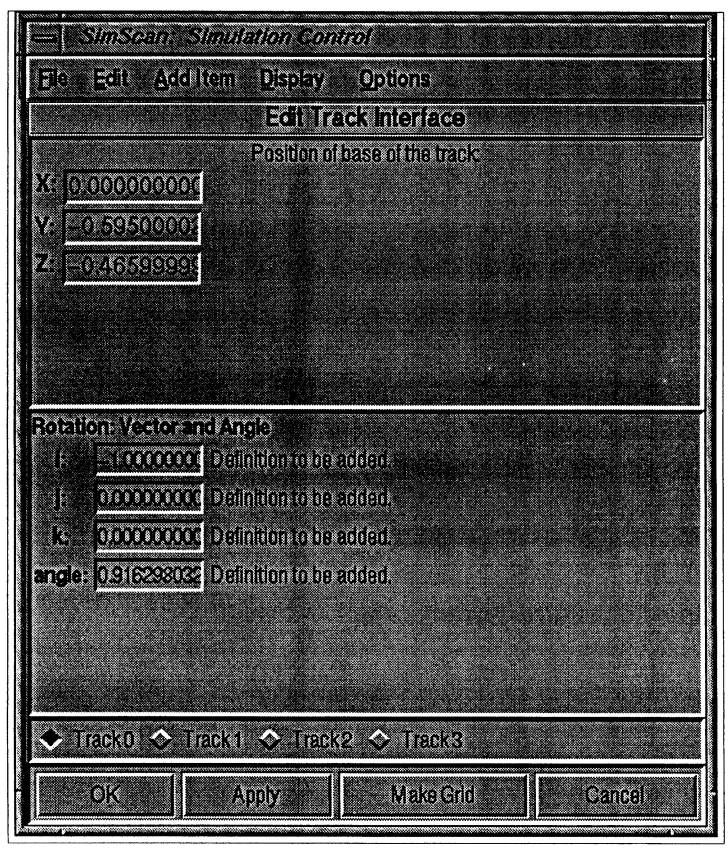

Figure 21: UIW in the edit track interface mode.

The position for the base of the track and the orientation of the track may be changed through these controls. Consider Figures 22 and 23. The figures depict a scanner simulator with two tracks and two scan heads. Also present is the subject, a chess piece placed on the table. Using the edit track mode the orientation of the track is changed. Figure 22 and 23 show the track (in the center of the figures) before and after the change.

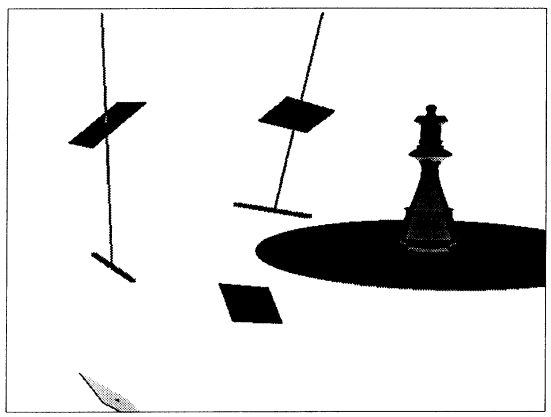

Figure 22: Track before orientation change.

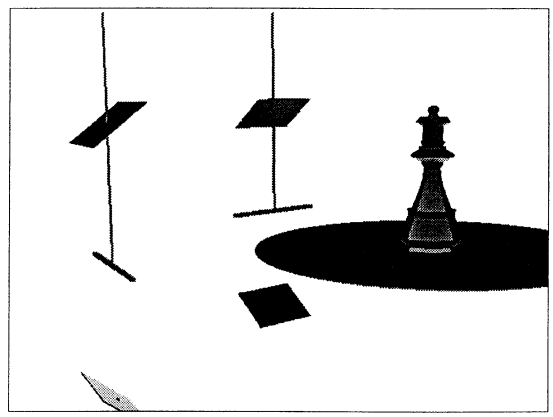

Figure 23: Track after orientation change.

c. ***Edit camera (intrinsic) mode and Edit camera (extrinsic) mode***

When the Simulation Control window is in the edit camera modes, the controls for editing the camera are placed into the UIW as shown in Figure 24.

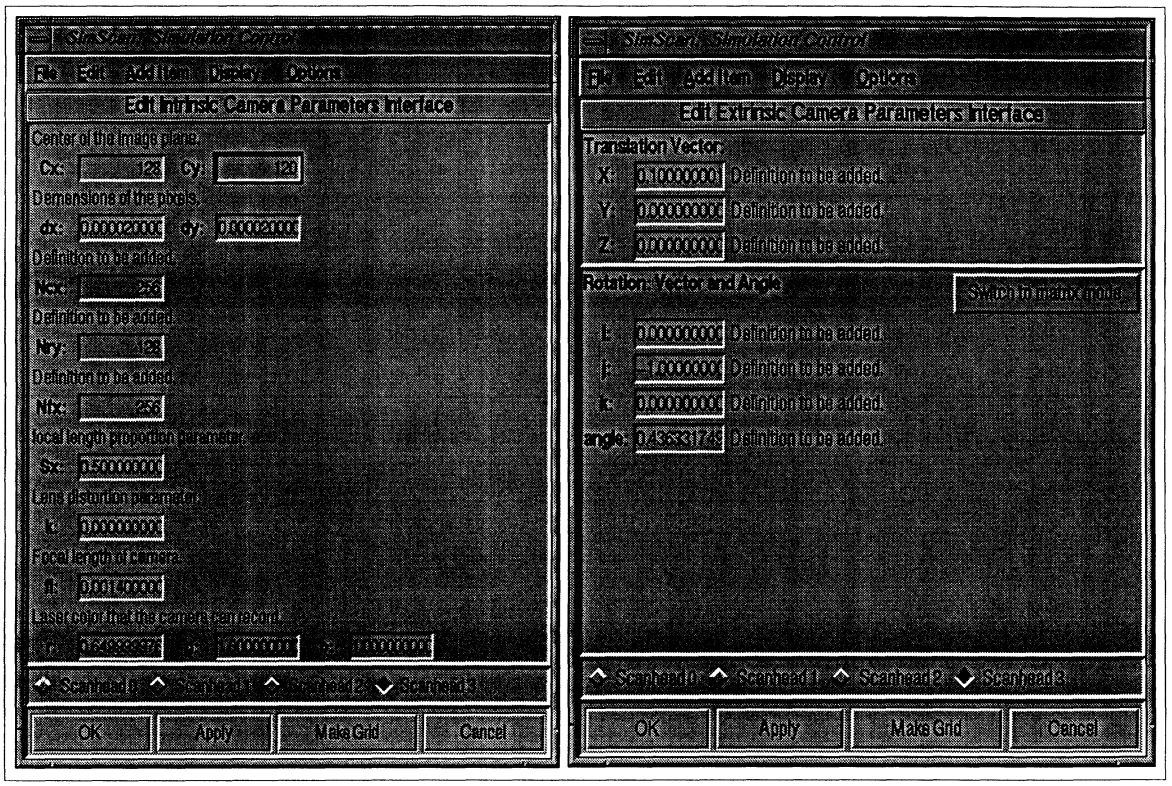

Figure 24: SimScan window for managing the simulated scanner.

These controls allow the user to change all the parameters related to the camera. Some parameters, such as focal length (fl) and pixel size(dx, dy), are not readily visible in the display area. However, parameters such as the image plane size and the image plane orientation are displayed to aid the user, as shown in Figures 25 and 26.

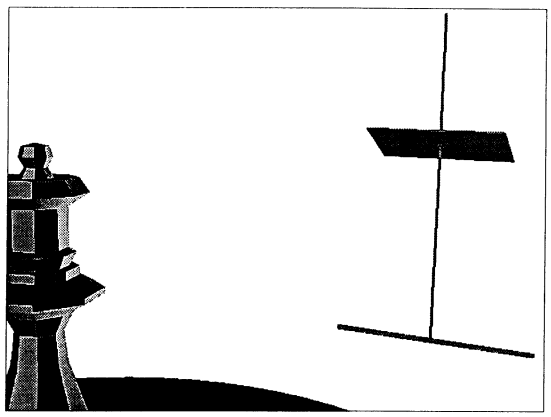

Figure 25: Image plane before alterations.

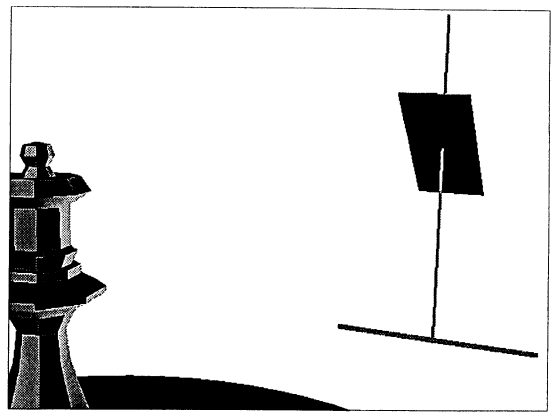

Figure 26: Image plane after alterations.

Figure 25 shows a scan head aimed at a subject (the chess piece). In Figure 26 the orientation and size of the image plane have been changed by using the edit camera mode. The color of laser plane that can be detected by the camera is set in edit camera (intrinsic) mode.

d. ***Edit laser plane mode***

The edit laser plane mode allows the user to set specifications related to the laser plane. In particular the user can define a laser plane using the width, resolution and focal point. Figure 27 shows the "SimScan: Simulation Control" window in edit laser plane mode.

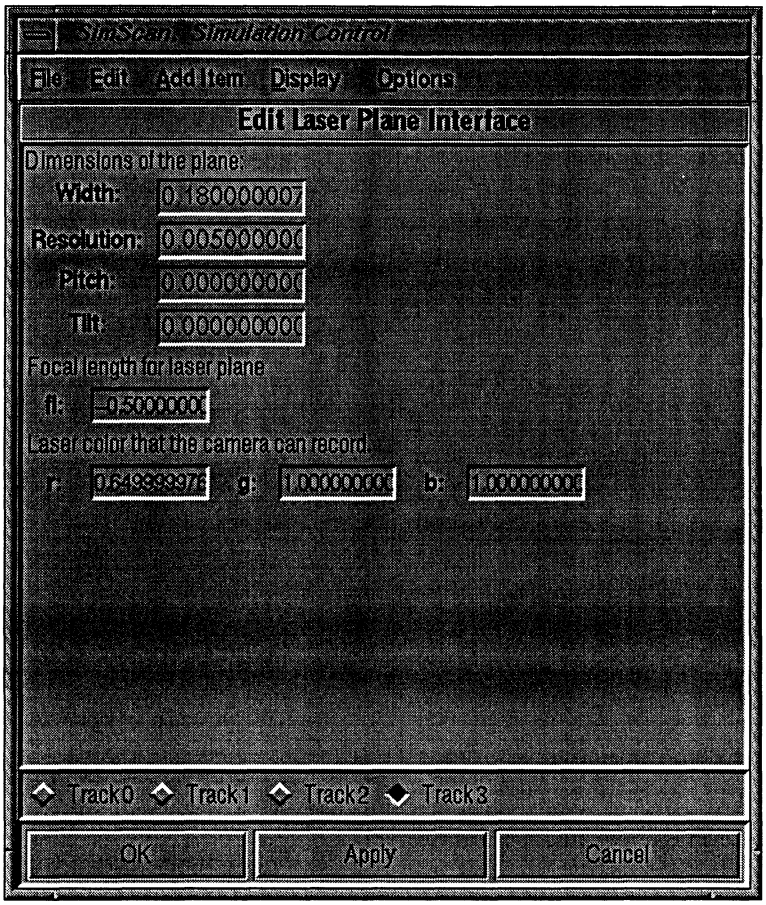

Figure 27: UIW in laser plane interface mode.

A laser plane also has a color associated with it. This color can be defined through the edit laser plane mode. The user may also set a pitch or tilt to examine the consequences of having the laser plane misaligned.

e. ***Output mode***

The output mode is used to display statistics related to a current scan. In particular the minimum error, maximum error, mean error, and standard deviation of the error are given.

4. SimScan action buttons

The action buttons are found at the bottom of the simulation control window as shown in Figure 18. These buttons allow the user to perform a variety of tasks such as "start the scan" or "make a test grid". The buttons change to reflect the type of input needed from the user based on the current mode of the UIW.

5. Example of SimScan usage

SimScan is started from the CyScan main menu under an item named OU Research Tools. When SimScan is started, the Simulation Control window appears. The user creates a virtual scanner by adding and editing tracks and scan heads. The internal parameters associated with each component can be changed. The following describes the method for creating a simulated laser triangulation system. Once created, the scanner model can be saved through the file menu for future use.

The first step is to create tracks on which the scan heads will travel. Tracks can be added through the add item menu. The position and orientation of each track are set through the edit track controls that will appear in the UIW as described in the previous section (see Figure 21). Four tracks are added and a scan head must be attached to each.

To add a scan head, the desired track is selected and the add item menu is used. Adding a scan head creates two virtual cameras and a virtual laser plane. The scan heads and tracks can be positioned around the scanning space as shown in Figure 28.

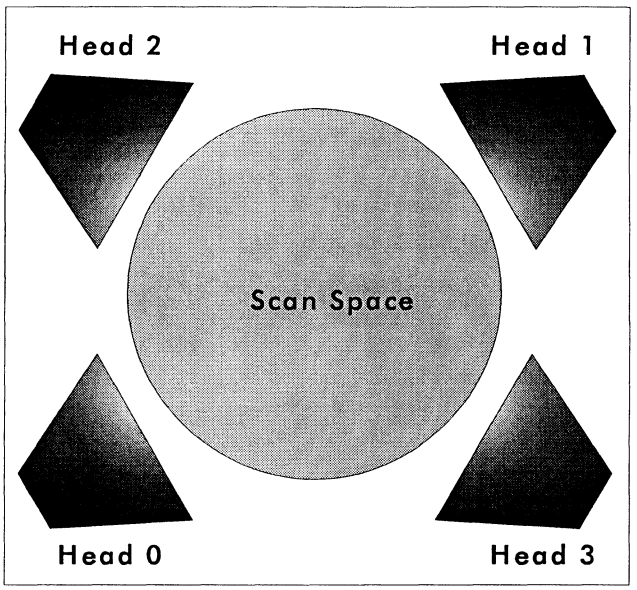

Figure 28: A diagram of a scanner configuration.

In Figure 28, the scan heads are depicted as trapezoids with the long base in the direction in which scanning is performed. The scan heads are labeled head 0, head 1, head 2 and head 3. In the center of the four scan heads is the scan space. The scan heads can collect data in this region.

For both a real and a virtual camera, each pixel on the camera's image plane corresponds to a region of scan space. If these pixels are projected into the scan space and connected, a grid is formed. The mathematical basis for this grid is discussed in chapter two. The grid is unique for each scan head of the scanner. SimScan generates a set of points corresponding to the intersections of these grids.

SimScan generates a point corresponding to every tenth pixel on the focal plane. Using the grids, it is easy to see how the region of scan space is covered by the cameras in a scan head. Figure 29 shows the projection grid for a scan head.

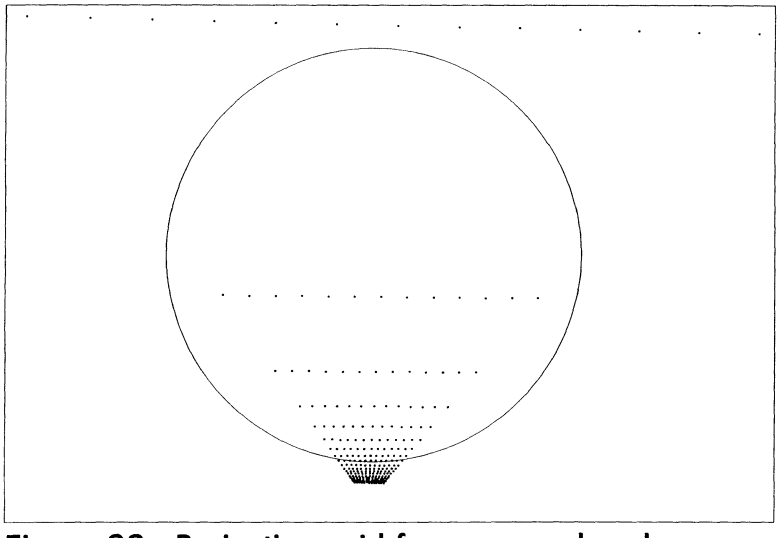

Figure 29: Projection grid from a scan head.

The points shown are from the SimScan projection of the image plane. The circle represents the scan space. It is clear that the area closer to the camera is detected by a larger number of pixels. Therefore the quality of data will decrease as the subject is moved away from the scan head.

The camera parameters may be adjusted through the edit menu item: Edit Camera. When the Edit Camera command is chosen, the user may edit camera parameters through the UIW (see Figure 25). The camera can be adjusted to achieve better coverage of the scanning space. Figure 30 shows the projection grid after improvements to the scan head have been made.

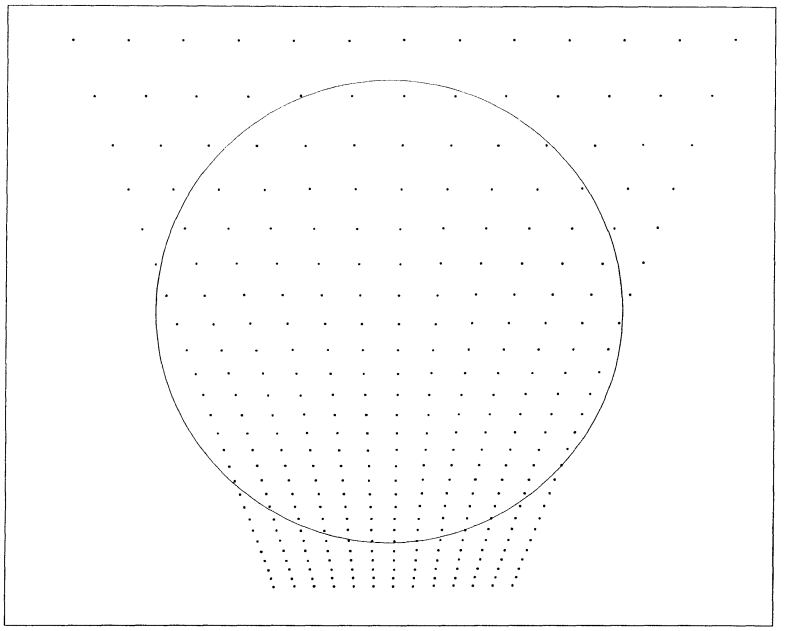

Figure 30: Improved projection grid

In Figure 30, the grid from an improved scan head is projected to the scanning space. We see that the scan head covers more of the scan space and that the space between grid points is more uniform.

Once a scanner is created, a subject must be placed in the scanning area. By default a sphere is the assumed subject. To use any other subject, an Open Inventor file of the subject must be available. The Open Inventor subject can be loaded into SimScan, using the file menu item: Get Subject.

Once the subject is in place, the user should select the number of slices to be scanned and the spacing between slices. This is done through the edit menu item: Edit Scanner. The simulation is activated by the action button labeled Start Scan. Starting at the base, SimScan scans up the subject.

While the simulation takes place, a blue sliding bar shows how much of the scan has been completed. After the simulation is complete, information about the quality of data will appear in the UIW. Specifically the minimum, maximum, mean and standard deviation of the error will be given. The simulated points can be imported to CyScan to be viewed and manipulated with CyScan tools, just like data collected from hardware.

V. Results of software use

A. Introduction

As result of this research, a program can be used to examine the trade-offs between occlusion and scanner precision. The program that implements the simulator is given in Appendix A. This chapter will give examples of how the simulator can be used to observe the trade-offs between data precision and data occlusion. High resolution cameras are capable of generating enormous amounts of data. When the amount of data exceeds 10,000 points per scan head images can not be effectively depicted on paper. As such, the resolution of the cameras used has been kept at a level for which they would produce a reasonable printed image.

The first example given in this chapter does not involve a simulation but demonstrates how the simulator may be compared to an actual scanner. Next is a description of the results of two cylinders measured with the simulator. Cylinders are used because the data generated is more easily described and understood. Also, due to the simplicity of the objects, the simulation process was accomplished more rapidly, allowing for a greater number of tests to be conducted. Cylinders are like many real world objects such as an arm or torso of a human body. After the cylinders, a human body model is used in the simulator. The purpose of using the human body is twofold. First, it demonstrates that the simulator can work for complex real world subjects. Second, scanner manufacturers such as Cyberware and Textile Clothing Technology Corporation have an interest in making scanners capable of extracting clothing measurements from human subjects.

B. Comparison to an actual scanner

To validate the simulation software, a comparison with a real laser triangulation system was made. The laser triangulation system that was used in the comparison was the Cyberware whole body four head scanner (Cyberware WB4). Two approaches of comparing the simulator to the Cyberware WB4 were used. One approach was to compare the resulting data from the hardware and the simulator. While the data sets from each system look identical, there are a number of issues that prevent quantification in this approach. Numerical inconsistences in the comparison could be a result of discrepancies between the computer solid model and the actual object. A far more useful, and less subjective solution is to compare the test grid generated by the simulator to a similar pattern generated by the Cyberware WB4.

A test grid represents the configuration of the scanner and can be generated for the Cyberware WB4. A similar grid can be generated by SimScan. The following figure shows a comparison of the test grids generated by SimScan and CyScan.

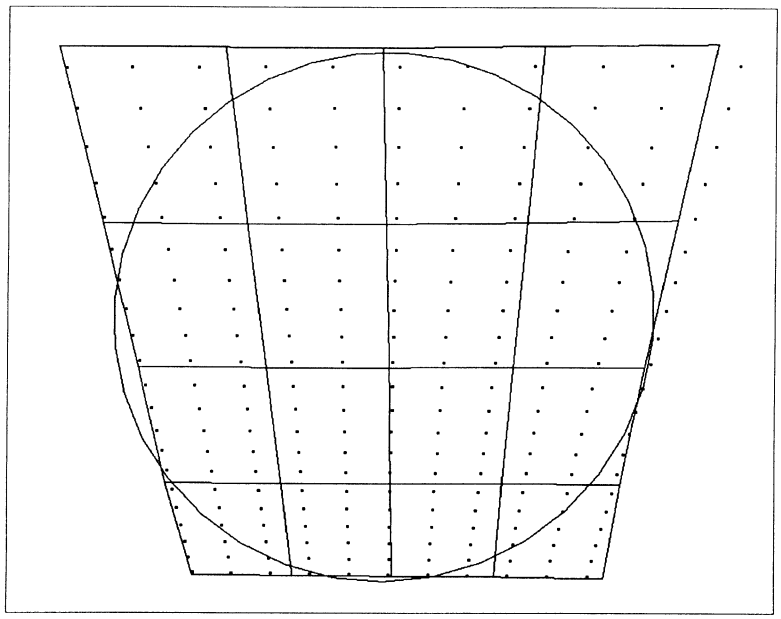

Figure 31: Cyberware WB4 test grid and SimScan test grid points.

In Figure 31 a test grid generated for the Cyberware WB4 is shown with lines. Test grid points that represent the scanner modeled by SimScan are displayed on top of the Cyberware WB4 test grid. The test grid generated by CyScan contains five rows of points (depicted as the intersection of lines) with five points in each row. The points on the boarder of the grid are on the boarder of the used image plane. This SimScan grid is generated using every 50^{th} pixel starting with the first point on the image plane. A result of the test grids being generated in different fashions is that only the first row and column of points correspond precisely with the Cyberware WB4 grid. However, the overall pattern of the grids should match. Figure 31 shows that the actual scanner and the simulated scanner match quite well. The row of points on the bottom and the column of

points on the left match precisely. There are discrepancies due to the differences in the way the grids are generated and slight deviations from the optics of the actual scanner. In particular, the ratio of focal lengths for the anamorphic lens, the position and orientation of the lens and image plane center are needed for a precise hardware match. These specifications were considered proprietary information by the scanner manufacture and where not made available.

C. Analysis with cylinders

For each of the tests conducted in this section, two cylinders were used. An image displaying the two cylinders is shown in Figure 32

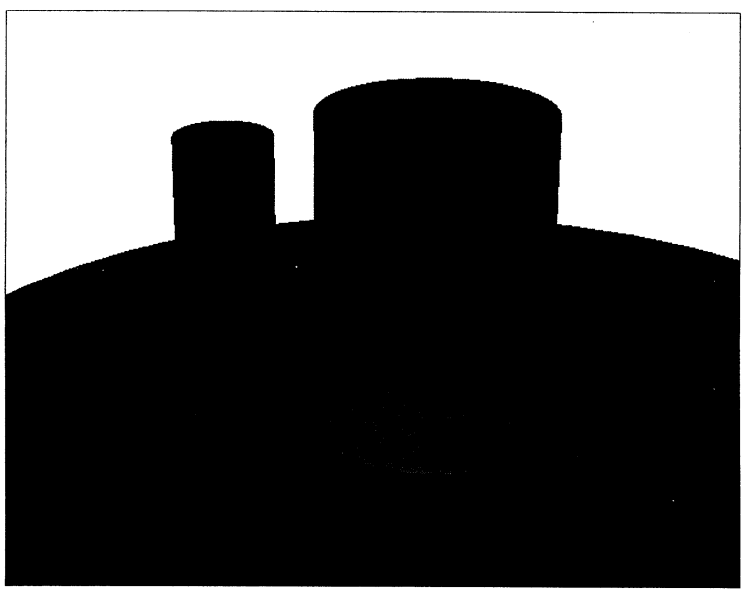

Figure 32: Two cylinders used in SimScan.

The smaller cylinder had a radius of 0.04 meters and a height of 0.3 meters. The larger cylinder had a radius of 0.1 meters and a height of 0.3 meters. The cylinders were

located on a scanning platform because such a platform, which would exist for a real scanner, can impact the performance of a scanner.

1. Two head vs. four head

The first issue that was considered was a two head scanner versus a four head scanner. Figure 33 shows the basic layout of the scanner and the orientation of the cylinders.

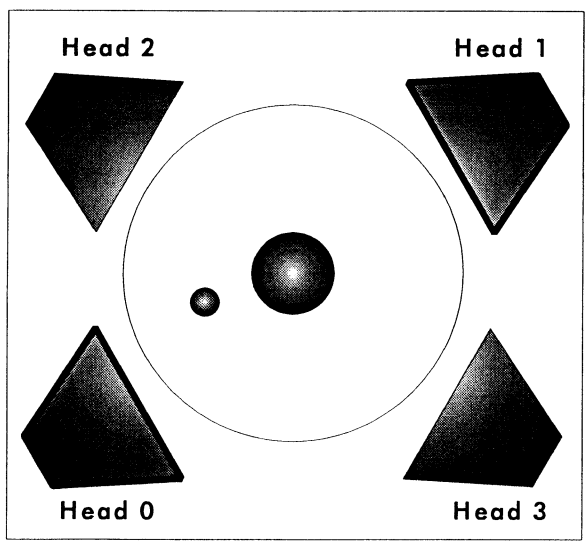

Figure 33: Diagram showing scanner subject orientation.

The scan heads are represented with trapezoids with the long base in the direction in which scanning was performed. They are labeled head 0, head 1, head 2 and head 3. Scan head 0 and scan head 1 were used for the two head model and all four scan heads were used for the four head model.

All four scan heads used the same basic parameters. The laser plane was parallel to the x-y plane with no pitch or tilt. The cameras were 0.25 meters from the laser plane and the camera angle was 45 degrees. A description of the scan heads specifications can be found in Appendix B under type A scan heads. Both the two and the four head scanners used two different colored lasers. Using multiple laser colors limits the amount that one scan head can use the laser plane created by another scan head.

The results of the scans are shown in Figure 34 for the two head scanner and Figure 35 for the four head scanner.

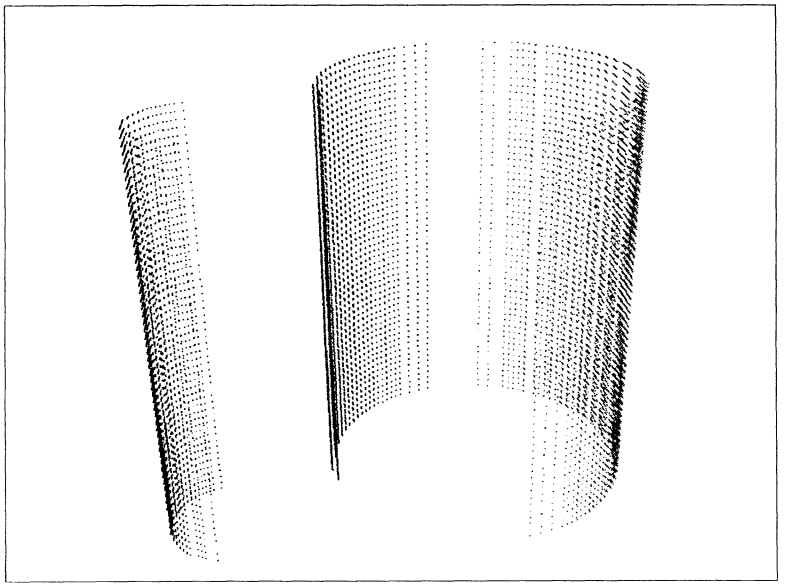

Figure 34: Results of a two head scanner simulator

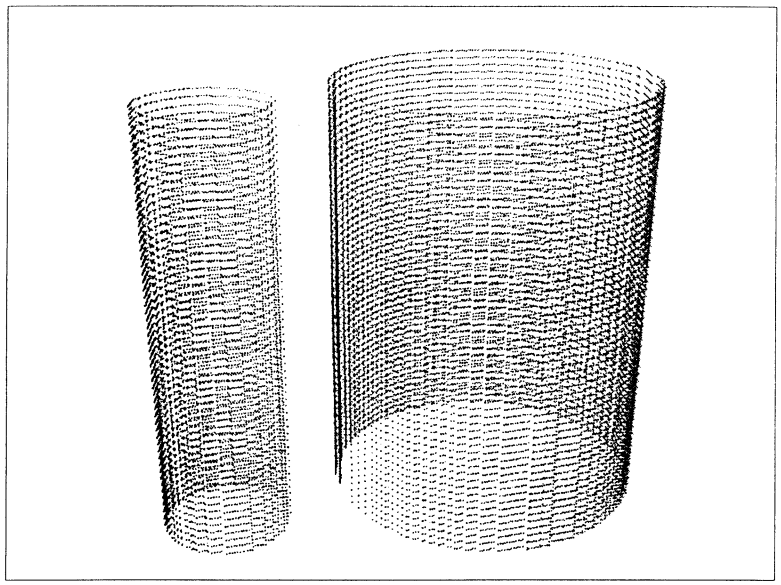

Figure 35: Results of a four head scanner simulator

Figures 34 and 35 show a rendering of points in 3D space that were generated by the two and four head scanners respectively. Points generated by each scan head are depicted in a different color. The figures relating to the accuracy of the data are given in the following tables.

Table 1: Statistics for a two head simulated scanner

	Scan Head 0	Scan Head 1
Number of Points	7,280	6,664
Mean Error	.874 mm	.996 mm
Min Error	.092 mm	.062 mm
Max Error	1.74 mm	1.80 mm
Std. Deviation	.413 mm	.465 mm

Table 2: Statistics for a four head simulated scanner

	Scan Head 0	Scan Head 1	Scan Head 2	Scan Head 3
Number of Points	7280	6664	9240	9296
Mean Error	.874 mm	.996 mm	1.064 mm	1.045 mm
Min Error	.092 mm	.062 mm	.061 mm	.052 mm
Max Error	1.74 mm	1.80 mm	2.28 mm	2.21 mm
Std. Deviation	.413 mm	.465 mm	.482 mm	.465 mm

As seen in the tables, the accuracy of the data was about the same for the two head scanner and the four head scanner. However, as seen in Figures 34 and 35, the four head scanner did not suffer from the occlusion problems present in the two head scanner.

2. Orientation of the subject

To consider what impact the orientation of the subject has on the results, consider once again the case of the two head scanner. Instead of comparing the results to that of a four head scanner, the two head scanner results were compared to the results of the same scanner with the subject rotated 90 degrees. The scan head parameters are still of the type A scanner as given in appendix B. Figure 36 illustrates the layout of the two scanning simulations that were conducted and compared.

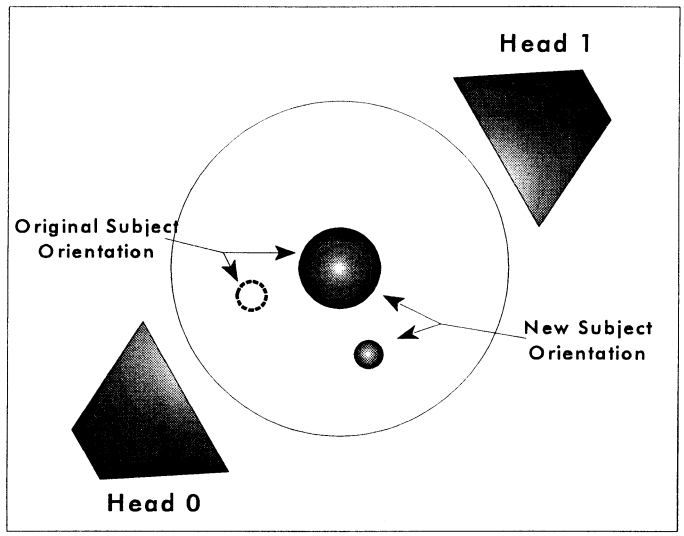

Figure 36: Layout diagram showing old and new subject orientation

In Figure 36 the original cylinder locations are shown with a dashed line (although the center cylinder is the same in both orientations). The solid objects represent the new location on which the analysis was performed. Scan head 0 and scan head 1 have not changed position, but the cylinders have been rotated 90 degrees about the center of the platform.

The results for the cylinders in their original position have already been given in Figure 36 and Table 1. The results that are generated after the cylinder is turned 90 degrees is shown in Figure 37 and Table 3.

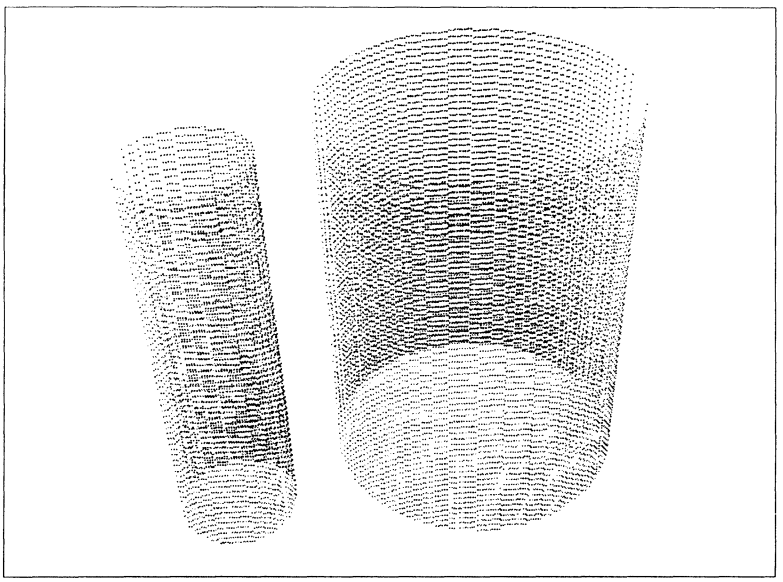

Figure 37: Results of a two head scanner simulator with subject rotated

In Figure 37 we can see that there was more data between the two cylinders than the scan in Figure 34. Overall there was significantly less occlusion in the second orientation than the first.

Because the scan head configuration has not changed at all (we are still using Type A scan heads) the precision of the data as described by the statistical information looks very much the same as for the other two head scanner. The statistical information is given in Table 3.

Table 3: Statistics for a two head simulated scanner; rotated subject

	Scan Head 0	Scan Head 1
Number of Points	9184	9464
Mean Error	1.08 mm	.994 mm
Min Error	.068 mm	.062 mm
Max Error	2.23 mm	1.87 mm

These tests indicate that the occlusion present in a scan was dependent on the subject and its orientation as well as the number of scan heads used. They also show that the data quality is not drastically impacted by the subject or the number of scan heads. In examples with a human model the dependence of occlusion on scanner geometry will be further demonstrated.

D. Analysis with a human model

To demonstrate the simulators effectiveness with a more complex real world object, a human body model was created and named Sam (simulation analysis man). Sam was created with demo tool called gview that is provided by Silicon Graphics as a standard software package. Figure 38 shows a picture of Sam.

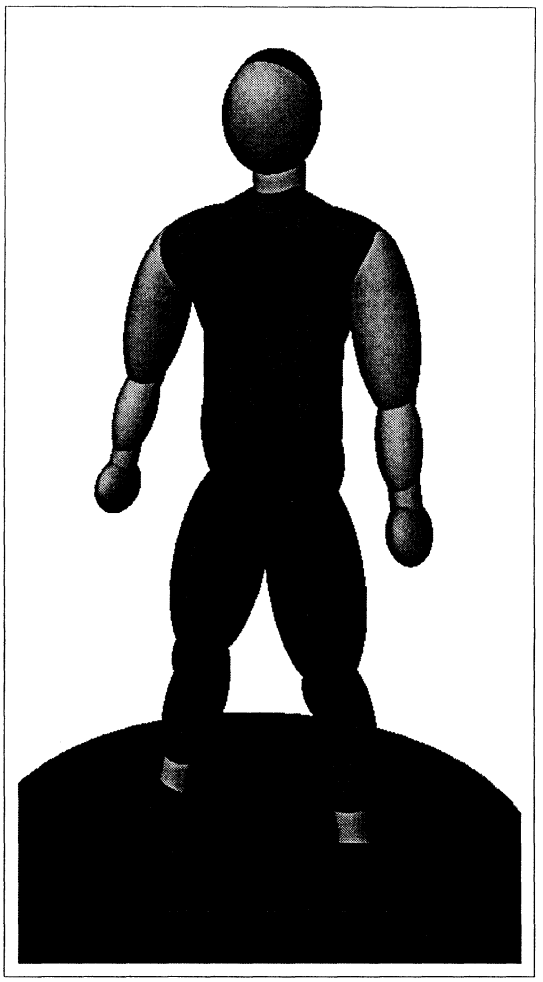

Figure 38: Sam the scan analysis man.

In Figure 38, Sam is standing in a common anthropometric pose. He is standing upright with his head facing the same direction as his body. His arms and legs are spread slightly apart. Sam stands about five feet ten inches tall. He has features and proportions similar to an actual human body.

Two simulations will be conducted on this subject. In the first, the simulator is set up to obtain relatively precise results. In the second, the simulator is changed to reduce

the amount of occlusion. When the simulator is set up to reduce the amount of occlusion, a reduction in data quality can also be observed. For both simulations, the scanner will be in the configuration shown in Figure 39.

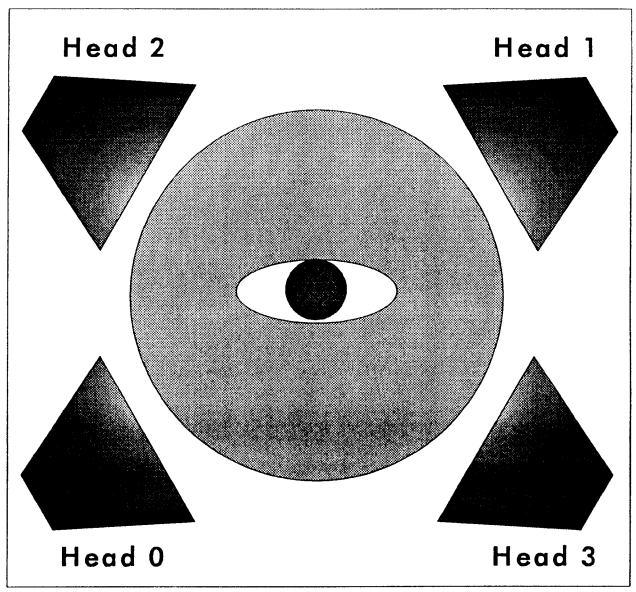

Figure 39: Sam's orientation relative to the scan heads.

In this figure the scan heads are shown as trapezoids with the large side facing the direction of the subject. Sam's head and shoulders are depicted by a circle and ellipse respectively. Sam is facing in the direction of the ellipse's minor axis.

1. Scan for precision

The scanner set up for precision has a baseline distance between the camera and laser plane of .6 meters and a camera angle of 60 degrees. For a complete listing of parameters used for the scan heads see type B scan heads in appendix B. The human

body model was positioned in the simulator so that it was facing between scan head 0 and scan head 3 shown in Figure 39.

Given the resolution of the camera, the results of the scan are very precise. However, there are portions of the scan that are not covered. Figure 38 shows some of the results from this scan.

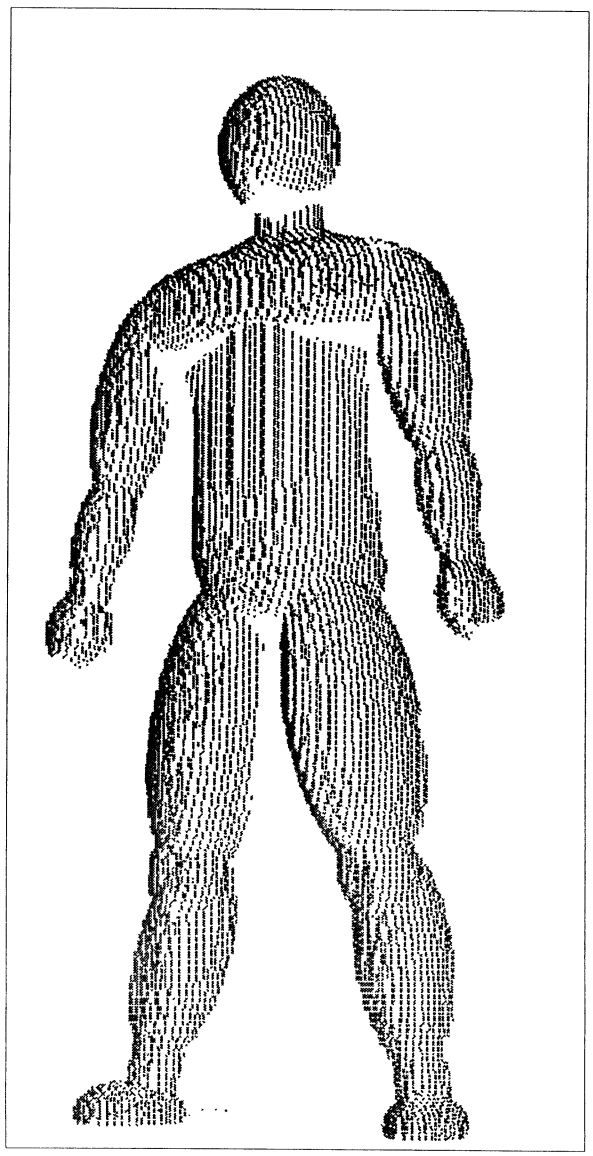

Figure 38: Results of a precise simulated scan of Sam.

There is too much information from all four scan heads to view the results in a static two dimensional image. Data from the two scan heads covering Sam's back have been turned off to simplify viewing. We can see where information between the subject's legs, under the subjects arm pits, and under the subjects chin has been occluded. The statistics on the error of the points, as summarized in Table 4, also indicate that the measurement error is very precise.

Table 4: Statistics for a precise four head scanner

	Scan Head 0	Scan Head 1	Scan Head 2	Scan Head 3
Number of Points	9987	10121	10139	9993
Mean Error	3.41 mm	3.48 mm	3.62 mm	3.52 mm
Min Error	.006 mm	.007 mm	.019	.007 mm
Max Error	10.1 mm	9.80 mm	9.79 mm	10.1 mm
Std. Deviation	1.50 mm	1.58 mm	1.59 mm	1.53 mm

2. Scan to prevent occlusion

It is clear that the system described above suffers from occlusion problems despite the fact that four heads are used. To overcome this problem, the baseline will be reduced from 0.6 meters to 0.1 meters and the camera angle decreased to 25 degrees. The remaining camera parameters are identical to those in previous section. For complete details see Appendix B type C.

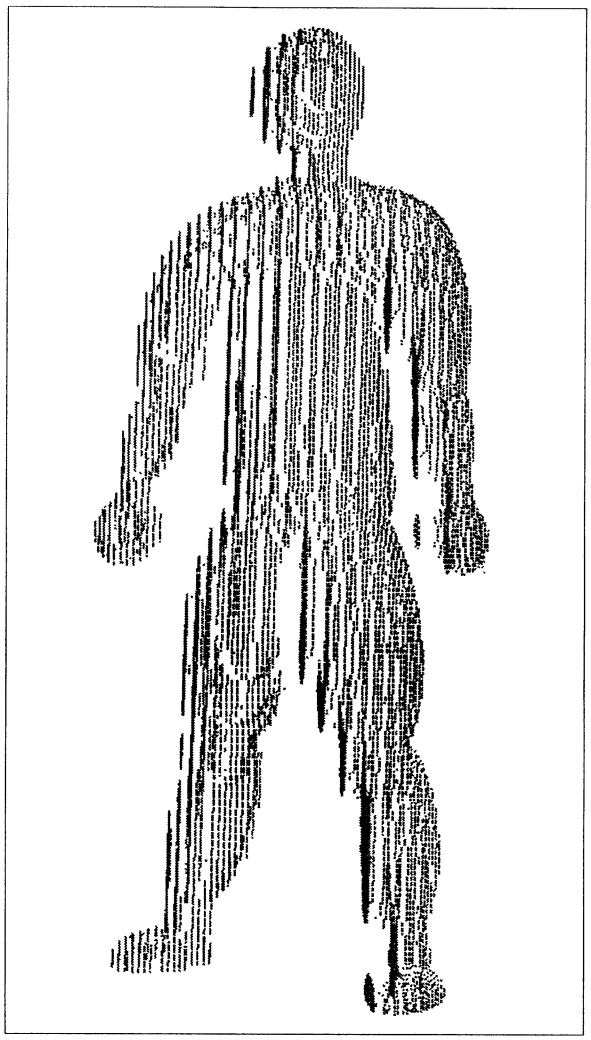

Figure 39: Results of an occlusion preventing simulated scan of Sam

A scan with the altered system reveals better coverage, but a significant reduction in data quality. Figure 39 depicts the results of this scan. Compared to the previous figure, there were no gaps in the data due to occlusion. The obvious gaps in this image were due to data quality.

By observing Sam's head, it is clear that the chin (which was occluded in the previous scan) has been covered. It is also clear that, due to the poor geometry, the data is grouped together in what looks like vertical lines. This is due to the camera being at such an extreme angle that the space covered by a single pixel is quite large.

In addition to the qualitative differences, the statistics show quantitatively that the type B scanner generates points of a lower resolution. Table 5 presents the statistics.

Table 5: Statistics for a low occlusion four head simulated scanner

	Scan Head 0	Scan Head 1	Scan Head 2	Scan Head 3
Number of Points	11531	12116	12109	11547
Mean Error	7.83 mm	8.06 mm	8.12 mm	7.77 mm
Min Error	.037 mm	.035 mm	.035 mm	.037 mm
Max Error	35.2 mm	35.3 mm	35.3 mm	36.6 mm
Std. Deviation	5.24 mm	5.79 mm	5.83 mm	5.21 mm

Notice that not only does the mean error increase, but the variation in the error also increases. This is expected and can be understood through the mathematical example given at the end of chapter 2 or through the projection grids in appendix B. The total number of points increased slightly which could also indicate fewer points were occluded.

As mentioned at the beginning of this chapter, the resolution of the scan heads were intentionally designed to produce a limited number of points for demonstrative

purposes. The scans of Sam were conducted with cameras having about four times the resolution. This was accomplished by increasing the resolution of the cameras. The results were similar with more data (about 120,000 points per scan head) and better precision. The results for system with a large baseline produced a mean error of about 0.3 mm and the system with a short baseline produced a mean error of about 0.6 mm.

VI. Discussion

SimScan, a laser triangulation simulator, is a powerful tool for examining the performance of various hardware designs. SimScan has many advantages over building and testing prototype designs which include ease of use and a high degree of flexibility. SimScan has the ability to test many different designs with many different subjects.

SimScan can be used in the design process of a new piece of scanning equipment by revealing strengths and weaknesses of particular scanner geometries. SimScan can help determine if a particular piece of equipment to be purchased will meet the requirements of a given application. SimScan could also be used to determine the best orientation for scanning a particular subject.

To develop an accurate simulator, it was important to study the various ways in which measurement inaccuracy can result. First, an analysis of error in stereo vision systems by Nurre [12] was extended to describe the sources of error in a laser triangulation system. Next, the sources of occlusion in a laser triangulation system were discussed. Finally, the impact of the sources of error and occlusion were incorporated into computer models that could be implemented through software.

SimScan, written with C++, Open Inventor and TCL/TK, was implemented in an object oriented fashion. Hooks were left in the software to accommodate future researchers who wish to use SimScan. A fully functional menu driven GUI was written using TCL/TK. This makes use of the software simple and intuitive to use. All information relating to a particular scanner configuration was stored in Open Inventor

scene graphs. This allows scanners to be saved and retrieved once they have been created.

The contribution of this research is simulation software and the underlying laser plane model. In addition, results were given that demonstrate the effectiveness of certain scanner configurations. These results address the issue of the number of scan heads, the orientation of the subject and the length of the baseline distance. The issues were addressed through several simulations. The simulations generated qualitative results in the form of a graphical depiction of the simulated data and quantitative results in the form of statistics relating to the simulated data.

As shown in this thesis, the software simulation of a laser triangulation system is an effective research tool which is highly adaptable to current and future applications.

References

[1] *Manual of Photogrammetry,* 4 ed. Falls Church VA: American Society of Photogrammetry, 1980, pp. 117-138

[2] *Open Inventor C++ Reference Manual,* Reading, MA: Addison-Wesley Publishing Company, 1994

[3] Clark J., "Variable Resolution Depth Imaging by Using Elliptical Mirrors," *Applied Optics,* vol. 36, no. 7, pp. 1621-1625, Mar. 1997.

[4] Dalgish, R. L. *et al.*, "Hardware Architecture for Real Time Laser-Range Finding Sensing by Triangulation," *Review of Scientific Instruments,* vol. 65, pp. 485-491, Feb. 1994.

[5] Dorsch, R. G. *et al.*, "Laser Triangulation: Fundamental Uncertainty in Distance Measurement," *Applied Optics,* vol. 33, no. 7, pp. 1306-1314, Mar. 1994.

[6] Dwulet, R. J., "Cutting Costs With Laser Triangulation," *Machine Design,* pp. 110-112, Nov. 1994.

[7] Farcy, R. and Damaschini, R., "Triangulation Laser Profilometer As a 3D Space Perception System for the Blind," *Applied Optics,* vol. 36, no. 31, pp. 8227-8232, Nov. 1997.

[8] Farcy, R. and Damaschini, R., "Triangulation Laser Profilometer As a Navigational Aid for the Blind: Optical Aspects," *Applied Optics,* vol. 35, no. 7, pp. 1161-1166, Mar. 1996.

[9] Fu, K. S. *et al.*, *Robotics: Control, Sensing, Vision and Intelligence,* New York: McGraw-Hill Book Company, 1987, pp. 304-328

[10] McCullough, R. *et al.*, "Laser-Optical Triangulation Systems Provide New Capabilities for Remote Inspection of Interior Surfaces," *Materials Evaluation,* vol. 53, pp. 1338-1345, Dec. 1995.

[11] Mizunama, M., "A Displacement Measurement Method by Laser Beam Scanning for Mapping the Interior Geometry of Pipes," *Journal of Pressure Vessel Technology,* vol. 16, pp. 188-192, May 1994.

[12] Nurre, J. H. and Hall, E. L., "Error Analysis for a Stereo Vision System," *Conference Proceedings of SME,* pp. 2-33 to 2-52, June 1986.

[13] Reimar K. and Tsai, R. Y., "Techniques for Calibration of the Scale Factor and Image Center for High Accuracy 3D Machine Vision Metrology," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 10, no. 5, pp. 713-720, Sept. 1988.

[14] Rioux, M., "Laser Range Finder Based on Synchronized Scanners," *Applied Optics,* vol. 23, no. 21, pp. 3837-3844, Nov. 1984.

[15] Saint-Marc P. *et al.*, "A Versitile PC Based Range Finding System," *IEEE Jounal of Robotics and Automation,* vol. 7, no. 2, pp. 250-256, Apr. 1991.

[16] Tsai, R. Y., "A Versatile Camera Calibration Technique for High-Accuracy 3D Machine Vision Metrology Using Off-the-Shelf TV Cameras and Lenses," *IEEE Jounal of Robotics and Automation,* vol. RA-3, no. 4, pp. 323-344, Aug. 1987.

[17] Welch, B. B., *Practical Programming in Tcl and Tk,* Upper Saddle River, New Jersey: Prentice Hall PTR, 1995

[18] Wernecke, J., *The Inventor Toolmaker,* Reading MA: Addison-Wesley Publishing Company, 1994

[19] Wernecke, J., *The Inventor Mentor,* Reading MA: Addison-Wesley Publishing Company, 1994

[20] Wu, J. *et al.*, "Weld Bead Placement System for Multipass Welding," *IEEE Proceedings of Science Measurement and Technology,* vol. 143, pp. 85-90, Mar. 1996.

[21] Zeng, L. *et al.*, "Two-Directional Scanning Method for Reducing the Shadow Effects in Laser Triangulation," *Measurement Science Technology,* vol. 8, no. 3, pp. 262-266, Mar. 1997.

Appendix A

On the following pages is a program that implements a laser triangulation system simulator. The files are given in the following order:

OUsBase.c++
OUsBase.h
OUsCamera.c++
OUsCamera.h
OUsCameraInv.c++
OUsCameraInv.h
OUsContainers.c++
OUsContainers.h
OUsControls.c++
OUsControls.h
OUsDataSet.c++
OUsDataSet.h
OUsDispObjs.c++
OUsDispObjs.h
OUsImagePlane.c++
OUsImagePlane.h
OUsLaserPlane.c++
OUsLaserPlane.h
OUsLaserPlaneInv.c++
OUsLaserPlaneInv.h
OUsRay.c++
OUsRay.h
OUsScanHead.c++
OUsScanHead.h
OUsSimulator.c++
OUsSimulator.h
OUsXfrmObjs.c++
OUsXfrmObjs.h

```
/*************************************************************************

  Filename:  OUsBase.c++
  Revision:  2.00

  Date:      19 Feb 1998
  Author:    Jeff Collier

  Description:  For a description of this class see OUsBase.h

*/

#include "OUsBase.h"
#include <iostream.h>
#include <Inventor/So.h>

/*************************************************************************
Static and Global Declarations
*************************************************************************/
short OUsBase::num_OUsBases = 0;
SoSeparator *OUsBase::simulator_root = NULL;
SoSwitch *OUsBase::subjects = NULL;
SoSeparator *OUsBase::scanner_elements = NULL;
SoSeparator *OUsBase::simulator_data = NULL;
CyIvState *OUsBase::state = NULL;



/*************************************************************************
Constructors - Destructors
*************************************************************************/

/*************************************************************************

  Method:  Constructor

  Remark:  Creates a base object for all OU simulator classes.  A pointer
     to CyIvState allows access to all data maintained by CyScan.  Pointers
     to several important nodes on the simulator scene graph are also
     created.  The number of OUsBase objects created is kept.   All of this
     data is stored in static variables so that all derived classes will be
     using the same things.

   Params:  state - The first object should be created from a constructor
     in which state is CyIvState *.  All other call state should be NULL.

*/

//This code defines two constructors OUsBase() and OUsBase(CyIvState *)
OUsBase::OUsBase(CyIvState *newstate) {
```

```
//Initialization needed for all OUs classes.
if (newstate != NULL) state = newstate;
//Signal Warning if state not intialized.
if(state == NULL) cerr << "Warning -- state never initialized\n";

if(num_OUsBases == 0) {
  if ((simulator_root != NULL)  ||
          (subjects != NULL)        ||
          (scanner_elements != NULL))
    cerr << "Error -- Simulator Root exists before OUsBase object!!\n";

  else {
    //Creating the basic graph structure for the simulator
    simulator_root = new SoSeparator;
    simulator_root->setName(SbName("Simulator_Root"));

    scanner_elements = new SoSeparator;
    scanner_elements->setName(SbName("Scanner_Elements"));

    subjects = new SoSwitch;
    subjects->whichChild = SO_SWITCH_ALL;
    subjects->setName(SbName("Subjects"));

    simulator_data = new SoSeparator;
    simulator_data->setName("Simulator_Data");

    //Inserting  the simulator into CyScan's scene graph
    state->root->addChild(simulator_root);
    simulator_root->addChild(scanner_elements);
    simulator_root->addChild(subjects);
    simulator_root->addChild(simulator_data);

    //Create a default primative under subjects
    //Sphere
    SoSphere *prim = new SoSphere;
    prim->radius = .1;

    //Cylinder
    //       SoCylinder *prim = new SoCylinder;
    //       prim->radius = .1;
    //       prim->height = .1;

    //Cube
    //       SoCube *prim = new SoCube;
    //       prim->width = .1;
    //       prim->height = .1;
    //       prim->depth = .1;
    subjects->addChild(prim);
  }
}
num_OUsBases++;
```

```
  //cerr << "Number of OUsBases:  " << num_OUsBases << endl;
}

/*****************************************************************************

  Method: Destructor

  Description:  Reduce the number of OUsBase objects until there are none
    left.  If it is the last OUsBase object clean up scene graph.

*/
OUsBase::~OUsBase() {

  //Clean up OUs class
  num_OUsBases--;

  if (num_OUsBases == 0) {
    //All OUsBase objects have been destroyed.
    //Remove all inserted nodes from scene graph.
    simulator_root->removeAllChildren();
    state->root->removeChild(simulator_root);

    simulator_root = NULL;
    subjects = NULL;
    scanner_elements = NULL;
  }

  //Debug
  //cerr << "Number of OUsBases:  " << num_OUsBases << endl;
}


/*****************************************************************************

  Method:  Read File

  Description:  Any OUs object may need the ability to open a file and read
    a scene graph from it.

*/
SoSeparator *OUsBase::readFile(const char *filename) {

  //Open the input file
  SoInput scene_input;

  if (!scene_input.openFile(filename)) {
    cerr << "Cannot open file " << filename << "!\n";
    return NULL;
  }

  //Read the whole file into the database
```

```
  SoSeparator *graph = SoDB::readAll(&scene_input);

  if (graph == NULL) {
    cerr << "Problem reading file\n";
  }

  scene_input.closeFile();
  return graph;
}


/******************************************************************************

  Method:  Write Full Graph

  Description:  This writes the scene graph created by OUsBase to a file.

*/

int OUsBase::writeFullGraph(char *filename) {


  SoOutput *out = new SoOutput;
  FILE *output;
  output = fopen(filename, "w");
  out->setFilePointer(output);

  SoWriteAction *Write = new SoWriteAction(out);
  Write->apply(simulator_root);

  fclose(output);

  return (1);
}
/******************************************************************************

  Filename:  OUsBase.h
  Revision:  2.00

  Date:    19 Feb 1998
  Author:  Jeff Collier

  Description:  This class is a foundational class for a simulator of
    a structured light scanner.

*/
#include "../../src/CyTclKit.h"
#include "../../src/CyScan.h"
#include "../../src/CyTcl.h"
#include <iostream.h>
```

```
#ifndef _OUSBASE_
#define _OUSBASE_

class OUsBase {

public:
  //Constructors -- Destructors
  OUsBase(CyIvState *newstate = NULL);
  ~OUsBase();

  SoSeparator *readFile(const char *filename);
  int writeFullGraph(char *filename = "graph.iv");

protected:
  //Inventer nodes for easy access to scenegraph.
  static SoSeparator *simulator_root;   //Entire Sim under this node.
  static SoSwitch *subjects;            //Objects to be scanned by Sim.
  static SoSeparator *scanner_elements; //Items that make up the scanner
  static SoSeparator *simulator_data;   //Points generated by Simulator

  //Other helpful variables
  static CyIvState *state;              //Pointer to CyScan's "state"

protected:
  static short num_OUsBases;            //Number of objects instanciated.

};

#endif
/*****************************************************************************

  Filename: OUsCamera.c++
  Revision: 2.0

  Date:     18 Feb 98
  Author:   Jeff Collier

  Description:   See OUsCamera.h for details.

*/
#include "OUsCamera.h"

/*****************************************************************************

  Method:  Constructors

  Description: Constructors call method constructCam to keep code together.

*/
OUsCamera::OUsCamera(OUsDataSet &data) {constructCam(data);}
```

```
OUsCamera::OUsCamera(OUsDataSet &data, SoSeparator *new_camera):
  OUsCameraInv(new_camera) {constructCam(data);}

void OUsCamera::constructCam(OUsDataSet &data) {
    camera_points = data.addDataSet();
    intersection_points = data.addDataSet();
}
/******************************************************************************
General Functionality
******************************************************************************/

/******************************************************************************

  Method:  Get Camera Info

  Description:  This function gets many of the image plane attributes and
    passes back values so that they may be used in an interface.

*/
int OUsCamera::getCameraInfo(int &oCx, int &oCy, float &odx, float &ody,
                             int &oNcx, int &oNry, int &oNfx, float &oSx,
                             float &ok, float &fl) {

  oCx = Cx; oCy = Cy;
  odx = dx; ody = dy;
  oNcx = Ncx; oNry = Nry; oNfx = Nfx;
  oSx = Sx; ok = k;
  fl = getFocalPoint()[2];

  return(1);
}


/******************************************************************************

  Method:  Get Camera Pos(ition) -- Set Camera Pos(ition)

  Description:  These methods make use of Open Inventor data types to pass
    back and forth information about the cameras position and orientation
    to the interface code.
*/
int OUsCamera::getCameraPos(SbRotation &rm, SbVec3f &ta) {
  ta = camera_position->translation.getValue();
  rm = camera_rotation->rotation.getValue();
  return(1);
}

int OUsCamera::setCameraPos(SbRotation &rm, SbVec3f &ta) {
  camera_position->translation.setValue(ta);
  camera_rotation->rotation.setValue(rm);
  return(1);
```

```
}
/*****************************************************************************

  Method:  Capture Frame

  Description:  Finds out if a set of points (capturepoints) will be seen
    by an image plane.  If they are the points are turned on in onpixels.

*/
int OUsCamera::captureFrame() {

  //Reset image plane for new info.
  allPixelsOff();

  generateCameraPoints();
  distortByLens();
  quantizePoints();

  return(1);
}


/*****************************************************************************

  Method:  Generate Camera Points -- Generate World Ponts

  Description:  This method takes a set of points and finds the points that
    would be generated on the image plane of the camera.  The inverse is to
    take a set of points in the camera and find the world points.
*/
int OUsCamera::generateCameraPoints() {

  int j = 0;
  SbVec3f tpt;

  SoCoordinate3 *laser_points = camera_points->getHead()->
    getDataByEmColor(color->emissiveColor[0])->getPoints();

  SoCoordinate3 *intersect_pts = intersection_points->getPoints();
  intersect_pts->point.deleteValues(0);
  image_points->point.deleteValues(0);

  num_points = laser_points->point.getNum();

  //Map each point to image plane if it is not ocluded.
  for (int i = 0; i < num_points; i++) {
    //If point falls on plane add to image plane.
    if (isOccludedBySubject(laser_points->point[i])) {
    }
    else {
      intersect_pts->point.set1Value(j, laser_points->point[i]);
```

```
        image_points->point.set1Value(j,capturePoint(laser_points->point[i]));
        j++;
      }
    }
    num_points = j;

    return(1);
}

int OUsCamera::generateWorldPoints(SoCoordinate3 *all_points,
                                   SoCoordinate3 *in_pts) {

    directGeneration(all_points, in_pts);
    //interpolatedGeneration(new_world_points);

    return(1);
}


/*****************************************************************************

    Method:  Distort By Lens -- Undistort By Lens

    Description:  The purpose of this method is to add in lense distortion as
      specified by the camera model.  Also an undistort function is present to
      undo what this distortion does.

*/
int OUsCamera::distortByLens() {

    SbVec3f *points = image_points->point.startEditing();

    //Distort each points as specified by the lense.
    for (int i = 0; i < num_points; i++)
      distortPoint(points[i][0], points[i][1]);

    image_points->point.finishEditing();
    return(1);
}

int OUsCamera::undistortByLens(SoCoordinate3 *modify_points) {

    int num_pts = modify_points->point.getNum();
    SbVec3f *points = modify_points->point.startEditing();

    for (int i = 0; i < num_points; i++)
      distortPoint(points[i][0], points[i][1], 1);

    modify_points->point.finishEditing();
    return(1);
}
```

```
/*****************************************************************************

  Method:  Quantize Points

  Description:  Puts the points into an array that represents the image
    plane of the actual camera.

*/
int OUsCamera::quantizePoints() {

  for (int i = 0; i < num_points; i++) {
    turnOnPixel(image_points->point[i], i);
  }

  return(1);
}


/*****************************************************************************

  Method:  Merege With (another camera)

  Description:  Takes the onpixels in this camera and puts them in another
    camera.

*/
int OUsCamera::mergeWith(OUsCamera &other_camera) {

   int num_o = other_camera.RZcoord->point.getNum();

//    for (int i = 0; i < num_o; i++)
//      if (other_camera.RZcoord->point[i][1] > 0) {
//        turnOnPixel(Nry - i, (int)rint(other_camera.RZcoord->point[i][2]),
//               (int)rint(other_camera.RZcoord->point[i][0]));
//      }
  return(1);
}


/*****************************************************************************

  Method:  Generate Test Grid

*/
int OUsCamera::createTestGrid(int rhs_nh, int rhs_nw) {

  allPixelsOff();
  num_h = rhs_nh;
  num_w = rhs_nw;
```

```
  RZcoord->point.deleteValues(0);
  RZcoord->point.set1Value(0,-1,0,0);

  return(1);
}


/*****************************************************************************

  Method:  Direct Generation (of world points)

*/
int OUsCamera::directGeneration(SoCoordinate3 *all_wp, SoCoordinate3 *in_pts) {


  SbVec3f *points;
  int tot_RZ = RZcoord->point.getNum();
  int tot_int;
  int index;
  int num_wp = 0;
  float x, y;

  SoCoordinate3 *intersect_pts = intersection_points->getPoints();
  tot_int = intersect_pts->point.getNum();

  points = RZcoord->point.startEditing();
  index = (int)rint(points[0][0]);



  if ((index == -1) && (tot_RZ == 1)) {

    float yy = (dy * Nry) / 2.0;
    float xx = (dx * Sx * Ncx) / 2.0;
    float y = -yy;
    float step_x = (dx * Sx * num_w);
    float step_y = (dy * num_h);

    while(y < yy) {
      x = -xx;

      while(x < xx) {
       all_wp->point.set1Value(num_wp, frameToWorldPoints(x, y));
       in_pts->point.set1Value(num_wp, frameToWorldPoints(x, y));
       num_wp++;
       x = x + step_x;
      }

      y = y + step_y;
    }
  }
```

```
  else {
    for(int i = 0; i < tot_RZ; i++) {

      if( (int)rint(points[i][1]) > 0) {
       cerr << "r,c,ind: " << i << "," << points[i][2] << "," << points[i][1]
            << endl;
       pixelToIP(i, (int)rint(points[i][2]), x, y);
       distortPoint(x, y, 1);
       all_wp->point.set1Value(num_wp, frameToWorldPoints(x, y));
       index = (int)rint(points[i][0]);
       in_pts->point.set1Value(num_wp,
                               intersect_pts->point[index]);
       num_wp++;
      }
    }
  }
  RZcoord->point.finishEditing();
  return(1);
}


/*****************************************************************************

  Method:  Interpolated Generation (of world points)

*/
int OUsCamera::interpolatedGeneration(SoCoordinate3 *,
                                      SoCoordinate3 *) {

  //Should be implimented like CyScan using template grid.
  cerr << "Not implimented yet!" << endl;
  return(1);
}

/*****************************************************************************

  Filename:  OUsCamera.h
  Revision:  2.00

  Date:     17 April 98
  Author:   Jeff Collier

  Description:  This class is used to simulate a camera for a structure
                light scanner.  Much of the functionality of this class
                is inherited from OUsCameraInv and OUsImagePlane.

*/
#include "OUsCameraInv.h"
#include "OUsDataSet.h"
#include "OUsImagePlane.h"
```

```
#define GRID_SIZE 10

/*****************************************************************************

  Class:  OUsCamera

  Description:  This class provides high level methods for a set of code
    that models a camera.

  Super Class:  OUsBase -> OUsXfrmObjs -> OUsDispObjs -> OUsImagePlane

  Inherited from
  OUsXfrmObjs:

    //Perform Transforms
    SoCoordinate3 &transFromWorld(SoCoordinate3 *points)
    SbVec3f &transFromWorld(SbVec3f &point)
    SoCoordinate3 &transToWorld(SoCoordinate3 *points);
    SbVec3f &transToWorld(SbVec3f &point)

  Inherited from
  OUsDispObjs:
    //Transformation functions
    virtual int setTransformPath()
    virtual int setTransform()

    //show/hide objects on the scene graph
    virtual OUsDispObjs &show()
    virtual OUsDispObjs &hide()

  Inherited from
  OUsImagePlane
    //Image Plane Functionality
    OUsImagePlane & allPixelsOff();
    int turnOnPixel(const SbVec3f & point, int ind);
    int turnOnPixel(int r, int c, int ind);

    SbVec3f capturePoint(const SbVec3f & point);
    virtual SbVec3f frameToWorldPoints(float x, float y);

    int ipToPixel(const float x, const float y, int &r, int &c);
    int pixelToIP(const int r, const int c, float &x, float &y);

    virtual int distortPoint(float x, float y, int dir = -1);

    void setToSlice(){};
    void setToScan();

    //Interface functions
    OUsImagePlane & setCenter(int rhsCx, int rhsCy)
```

```
    OUsImagePlane & setPixelDim(float rhsdx, float rhsdy)
    OUsImagePlane & setNumberSensors(int rhsNcx, int rhsNry, int rhsNfx)
    OUsImagePlane & setSx(float rhsSx)
    OUsImagePlane & setDistortion(float rhsk)
    OUsImagePlane & setFocalPoint(int rhsCx, int rhsCy, float rhsfl)

  Inherited from
  OUsCameraInv

        int isOccludedBySubject(const SbVec3f &point);

      SbVec3f getFocalPoint() {return(focal_point->point[0]);}
            OUsCameraInv &setEmissiveColor(SbColor cl)
    SbColor getEmissiveColor() {return(color->emissiveColor[0]);}

    int setSceneGraph();
    int getFromSceneGraph();

*/
class OUsCamera : public OUsCameraInv {

public:
  //Constructors -- Destructors
  OUsCamera(OUsDataSet &data);
  OUsCamera(OUsDataSet &data, SoSeparator *new_camera);
  ~OUsCamera() {camera_points = NULL;}

  //General Functionality
  int getCameraInfo(int &oCx, int &oCy, float &odx, float &ody, int &oNcx,
                    int &oNry, int &oNfx, float &oSx, float &ok, float &fl);
  int getCameraPos(SbRotation &rm, SbVec3f &ta);
  int setCameraPos(SbRotation &rm, SbVec3f &ta);

  int captureFrame();
  virtual int generateCameraPoints();
  virtual int generateWorldPoints(SoCoordinate3 *modify_points,
                                  SoCoordinate3 *in_pts);
  virtual int distortByLens();
  virtual int undistortByLens(SoCoordinate3 *modify_points);
  virtual int quantizePoints();

  virtual int mergeWith(OUsCamera &other_camera);

  virtual int createTestGrid(int nh = GRID_SIZE, int nw = GRID_SIZE);

  int directGeneration(SoCoordinate3 *nw_pts, SoCoordinate3 *in_pts);
  int interpolatedGeneration(SoCoordinate3 *nw_pts, SoCoordinate3 *in_pts);

  void clearScan() {
    camera_points->clearPoints(); intersection_points->clearPoints();}
```

```
private:
  void constructCam(OUsDataSet &data);

  int num_points;
  OUsDataSet *camera_points;
  OUsDataSet *intersection_points;
  int num_w, num_h;

};
/******************************************************************************

  Filename:  OUsCameraInv.c++
  Revision:  2.00

  Date:      18 April 98
  Author:    Jeff Collier

  Description:   See OUsCameraInv.h for details about the class.

*/
#include "OUsCameraInv.h"

/******************************************************************************
Constructors -- Destructors
******************************************************************************/

/******************************************************************************

  Method:  Constructor

  Description:  Creates a new camera and inserts it into the scene graph.

*/
OUsCameraInv::OUsCameraInv() {

  //Create pixel node
  pixel = new SoCube;
  pixel->setName("Pixel");

  //Create lense distortion node
  lens_distort = new SoCoordinate3;
  lens_distort->setName(SbName("Lens_Distortion"));

  //Create focal point node
  focal_point = new SoCoordinate3;
  focal_point->setName("Focal_Point");

  //Create node for displaying points on the image plane.
  image_points = new SoCoordinate3;
  image_points->setName(SbName("Image_Points"));
```

```
//Scale image points for debuging
SoScale *simp = new SoScale;
simp->scaleFactor.setValue(IP_SCALE, IP_SCALE, 1);

//Define color of laser.
color = new SoMaterial;
color->setName("Color");
color->emissiveColor = RED_LASER;
color->ambientColor = RED_LASER;
color->diffuseColor = RED_LASER;

/************************************************
Create Debug Display Switch and branch.
************************************************/
SoSwitch *disp_debug = new SoSwitch;
disp_debug->whichChild = DEBUG_DISP;
disp_debug->addChild(color);
disp_debug->addChild(simp);
disp_debug->addChild(image_points);
disp_debug->addChild(new SoPointSet);
disp_debug->addChild(focal_point);
disp_debug->addChild(new SoPointSet);
disp_debug->addChild(pixel);
disp_debug->addChild(lens_distort);
/************************************************/

//Create image plane cube.
image_plane = new SoCube;
image_plane->setName("Image_Plane");

//Create image plane scale factor Sx.
scale_Sx = new SoScale;
scale_Sx->setName("Scale_Sx");

//Compensate for the depth of the image_plane.
SoTranslation *translate_ip = new SoTranslation;
translate_ip->translation = SbVec3f(0, 0, -PLANE_WIDTH);

//Create Material style for the image plane
SoMaterial *ip_color = new SoMaterial;
ip_color->ambientColor = IP_AMB_COLOR;
ip_color->diffuseColor = IP_DIF_COLOR;
ip_color->specularColor = IP_SPE_COLOR;
ip_color->shininess = IP_SHINE;

/************************************************
Create Display Switch and branch.
************************************************/
display->addChild(ip_color);
display->addChild(translate_ip);
display->addChild(scale_Sx);
```

```
  display->addChild(image_plane);
  /***********************************************/

  //Each display node must be on a separator named Display.
  SoSeparator * disp = new SoSeparator;
  disp->setName(SbName("Display"));
  disp->addChild(display);

  //Set up camera rotation.
  coop_rotation = new SoRotation;
  coop_rotation->setName(SbName("Coop_Rotation"));
  coop_rotation->rotation = SbRotation(SbVec3f(0, 1, 0), 0);

  //Set up camera position.
  camera_position = new SoTranslation;
  camera_position->setName(SbName("Camera_Position"));
  camera_position->translation = SbVec3f(CAM_DIST, 0, 0);

  //Set up camera rotation.
  camera_rotation = new SoRotation;
  camera_rotation->setName(SbName("Camera_Rotation"));
  camera_rotation->rotation = SbRotation(SbVec3f(0, 1, 0), CAM_ANGLE);

  //Set Up Camera Root
  root = new SoSeparator;
  root->setName(SbName("Camera_Root"));

  /***********************************************
  Finish creating scene graph.
  ***********************************************/
  root->addChild(coop_rotation);
  root->addChild(camera_position);
  root->addChild(camera_rotation);
  root->addChild(disp);
  root->addChild(disp_debug);

  setSceneGraph();
}


/*****************************************************************************

  Method:  Constructor

  Description:  Construct a camera from a given scene graph

*/
OUsCameraInv::OUsCameraInv(SoSeparator *new_camera):
OUsImagePlane(new_camera) {

  SoSearchAction *searcher = new SoSearchAction;
```

```
root = new_camera;

//Camera Location
searcher->setName(SbName("Camera_Position"));
searcher->apply(root);
if (searcher->getPath() == NULL) camera_position = NULL;
camera_position = (SoTranslation *)searcher->getPath()->getTail();

searcher->reset();
searcher->setName(SbName("Coop_Rotation"));
searcher->apply(root);
if (searcher->getPath() == NULL) coop_rotation = NULL;
coop_rotation = (SoRotation *)searcher->getPath()->getTail();

searcher->reset();
searcher->setName(SbName("Camera_Rotation"));
searcher->apply(root);
if (searcher->getPath() == NULL) camera_rotation = NULL;
camera_rotation = (SoRotation *)searcher->getPath()->getTail();

searcher->reset();
searcher->setName(SbName("Focal_Point"));
searcher->apply(root);
if (searcher->getPath() == NULL) focal_point = NULL;
focal_point = (SoCoordinate3 *)searcher->getPath()->getTail();

searcher->reset();
searcher->setName(SbName("Image_Plane"));
searcher->apply(root);
if (searcher->getPath() == NULL) image_plane = NULL;
image_plane = (SoCube *)searcher->getPath()->getTail();

searcher->reset();
searcher->setName(SbName("Pixel"));
searcher->apply(root);
if (searcher->getPath() == NULL) pixel = NULL;
pixel = (SoCube *)searcher->getPath()->getTail();

searcher->reset();
searcher->setName(SbName("Scale_Sx"));
searcher->apply(root);
if (searcher->getPath() == NULL) scale_Sx = NULL;
scale_Sx = (SoScale *)searcher->getPath()->getTail();

searcher->reset();
searcher->setName(SbName("Color"));
searcher->apply(root);
if (searcher->getPath() == NULL) image_points = NULL;
color = (SoMaterial *)searcher->getPath()->getTail();

searcher->reset();
```

```
    searcher->setName(SbName("Image_Points"));
    searcher->apply(root);
    if (searcher->getPath() == NULL) image_points = NULL;
    image_points = (SoCoordinate3 *)searcher->getPath()->getTail();

    searcher->reset();
    searcher->setName(SbName("Lens_Distortion"));
    searcher->apply(root);
    if (searcher->getPath() == NULL) lens_distort = NULL;
    lens_distort = (SoCoordinate3 *)searcher->getPath()->getTail();

    searcher->reset();
    searcher->setName(SbName("Display"));
    searcher->apply(root);
    if (searcher->getPath() == NULL) display = NULL;
    SoSeparator *disp = (SoSeparator *)searcher->getPath()->getTail();
    display = (SoSwitch *)disp->getChild(0);

    if ((camera_position == NULL)  ||
        (coop_rotation == NULL)    ||
        (camera_rotation == NULL)  ||
        (focal_point == NULL)      ||
        (image_plane == NULL)      ||
        (pixel == NULL)            ||
        (scale_Sx == NULL)         ||
        (color == NULL)            ||
        (image_points == NULL)     ||
        (lens_distort == NULL)     ||
        (display == NULL)) {
      cerr << "ERROR -- File not successfully loaded! (CameraInv Constructor)\n";
      cerr << "CyScan must be restarted."  << endl;
      exit(1);
    }

    getFromSceneGraph();
    setTransformPath();
}

/*****************************************************************************
General Functionality
*****************************************************************************/

/*****************************************************************************

    Method:  Is (the point) Occluded by the subject

    Description:  Determines if the point is occluded by the subject.

    Return:  int -- 1 if subject is occluding camera
                    0 if subject is not occluding the camera
```

```
*/
int OUsCameraInv::isOccludedBySubject(const SbVec3f &point) {

  SbVec3f fp = getFocalPoint();
  ray.setRayInternal(transToWorld(fp), point);

  if (ray.fireRay(subjects) != NULL)
    return(1);

  return (0);
}


/*****************************************************************************

  Method:  Set Scene Graph -- Get From Scene Graph

  Description:  These methods take points that are stored in the
    OUsImagePlane class and updates the OUsCameraInv class (updating Open
    Inventor Nodes) or vise versa.

*/
int OUsCameraInv::setSceneGraph() {

  pixel->height = dy;
  pixel->width = dx;
  pixel->depth = PLANE_WIDTH;

  pixelToIP(Cy, Cx, cx, cy);
  focal_point->point.setValue(cx, cy, fl);

  image_plane->width = Ncx * dx * IP_SCALE;
  image_plane->height = Nry * dy * IP_SCALE;
  image_plane->depth = PLANE_WIDTH;

  scale_Sx->scaleFactor.setValue(1, Sx, 1);

  lens_distort->point.setValue(k, 0, 0);

  return(OUsImagePlane::setSceneGraph());
}

int OUsCameraInv::getFromSceneGraph() {

  cx = focal_point->point[0][0];
  cy = focal_point->point[0][1];
  fl = focal_point->point[0][2];

  dx = pixel->width.getValue();
  dy = pixel->height.getValue();
  Nfx = (int)rint((image_plane->width.getValue() / IP_SCALE) / dx);
```

```
    Nry = (int)rint((image_plane->height.getValue() / IP_SCALE) / dy);
    Ncx = Nfx;
    setCache();

    ipToPixel(cx, cy, Cy, Cx);

    Sx = scale_Sx->scaleFactor.getValue()[1];

    k = lens_distort->point[0][0];


    return(OUsImagePlane::getFromSceneGraph());
}
/*****************************************************************************

    Filename:  OUsCameraInv.h
    Revision:  2.00

    Date:      18 April 97
    Author:    Jeff Collier

    Description: This class is designed to handle most of the Open
                 Inventor interface.  It is used to simulate a camera.
                 It derives from OUsImagePlane and contains pointers to
                 Open Inventor nodes.

*/
#include "OUsDispObjs.h"
#include "OUsImagePlane.h"
#include <Inventor/Xt/SoXtTransformSliderSet.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/SoDB.h>
#include <Inventor/Xt/SoXt.h>


#ifndef OU_CAM_INV
#define OU_CAM_INV

/*****************************************************************************
Constant Definitions
*****************************************************************************/
#define PLANE_WIDTH .001              //Rendered image plane width
#define DEBUG_DISP SO_SWITCH_ALL      //Current level of debugging

#define CAM_ANGLE -M_PI_4             //Default Camera angle.
#define CAM_DIST .2                   //Default baseline
#define IP_SCALE 30                   //Scales the image plane so that it
                                      //may be seen.

const SbColor IP_AMB_COLOR(.2, .2, .2);  //The following definitions are
const SbColor IP_DIF_COLOR(.6, .6, .6);  //used to define the look of the
```

```
const SbColor IP_SPE_COLOR(.5, .5, .5);  //image plane.
const float IP_SHINE = .5;

/*****************************************************************************

  Class:  OUsCameraInv

  Description:  This class provides the open inventor functionality that
    bridges the classes OUsImagePlane and OUsCamera.  It contains low level
    camera operations that rely primaraly on Open Inventor

  Super Class:  OUsBase -> OUsXfrmObjs -> OUsDispObjs -> OUsImagePlane

  Inherited from
  OUsXfrmObjs:

    //Perform Transforms
    SoCoordinate3 &transFromWorld(SoCoordinate3 *points)
    SbVec3f &transFromWorld(SbVec3f &point)
    SoCoordinate3 &transToWorld(SoCoordinate3 *points);
    SbVec3f &transToWorld(SbVec3f &point)

  Inherited from
  OUsDispObjs:
    //Transformation functions
    virtual int setTransformPath()
    virtual int setTransform()

    //show/hide objects on the scene graph
    virtual OUsDispObjs &show()
    virtual OUsDispObjs &hide()

  Inherited from
    //Image Plane Functionality
    OUsImagePlane & allPixelsOff();
    int turnOnPixel(const SbVec3f & point, int ind);
    int turnOnPixel(int r, int c, int ind);

    SbVec3f capturePoint(const SbVec3f & point);
    virtual SbVec3f frameToWorldPoints(float x, float y);

    int ipToPixel(const float x, const float y, int &r, int &c);
    int pixelToIP(const int r, const int c, float &x, float &y);

    virtual int distortPoint(float x, float y, int dir = -1);

    void setToSlice(){};
    void setToScan();

    //Interface functions
    OUsImagePlane & setCenter(int rhsCx, int rhsCy)
```

```
    OUsImagePlane & setPixelDim(float rhsdx, float rhsdy)
    OUsImagePlane & setNumberSensors(int rhsNcx, int rhsNry, int rhsNfx)
    OUsImagePlane & setSx(float rhsSx)
    OUsImagePlane & setDistortion(float rhsk)
    OUsImagePlane & setFocalPoint(int rhsCx, int rhsCy, float rhsfl)

    virtual int setSceneGraph()
    virtual int getFromSceneGraph(){return(OUsImagePlane::setSceneGraph())

*/
class OUsCameraInv : public OUsImagePlane {

public:
  //Constructors -- Destructors
  OUsCameraInv();
  OUsCameraInv(SoSeparator *new_camera);
  OUsCameraInv(const OUsCameraInv &rhs) {*this = rhs;}

  int isOccludedBySubject(const SbVec3f &point);

  SbVec3f getFocalPoint() {return(focal_point->point[0]);}
  int makeCoopCamera() {
    coop_rotation->rotation = SbRotation(SbVec3f(0, 0, 1), M_PI); return(1);}
  OUsCameraInv &setEmissiveColor(SbColor cl) {
    color->emissiveColor = cl; return(*this);}
  SbColor getEmissiveColor() {return(color->emissiveColor[0]);}

  int setSceneGraph();
  int getFromSceneGraph();

  virtual void clearScan(){};

protected:
  //Scene Graph Pointers
  SoTranslation *camera_position;
  SoRotation *coop_rotation;
  SoRotation *camera_rotation;
  SoCoordinate3 *focal_point;
  SoCube *image_plane;
  SoCube *pixel;
  SoScale *scale_Sx;
  SoMaterial *color;
  SoCoordinate3 *image_points;
  SoCoordinate3 *lens_distort;

private:
  OUsRay ray;
};

#endif
/*****************************************************************************
```

```
  Filename:  OUsContainers.c++
  Revision:  2.00

  Date:    19 Feb 1998
  Author:  Jeff Collier

  Description:  See OUsContainers.h

*/
#include "OUsContainers.h"


/*****************************************************************************

  Method:  Set Transform Path

*/
int OUsContainers::setTransformPath() {

  for (int i = 0; i < num_objs; i++)
    objects[i]->setTransformPath();

  OUsDispObjs::setTransformPath();

  return(1);
}


/*****************************************************************************

  Method:  Set Transform

*/
int OUsContainers::setTransform() {

  for (int i = 0; i < num_objs; i++)
    objects[i]->setTransform();

  OUsDispObjs::setTransform();

  return(1);
}

/*****************************************************************************

  Method:  Set To Scan

*/
void OUsContainers::setToScan() {

  for (int i = 0; i < num_objs; i++)
```

```
    objects[i]->setToScan();

  OUsDispObjs::setToScan();
}


/*****************************************************************************

  Method:  Set To Slice

*/
void OUsContainers::setToSlice() {

  for (int i = 0; i < num_objs; i++)
    objects[i]->setToSlice();

  OUsDispObjs::setToSlice();
}


/*****************************************************************************

  Method:  Clear Scan

*/
void OUsContainers::clearScan() {

  for (int i = 0; i < num_objs; i++)
    objects[i]->clearScan();

  OUsDispObjs::clearScan();
}

/*****************************************************************************

  Method:  Add Object -- Add Object Scene Graph

  Description:  Adds an object to this objects lists and adds it to the
    scene graph.  Add Object does the same thing but does not have to and
    the item to the scene graph.

*/
OUsContainers & OUsContainers::addObject(OUsDispObjs * obj_to_add) {

  objects[num_objs] = obj_to_add;
  root->addChild(obj_to_add->getRoot());
  num_objs++;

  return(*this);
}
```

```
OUsContainers & OUsContainers::addObjectSG(OUsDispObjs * obj_to_add) {

  objects[num_objs] = obj_to_add;
  num_objs++;

  return(*this);
}


/*****************************************************************************

  Method:  show

*/
OUsDispObjs & OUsContainers::show() {

  for (int i = 0; i < num_objs; i++)
    objects[i]->show();

  return(OUsDispObjs::show());
}


/*****************************************************************************

  Method:  hide

*/
OUsDispObjs & OUsContainers::hide() {

  for (int i = 0; i< num_objs; i++)
    objects[i]->hide();

  return(OUsDispObjs::hide());
}
/*****************************************************************************

  Filename:  OUsContainers.h
  Revision:  2.00

  Date:  19 March 1998
  Author:  Jeff Collier

*/

#ifndef _CONTAINS_
#define _CONTAINS_

#include "OUsDispObjs.h"

/*****************************************************************************
```

```
Definitions
*************************************************************************/
#define NUM_CONT_OBJS 100


/*************************************************************************

  Class:  OUsContainers

  Description:  This provides the functionality for adding objects as part
    of the inherited object.  This allows the inherited object the ability
    to contain an object of OUsDispObj type.

  Superclass:  OUsBase -> OUsXfrmObjects -> OUsDispObjs

  Inherited from
  OUsXfrmObjects:
    int copyMatrix(const SbMatrix *to_world, const SbMatrix *from_world);
    int linkMatrix(SbMatrix *to_world, SbMatrix *from_world);
    SbMatrix *getTransToWorld() {return trans_to_world;}
    SbMatrix *getTransFromWorld() {return trans_from_world;}
    void displayMatrix();

    //Perform Transforms
    SoCoordinate3 &transFromWorld(SoCoordinate3 *points);
    SbVec3f &transFromWorld(SbVec3f &point);
    SoCoordinate3 &transToWorld(SoCoordinate3 *points);
    SbVec3f &transToWorld(SbVec3f &point);

  Inherited from
  OUsDispObjs:
    //Transformation functions
    virtual int setTransformPath()
    virtual int setTransform()

       //Initialization Methods
    virtual void setToScan(){};
    virtual void setToSlice(){};

    //show/hide objects on the scene graph
    virtual OUsDispObjs & show()
    virtual OUsDispObjs & hide()

    //Interaction methods
    SoSeparator *getRoot() {return(root);}

*/
class OUsContainers : public OUsDispObjs {

public:
  OUsContainers() {num_objs = 0;}
```

```
    OUsContainers(SoSeparator *root_node): OUsDispObjs(root_node) {
      num_objs = 0; }
    ~OUsContainers() {delete [] objects;}

    virtual int setTransformPath();
    virtual int setTransform();

    virtual void setToScan();
    virtual void setToSlice();
    virtual void clearScan();

    int getNumObjects() {return(num_objs);}
    OUsContainers & addObjectSG(OUsDispObjs * obj_to_add);
    OUsContainers & addObject(OUsDispObjs * obj_to_add);
    OUsDispObjs * getObject(int ind) {return(objects[ind]);}

    OUsDispObjs & show();
    OUsDispObjs & hide();

  protected:
    int num_objs;
    OUsDispObjs *objects[NUM_CONT_OBJS];
  };

  #endif
  /*****************************************************************************

    Filename:  OUsControls.c++
    Revision:  2.00

    Date:  15 March 1998
    Author:  Jeff Collier

    Description: For Details see OUsControls.h

  */
  #include <iostream.h>
  #include <stdio.h>
  #include <string.h>
  #include "OUsBase.h"
  #include "OUsControls.h"
  #include "OUsTrack.h"

  /*****************************************************************************
  File Menu
  *****************************************************************************/

  /*****************************************************************************

         Method: Get Subject
```

```
  Description:  Get an Open Inventor File to use as a scanning object.

*/
SbBool OUsControls::getSubject(char *filename, Tcl_Interp *tcl_string) {

  SoSeparator *new_subject = readFile(filename);

  if (new_subject == NULL) {
    tcl_string->result = "GetSubject: No valid subject in this file!";
    cerr << "GetSubject: No valid subject in this file!" << endl;
    return TCL_ERROR;
  }

  subjects->removeAllChildren();
  subjects->addChild(new_subject);
  return TCL_OK;
}


/*****************************************************************************

  Method:  Get Scanner

  Description: Get a scanner that has been saved in an Open Inventor format.

*/
SbBool OUsControls::getScanner(char *filename, Tcl_Interp *tcl_string){

  if(filename == NULL) return TCL_OK;
  SoSeparator *new_scanner = readFile(filename);

  if (new_scanner == NULL) {
    tcl_string->result =  "GetScanner: No valid scanner in this file!\n";
       cerr << "GetScanner: No valid scanner in this file!\n" << endl;
    return TCL_ERROR;
  }

  insertScanner(new_scanner);
  return TCL_OK;
}


/*****************************************************************************

  Method:  Save Scanner

  Description:  Saves the Inventor graph describing the scanner in an
    Inventor file format.

*/
int OUsControls::saveScanner(char *filename, Tcl_Interp *tcl_string) {
```

```
  if (writeToFile(filename))
      return TCL_OK;

  tcl_string->result = "saveScanner: Unable to write scanner to file!";
  cerr << "SaveScanner: Unable to write scanner to file!" << endl;
  return TCL_ERROR;
}

int OUsControls::saveGraph(char *filename, Tcl_Interp *tcl_string) {

  if (writeFullGraph(filename))
      return TCL_OK;

  tcl_string->result = "saveGraph: Unable to write scanner to file!";
  cerr << "saveGraph: Unable to write scanner to file!" << endl;
  return TCL_ERROR;
}
/*****************************************************************************

  Methods: Save Actual Points and Save Scanned Points

  Description:  Not yet implemented.

*/
SbBool OUsControls::saveActualPoints(Tcl_Interp *tcl_string) {
  tcl_string->result = "-- SaveActualPoints --\nNot yet implimented!\n";
  cerr << "-- SaveActualPoints --\nNot yet implimented!\n" << endl;
  return TCL_ERROR;
}
SbBool OUsControls::saveScannedPoints(Tcl_Interp *tcl_string){
  tcl_string->result = "-- SaveScannedPoints --\nNot yet implimented!\n";
  cerr << "-- SaveScannedPoints --\nNot yet implimented!\n" << endl;
  return TCL_ERROR;
}


/*****************************************************************************
Interaction Methods
*****************************************************************************/

int OUsControls::getNumScanheads(Tcl_Interp *tcl_string) {
  sprintf(tcl_string->result, "%d",  num_scanheads);
  return TCL_OK;
}
int OUsControls::getNumTracks(Tcl_Interp *tcl_string) {
  sprintf(tcl_string->result, "%d",  num_tracks);
  return TCL_OK;
}
int OUsControls::getNumCameras(Tcl_Interp *tcl_string) {
  sprintf(tcl_string->result, "%d",  num_cameras);
  return TCL_OK;
```

```
}

/*****************************************************************************

  Method:  Edit Scanner

  Description:  Records the number of slices and the spacing between slices
    with each track.

*/
int OUsControls::setScannerInfo(float step_size, int steps) {

  for (int i = 0; i < num_tracks; i++) {
    track[i]->setIncriment(step_size);
    track[i]->setTotalInc(steps);
  }
  return TCL_OK;
}

/*****************************************************************************

  Method:  Get Scanner Info

  Description:  Method to return to number of slices and spacing between
    slices to TCL.

*/
int OUsControls::getScannerInfo(Tcl_Interp *tcl_string) {

  if (num_tracks < 1) {
    tcl_string->result = "getScannerInfo:\nNo Scanner for parameters";
    return TCL_ERROR;
  }

  float inc = track[0]->getIncriment();
  float tot = track[0]->getTotalInc();

  sprintf(tcl_string->result, "%.9f %.9f", inc, tot);

  return TCL_OK;
}


int OUsControls::setCameraInfo(Tcl_Interp *tcl_string, int scanhead_num,
                               int Cx, int Cy, float dx, float dy, int Ncx,
                               int Nry, int Nfx, float Sx, float k, float fl,
                               float r, float g, float b) {

  if((scanhead_num < 0) || (scanhead_num >= num_scanheads)) {
    tcl_string->result = "setCameraInfo:\nscanhead_num out of range!";
    return TCL_ERROR;
```

```
  }
  scanhead[scanhead_num]->getTopCamera()->setFocalPoint(Cx, Cy, fl);
  scanhead[scanhead_num]->getTopCamera()->setPixelDim(dx, dy);
  scanhead[scanhead_num]->getTopCamera()->setNumberSensors(Ncx, Nry, Nfx);
  scanhead[scanhead_num]->getTopCamera()->setSx(Sx);
  scanhead[scanhead_num]->getTopCamera()->setDistortion(k);
  scanhead[scanhead_num]->getTopCamera()->setEmissiveColor(SbColor(r, g, b));

  scanhead[scanhead_num]->getBottomCamera()->setFocalPoint(Cx, Cy, fl);
  scanhead[scanhead_num]->getBottomCamera()->setPixelDim(dx, dy);
  scanhead[scanhead_num]->getBottomCamera()->setNumberSensors(Ncx, Nry, Nfx);
  scanhead[scanhead_num]->getBottomCamera()->setSx(Sx);
  scanhead[scanhead_num]->getBottomCamera()->setDistortion(k);
  scanhead[scanhead_num]->getBottomCamera()->setEmissiveColor(SbColor(r, g, b));
  return TCL_OK;
}


int OUsControls::getCameraInfo(Tcl_Interp *tcl_string, int scanhead_num) {

   int Cx, Cy, Ncx, Nry, Nfx;
   float dx, dy, Sx, k, fl;
   SbColor color;

   if (scanhead_num < num_scanheads) {
     scanhead[scanhead_num]->getTopCamera()->getCameraInfo(Cx, Cy, dx, dy, Ncx,
                                                    Nry, Nfx, Sx, k, fl);
     color = scanhead[scanhead_num]->getTopCamera()->getEmissiveColor();

     sprintf(tcl_string->result,"%d %d %.9f %.9f %d %d %d %.9f %.9f %.9f %.9f %.9f
%.9f",
             Cx, Cy, dx, dy, Ncx, Nry, Nfx, Sx, k, fl,
             color[0], color[1], color[2]);
   }
   else
     sprintf(tcl_string->result, "-- -- -- -- -- -- -- -- -- --");

   return TCL_OK;
}


int OUsControls::setCameraPos(Tcl_Interp *tcl_string, int scanhead_num,
                           float X, float Y, float Z, float i, float j,
                           float k, float ang) {

  if((scanhead_num < 0) || (scanhead_num >= num_scanheads)) {
    tcl_string->result = "setCameraPos:\nInvalid scanhead number";
    return TCL_ERROR;
  }
  SbRotation rt(SbVec3f(i,j,k), ang);
  SbRotation rb(SbVec3f(-i,j,-k), ang);
```

```
  SbVec3f pt(X, Y, Z);
  SbVec3f pb(X, -Y, Z);

  scanhead[scanhead_num]->getTopCamera()->setCameraPos(rt, pt);
  scanhead[scanhead_num]->getBottomCamera()->setCameraPos(rb, pb);

  return TCL_OK;
}

int OUsControls::getCameraPos(Tcl_Interp *tcl_string, int scanhead_num) {

  SbRotation rm;
  SbVec3f ta, temp;
  float x, y, z, i, j, k, ang;

  if (scanhead_num < num_scanheads) {
    scanhead[scanhead_num]->getTopCamera()->getCameraPos(rm, ta);
    x = ta[0]; y = ta[1]; z = ta[2];
    rm.getValue(temp, ang); i = temp[0]; j = temp[1]; k = temp[2];

    sprintf(tcl_string->result, "%.9f %.9f %.9f %.9f %.9f %.9f %.9f",
           x, y, z, i, j, k, ang);
  }
  else
    sprintf(tcl_string->result, "-- -- -- -- -- -- --");

  return TCL_OK;
}

int OUsControls::setCameraPosMat(Tcl_Interp *tcl_string, int scanhead_num,
                            float X, float Y, float Z,
                            float a11, float a12, float a13, float a14,
                            float a21, float a22, float a23, float a24,
                            float a31, float a32, float a33, float a34,
                            float a41, float a42, float a43, float a44) {
  float i, j, k, ang;
  SbVec3f temp, pt(X, Y, Z), pb(X, -Y, Z);
  SbMatrix m(a11, a12, a13, a14,
             a21, a22, a23, a24,
             a31, a32, a33, a34,
             a41, a42, a43, a44);

  SbRotation r(m);

  if((scanhead_num < 0) || (scanhead_num >= num_scanheads)) {
    tcl_string->result = "setCameraPos:\nInvalid scanhead number";
    return TCL_ERROR;
  }

  scanhead[scanhead_num]->getTopCamera()->setCameraPos(r, pt);
```

```
    r.getValue(temp, ang);
    temp.getValue(i, j, k);
    temp.setValue(-i, j, -k);

    scanhead[scanhead_num]->getBottomCamera()->
      setCameraPos(SbRotation(temp, ang), pb);

    return TCL_OK;
}
int OUsControls::getCameraPosMat(Tcl_Interp *tcl_string, int scanhead_num) {

    SbRotation rm;
    SbMatrix m;
    SbVec3f ta, temp;
    float x, y, z;

    if (scanhead_num < num_scanheads) {
      scanhead[scanhead_num]->getTopCamera()->getCameraPos(rm, ta);
      x = ta[0]; y = ta[1]; z = ta[2];
      rm.getValue(m);

      sprintf(tcl_string->result,
            "%.9f %.9f %.9f %.9f %.9f %.9f %.9f %.9f %.9f %.9f %.9f %.9f %.9f %.9f %.9f
%.9f %.9f %.9f %.9f %.9f %.9f",
            x, y, z, m[0][0], m[0][1], m[0][2], m[0][3],
            m[1][0], m[1][1], m[1][2], m[1][3],
            m[2][0], m[2][1], m[2][2], m[2][3],
            m[3][0], m[3][1], m[3][2], m[3][3]);
    }
    else
      sprintf(tcl_string->result,
            "-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --");

    return TCL_OK;
}

/*****************************************************************************

    Method:  Set Track Position

    Description:  Moves the base of the track to a new location

*/
int OUsControls::setTrackPos(Tcl_Interp *tcl_string, int track_num,
                             float x, float y, float z,
                             float i, float j, float k, float angle) {

    if ((track_num >= 0) && (track_num < num_tracks)) {
      track[track_num]->setRotation(SbRotation(SbVec3f(i, j, k), angle));
      track[track_num]->setOrigin(SbVec3f(x, y, z));
    }
```

```
  else {
    tcl_string->result = "setTrackPos: track number out of range!";
    return TCL_ERROR;
  }

  return TCL_OK;
}



int OUsControls::getTrackPos(Tcl_Interp *tcl_string, int track_num) {

  SbVec3f temp;
  SbRotation rm;
  float x, y, z, angle;

   if (track_num < num_tracks) {
     track[track_num]->getOrigin(x, y, z);
     track[track_num]->getRotation(rm);
     rm.getValue(temp, angle);

     sprintf(tcl_string->result, "%.9f %.9f %.9f %.9f %.9f %.9f %.9f",
            x, y, z, temp[0], temp[1], temp[2], angle);
   }
   else
     sprintf(tcl_string->result, "-- -- -- -- -- -- --");

   return TCL_OK;
}

int OUsControls::setLaserPlaneInfo(Tcl_Interp *tcl_string, const int sh_num,
                                   const float resolution, const float width,
                                   const float pitch, const float tilt,
                                   const float r, const float g, const float b,
                                   const float fl){


  if((sh_num < 0) || (sh_num >= num_scanheads)) {
    tcl_string->result = "setLaserPlaneInfo:\nInvalid scanhead number";
    cerr << "-- setLaserPlaneInfo:\nInvalid scanhead number\n" << endl;
    return TCL_ERROR;
  }

  scanhead[sh_num]->getLaserPlane()->
    setDefiningPoints(resolution, width, pitch, tilt);

  scanhead[sh_num]->getLaserPlane()->setColor(SbColor(r, g, b));
  scanhead[sh_num]->getLaserPlane()->setFocalLength(fl);

  return TCL_OK;
}
```

```
int OUsControls::getLaserPlaneInfo(Tcl_Interp *tcl_string,
                                   const int scanhead_num) {

  SbColor color;

  float res = 0, wid = 0, pitch = 0, tilt = 0, fl;

  if (scanhead_num < num_scanheads) {
    scanhead[scanhead_num]->getLaserPlane()->
      getDefiningPoints(res, wid, pitch, tilt);

    fl = scanhead[scanhead_num]->getLaserPlane()->getFocalLength();
    color = scanhead[scanhead_num]->getLaserPlane()->getEmissiveColor();

    sprintf(tcl_string->result, "%.9f %.9f %.9f %.9f %.9f %.9f %.9f %.9f",
           res, wid, pitch, tilt, color[0], color[1], color[2], fl);
  }
  else
    sprintf(tcl_string->result, "-- -- -- -- -- -- --");

  return TCL_OK;
}


int OUsControls::getOutput(Tcl_Interp *tcl_string, int scanhead_num) {

  int num_points;
  float min, max, mean, sv;

  if (scanhead_num < num_scanheads) {
    scanhead[scanhead_num]->getNumScannedPoints(num_points);
    scanhead[scanhead_num]->getStats(min, max, mean, sv);
    sprintf(tcl_string->result, "%d %.9f %.9f %.9f %.9f",
           num_points, min, max, mean, sv);
  }
  else
    sprintf(tcl_string->result, "--");

  return TCL_OK;
}

//==== Display Menu ============================================================\\

SbBool OUsControls::display(int trackon, int scanheadon,
                            int objecton, int dataon) {

  //Show/Hide Object
  if (objecton) subjects->whichChild = SO_SWITCH_ALL;
  else subjects->whichChild = SO_SWITCH_NONE;

  //Show/Hide Track
```

```
  if (trackon)
    for (int i = 0; i < num_tracks; i++)
      track[i]->show();
  else
    for (int i = 0; i < num_tracks; i++)
      track[i]->hide();

  //Show/Hide Scanhead
  if (scanheadon)
    for (int i = 0; i < num_scanheads; i++)
      scanhead[i]->show();
  else
    for (int i = 0; i < num_scanheads; i++)
      scanhead[i]->hide();

  //Show/Hide Data
  if(dataon)
    data_sets.showAll();
  else
    data_sets.hideAll();

  return TCL_OK;
}

int OUsControls::dataToCyScan() {

  for (int i = 0; i < num_scanheads; i++)
    scanhead[i]->dataToCyScan(i);

  return(1);
}

/**************************************************************************
TCL Interface
**************************************************************************/

/**************************************************************************

  Function:  Cmd SC control

  Description:  This function provides the interface for interaction with
    the class OUsControl object.

*/
int CmdSCcontrol(ClientData clientData, Tcl_Interp *tcl_string,
                 int argc, char *argv[]) {

  CyIvState * state = (CyIvState *) clientData;
  OUsControls *menu = (OUsControls *)state->ResearchClass;

  //If now argument is passed then control object should be created.
```

```
 if (argc == 1) {
   state->ResearchClass = new OUsControls(state);
   return TCL_OK;
 }
 if (menu == NULL) {
   tcl_string->result = "state->ResearchClass should not contain a NULL pointer";
   return TCL_ERROR;
 }
 if(strcmp(argv[1], "delete") == 0) {
   delete menu;
   state->ResearchClass = NULL;
   return TCL_OK;
 }
 //File Menu to Control Class.
 if(strcmp(argv[1], "getsubject") == 0) {
   return(menu->getSubject(argv[2], tcl_string));
 }
 if(strcmp(argv[1], "getscanner") == 0) {
   return(menu->getScanner(argv[2], tcl_string));
 }
 if(strcmp(argv[1], "savegraph") == 0) {
   return(menu->saveGraph(argv[2], tcl_string));
 }
 if(strcmp(argv[1], "savescanner") == 0) {
   return(menu->saveScanner(argv[2], tcl_string));
 }
 if(strcmp(argv[1], "saveactualpoints") == 0) {
   return(menu->saveActualPoints(tcl_string));
 }
 if(strcmp(argv[1], "savescannedpoints") == 0) {
   return(menu->saveScannedPoints(tcl_string));
 }
 //Info Exchange Methods
 if(strcmp(argv[1], "getnumscanheads") == 0) {
   return menu->getNumScanheads(tcl_string);
 }
 if(strcmp(argv[1], "getnumtracks") == 0) {
   return menu->getNumTracks(tcl_string);
 }
 if(strcmp(argv[1], "getnumcameras") == 0) {
   return menu->getNumCameras(tcl_string);
 }
 if(strcmp(argv[1], "setscannerinfo") == 0) {
   return (menu->setScannerInfo(atof(argv[2]), atoi(argv[3])));
 }
 if(strcmp(argv[1], "getscannerinfo") == 0) {
   return(menu->getScannerInfo(tcl_string));
 }
 if(strcmp(argv[1], "setcamerainfo") == 0) {
   return(menu->setCameraInfo(tcl_string, atoi(argv[2]), atoi(argv[3]),
                              atoi(argv[4]), atof(argv[5]), atof(argv[6]),
```

```
                        atoi(argv[7]), atoi(argv[8]), atoi(argv[9]),
                        atof(argv[10]), atof(argv[11]), atof(argv[12]),
                        atof(argv[13]), atof(argv[14]), atof(argv[15])));
  }
  if(strcmp(argv[1], "getcamerainfo") == 0) {
    return menu->getCameraInfo(tcl_string, atoi(argv[2]));
  }
  if(strcmp(argv[1], "setcamerapos") == 0) {
    return(menu->setCameraPos(tcl_string, atoi(argv[2]), atof(argv[3]),
                        atof(argv[4]), atof(argv[5]), atof(argv[6]),
                        atof(argv[7]), atof(argv[8]), atof(argv[9])));
  }
  if(strcmp(argv[1], "getcamerapos") == 0) {
    return menu->getCameraPos(tcl_string, atoi(argv[2]));
  }
  if(strcmp(argv[1], "setcameraposmat") == 0) {
    return(menu->setCameraPosMat(tcl_string, atoi(argv[2]), atof(argv[3]),
                        atof(argv[4]), atof(argv[5]), atof(argv[6]),
                        atof(argv[7]), atof(argv[8]), atof(argv[9]),
                        atof(argv[10]), atof(argv[11]), atof(argv[12]),
                        atof(argv[13]), atof(argv[14]), atof(argv[15]),
                        atof(argv[16]), atof(argv[17]), atof(argv[18]),
                        atof(argv[19]), atof(argv[20]), atof(argv[21])));
  }
  if(strcmp(argv[1], "getcameraposmat") == 0) {
    return menu->getCameraPosMat(tcl_string, atoi(argv[2]));
  }
  if(strcmp(argv[1], "settrackpos") == 0) {
    return menu->setTrackPos(tcl_string, atoi(argv[2]),
                        atof(argv[3]), atof(argv[4]), atof(argv[5]),
                        atof(argv[6]), atof(argv[7]), atof(argv[8]),
                        atof(argv[9]));
  }
  if(strcmp(argv[1], "gettrackpos") == 0) {
    menu->getTrackPos(tcl_string, atoi(argv[2]));
    return TCL_OK;
  }
  if(strcmp(argv[1], "setlaserplaneinfo") == 0) {
    return(menu->setLaserPlaneInfo(tcl_string, atoi(argv[2]), atof(argv[3]),
                            atof(argv[4]), atof(argv[5]), atof(argv[6]),
                            atof(argv[7]), atof(argv[8]), atof(argv[9]),
                            atof(argv[10])));
  }
  if(strcmp(argv[1], "getlaserplaneinfo") == 0) {
    return menu->getLaserPlaneInfo(tcl_string, atoi(argv[2]));
  }

  //Add Menu
  if(strcmp(argv[1], "addtrack") == 0) {
    return menu->addTrack(new OUsTrack());
  }
```

```
    if(strcmp(argv[1], "addscanhead") == 0) {
      return menu->addScanHead(atoi(argv[2]));
    }
    if(strcmp(argv[1], "addscanner") == 0) {
      return (menu->addScanner());
    }
    //Display Menu
    if(strcmp(argv[1], "getoutput") == 0) {
      return menu->getOutput(tcl_string, atoi(argv[2]));
    }
    if(strcmp(argv[1], "display") == 0) {
      return menu->display(atoi(argv[2]),atoi(argv[3]),
                           atoi(argv[4]),atoi(argv[5]));
    }

    //Action Buttons
    if(strcmp(argv[1], "datatocyscan") == 0) {
      if(menu->dataToCyScan()) return TCL_OK;
    }
    if(strcmp(argv[1], "scanslice") == 0) {
      if(menu->scanSlice()) return TCL_OK;
    }
    if(strcmp(argv[1], "reset") == 0) {
      menu->setToScan();
      return TCL_OK;
    }
    if(strcmp(argv[1], "setgrid") == 0) {
      if(menu->createTestGrid()) return TCL_OK;
      tcl_string->result = "Could not create test grid!";
      return TCL_ERROR;
    }
    if(strcmp(argv[1], "clear") == 0) {
      if(menu->clearScannedPoints()) return TCL_OK;
      tcl_string->result = "Could not clear scanned points!";
      return TCL_ERROR;
    }
    sprintf(tcl_string->result, "%s  does not exist for OUSC", argv[1]);
    return TCL_ERROR;
}

void CreateTclOUsScanner(CyIvState *state) {

    TCLCMD("OUSC",                  CmdSCcontrol);

}
/*****************************************************************************

    Filename:  OUsControls.h
    Revision:  2.00

    Date:  7 March 1998
```

```
  Author:  Jeff Collier

  Description:  Maintains the interface between the simulator code and the
    GUIs that control it.  In this case the GUI is created with TCL/TK.
*/

#ifndef OU_CONTROLS
#define OU_CONTROLS

/*****************************************************************************
Include Files
*****************************************************************************/
#include "OUsBase.h"
#include "OUsSimulator.h"

/*****************************************************************************

  Class:  OUsControls

  Description:  This class is used to control the interface between the GUIs
    and the simulator.

  Superclasses:  ResearchBase, OUsSimulator

  Inherited From
  ResearchBase:  Nothing.

  Inherited From
  OUsSimulator:   int removeSimulator();

                //Add Functions
                int addTrack(OUsTrack *new_track)
                OUsLaserPlane & addLaser();
                OUsCamera & addCamera();
                int addScanHead(short track_num = -1,
                OUsScanHead *new_scanhead);

               //Simulator Functions
                int scanSlice();
                int setTestGrid();
                int resetScanner();
                int clearScannedPoints();

               //File Functions
                int replaceScanner(SoSeparator *scanner);
                int insertScanner(SoSeparator *scanner);
                int writeToFile(char *filename = "scanner.iv");

*/
class OUsControls : public ResearchBase, public OUsSimulator {
```

```
public:
  //Constructors -- Destructors
  OUsControls(CyIvState *state = NULL):
              OUsSimulator(state) {};

  //File Menu
  int getSubject(char *filename, Tcl_Interp *tcl_string);
  int getScanner(char *filename, Tcl_Interp *tcl_string);
  int saveScanner(char *filename, Tcl_Interp *tcl_string);
  int saveGraph(char *filename, Tcl_Interp *tcl_string);
  int saveActualPoints(Tcl_Interp *tcl_string);
  int saveScannedPoints(Tcl_Interp *tcl_string);


  //Info Exchange Methods
  int getNumScanheads(Tcl_Interp *tcl_string);
  int getNumTracks(Tcl_Interp *tcl_string);
  int getNumCameras(Tcl_Interp *tcl_string);

  int setScannerInfo(float stepsize, int steps);
  int getScannerInfo(Tcl_Interp *interp);

  int setCameraInfo(Tcl_Interp *tcl_string, int scanhead_num,
                  int Cx, int Cy, float dx, float dy, int Ncx,
                  int Nry, int Nfx, float Sx, float k, float fl,
                  float r, float g, float b);
  int getCameraInfo(Tcl_Interp *interp, int scanhead_num);

  int setCameraPos(Tcl_Interp *tcl_string, int scanhead_num,
                 float X, float Y, float Z, float i, float j,
                 float k, float ang);
  int getCameraPos(Tcl_Interp *interp, int scanhead_num);

  int setCameraPosMat(Tcl_Interp *tcl_string, int scanhead_num,
                    float X, float Y, float Z,
                    float a11, float a12, float a13, float a14,
                    float a21, float a22, float a23, float a24,
                    float a31, float a32, float a33, float a34,
                    float a41, float a42, float a43, float a44);
  int getCameraPosMat(Tcl_Interp *tcl_string, int scanhead_num);

  int setTrackPos(Tcl_Interp *tcl_string, int track_num,
                float x, float y, float z,
                float i, float j, float k, float angle);
  int getTrackPos(Tcl_Interp *tcl_string, int track_num);

  int setLaserPlaneInfo(Tcl_Interp *tcl_string, const int sh_num,
                    const float resolution, const float width,
                    const float pitch, const float tilt,
                    const float r, const float g, const float b,
                    const float fl);
```

```
  int getLaserPlaneInfo(Tcl_Interp *tcl_string, const int scanhead_num);


  //Add Menu
  int addScanner() {addScanHead(addTrack(new OUsTrack())); return TCL_OK;}

  //Display Menu
  int getOutput(Tcl_Interp *tcl_string, int scanhead_num);
  int display(int track, int headscan, int object, int data);

  //Action Buttons
  int dataToCyScan();

};

/*****************************************************************************
TCL Interface Global Functions
*****************************************************************************/
void CreateTclOUsScanner(CyIvState *state);

#endif
/*****************************************************************************

  Filename:  OUsDataSet.c++
  Revision:  2.00

  Date:  7 March 1998
  Author:  Jeff Collier

  Description:  See OUsDataSet.h

*/
#include "OUsDataSet.h"

/*****************************************************************************
Static member declarations
*****************************************************************************/
OUsDataSet *OUsDataSet::head = NULL;

/*****************************************************************************

  Method:  Constructor

  Description:  Creates a new Data set and stores it on the scene graph.

*/
OUsDataSet::OUsDataSet() {

  if(head == NULL) head = this;

  //Set Up Display Root
```

```
    root = new SoSeparator;
    root->setName(SbName("DataSet_Root"));

    //Each display node must be on a separator named Display.
    SoSeparator *disp = new SoSeparator;
    disp->setName(SbName("Display"));
    disp->addChild(display);
    root->addChild(disp);

    color = new SoMaterial;
    color->setName(SbName("Color"));
    display->addChild(color);

    data_points = new SoCoordinate3;
    data_points->setName(SbName("Data_Points"));
    data_points->point.deleteValues(0);
    display->addChild(data_points);

    mat_binding = new SoMaterialBinding;
    //mat_binding->value = SoMaterialBinding::PER_PART_INDEXED;
    mat_binding->setName(SbName("Material_Binding"));
    display->addChild(mat_binding);

    draw_properties = new SoDrawStyle;
    draw_properties->pointSize = 2;
    display->addChild(draw_properties);
    display->addChild(new SoPointSet);

    simulator_data->addChild(root);

    next = NULL;
}


/******************************************************************************

  Method:  Show Allm -- Hide All

  Description:  Moves through the entire list and sets all the data to show
    itself or sets it to hide itself

*/
OUsDataSet & OUsDataSet::showAll() {

  OUsDataSet *cur = getHead();
  while(cur != NULL) {
    cur->show();
    cur = cur->next;
  }
  return (*this);
}
```

```
OUsDataSet & OUsDataSet::hideAll() {

  OUsDataSet *cur = getHead();
  while(cur != NULL) {
    cur->hide();
    cur = cur->next;
  }
  return (*this);
}


/******************************************************************************

  Method:  Add Data Set

  Description:  Adds a data set to the end of the linked list.  It then
    returns the pointer to that list.

*/
OUsDataSet * OUsDataSet::addDataSet() {

  OUsDataSet *cur = getHead();
  if (cur == NULL) return(new OUsDataSet);
  while(cur->next != NULL)  {
    cur = cur->next;
  }

  return(cur->next = new OUsDataSet);
}


/******************************************************************************

  Method:  Get Data Set By Emissive Color

  Description:  Finds the first data set that has the specified color.  If
    no such data set exists, one is created.

  Params:  SbColor rhs_color -- rhs_color is the color of the data that should
    be found if posible.

  Return:  Reference to the data set of the correct color.

*/
OUsDataSet *OUsDataSet::getDataByEmColor(const SbColor &rhs_color) {

  OUsDataSet *current = getHead();
  SoMaterial *test_color;

  int i = 0;
```

```
  while(current != NULL) {
    test_color = current->getColor();

    if(test_color->emissiveColor[0] == rhs_color) {
      return(current);
    }
    current = current->next;
  }

  OUsDataSet *tmp = addDataSet();
  tmp->setEmColor(rhs_color);

  return(tmp);
}


/*****************************************************************************

  Method:  Add Points

  Description:  Adds a set of points to the points already in the data set

  Params:  SoCoordinate3* pts -- The points to add to the data set.

*/
OUsDataSet &OUsDataSet::addPoints(SoCoordinate3 *pts) {

  int num_add = pts->point.getNum();
  int num_tot = data_points->point.getNum();

  data_points->point.setValues(num_tot, num_add, pts->point.getValues(0));

  return (*this);
}


/*****************************************************************************

  Method:  Set Color

  Description:  Sets the color for the data set on the Open Inventor scene
    graph.
*/
OUsDataSet & OUsDataSet::setDifColor(const SbColor &rhs_color) {

  color->diffuseColor = rhs_color;

  return(*this);
}

OUsDataSet &OUsDataSet::setEmColor(const SbColor &rhs_color) {
```

```
  color->emissiveColor = rhs_color;

  return(*this);
}


/*****************************************************************************

  Method:  Place In CyScan

  Description:  Places the data in the node into the CyScan scene graph so
    that testing may be done on the node.

*/
OUsDataSet & OUsDataSet::placeInCyScan(int graph) {

  //Get an empty SoCoordinate3 node for CyScan XYZ and FRZ
  SoCoordinate3 *XYZCoords, *FRZCoords;
  XYZCoords = state->scan[graph]->XYZCoordsNode;

  if(XYZCoords == NULL) {
    cerr << "No XYZ coord without file" << endl;
    exit(1);
  }
  XYZCoords->point.deleteValues(0);
  FRZCoords = state->scan[graph]->FRZCoordsNode;
  FRZCoords->point.deleteValues(0);

  cerr << "Inserting simulated data into CyScan" << endl;
  XYZCoords->point.setValues(0, data_points->point.getNum(),
                             data_points->point.getValues(0));

  int num_pt = XYZCoords->point.getNum();
  FRZCoords->point.deleteValues(0);
  FRZCoords->point.setNum(num_pt);

  float last_F = XYZCoords->point[0][0];
  float Z;
  int num = 0, F = 1, R = 0;

  for(int j = 0; j < num_pt; j++) {
   if(((last_F + .002) > XYZCoords->point[j][0]) &&
      ((last_F - .002) < XYZCoords->point[j][0])){
     //Same frame value
     R++;
   }
   else {
     //New Frame
     F++;
     last_F = XYZCoords->point[j][0];
     R = 0;
```

```
    }

    Z = XYZCoords->point[j][2];
    FRZCoords->point.set1Value(j, F, R, Z);
  }

 return (*this);
}


/******************************************************************************

  Filename:  OUsDataSet.h
  Revision:  2.00

  Date:  9 Dec 1997
  Author:  Jeff Collier

  Description:  This class helps keep track of data generated by a scanner
    simulator.  It is stored on a scenegraph and has a means of displaying
    itself.

*/
#ifndef OU_DSET
#define OU_DSET

#include "OUsDispObjs.h"
#include "Inventor/nodes/SoMaterial.h"

/******************************************************************************
Constant Definition
******************************************************************************/
#define HEAD0_COLOR SbColor(1, 1, 0)
#define HEAD1_COLOR SbColor(1, 0, 1)
#define HEAD2_COLOR SbColor(0, 1, 1)
#define HEAD3_COLOR SbColor(1, 0, 0)
const SbColor DEF_DIF(.7, .1, .7);


/******************************************************************************

  Class:  OUsDataSet.h

  Description:  This class stores data on a scene graph.  There are many
    methods to make the objects here render themselves in a helpful way.

  Superclass:  OUsBase -> OUsXfrmObjs -> OUsDispObjs

  Inherited from
  OUsBase:
```

```
        SoSeparator *ReadFile(const char *filename)
      int writeFullGraph(char *filename = "graph.iv")

    Inherited from
    OUsXfrmObjects:
      int copyMatrix(const SbMatrix *to_world, const SbMatrix *from_world);
      int linkMatrix(SbMatrix *to_world, SbMatrix *from_world)
      SbMatrix *getTransToWorld() {return trans_to_world;}
      SbMatrix *getTransFromWorld() {return trans_from_world;}
      void displayMatrix()

      //Perform Transforms
      SoCoordinate3 &transFromWorld(SoCoordinate3 *points)
      SbVec3f &transFromWorld(SbVec3f &point)
      SoCoordinate3 &transToWorld(SoCoordinate3 *points);
      SbVec3f &transToWorld(SbVec3f &point)

    Inherited from
    OUsDispObjs:
      //Transformation functions
      virtual int setTransformPath()
      virtual int setTransform()

      //Initialization Methods
      virtual void setToScan(){}
      virtual void setToSlice(){}

      //show/hide objects on the scene graph
      virtual OUsDispObjs &show()
      virtual OUsDispObjs &hide()

      //Interaction methods
      SoSeparator *getRoot() {return(root);}

*/
class OUsDataSet : public OUsDispObjs {

public:
  //Constructors -- Destructors
  OUsDataSet();
  ~OUsDataSet() {
    if(next != NULL) delete next;}

  //Methods affecting entire class
  OUsDataSet &showAll();
  OUsDataSet &hideAll();
  OUsDataSet &clearPoints() {
    data_points->point.deleteValues(0); return (*this);}

  //Linked list operations
  OUsDataSet *getHead() {return(head);}
```

```
    OUsDataSet *addDataSet();
    OUsDataSet *getDataByEmColor(const SbColor &color);

    //Dealing with objects points
    OUsDataSet &addPoints(SoCoordinate3 *pts);
    OUsDataSet &setPoint(int index, SbVec3f &pt);
    SoCoordinate3 *getPoints() {return(data_points);}

    //Other ojbect functionality
    OUsDataSet &setDifColor(const SbColor &color = DEF_DIF);
    OUsDataSet &setEmColor(const SbColor &color = RED_LASER);
    SoMaterial *getColor() {return(color);}
    SoMFColor *getDiffuseColor() {return(&color->diffuseColor);}
    OUsDataSet &placeInCyScan(int graph);

  private:
    OUsDataSet * next;
    static OUsDataSet * head;

    SoCoordinate3 *data_points;
    SoDrawStyle *draw_properties;
    SoMaterialBinding *mat_binding;
    SoMaterial *color;
};

#endif
/******************************************************************************

    Filename:  OUsDispObjs.c++
    Revision:  2.00

    Date:      14 Feb 98
    Author:    Jeff Collier

    Description:  For details see OUsDispObjs.h

*/
#include "OUsDispObjs.h"

/******************************************************************************

    Method:  Default Constructor

*/
OUsDispObjs::OUsDispObjs(SoSeparator *disp_obj_root) {

    SoSearchAction *searcher = new SoSearchAction;
    root = disp_obj_root;

    if (root != NULL) {
```

```
    searcher->setName(SbName("Display"));
    searcher->apply(root);

    if(searcher->getPath() == NULL) {
      cerr << "OUsDispObjs::Constructor:  Problem with scene graph!"
           << "  Bad scanner file." << endl;
      exit(1);
    }
    else {
      SoSeparator *disp = (SoSeparator *)searcher->getPath()->getTail();
      display = (SoSwitch *)disp->getChild(0);
    }
  }
  else display = new SoSwitch;
  display->ref();

  sprintf(display_name, "display-%d", num_OUsBases);
  display->setName(SbName(display_name));

  display->whichChild = SO_SWITCH_ALL;
}


/******************************************************************************
Transfromation Methods
******************************************************************************/


/******************************************************************************

  Method:  Set Transform Path

  Description:  Defines a path on the scene graph so that a transformation
    Through that path can be found.

*/
int OUsDispObjs::setTransformPath() {

  SbViewportRegion vr;
  SoGetMatrixAction *matrix = new SoGetMatrixAction(vr);

  SoSearchAction *search_path = new SoSearchAction;

  search_path->setName(SbName(display_name));
  search_path->setSearchingAll(FALSE);
  search_path->setInterest(SoSearchAction::FIRST);
  search_path->apply(scanner_elements);

  //Get transform path, transform path should not change.
  transform_path = search_path->getPath();
```

```
  if (transform_path == NULL) {
    cerr << "problem with transfrom path" << endl;
    exit(1);
  }

  return(1);
}


/******************************************************************************

  Method:  Set Transforms

  Discription:  This sets the transforms for world to object space
    conversion.  The transformation is based on the objects display node.

*/
int OUsDispObjs::setTransform() {

  SbViewportRegion vr;
  SoGetMatrixAction *matrix = new SoGetMatrixAction(vr);

  matrix->apply(transform_path);
  copyMatrix (&matrix->getMatrix(), &matrix->getInverse());

  return(1);
}
/******************************************************************************

  Filename:  OUsDispObjs.h
  Revision:  2.00

  Date:      08 April 98
  Author:    Jeff Collier

*/
#ifndef _OUSELEM
#define _OUSELEM

/******************************************************************************
Include files
******************************************************************************/
#include "OUsRay.h"
#include "OUsXfrmObjs.h"
#include <Inventor/So.h>
#include <Inventor/SbViewportRegion.h>
#include <Inventor/actions/SoGetMatrixAction.h>
#include <iostream.h>


/******************************************************************************
```

```
Definitions
*************************************************************************/
#define DISP_LINE_WIDTH 3
#define DISP_POINT_SIZE 5
#define DISP_NAME_SIZE 20

const SbColor RED_LASER(1, .65, .65);
const SbColor INFRARED_LASER(.65, 1, 1);


/*************************************************************************

  Class OUsDispObjs

  Description:  This class is a foundational class for all scanning
                DispObjs simulated by this program.  It inherits from
                OUsBase.

                It contains transformations of the objects to and from
                worldspace.  It also provides the functionality of
                placing objects on a track.  An OUsRay is contained and
                can be used to find the intersection of objects and
                rays from a point through a focal point.

                Pointers to XYZCoord nodes are included so Captured
                points may be displayed

  Superclass:  OUsBase -> OUsXfrmObjects

  Inherited from
  OUsBase:
    SoSeparator *ReadFile(const char *filename)
    int writeFullGraph(char *filename = "graph.iv")

  Inherited from
  OUsXfrmObjects:
    int copyMatrix(const SbMatrix *to_world, const SbMatrix *from_world);
    int linkMatrix(SbMatrix *to_world, SbMatrix *from_world);
    SbMatrix *getTransToWorld() {return trans_to_world;}
    SbMatrix *getTransFromWorld() {return trans_from_world;}
    void displayMatrix();

    //Perform Transforms
    SoCoordinate3 &transFromWorld(SoCoordinate3 *points);
    SbVec3f &transFromWorld(SbVec3f &point);
    SoCoordinate3 &transToWorld(SoCoordinate3 *points);
    SbVec3f &transToWorld(SbVec3f &point);

*/
class OUsDispObjs : public OUsXfrmObjs {
```

```
public:
  //Constructors -- Destructors
  OUsDispObjs(SoSeparator *disp_obj_root = NULL);
  ~OUsDispObjs(){}

  //Transformation functions
  virtual int setTransformPath();
  virtual int setTransform();

  //Initialization Methods
  virtual void setToScan(){};
  virtual void setToSlice(){};
  virtual void clearScan(){};

  //show/hide objects on the scene graph
  virtual OUsDispObjs & show() {
    display->whichChild = SO_SWITCH_ALL; return(*this);}
  virtual OUsDispObjs & hide() {
    display->whichChild = SO_SWITCH_NONE; return(*this);}

  //Interaction methods
  SoSeparator *getRoot() {return(root);}

private:
  char display_name[DISP_NAME_SIZE]; //Name given to the display node

protected:
  SoSeparator *root;
  SoSwitch *display;
  SoPath *transform_path;
};

#endif
/******************************************************************************

  Filename:  OUsImagePlane.c++
  Revision:  2.00

  Date:      11 March 1998
  Author:    Jeff Collier

  Description:  See OUsImagePlane.h for details.

*/
#include "OUsImagePlane.h"

/******************************************************************************
Constructors -- Destructors
******************************************************************************/

/******************************************************************************
```

```
  Method:  Constructor

  Description:  This method constructs an image plane from a list of image
    plane paramters.  See OUsImagePlane.h for info on paramters.

*/
OUsImagePlane::OUsImagePlane():
Cx(DEF_CX),
Cy(DEF_CY),
dx(DEF_DX),
dy(DEF_DY),
Ncx(DEF_NCX),
Nry(DEF_NRY),
Nfx(DEF_NFX),
fl(DEF_CAM_FL),
k(DEF_K),
Sx(DEF_SX) {
  construct();
};

OUsImagePlane::OUsImagePlane(SoSeparator *rt):
OUsDispObjs(rt),
Cy(DEF_CY),
dx(DEF_DX),
dy(DEF_DY),
Ncx(DEF_NCX),
Nry(DEF_NRY),
Nfx(DEF_NFX),
fl(DEF_CAM_FL),
k(DEF_K),
Sx(DEF_SX) {
  construct();
};

void OUsImagePlane::construct() {
  setCache();
  RZcoord = new SoCoordinate3;
  RZcoord->ref();
  allPixelsOff();
}

/*****************************************************************************
Image Plane Functionality
*****************************************************************************/

/*****************************************************************************

  Method:  All Pixels Off

  Description:  Sets the array that contains information about pixels so
    that all the values are -1 which indicates an "off" value.
```

```
*/

OUsImagePlane & OUsImagePlane::allPixelsOff() {

  RZcoord->point.deleteValues(0);
  RZcoord->point.setNum(Nry);
  SbVec3f *pt = RZcoord->point.startEditing();
  for (int i = 0; i < Nry; i++)
    pt[i].setValue(0, 0, 0);
  RZcoord->point.finishEditing();
  num_on = 0;

  return (*this);
}

/*****************************************************************************

  Method:  Turn On Pixel

  Description:  Takes a point on the image plane and finds the corresponding
    pixel that should be turned on.

*/
int OUsImagePlane::turnOnPixel(const SbVec3f & point, int ind) {

  //Points converted to image plane space.
  int r, c;

  if(ipToPixel(point[0], point[1], r, c)) {
    return(turnOnPixel(r, c, ind));
  }
  return(0);
}

int OUsImagePlane::turnOnPixel(int r, int c, int ind) {

  int in = (int)rint(RZcoord->point[r][1]);

  if (in > 0) return(0);
    // {
//     if (c > RZcoord->point[r][2])
//       return(0);
//     num_on--;
//   }
  num_on++;
  in++;
  RZcoord->point.set1Value(r, ind, in, c);
  return(1);
}
```

```
/*****************************************************************************

  Method:  Capture Point

  Description:  Takes a point and multiplies it by the appropriate
    transformation matrix so that the point is captured by the image plane.

*/
SbVec3f OUsImagePlane::capturePoint(const SbVec3f &point) {

  SbVec3f dst, src;
  src = point;
  src = transFromWorld(src);

  projection_matrix.multVecMatrix(src, dst);

  return(dst);
}


/*****************************************************************************

  Method:  Frame To World Points

  Description:  This method takes the points stored in the frame buffer and
    converts them to coordinates in world space.

*/
SbVec3f OUsImagePlane::frameToWorldPoints(float x, float y) {

  SbVec3f nw_pt(0,0,0);
  float z;

  //This equation uses the transformation matrix of the camera and the
  //distance of the camera from the laser plane to determin a depth so
  //that the point may be converted back to a world location.

  z = ((x * proj_to_world[0][0] + y * proj_to_world[1][0] +
       proj_to_world[3][0]) / (-1.0 * proj_to_world[2][0]));

  //Note:  The above equation has been simplified by omiting terms that
  //       will because the camera is always in the same place relative
  //       to the laser plane.

  proj_to_world.multVecMatrix(SbVec3f(x, y, z), nw_pt);

  if(nw_pt.length() > 1) cerr << "Problem point" << endl;

  return(nw_pt);
}
```

```
/*****************************************************************************

  Method: IP to Pixel -- Pixel to IP

  Description:  This function does and undoes the quantization that occurs
    with a CCD image plane device.  If a point on the image plane was
    converted with IP to pixel it would give the pixel value of the
    position.  If that pixel value was used to generate the IP value (using
    Pixel to IP the original point would not be recovered, instead the
    center of the pixel would be returned.

 */
int OUsImagePlane::ipToPixel(const float x, const float y, int &r, int &c) {

  float rr, cc;

  //Conversion of x
  if(x > cen_x) return(0);
  cc = cen_x + x;
  if(cc < 0) return(0);

  //Conversion of y
  if(y > cen_y) return(0);
  rr = cen_y - y;
  if(rr < 0) return(0);

  c = (int)rint(cc/(dx * Sx));
  r = (int)rint(rr/dy);
  return(1);
}

int OUsImagePlane::pixelToIP(const int r, const int c, float &x, float &y) {

  x = c * dx * Sx - cen_x;
  y = cen_y - (r * dy);

  return(1);
}

/*****************************************************************************

  Method:  Distort Point

  Description:  Used Tsai's method for determining lens distortion.
    The function uses real image point values not pixel values.

 */
int OUsImagePlane::distortPoint(float x, float y, int dir) {

  float kp;
```

```
    if(dir > 0) dir = 1;
    else dir = -1;

    x = cx - x;
    y = cy - y;
    kp = 1 + k * (x * x + y * y);

    x = cx + dir * x/kp;
    y = cy + dir * y/kp;

    return(1);
}

/*****************************************************************************

    Method:  Set To Scan

    Description:  Creates a matrix that can project a point from the image
      plane back into the scene graph.  This should only done once per scan
      because only coordinate that changes is an offset in X.

 */
void OUsImagePlane::setToScan() {

    setTransformPath();
    setTransform();

    proj_to_world = *getTransToWorld();
    proj_to_world.multLeft(inv_projection_matrix);

};

int OUsImagePlane::setSceneGraph(){

    setCache();

    projection_matrix.makeIdentity();
    projection_matrix[2][3] = -1/fl;
    inv_projection_matrix.makeIdentity();
    inv_projection_matrix[2][3] = 1/fl;

    return(1);
}


/*****************************************************************************

    Filename:  OUsImagePlane.h
    Revision:  2.00

    Date:      21 March 1998
```

```
  Author:   Jeff Collier

*/

#ifndef IMAGE_PLANE
#define IMAGE_PLANE

/******************************************************************************
Include Files
******************************************************************************/
#include "OUsDispObjs.h"
#include <Inventor/Sb.h>
#include <Inventor/So.h>
#include <Inventor/fields/SoMFVec3f.h>
#include <math.h>

/******************************************************************************
Define constants
******************************************************************************/
#define MAX_NUM_ROWS 300
#define DEF_CX 300
#define DEF_CY 120            //Focal center on imageplane.
#define DEF_DX  .000007
#define DEF_DY  .000019       //Distance from one pixel to the next.
#define DEF_NCX  680          //Number of sensing elements in a row.
#define DEF_NRY  240
#define DEF_NFX  100          //Number of pixels in a row.
#define DEF_SX   .87          //Parameter relates the ratio of x/y.
#define DEF_K    .01
#define DEF_CAM_FL .002


/******************************************************************************

  Class:

  Description:  This class keeps track of image plane information and
                functionality as it relates to the pixels on the image
                plane.
                It inherits from OUsBasic and contains Open Inventor
                classes.

  Super Class:  OUsBase -> OUsXfrmObjs -> OUsDispObjs

  Inherited from
  OUsBase:
       SoSeparator *ReadFile(const char *filename)
    int writeFullGraph(char *filename = "graph.iv")

  Inherited from
  OUsXfrmObjs:
```

```
    int copyMatrix(const SbMatrix *to_world, const SbMatrix *from_world);
    int linkMatrix(SbMatrix *to_world, SbMatrix *from_world)
    SbMatrix *getTransToWorld() {return trans_to_world;}
    SbMatrix *getTransFromWorld() {return trans_from_world;}
    void displayMatrix()

    //Perform Transforms
    SoCoordinate3 &transFromWorld(SoCoordinate3 *points)
    SbVec3f &transFromWorld(SbVec3f &point)
    SoCoordinate3 &transToWorld(SoCoordinate3 *points);
    SbVec3f &transToWorld(SbVec3f &point)

  Inherited from
  OUsDispObjs:
    //Transformation functions
    virtual int setTransformPath()
    virtual int setTransform()

    //Initialization Methods
    virtual void setToScan(){}
    virtual void setToSlice(){}

    //show/hide objects on the scene graph
    virtual OUsDispObjs &show()
    virtual OUsDispObjs &hide()

    //Interaction methods
    SoSeparator *getRoot() {return(root);}

*/
class OUsImagePlane : public OUsDispObjs {

public:
  //Constructors -- Destructors
  OUsImagePlane();
  OUsImagePlane(SoSeparator *rt);
  ~OUsImagePlane() {RZcoord->unref();}

  //Image Plane Functionality
  OUsImagePlane & allPixelsOff();
  int turnOnPixel(const SbVec3f & point, int ind);
  int turnOnPixel(int r, int c, int ind);

  SbVec3f capturePoint(const SbVec3f & point);
  virtual SbVec3f frameToWorldPoints(float x, float y);

  int ipToPixel(const float x, const float y, int &r, int &c);
  int pixelToIP(const int r, const int c, float &x, float &y);

  virtual int distortPoint(float x, float y, int dir = -1);
```

```
  void setToSlice(){};
  void setToScan();
  virtual void clearScan(){};

  //Interface functions
  OUsImagePlane & setCenter(int rhsCx, int rhsCy) {
    Cx = rhsCx; Cy = rhsCy; setSceneGraph(); return(*this);}
  OUsImagePlane & setPixelDim(float rhsdx, float rhsdy) {
    dx = rhsdx; dy = rhsdy; setSceneGraph(); return(*this);}
  OUsImagePlane & setNumberSensors(int rhsNcx, int rhsNry, int rhsNfx) {
    Ncx = rhsNcx; Nry = rhsNry; Nfx = rhsNfx; setSceneGraph(); return(*this);}
  OUsImagePlane & setSx(float rhsSx) {
    Sx = rhsSx; setSceneGraph(); return(*this);}
  OUsImagePlane & setDistortion(float rhsk) {
    k = rhsk; setSceneGraph(); return(*this);}
  OUsImagePlane & setFocalPoint(int rhsCx, int rhsCy, float rhsfl) {
    Cx = rhsCx; Cy = rhsCy; fl = rhsfl; setSceneGraph(); return(*this);}

  OUsImagePlane & setCache() {
    cen_x = (dx*Sx*Ncx)/2;  cen_y = (dy*Nry)/2; return(*this);}

  virtual int setSceneGraph();
  virtual int getFromSceneGraph(){return(OUsImagePlane::setSceneGraph());}

protected:
  //Image plane and camera specifications.
  int Cx, Cy;               //Focal center on imageplane in pixels.
  float cx, cy;             //Focal center of imageplane in actual coordinates
  float dx, dy;             //Distance from one sensing element to the next.
  int Ncx;                  //Number of sensing elements in a row.
  int Nry;                  //Number of rows of sensing elements
  int Nfx;                  //Number of pixels in a row.
  float Sx;                 //Parameter relates the ratio of x/y.
  float k;                  //First term of lens distortion
  float fl;                 //Focal length of the image plane lens

  //Data capture on the image plane.
  SoCoordinate3 *RZcoord;          //Row is index, 0 undefined, 1 intensity,
                                   //2 is column
  int num_on;                      //Indicates the number of pixels turned on.
  SbMatrix projection_matrix;      //Matrices used to speed up the process
  SbMatrix inv_projection_matrix;  //of projecting a set of points back and
  SbMatrix proj_to_world;          //forth from world space.

private:
  void construct();                //Function used by the constructors

  //Information cached
  float cen_x;
  float cen_y;
```

```
};

#endif
/*****************************************************************************

  Filename:  OUsLaserPlane.c++
  Revision:  2.00

  Date:      15 April 97
  Author:    Jeff Collier

  Description:

*/
#include "OUsLaserPlane.h"
#include <iostream.h>

/*****************************************************************************

  Method:  Constructor -- Destructor

  Description:  The constructor methods call a method called construct
    because the cod is the same but the method needed to take different
    arguments for the base class.

*/
OUsLaserPlane::OUsLaserPlane(OUsDataSet & data):
    intersect_points(NULL), OUsLaserPlaneInv() {
      construct(data);
}

OUsLaserPlane::OUsLaserPlane(OUsDataSet & data, SoSeparator *new_plane):
    intersect_points(NULL), OUsLaserPlaneInv(new_plane) {
      construct(data);
}

void OUsLaserPlane::construct(OUsDataSet &data) {
  intersect_points = data.getDataByEmColor(color->emissiveColor[0]);
  laserbeam.linkMatrix(getTransToWorld(), getTransFromWorld());
}

/*****************************************************************************
General Functionality
*****************************************************************************/

/*****************************************************************************

  Method:  Activate Plane

  Description:  Returns a set of points that represent the points on the
    object intersected by the laser plane.
```

```
*/
OUsLaserPlane &OUsLaserPlane::activatePlane() {

  int num_points = control_points->point.getNum();
  SoPickedPoint *picked_point;
  SbVec3f t_cp, t_o;
  t_o = origin;
  transToWorld(t_o);

  SoCoordinate3 *intersects = intersect_points->getPoints();
  int num_intersects = intersects->point.getNum();

  //Start with 1 because first point is origin
  for (int i = 1; i < num_points; i++) {
    t_cp = control_points->point[i];
    transToWorld(t_cp);
    //Set up laser beam and find intersecting point.
    laserbeam.setRayExternal(t_cp,t_o);
    picked_point = laserbeam.fireRay(subjects);

    //If intersection exists save point.
    if (picked_point != NULL) {
      intersects->point.set1Value(num_intersects,
                          picked_point->getPoint().getValue());
      num_intersects++;
    }
  }
  return(*this);
}



/******************************************************************************

  Filename:  OUsLaserPlane.h
  Revision:  2.00

  Date:      11 April 1998
  Author:    Jeff Collier

  Description:

*/
#include "OUsLaserPlaneInv.h"
#include "OUsDataSet.h"

/******************************************************************************

  Class:  OUsLaserPlane

  Description:  This class models a laser plane of light used in a laser
```

```
    triangulation system.

  Super Class: OUsBase -> OUsXfrmObjs -> OUsDispObjs -> OUsLaserPlaneInv

  Inherited from
  OUsDispObjs:
    //Transformation functions
    virtual int setTransformPath()
    virtual int setTransform()

    //Initialization Methods
    virtual void setToScan(){}
    virtual void setToSlice(){}

    //show/hide objects on the scene graph
    virtual OUsDispObjs &show()
    virtual OUsDispObjs &hide()

  Inherited from
  OUsLaserPlane:
    //Interaction methods
    SoSeparator      *getRoot() {return(root);}

    //Interaction Methods
    int setDefiningPoints(const float resolution = LP_RESOLUTION,
                    const float width = LP_WIDTH,
      const float pitch = LP_PITCH,
                    const float tilt = LP_TILT);
    int getDefiningPoints(float &resolution, float &width,
                    float &pitch, float &tilt);
    int setControlPoints();

    void setFocalLength(const float fl = -LP_FOCAL)
         float getFocalLength()
    OUsLaserPlaneInv &setEmissiveColor(SbColor cl)
    SbColor getEmissiveColor() {return(color->emissiveColor[0]);}


*/
class OUsLaserPlane : public OUsLaserPlaneInv {

public:
  OUsLaserPlane(OUsDataSet & data);
  OUsLaserPlane(OUsDataSet & data, SoSeparator *new_plane);
  ~OUsLaserPlane() {intersect_points = NULL;}

  virtual OUsLaserPlane &activatePlane();
  virtual void setToSlice() {intersect_points->clearPoints();};
  virtual void clearScan() {intersect_points->clearPoints();}

  int checkPlane();                //Not yet implimented
```

```
  void setColor(SbColor cl) {
    setEmissiveColor(cl);
    intersect_points = intersect_points->getDataByEmColor(color->emissiveColor[0]);
  }

private:
  void construct(OUsDataSet &data);

  OUsDataSet *intersect_points;
  OUsRay laserbeam;
};
/******************************************************************************

  Filename:  OUsLaserPlaneInv.c++
  Revision:  2.00

  Date:      25 July 97
  Author:    Jeff Collier

  Description:  This file impliements the code for the class OUsLaserPlane.

*/
#include <iostream.h>
#include "OUsLaserPlaneInv.h"

/******************************************************************************
Constructors -- Destructors
******************************************************************************/

/******************************************************************************

  Method:  Default Constructor

*/
OUsLaserPlaneInv::OUsLaserPlaneInv() {

  //Create new nodes for all of the classes pointers.
  root = new SoSeparator;
  root->setName(SbName("Laser_Plane_Root"));

  position = new SoTranslation;
  position->setName(SbName("Position"));

  orientation = new SoRotation;
  orientation->setName(SbName("Orientation"));

  control_points = new SoCoordinate3;
  control_points->setName(SbName("Control_Points"));

  defining_points = new SoCoordinate3;
  defining_points->setName(SbName("Defining_Points"));
```

```
    focal_point = new SoTranslation;
    focal_point->setName(SbName("Focal_Point"));

    SoSeparator *disp = new SoSeparator;
    disp->setName(SbName("Display"));

    //Define color of laser.
    color = new SoMaterial;
    color->setName(SbName("LP_Color"));
    color->ambientColor = RED_LASER;
    color->diffuseColor = RED_LASER;
    color->emissiveColor = RED_LASER;

    display->addChild(new SoPointSet);
    disp->addChild(display);

    //Insert the classes onto a scenegraph.
    root->addChild(position);
    root->addChild(orientation);
    root->addChild(focal_point);
    root->addChild(color);
    root->addChild(defining_points);
    root->addChild(control_points);
    root->addChild(disp);

    setDefiningPoints();
    setFocalLength();
}


/******************************************************************************

    Method:  Constructor based on scene graph.

    Description:  Construct a laser plane from a given scene graph

*/
OUsLaserPlaneInv::OUsLaserPlaneInv(SoSeparator *new_laserplane):
OUsDispObjs(new_laserplane) {

    //Debug
    //cerr << "OUsLaserPlaneInv::Scene graph constructor" << endl;

    SoSearchAction *searcher = new SoSearchAction;
    root = new_laserplane;

    searcher->setName(SbName("Position"));
    searcher->apply(root);
    if (searcher->getPath() == NULL) {
      cerr << "File type incorrect(Position).  CyScan must be restarted." << endl;
      exit(1);
```

```
  }
  position = (SoTranslation *)searcher->getPath()->getTail();

  searcher->reset();
  searcher->setName(SbName("Orientation"));
  searcher->apply(root);
  if (searcher->getPath() == NULL) {
    cerr << "File type incorrect(Orientation). CyScan must be restarted." << endl;
    exit(1);
  }
  orientation = (SoRotation *)searcher->getPath()->getTail();

  searcher->reset();
  searcher->setName(SbName("Control_Points"));
  searcher->apply(root);
  if (searcher->getPath() == NULL) {
    cerr << "File type incorrect(Control_Points).  CyScan must be restarted." <<
endl;
    exit(1);
  }
  control_points = (SoCoordinate3 *)searcher->getPath()->getTail();

  searcher->reset();
  searcher->setName(SbName("Focal_Point"));
  searcher->apply(root);
  if (searcher->getPath() == NULL) {
    cerr << "File type incorrect(Focal_Point).  CyScan must be restarted." << endl;
    exit(1);
  }
  focal_point = (SoTranslation *)searcher->getPath()->getTail();

  searcher->reset();
  searcher->setName(SbName("LP_Color"));
  searcher->apply(root);
  if (searcher->getPath() == NULL) {
    cerr << "File type incorrect(Color).  CyScan must be restarted." << endl;
    exit(1);
  }
  color = (SoMaterial *)searcher->getPath()->getTail();

  searcher->reset();
  searcher->setName(SbName("Defining_Points"));
  searcher->apply(root);
  if (searcher->getPath() == NULL) {
    cerr << "File type incorrect(Defining_Points).  CyScan must be restarted." <<
endl;
    exit(1);
  }
  defining_points = (SoCoordinate3 *)searcher->getPath()->getTail();

  setControlPoints();
```

```
}

/*****************************************************************************

  Method:  Set Defining Points -- Get Defining Points

  Description:  A small set of points can be used to generate the control
    points that describe the laser plane.  This method sets and gets those
    defining points.

*/
int OUsLaserPlaneInv::setDefiningPoints(const float resolution,
                                        const float width,
                                        const float pitch, const float tilt) {

  defining_points->point.set1Value(0, resolution, width, 0);
  defining_points->point.set1Value(1, pitch, tilt, 0);
  setControlPoints();

  return(1);
}

int OUsLaserPlaneInv::getDefiningPoints(float &resolution, float &width,
                     float &pitch, float &tilt) {

  resolution = defining_points->point[0][0];
  width = defining_points->point[0][1];
  pitch = defining_points->point[1][0];
  tilt = defining_points->point[1][1];

  return(1);
}


/*****************************************************************************

  Method:  Set Control Points

  Description:  Sets the points that define the laser plane.  The plane is
    defined as a set of rays from the origin (in the laser plane space)
    through each control point.

  Params:  float lp_resolution -- space between control points.
           float lp_width -- the width of the plane emiter.

*/
int OUsLaserPlaneInv::setControlPoints() {

  float d, h;
```

```
  float lp_resolution = defining_points->point[0][0];
  float lp_width = defining_points->point[0][1];
  float lp_pitch = defining_points->point[1][0];
  float lp_tilt = defining_points->point[1][1];

  float count = -lp_width;
  int index = 0;

  d = focal_point->translation.getValue()[2] * tan(lp_pitch);

  control_points->point.deleteValues(0);
  control_points->point.set1Value(index++, 0, 0, d);
  while (count <= lp_width) {
    //List of points that represent the direction of a laserbeam
    h = count * tan(lp_tilt);
    control_points->point.set1Value(index, h, count,
                                -focal_point->translation.getValue()[2]);
    index++;
    count += lp_resolution;
  }
  return(1);
}


/******************************************************************************

  Filename:  OUsLaserPlaneInv.h
  Revision:  2.00

  Date:      23 July 97
  Author:    Jeff Collier

  Description:

*/
#include "OUsDispObjs.h"
#include "OUsDataSet.h"

/******************************************************************************
Constants related to laser plane (all values in meters).
******************************************************************************/
#define LP_FOCAL 5                  //Laser plane focal length.
#define LP_WIDTH .15                //Half the width of the laser plane
                                    //at scanhead.
#define LP_RESOLUTION .01           //Spacing between beams on the plane.
#define LP_TILT 0                   //Tilt of the laser plane
#define LP_PITCH 0                  //Pitch of the laser plane

/******************************************************************************

  Class:  OUsLaserPlaneInv
```

```
  Description:  Simulates a laser plane.  One point (called the focal point
    not to be confused the focal point of a camera) is taken to be the
    origin for all the rays.  A set of points (called control points)
    indicates a path through which the rays pass.  If the points are co-
    linear the laser plane is a plane that looks like a fan.

    The purpose of this class is to implement the aspects of the laser
    plane that deal with Open Inventor

  Superclass:  OUsBase -> OUsXfrmObjs -> OUsDispObjs
                    Inherited from
  OUsBase:
       SoSeparator *ReadFile(const char *filename)
    int writeFullGraph(char *filename = "graph.iv")

  Inherited from
  OUsXfrmObjects:
    int copyMatrix(const SbMatrix *to_world, const SbMatrix *from_world);
    int linkMatrix(SbMatrix *to_world, SbMatrix *from_world)
    SbMatrix *getTransToWorld() {return trans_to_world;}
    SbMatrix *getTransFromWorld() {return trans_from_world;}
    void displayMatrix()

    //Perform Transforms
    SoCoordinate3 &transFromWorld(SoCoordinate3 *points)
    SbVec3f &transFromWorld(SbVec3f &point)
    SoCoordinate3 &transToWorld(SoCoordinate3 *points);
    SbVec3f &transToWorld(SbVec3f &point)

  Inherited from
  OUsDispObjs:
    //Transformation functions
    virtual int setTransformPath()
    virtual int setTransform()

    //Initialization Methods
    virtual void setToScan(){}
    virtual void setToSlice(){}

    //show/hide objects on the scene graph
    virtual OUsDispObjs &show()
    virtual OUsDispObjs &hide()

    //Interaction methods
    SoSeparator *getRoot() {return(root);}


*/
class OUsLaserPlaneInv : public OUsDispObjs {

public:
```

```
  //Constructors -- Destructors
  OUsLaserPlaneInv();
  OUsLaserPlaneInv(SoSeparator *newlaserplane);

  //Interaction Methods
  int setDefiningPoints(const float resolution = LP_RESOLUTION,
                        const float width = LP_WIDTH,
                        const float pitch = LP_PITCH,
                        const float tilt = LP_TILT);
  int getDefiningPoints(float &resolution, float &width,
                        float &pitch, float &tilt);
  int setControlPoints();

  void setFocalLength(const float fl = -LP_FOCAL) {
    focal_point->translation = SbVec3f(0, 0, fl); setControlPoints();}
  float getFocalLength() {
    return(focal_point->translation.getValue()[2]);}

  OUsLaserPlaneInv &setEmissiveColor(SbColor cl) {
    color->emissiveColor = cl; return(*this);}
  SbColor getEmissiveColor() {return(color->emissiveColor[0]);}

  virtual void clearScan() {}

protected:

  SoTranslation *position;
  SoTranslation *focal_point;
  SoRotation *orientation;
  SoCoordinate3 *control_points;
  SoMaterial *color;
  SoCoordinate3 *defining_points; //First resolution and width
                                  //Second point is pitch and tilt
};
/******************************************************************************

  Filename:  OUsRay.c++
  Revision:  2.20

  Date:      16 Dec 97
  Author:    Jeff Collier

  Description: See OUsRay.h for details.

*/
#include "OUsRay.h"

/******************************************************************************

  Method:  Constructor
```

```
  Description:  Set up to make Fire Ray a little faster.

*/
OUsRay::OUsRay() {

  SbViewportRegion vr;
  laseraction = new SoRayPickAction(vr);
}


/*****************************************************************************

  Method:  Set Ray Internal -- Set Ray External

       Description:  A ray is defined from two points.  First the direction
    is established by subtracting the two points.  Next the starting point
    is defined.  For an internal ray the starting point is defined so that
    the ray starts at a point and passes through the other.  For external
    the ray starts at the point for which the ray would travel away from
    the other point.
*/
OUsRay &OUsRay::setRayInternal(const SbVec3f p1, const SbVec3f p0) {

  dir = p1 - p0;
  startpoint = p0;
  laseraction->setRay(startpoint, dir, RAY_RANGE, 1);

  return(*this);
}

OUsRay &OUsRay::setRayExternal(const SbVec3f p1, const SbVec3f p0) {

  dir = p1 - p0;
  startpoint = p1;
  laseraction->setRay(startpoint, dir);

  return(*this);
}


/*****************************************************************************

  Method:  Fire Ray

  Description:  This applies the defined ray to a portion of an Open
    Inventor scene graph.  The result is returned through an SoPickedPoint
    object.

*/
SoPickedPoint *OUsRay::fireRay(SoGroup *graph) {
```

```
  //Ray fired at object or entire scene
  if (graph == NULL) {
    graph = subjects;
  }

  laseraction->apply(graph);

  return(laseraction->getPickedPoint());
}
```

```
/*****************************************************************************

  Filname:  OUsRay.h
  Revision: 2.0

  Date:     16 Dec  97
  Author:   Jeff Collier

  Description: This class handles all intersecting rays needed by the
               simulator.  The rays are primarily used by the laser
               plane but are also used by the camera

*/
#ifndef _OUSRAY
#define _OUSRAY

/*****************************************************************************
Include Files
*****************************************************************************/
#include "OUsBase.h"
#include "OUsXfrmObjs.h"
#include <Inventor/Sb.h>
#include <Inventor/So.h>
#include <Inventor/SoPickedPoint.h>


/*****************************************************************************
Constant and macro definition
*****************************************************************************/
#define RAY_RANGE .001            //Restricts the distance for ray
                                  // intersection.

const SbVec3f origin(0,0,0);      //Defines the origin for use as a
```

```
                                    // default value.

/*************************************************************************

  Class:  OUsRay

  Description:  Simulates a ray from a laser.  It is used to select a point
    on an object.

  Superclass:  OUsBase -> OUsXfrmObjs

  Inherited from
  OUsBase:
       SoSeparator *ReadFile(const char *filename)
    int writeFullGraph(char *filename = "graph.iv")

  Inherited from
  OUsXfrmObjects:
    int copyMatrix(const SbMatrix *to_world, const SbMatrix *from_world);
    int linkMatrix(SbMatrix *to_world, SbMatrix *from_world);
    SbMatrix *getTransToWorld() {return trans_to_world;}
    SbMatrix *getTransFromWorld() {return trans_from_world;}
    void displayMatrix();

    //Perform Transforms
    SoCoordinate3 &transFromWorld(SoCoordinate3 *points);
    SbVec3f &transFromWorld(SbVec3f &point);
    SoCoordinate3 &transToWorld(SoCoordinate3 *points);
    SbVec3f &transToWorld(SbVec3f &point);

*/
class OUsRay : public OUsXfrmObjs {

public:
  //Constructor
  OUsRay();

  //Pick Ray Set Up
  OUsRay &setRayInternal(const SbVec3f p1, const SbVec3f p0 = origin);
  OUsRay &setRayExternal(const SbVec3f p1, const SbVec3f p0 = origin);

  //Ray Execution
  SoPickedPoint * fireRay(SoGroup *graph = NULL);

private:
  //Information related to the direction placement of the ray.
  SbVec3f startpoint;
  SbVec3f dir;

  //Helpful object for Fire Ray
```

```
  SoRayPickAction *laseraction;
};


#endif
/*****************************************************************************

  Filename:  OUsScanHead.c++
  Revision:  2.00

  Date:      19 Feb 1998
  Author:    Jeff

  Description: See OUsScanhead.h for details.

*/
#include <iostream.h>
#include "OUsScanHead.h"
#include <math.h>

int OUsScanHead::sh_num = 0;

/*****************************************************************************
Constructors -- Destructors
*****************************************************************************/

/*****************************************************************************

  Method:  Constructor

  Description:

*/
OUsScanHead::OUsScanHead(OUsDataSet & data, SoSeparator *new_scanhead):
min(0),
max(0),
mean(0),
sv(0) {

  scan_data = data.addDataSet();
  intersect_data = data.addDataSet();

  setColor();

  if (new_scanhead == NULL) {
    //New Scanhead.
    root = new SoSeparator;
    root->setName(SbName("Scanhead_Root"));
    addObject(new OUsCamera(data));
    addObject(new OUsLaserPlane(data));
    addObject(new OUsCamera(data));
```

```
    top_camera = (OUsCamera *)getObject(0);
    bottom_camera = (OUsCamera *)getObject(2);
    laser_plane = (OUsLaserPlane *)getObject(1);

    connectCameras();
  } else {

    //Scanhead from graph.
    if(new_scanhead->getName() != SbName("Scanhead_Root")) {
      cerr << "Error -- File type incorrect, no scanhead found\n";
      exit(1);
    }
  }
  sh_num++;
}


/******************************************************************************

  Method:  Activate Cameras

*/
OUsScanHead & OUsScanHead::activateCameras() {

  //Capture Points with cameras.
  top_camera->captureFrame();
  bottom_camera->captureFrame();
  top_camera->mergeWith(*bottom_camera);

 return (*this);
}


/******************************************************************************

  Method:  Connect Cameras

  Description:  This causes two cameras to be linked together to work as a
    pair.  So that only one camera needs to be considered by the designer
    and the second follows the first.

 */
OUsScanHead & OUsScanHead::connectCameras() {

    bottom_camera->makeCoopCamera();
    return(*this);
}


/******************************************************************************
```

```
  Method:  Recover Points

  Description:  This gets the points from the main camera and determines
    which world points correspond.

 */
OUsScanHead & OUsScanHead::recoverPoints(float track_tran) {

  SoCoordinate3 *new_data = new SoCoordinate3;
  SoCoordinate3 *new_intersects = new SoCoordinate3;

  new_data->point.deleteValues(0);
  new_intersects->point.deleteValues(0);

  top_camera->generateWorldPoints(new_data, new_intersects);
  adjustForTrack(new_data, track_tran);

  scan_data->addPoints(new_data);
  intersect_data->addPoints(new_intersects);

  return (*this);
}


/*****************************************************************************

  Method: Create Test Grid

  Description:  Uses the camera's ability to create a test grid based on the
    position of the camera.

*/
OUsScanHead & OUsScanHead::createTestGrid() {

  clearScan();
  top_camera->createTestGrid();
  recoverPoints(0);

  return (*this);
}


/*****************************************************************************

  Method:  Clear Scan

  Description:  Clears out all scanned data relating to a particular scan.

 */
void OUsScanHead::clearScan() {
```

```
  scan_data->clearPoints();
  intersect_data->clearPoints();

  OUsContainers::clearScan();
}

/*****************************************************************************

  Method:  Set color.

  Description:  Simple little scheme that alters the color for the data
    for the first four scan heads created.

 */
void OUsScanHead::setColor() {

  switch (sh_num) {
  case 0: scan_data->setDifColor(HEAD0_COLOR);
    break;
  case 1: scan_data->setDifColor(HEAD1_COLOR);
    break;
  case 2: scan_data->setDifColor(HEAD2_COLOR);
    break;
  case 3: scan_data->setDifColor(HEAD3_COLOR);
    break;
  }
}

/*****************************************************************************

  Method:  Adjust for track

  Description:  The scan head only generates a Z-Y coordinate as though the
    scan head is not moving.  This method adds the offset of the position
    on the track


 */
OUsScanHead & OUsScanHead::adjustForTrack(SoCoordinate3 * new_data,
                                          float track_tran) {

  int num = new_data->point.getNum();
  SbVec3f* points;

  points = new_data->point.startEditing();

  for(int i = 0; i < num; i++)
    points[i][0] = track_tran;

  new_data->point.finishEditing();
```

```
 return (*this);
}


/*****************************************************************************

  Method:  Set Transform Path

*/
int OUsScanHead::setTransformPath() {

  for (int i = 0; i < num_objs; i++)
    objects[i]->setTransformPath();

  return(1);
}


/*****************************************************************************

  Method:  Set Transform

*/
int OUsScanHead::setTransform() {

  for (int i = 0; i < num_objs; i++)
    objects[i]->setTransform();

  return(1);
}


/*****************************************************************************

  Method:  Get Stats

  Description:  This method calculates the statistics on a set of point that
    have been simulated so the designer can get a feel for the amount of
    precision in the data.


 */
OUsScanHead &OUsScanHead::getStats(float &rhs_min, float &rhs_max,
                                   float &rhs_mean, float &rhs_sv) {

  getStats();

  rhs_min = min;
  rhs_max = max;
  rhs_mean = mean;
  rhs_sv = sv;
```

```
  return (*this);
}

OUsScanHead &OUsScanHead::getStats() {

  float *dif, tot, range;
  SbVec3f dif_vec;
  float d2 = 0, Ex2 = 0;
  int i = 0;
  int num = intersect_data->getPoints()->point.getNum();
  int num_p = scan_data->getPoints()->point.getNum();
  dif = new float[num];

  if(num != num_p) cerr << "number intersects, points" << endl;
  if((num > 0) && (num_p > 0) && (num == num_p)) {
    SbVec3f *in_pts = intersect_data->getPoints()->point.startEditing();
    SbVec3f *scan_pts = scan_data->getPoints()->point.startEditing();

    dif_vec = scan_pts[i] - in_pts[i];
    dif[i] = dif_vec.length();
    tot = min = max = dif[i];

    for(i = 1; i < num; i++) {
      dif_vec = scan_pts[i] - in_pts[i];
      dif[i] = dif_vec.length();

      if (dif[i] > max) max = dif[i];
      if (dif[i] < min) min = dif[i];

      d2 = (dif[i] * dif[i]);
      Ex2 += d2;
      tot += dif[i];


    }


    float Sxx = Ex2 - ((tot * tot) / num);
    sv = sqrt(Sxx / (num - 1));

    if(i != 0) mean = tot / (num - 1);
    range = max - min;

    intersect_data->getPoints()->point.finishEditing();
    scan_data->getPoints()->point.finishEditing();

    scan_data->setDifColor(SbColor(1,0,0));
    SoMFColor *color_map = scan_data->getDiffuseColor();
    float hue;

    for(i = 0; i < num; i++) {
```

```
      hue = (sv - dif[i]) * (.75 / sv);
      if(hue = 0) hue = 1;
      color_map->set1HSVValue(i, hue, 1, 1);
    }

  }

  intersect_data->hide();
  return(*this);
}
/*****************************************************************************

  Filename:  OUsScanHead.h
  Revision:  2.00

  Date:     19 Feb 1998
  Author:   Jeff Collier

  Description: This class simulates a scanhead which is a collection of
               a laserplane and cameras.

*/
#include "OUsContainers.h"
#include "OUsDataSet.h"
#include "OUsLaserPlane.h"
#include "OUsCamera.h"

/*****************************************************************************

  Method:  OUsScanHead

  Description:  This class models a scan head witch contains two cameras
    and a laser plane.  It helps collect the functionality of these devices
    so that simulation data may be collected more efficiently.

  Superclass:  OUsBase -> OUsXfrmObjects -> OUsDispObjs -> OUsContainers

  Inherited from
  OUsXfrmObjects:
    int copyMatrix(const SbMatrix *to_world, const SbMatrix *from_world);
    int linkMatrix(SbMatrix *to_world, SbMatrix *from_world);
    SbMatrix *getTransToWorld() {return trans_to_world;}
    SbMatrix *getTransFromWorld() {return trans_from_world;}
    void displayMatrix();

    //Perform Transforms
    SoCoordinate3 &transFromWorld(SoCoordinate3 *points);
    SbVec3f &transFromWorld(SbVec3f &point);
    SoCoordinate3 &transToWorld(SoCoordinate3 *points);
    SbVec3f &transToWorld(SbVec3f &point);
```

```
    Inherited from
    OUsDispObjs:
      //Transformation functions
         //Initialization Methods
      virtual void setToScan(){};
      virtual void setToSlice(){};

      //show/hide objects on the scene graph
      virtual OUsDispObjs & show()
      virtual OUsDispObjs & hide()

      //Interaction methods
      SoSeparator *getRoot() {return(root);}

    Inherited from
    OUsDispObjs:
      virtual int setTransformPath();
      virtual int setTransform();

      virtual void setToScan();
      virtual void setToSlice();

      int getNumObjects() {return(num_objs);}
      OUsContainers &addObjectSG(OUsDispObjs * obj_to_add);
      OUsContainers &addObject(OUsDispObjs * obj_to_add);
      OUsDispObjs *getObject(int ind) {return(objects[ind]);}

      OUsDispObjs &show();
      OUsDispObjs &hide();
*/
class OUsScanHead : public OUsContainers {

public:
  //Constructors -- Destructors
  OUsScanHead(OUsDataSet &data, SoSeparator *scanhead = NULL);
  ~OUsScanHead() {scan_data = NULL;}

  //Camera Functionality
  OUsScanHead &activateCameras();
  OUsScanHead &connectCameras();
  OUsScanHead &recoverPoints(float track_tran);
  OUsScanHead &createTestGrid();

  //Genera Functionality
  void clearScan();
  void setColor();
  OUsScanHead &dataToCyScan(int gr_num) {
    scan_data->placeInCyScan(gr_num); return(*this);}

  OUsScanHead &adjustForTrack(SoCoordinate3 * new_data, float track_tran);
```

```
    int setTransform();
    int setTransformPath();

    //Interface functions
    OUsCamera *getTopCamera() {
      return(top_camera = (OUsCamera *)getObject(0));}
    OUsCamera *getBottomCamera() {
      return(bottom_camera = (OUsCamera *)getObject(2));}
    OUsLaserPlane *getLaserPlane() {
      return(laser_plane = (OUsLaserPlane *)getObject(1));}
    OUsScanHead &getStats(float &rhs_min, float &rhs_max,
                          float &rhs_mean, float &rhs_sv);
    OUsScanHead &getNumScannedPoints(int &n) {
      n = scan_data->getPoints()->point.getNum(); return (*this);}

  private:
    OUsScanHead &getStats();

    OUsCamera *top_camera;
    OUsCamera *bottom_camera;
    OUsLaserPlane *laser_plane;

    OUsDataSet *scan_data;
    OUsDataSet *intersect_data;
    static int sh_num;
    SoCoordinate3 *FRZcoord;

    float min, max, mean, sv;
  };
  /*****************************************************************************

    Filename:  OUsSimulator.c++
    Revision:  2.00

    Date:      08 Aug 97
    Author:    Jeff Collier

    Description:  For details see OUsSimulator.h

  */
  #include "OUsSimulator.h"

  /*****************************************************************************
  Constructors -- Destructors
  *****************************************************************************/

  /*****************************************************************************

    Method:  Constructor based on CyIvState *

    Description:  Constructs a Simulator
```

```
*/
OUsSimulator::OUsSimulator(CyIvState *state) :
OUsBase(state),
num_tracks(0),
num_lasers(0),
num_cameras(0),
num_scanheads(0),
num_slices(TOTAL_INC) {}

/*****************************************************************************

  Method:  Remove Scanner

  Description:  Removes all the scan heads and tracks and sets the numbers
    of each to 0

*/
int OUsSimulator::removeSimulator() {

  num_lasers--;
  while (num_lasers >= 0) {
    delete laser[num_lasers];
    laser[num_lasers] = NULL;
    num_lasers--;
  }
  num_lasers = 0;

  num_cameras--;
  while (num_cameras >= 0) {
    delete camera[num_cameras];
    camera[num_cameras] = NULL;
    num_cameras--;
  }
  num_cameras = 0;

  num_scanheads--;
  while (num_scanheads >=0) {
    delete scanhead[num_scanheads];
    scanhead[num_scanheads] = NULL;
    num_scanheads--;
  }
  num_scanheads = 0;

  num_tracks--;
  while (num_tracks >= 0) {
    delete track[num_tracks];
    track[num_tracks] = NULL;
    num_tracks--;
  }
  num_tracks = 0;
```

```
  simulator_root->removeChild(0);
  return(1);
}


/*****************************************************************************
Add Functions
*****************************************************************************/

/*****************************************************************************

  Method:  Add Scan Head

  Description:  Adds a scanhead to a track specified by tracknum.  A new
    ScanHead is created.

  Params:  tracknum -- The number of the track to insert the scan head.
                       This paramerter defaults to -1 which indicates that
                       the last track added shouls be used.

           newscanhead -- The root of a scenegraph that defines a scanhead.
                          default value is NULL.

  Return:  TCL_OK -- if track is added.
           TCL_ERROR -- if track number out of range.

*/
int OUsSimulator::addScanHead(short track_num, OUsScanHead *new_scanhead) {

  //If track not specified use the latest track.
  if (track_num == -1) track_num = num_tracks - 1;

  if (track_num < num_tracks) {
    if (new_scanhead == NULL)
      new_scanhead = new OUsScanHead(data_sets);

    track[track_num]->addObject(new_scanhead);
    scanhead[num_scanheads++] = new_scanhead;
    camera[num_cameras++] = new_scanhead->getTopCamera();
    camera[num_cameras++] = new_scanhead->getBottomCamera();
    laser[num_lasers++] = new_scanhead->getLaserPlane();
  }
  else {
    return(0);
  }

  return(1);
}


/*****************************************************************************
```

```
  Method:  Add Scan Head

  Description:  Adds a scan head based on an open inventor description of one.
*/

OUsScanHead * OUsSimulator::addScanHead(SoSeparator *sr) {

  SoSeparator *rt;
  scanhead[num_scanheads++] = new OUsScanHead(data_sets, sr);
  int cur_sh_num = num_scanheads - 1;

  rt = (SoSeparator *)sr->getChild(0);
  if(rt->getName() != SbName("Camera_Root")) {
    cerr << "Error -- File type incorrect, first cammera not found\n";
    exit(1);
  }
  scanhead[cur_sh_num]->addObjectSG(addCamera(rt));
  scanhead[cur_sh_num]->getTopCamera();

  rt = (SoSeparator *)sr->getChild(1);
  if(rt->getName()  != SbName("Laser_Plane_Root")) {
    cerr << "Error -- File type incorrect, first cammera not found\n";
    exit(1);
  }
  scanhead[cur_sh_num]->addObjectSG(addLaserPlane(rt));
  scanhead[cur_sh_num]->getLaserPlane();

  rt = (SoSeparator *)sr->getChild(2);
  if(rt->getName() != SbName("Camera_Root")) {
    cerr << "Error -- File type incorrect, first cammera not found\n";
    exit(1);
  }
  scanhead[cur_sh_num]->addObjectSG(addCamera(rt));
  scanhead[cur_sh_num]->getBottomCamera();

  return scanhead[cur_sh_num];
}

 /*****************************************************************************

  Method:  Add Track

  Description:  Adds a track based on an open inventor scene graph description
    of a track.  This may require a recursive call because a track can
    contain a track.

 */
int OUsSimulator::addTrack(SoSeparator *tr) {

  track[num_tracks++] = new OUsTrack(tr);
```

```
    SoSearchAction *searcher = new SoSearchAction;

    //Add a scanhead if present
    searcher->setName(SbName("Scanhead_Root"));
    searcher->apply(tr);
    if (searcher->getPath() != NULL) {
      SoSeparator *sh_root = (SoSeparator *)searcher->getPath()->getTail();
      track[num_tracks - 1]->addObjectSG(addScanHead(sh_root));
    }

    //Search children for tracks.  If found add.
    int nodes = tr->getNumChildren();
    for (int i = 0; i < nodes; i++) {
      searcher->reset();
      searcher->setName(SbName("Track_Root"));
      searcher->apply(tr->getChild(i));

      if (searcher->getPath() != NULL) {
        SoSeparator *t_root = (SoSeparator *)searcher->getPath()->getTail();
        addTrack(t_root);
        cerr << "Found nested tracks!" << endl;
      }
    }

    return(1);
}

/*****************************************************************************

    Method:  Add Laser Plane -- Add Camera

    Description:  These methods return pointers to a camera or laserplane that
      was created from an Open Inventor scene graph.

 */
OUsLaserPlane * OUsSimulator::addLaserPlane(SoSeparator *lr) {
    return (laser[num_lasers++] = new OUsLaserPlane(data_sets, lr));
}

OUsCamera * OUsSimulator::addCamera(SoSeparator *cr) {
    return (camera[num_cameras++] = new OUsCamera(data_sets, cr));
}


/*****************************************************************************

    Method:  Scan Slice

    Description:  Gets the current position of the track, Scans a slice at
      that position, increments the position on the track.
```

```
*/
int OUsSimulator::scanSlice() {

  int i;

  for (i = 0; i < num_tracks; i++)
    track[i]->setToSlice();

  float offset;
  //Fire each laser plane.
  for (i = 0; i < num_lasers; i++)
    laser[i]->activatePlane();


  //Activate each camera
  for (i = 0; i < num_scanheads; i++) {
    scanhead[i]->activateCameras();
    offset = track[i]->getTranslation();
    scanhead[i]->recoverPoints(offset);
  }

  //Update each track
  for (i = 0; i < num_tracks; i++)
    track[i]->incriment();

  return(1);
}


/*****************************************************************************

  Method:  Create Test Grid

  Description:  Causes a test grid, relative to the scanners current
    position to be created.  The test grid should coraspond to the grid in
    the template file.
*/
int OUsSimulator::createTestGrid() {

  //Set the transforms to match the new position.
  for (int i = 0; i < num_scanheads; i++)
    scanhead[i]->createTestGrid();

  return(1);
}


/*****************************************************************************

  Method: Set to scan
```

```
  Description:  Initializes each track to the starting position.

*/
int OUsSimulator::setToScan() {

  //Reset the position for the scanheads on each track.
  for (int i = 0; i < num_tracks; i++) {
    track[i]->setToScan();
    track[i]->setToSlice();
  }

  return(1);
}


/*****************************************************************************

  Method:  Clear Scanned Points

  Description:  Clears all points that have been created by the simulator
    These points could be simulated or test grid points.

*/
int OUsSimulator::clearScannedPoints() {

  //Set the transforms to match the new position.

  for (int i = 0; i < num_scanheads; i++) {
    scanhead[i]->clearScan();
  }

  return(1);
}


/*****************************************************************************
File Functions
*****************************************************************************/

/*****************************************************************************

  Method:  Replace Scanner

  Description:  Takes a scanner in scene graph form and creates a scanner
    from it.  The old scanner is replaced by the one on the new scene graph

*/
int OUsSimulator::replaceScanner(SoSeparator *scanner) {

  SoSearchAction *searcher = new SoSearchAction;
  //Set up track_root and check to make sure it exists.
```

```
  scanner_elements = scanner;

  searcher->setName(SbName("Track_Root"));
  searcher->setInterest(SoSearchAction::ALL);
  searcher->apply(scanner_elements);

  SoPathList pl;
  pl = searcher->getPaths();
  int tracks = pl.getLength();

  if (tracks < 1) {
    cerr << "No scanner elements found on this scenegraph!\n";
    return (0);
  }

  //Remove old scanner from the scene graph and insert a new one.
  removeSimulator();
  simulator_root->insertChild(scanner_elements, 0);

  for(int i = 0; i < tracks; i++) {
    SoSeparator *tr = (SoSeparator *)searcher->getPaths()[i]->getTail();
    addTrack(tr);
  }

  setToScan();
  return (1);
}


/******************************************************************************

  Method:  Write To File

  Description:  Writes the scene graph to an open inventor file.

*/
int OUsSimulator::writeToFile(char *filename) {

  setToScan();

  SoOutput *out = new SoOutput;
  FILE *output;
  output = fopen(filename, "w");
  out->setFilePointer(output);

  SoWriteAction *Write = new SoWriteAction(out);
  Write->apply(scanner_elements);

  fclose(output);

  return (1);
```

```
}
/*************************************************************************

  Filename:  OUsSimulator.h
  Revision:  2.00

  Date:      18 April 1998
  Author:    Jeff Collier

  Description:  This class is the main simulator.

*/
#ifndef _OU_SIMULATE
#define _OU_SIMULATE

/*********************************************************************
Header Files
*********************************************************************/
#include "OUsTrack.h"
#include "OUsScanHead.h"
#include <stdio.h>
#include <iostream.h>

#define NUM_GRAPH 6
#define NUM_ITEMS 16

/*************************************************************************

  Class:  OUsSimulator

  Description:  This class impliments a laser light simulator.  It contains
    an arrays of pointers for OUsCameras OUsLaserPlanes OUsScanHeads and
    OUsDataSets.

  Superclass:  OUsBase

  Inherited From
  OUsBase:
       SoSeparator *ReadFile(const char *filename)
       int writeFullGraph(char *filename = "graph.iv")

*/
class OUsSimulator : public OUsBase {

public:
  //Constructor -- Destructors
  OUsSimulator(CyIvState *state);
  ~OUsSimulator() {removeSimulator();}

  int removeSimulator();
```

```
  //Add Functions
  int addTrack(OUsTrack *new_track) {
    track[num_tracks++] = new_track; return(num_tracks - 1);}
  int addScanHead(short track_num = -1,    OUsScanHead *new_scanhead = NULL);

  //Add based on scene graph
  OUsScanHead *addScanHead(SoSeparator *sr);
  int addTrack(SoSeparator *tr);
  OUsLaserPlane *addLaserPlane(SoSeparator *lr);
  OUsCamera *addCamera(SoSeparator *cr);

  //Simulator Functions
  int scanSlice();
  int setToScan();
  int createTestGrid();
  int clearScannedPoints();

  //File Functions
  int replaceScanner(SoSeparator *scanner);
  int insertScanner(SoSeparator *scanner) {return(replaceScanner(scanner));}
  int writeToFile(char *filename = "scanner.iv");

protected:

  OUsTrack *track[NUM_ITEMS];
  OUsLaserPlane *laser[NUM_ITEMS];
  OUsCamera *camera[NUM_ITEMS];
  OUsScanHead *scanhead[NUM_ITEMS];
  OUsDataSet data_sets;

  short num_tracks;
  short num_lasers;
  short num_cameras;
  short num_scanheads;

  int num_slices;
};

#endif
/******************************************************************************

  Method:  OUsTrack.c++
  Revision: 1.15

  Date:     23 July 97
  Author:   Jeff Collier

  Description:  For a description of this class see OUsTrack.h

*/
```

```
#include "OUsTrack.h"
#include <iostream.h>
/*****************************************************************************

  Method:  OUsTrack.h
  Revision: 1.15

  Date:     23 July 97
  Author:   Jeff Collier

  Description:  Class OUsTrack represents the track of a laser light
                plane body scanner.  The details of the track are stored
                on an Open Inventor scene graph so that the information
                may be used to simulate a scan.

*/
#ifndef _OU_TRACK
#define _OU_TRACK

/*****************************************************************************
Include Files
*****************************************************************************/
#include "OUsTrackInv.h"
#include <Inventor/nodes/SoTranslation.h>
#include <Inventor/nodes/SoRotation.h>
#include <Inventor/So.h>




class OUsTrack : public OUsTrackInv {
public:
  OUsTrack() {}
  OUsTrack(SoSeparator *tr): OUsTrackInv(tr) {}

};

#endif
/*****************************************************************************

  Method:  OUsTrackInv.c++
  Revision: 1.15

  Date:     23 July 97
  Author:   Jeff Collier

  Description:  For a description of this class see OUsTrackInv.h

*/
#include "OUsTrackInv.h"
```

```
#include <iostream.h>

/******************************************************************************
Static and Global Declarations
******************************************************************************/
float DEF_KNOT_PTS[8] = {0, 0, 0, 0, 3, 3, 3, 3};
float DEF_CONT_PTS[4][3] = {
  { 0.0, 0.0, 0.0 },
  { 0.0, 0.0, 0.0 },
  { 2.0, 0.0, 0.0 },
  { 2.0, 0.0, 0.0 }};


/******************************************************************************
Constructors -- Destructors
******************************************************************************/

/******************************************************************************

  Method:  Default Constructor

  Description:  Construct a completely new track or constructs a track from
    a scene graph of a track.  The constructor needs a pointer the
    trackroot of the scenegraph.

*/
OUsTrackInv::OUsTrackInv():
inc_val(INC_VAL),
total_inc(TOTAL_INC) {

  //Create all nodes on the scene graph (members of class)
  root = new SoSeparator;
  root->setName(SbName("Track_Root"));

  origin = new SoTranslation;
  origin->setName(SbName("Origin"));
  origin->translation = (SbVec3f(0, 0, TRACK_XPOS));

  control_points = new SoCoordinate3;
  control_points->setName(SbName("Control_Points"));

  track_rotation = new SoRotation;
  track_rotation->setName(SbName("Track_Rotation"));
  track_rotation->rotation = SbRotation(SbVec3f(0, 1, 0), 0);

  track_position = new SoTranslation;
  track_position->setName(SbName("Track_Position"));
  track_position->translation = (SbVec3f(0, 0, 0));

  inc_point = new SoCoordinate3;
  inc_point->setName(SbName("Incriment_Point"));
```

```
  inc_point->point.setValue(inc_val, total_inc, 0);


  //Create all nodes on the scene graph (non-members of class)
  SoGroup *track = new SoGroup;
  SoMaterial *track_material = new SoMaterial;
  SoDrawStyle *track_drawstyle = new SoDrawStyle;
  SoNurbsCurve *display_track = new SoNurbsCurve;

  //Set up to display the default track
  SoDrawStyle *track_draw_style = new SoDrawStyle;
  track_draw_style->lineWidth = DISP_LINE_WIDTH;
  track_draw_style->pointSize = DISP_POINT_SIZE;
  control_points->point.setValues(0, NUM_CONT_PTS, DEF_CONT_PTS);
  display_track->numControlPoints = NUM_CONT_PTS;
  display_track->knotVector.setValues(0, NUM_KNOT_PTS, DEF_KNOT_PTS);

  SoSeparator * disp = new SoSeparator;
  disp->setName(SbName("Display"));
  disp->addChild(display);
  display->whichChild = SO_SWITCH_ALL;

  //Create Graph (level 1)
  root->addChild(track);

  //Create Graph (level 2)
  track->addChild(origin);
  track->addChild(disp);
  track->addChild(inc_point);
  track->addChild(track_rotation);
  track->addChild(track_position);

  //Create Graph (level 3)
  display->addChild(track_material);
  display->addChild(track_draw_style);
  display->addChild(control_points);
  display->addChild(display_track);

  //Put graph on simulator scene graph.
  scanner_elements->addChild(root);
}

OUsTrackInv::OUsTrackInv(SoSeparator *new_track_root):
OUsContainers(new_track_root),
origin(NULL),
track_position(NULL),
track_rotation(NULL),
control_points(NULL),
inc_point(NULL) {

  SoSearchAction *searcher = new SoSearchAction;
```

```
  root = new_track_root;

#ifdef DEBUG_TRACKINV
  if(root == NULL) {
    cerr << "Error! -- scene graph NULL in construct from scene graph"
        << endl;
    exit(1);
  }
#endif

  searcher->setName(SbName("Origin"));
  searcher->apply(root);
  if(searcher->getPath() == NULL) origin = NULL;
  else origin = (SoTranslation *)searcher->getPath()->getTail();

  searcher->reset();
  searcher->setName(SbName("Control_Points"));
  searcher->apply(root);
  if(searcher->getPath() == NULL) control_points = NULL;
  else control_points = (SoCoordinate3 *)searcher->getPath()->getTail();

  searcher->reset();
  searcher->setName(SbName("Track_Rotation"));
  searcher->apply(root);
  if(searcher->getPath() == NULL) track_rotation = NULL;
  track_rotation = (SoRotation *)searcher->getPath()->getTail();

  searcher->reset();
  searcher->setName(SbName("Track_Position"));
  searcher->apply(root);
  if(searcher->getPath() == NULL) track_position = NULL;
  track_position = (SoTranslation *)searcher->getPath()->getTail();

  searcher->reset();
  searcher->setName(SbName("Incriment_Point"));
  searcher->apply(root);
  if(searcher->getPath() == NULL) inc_point = NULL;
  inc_point = (SoCoordinate3 *)searcher->getPath()->getTail();

  if ((root == NULL)         ||
      (track_rotation == NULL)  ||
      (track_position == NULL)  ||
      (origin == NULL)              ||
      (control_points == NULL)  ||
      (inc_point == NULL)) {
    cerr << "ERROR -- File not successfully loaded! (track constructor)\n";
    cerr << "Root: " << root << "\ntrack_position:  " << track_position
        << "\ntrack_rotation" << track_rotation
        << "\norigin:  " << origin << "\ncontrol points:  " << control_points
        << "\ninc_point:  " << inc_point << endl;
    exit(1);
```

```
  }

  inc_val = inc_point->point[0][0];
  total_inc = (int)rint(inc_point->point[0][1]);
}


/*****************************************************************************
General Functionality
*****************************************************************************/

/*****************************************************************************

  Method:  Get Origin

  Description:  This method puts the origin of a track into three floats
    to be altered in the calling program

  Params:  float x, y, z -- variables to be altered to contain data from
    translation of the origin.

*/
OUsTrackInv & OUsTrackInv::getOrigin(float &x, float &y, float &z) {

    x = origin->translation.getValue()[0];
    y = origin->translation.getValue()[1];
    z = origin->translation.getValue()[2];
    return(*this);
}


/*****************************************************************************

  Method:  Incriment Scan Head

  Description:  This translates the position of scanhead along the track.

*/
OUsTrackInv & OUsTrackInv::incriment() {

  SbVec3f temp;

  temp = track_position->translation.getValue();
  temp[0] += inc_val;
  track_position->translation = temp;

  return(*this);
}

/*****************************************************************************
```

```
  Method:  Set To Scan

  Description:  This method prepares a track to scan an entire subject.
    The track position is set to the starting point and the transform path
    is created.

*/
void OUsTrackInv::setToScan() {

  //Home track
  SbVec3f temp(0,0,0);
  track_position->translation = temp;

  setTransformPath();
  OUsContainers::setToScan();
}
/******************************************************************************

  Method:  OUsTrackInv.h
  Revision: 2.00

  Date:     3 Dec 97
  Author:   Jeff Collier

  Description:  Class OUsTrackInv represents the track of a laser light
                plane body scanner.  The details of the track are stored
                on an Open Inventor scene graph so that the information
                may be used to simulate a scan.

*/
#ifndef _OU_TRACK_INV
#define _OU_TRACK_INV
#define DEBUG_TRACKINV

/******************************************************************************
Include Files
******************************************************************************/
#include "OUsContainers.h"
#include <Inventor/nodes/SoTranslation.h>
#include <Inventor/nodes/SoRotation.h>
#include <Inventor/So.h>


/******************************************************************************
Definitions
******************************************************************************/
#define INC_VAL .002                    //Default distance to incriment.
#define TOTAL_INC 50                    //Total num of slices to incriment.
#define LINE_W 3                        //How the line should be displayed.
#define POINT_S 5                       //
#define NUM_KNOT_PTS 8                  //Number of points for NURBS
```

```
#define NUM_CONT_PTS 4                  //
#define TRACK_XPOS -0.75                //Default location for a track.
#define NUM_TEST_POINTS 10              //Demension of the test grid.


/*****************************************************************************

  Class OUsTrackInv

  Description: This class handles the open inventor functionality of a class
    that represents a track in a structured light scanner

  Superclass:  OUsBase -> OUsXfrmObjects -> OUsDispObjs -> OUsContainers

  Inherited from
  OUsXfrmObjects:
    int copyMatrix(const SbMatrix *to_world, const SbMatrix *from_world);
    int linkMatrix(SbMatrix *to_world, SbMatrix *from_world);
    SbMatrix *getTransToWorld() {return trans_to_world;}
    SbMatrix *getTransFromWorld() {return trans_from_world;}
    void displayMatrix();

    //Perform Transforms
    SoCoordinate3 &transFromWorld(SoCoordinate3 *points);
    SbVec3f &transFromWorld(SbVec3f &point);
    SoCoordinate3 &transToWorld(SoCoordinate3 *points);
    SbVec3f &transToWorld(SbVec3f &point);

  Inherited from
  OUsDispObjs:
    //Transformation functions
       //Initialization Methods
    virtual void setToScan(){};
    virtual void setToSlice(){};

    //show/hide objects on the scene graph
    virtual OUsDispObjs & show()
    virtual OUsDispObjs & hide()

    //Interaction methods
    SoSeparator *getRoot() {return(root);}

  Inherited from
  OUsDispObjs:
    virtual int setTransformPath();
    virtual int setTransform();

    virtual void setToScan();
    virtual void setToSlice();

    int getNumObjects() {return(num_objs);}
```

```
    OUsContainers &addObjectSG(OUsDispObjs * obj_to_add);
    OUsContainers &addObject(OUsDispObjs * obj_to_add);
    OUsDispObjs *getObject(int ind) {return(objects[ind]);}

    OUsDispObjs &show();
    OUsDispObjs &hide();

*/
class OUsTrackInv : public OUsContainers {

public:
  //Constructors -- Destructors
  OUsTrackInv();
  OUsTrackInv(SoSeparator *new_track_root);

  //General Functionality
  OUsTrackInv &setOrigin(SbVec3f or) {
       origin->translation = or; return(*this);}
  OUsTrackInv &getOrigin(float &x, float &y, float &z);
  OUsTrackInv &getRotation(SbRotation &rm) {
    rm = track_rotation->rotation.getValue(); return (*this);}
  OUsTrackInv & incriment();

  //Interface Functions
  OUsTrackInv & setRotation(SbRotation rm) {
    track_rotation->rotation = rm; return(*this);}
  float getTranslation() {return(track_position->translation.getValue()[0]);}
  OUsTrackInv & setIncriment(float inc) {
    inc_point->point.setValue(inc,total_inc,0); inc_val=inc; return(*this);}
  float getIncriment() {return(inc_val);}
  OUsTrackInv & setTotalInc(int tin) {
    inc_point->point.setValue(inc_val,tin,0); total_inc=tin; return(*this);}
  float getTotalInc() {return(total_inc);}

  void setToSlice() {setTransform(); OUsContainers::setToSlice();}
  void setToScan();



private:
  float inc_val;
  int total_inc;

  SoTranslation *origin;
  SoTranslation *track_position;
  SoRotation *track_rotation;
  SoCoordinate3 *control_points;
  SoCoordinate3 *inc_point;
};

#endif
```

```
/*****************************************************************************

  Filename:  OUsXfrmObjs.c++
  Revision:  2.00

  Date:      04 Dec 97
  Author:    Jeff Collier

  Description:  See OUsBasic.h for details.

*/
#include "OUsXfrmObjs.h"

/*****************************************************************************

  Method:  Constructor -- Destructor

*/
OUsXfrmObjs::OUsXfrmObjs() {

  trans_to_world = new SbMatrix;
  trans_from_world = new SbMatrix;
}

OUsXfrmObjs::~OUsXfrmObjs() {

  delete trans_to_world;
  delete trans_from_world;
}


/*****************************************************************************

  Method:  Copy Matrix -- Link Matrix

  Description:  These methods explictly set the transformation matrices.
    For copy the matrices are copied from the matrices passed.  For link
    The matrices are made to point to the matrices passed.

  Params:  SbMatrix * -- to_world, from_world:  These are the matrices to
    be copied from or linked to.

*/
int OUsXfrmObjs::copyMatrix(const SbMatrix *to_world,
                            const SbMatrix *from_world) {

  *trans_to_world = *to_world;
  *trans_from_world = *from_world;

  return(1);
}
```

```
int OUsXfrmObjs::linkMatrix(SbMatrix *to_world, SbMatrix *from_world) {

  delete trans_to_world;
  delete trans_from_world;

  trans_to_world = to_world;
  trans_from_world = from_world;

  return(1);
}


/*****************************************************************************

  Method:  Transform from World Space to Object Space

  Description:  These methods transform a point or set of points from world
    space to object space.  Both of these functions alter the information
    passed to them.  They also return references to the objects passed to
    them so that this method may be nested inside another.

*/
SoCoordinate3 &OUsXfrmObjs::transFromWorld(SoCoordinate3 *points){

  int num = points->point.getNum();
  SbVec3f *arr_points;
  arr_points = points->point.startEditing();

  for (int i = 0; i < num; i++)
    transFromWorld(arr_points[i]);

  points->point.finishEditing();
  return (*points);
}

SbVec3f &OUsXfrmObjs::transFromWorld(SbVec3f &point) {

  trans_from_world->multVecMatrix(point, point);
  return point;
}


/*****************************************************************************

  Method:  Transform from Object Space to World Space

  Description:  These methods work exactly the same way the transform from
    world to object methods do but in reverse order.  See Transform from
    World Space to Object space for more details.

*/
```

```
SoCoordinate3 & OUsXfrmObjs::transToWorld(SoCoordinate3 *points){

  int num = points->point.getNum();
  SbVec3f *arr_points;
  arr_points = points->point.startEditing();

  for (int i = 0; i < num; i++)
    transToWorld(arr_points[i]);

  points->point.finishEditing();
  return (*points);
}

SbVec3f & OUsXfrmObjs::transToWorld(SbVec3f &point) {

  trans_to_world->multVecMatrix(point, point);
  return point;
}

/******************************************************************************

       Method:  Display Matrix

       Description:  This method prints to cerr the matrices in this object.

*/

void OUsXfrmObjs::displayMatrix() {

  int i, j;

  cerr << "Marix trans_to_world:" << endl;
  for(i = 0; i < 4; i++) {
    for(j = 0; j < 4; j++)
      cerr << *trans_to_world[i][j] << "\t";
    cerr << endl;
  }

  cerr << "Marix trans_from_world:" << endl;
  for(i = 0; i < 4; i++) {
    for(j = 0; j < 4; j++)
      cerr << *trans_from_world[i][j] << "\t";
    cerr << endl;
  }
}

/******************************************************************************

  Filename:  OUsXfrmObjects.h
  Revision:  2.00
```

```
  Date:     04 Dec 97
  Author:   Jeff Collier

  Description:  This class is a foundational class for all basic
                elements used in the simulation of a structured light
                scanner.

                This object contains transformation matrices to and
                from object space to world space.

*/

#ifndef OU_XFRM
#define OU_XFRM

#include "OUsBase.h"

/******************************************************************************
Constant Definitions
******************************************************************************/
const SbMatrix empty_m(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0);


/******************************************************************************

  Class:  OUsXfrmObjs

  Description:  This is an abstract class to give all inherited classes
    the ability to keep track of world space and object space.

  Superclass:  OUsBase

  Inherited
  Methods:      SoSeparator *ReadFile(const char *filename)
                int writeFullGraph(char *filename = "graph.iv")

*/
class OUsXfrmObjs : public OUsBase {

public:
  OUsXfrmObjs();
  ~OUsXfrmObjs();

  int copyMatrix(const SbMatrix *to_world, const SbMatrix *from_world);
  int linkMatrix(SbMatrix *to_world, SbMatrix *from_world);
  SbMatrix *getTransToWorld() {return trans_to_world;}
  SbMatrix *getTransFromWorld() {return trans_from_world;}
  void displayMatrix();

  //Perform Transforms
  SoCoordinate3 &transFromWorld(SoCoordinate3 *points);
```

```
  SbVec3f &transFromWorld(SbVec3f &point);
  SoCoordinate3 &transToWorld(SoCoordinate3 *points);
  SbVec3f &transToWorld(SbVec3f &point);

private:
  //Matrices to keep a translation to and from worldspace.
  SbMatrix *trans_to_world;
  SbMatrix *trans_from_world;
};

#endif
```

Appendix B

A camera's calibration will be a major factor in the quality of a laser triangulation system. Among the most important parameters are baseline distance (the distance between the camera and the laser plane) and the camera angle (the angle between the optical axis of the camera and the laser plane). This appendix will define three sets of scan heads based on the camera's parameters. The three sets will be called type A, type B and type C. A projection grid of each camera is provided as an indication of the resolution of the scan head.

Table 6: Three different camera calibrations

Calibration Parameter	Type A	Type B	Type C
baseline	0.25 meters	0.6 meters	0.1 meters
camera angle	45 degrees	60 degrees	25 degrees
Cx, Cy	256, 256	256, 128	256, 128
dx, dy	10 μm, 10 μm	10 μm, 10 μm	10 μm, 10 μm
Ncx	512	512	512
Nry	512	256	256
Nfx	512	512	512
Sx	1	0.5	0.5
κ	0	0	0
fl	3.5 mm	1.4 mm	1.4 mm

As seen in the table the only difference between a type B scan head and a type C scan head is the baseline distance and the camera angle. However a there is a substantial

difference in the resolution of the scan head. The figures below are the projection grids of the three types of scan heads.

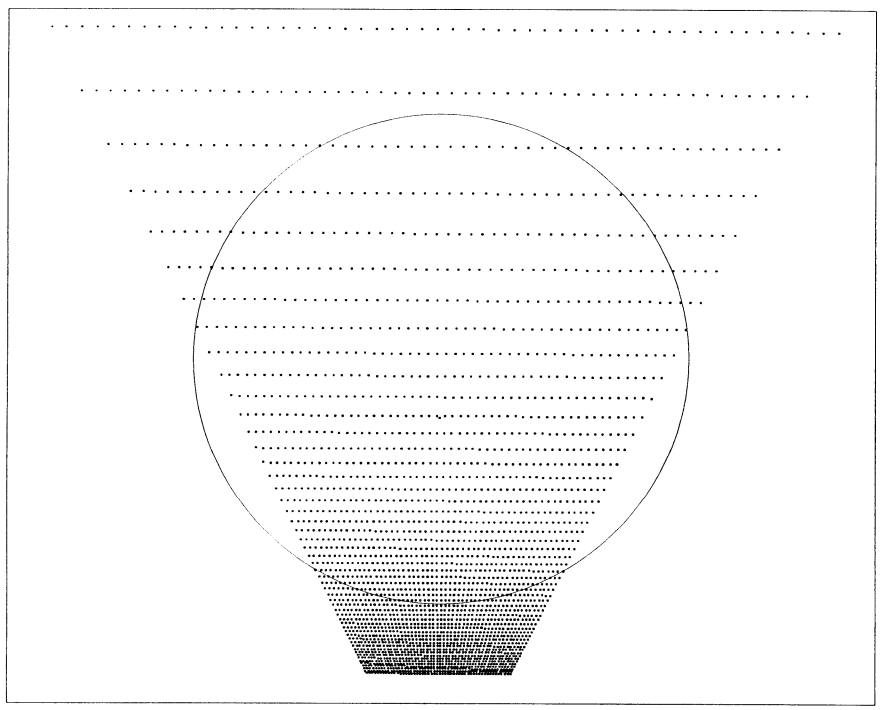

Figure 39: Projection grid for a type A camera.

Figure 39 is a projection grid for a type A camera. The circle represents the desired scanning space. While the resolution is good, parts of the scanning space are missed.

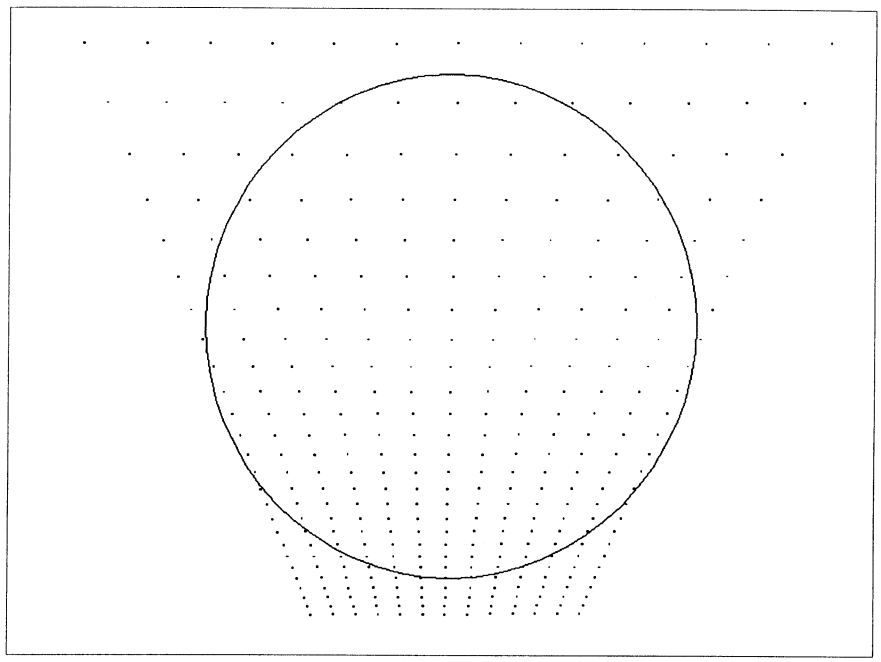

Figure 40: Projection grid for a type B camera.

In figure 40 the resolution of the camera has been reduced. However the longer base line improves the coverage of the scanned area. In addition there is less variation in resolution throughout the scanning space.

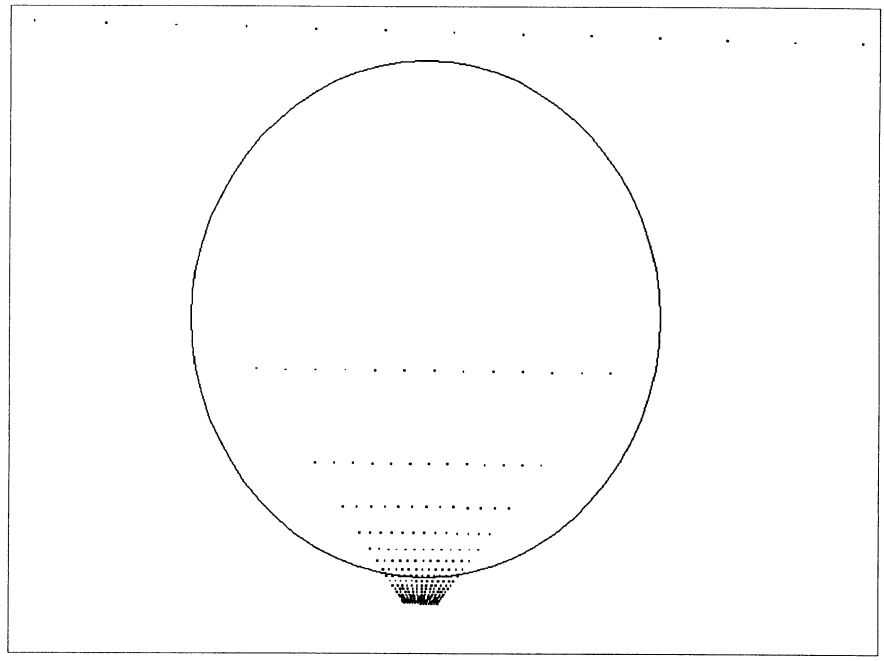

Figure 41: Projection grid for a type C camera.

In figure 41 the baseline has been reduced. The figure shows that the coverage of the scanner is not as good as a type A or type B scan head.