Multisensor Knowledge Systems: Interpreting 3-D Structure

T. Henderson, A. Mitiche, E. Weitz and C. Hansen

UUCS-87-028

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

November 19, 1987

Abstract

We describe an approach which facilitates and makes explicit the organization of the knowledge necessary to map multisensor system requirements onto an appropriate assembly of algorithms, processors, sensors, and actuators. We have previously introduced the Multisensor Kernel System and Logical Sensor Specifications as a means for high-level specification of multisensor systems. The main goals of such a characterization are: to develop a coherent treatment of multisensor information, to allow system reconfiguration for both fault tolerance and dynamic response to environmental conditions, and to permit the explicit description of control.

In this paper we show how Logical Sensors can be incorporated into an object-based approach for the interpretation of 3-D structure. Considering the inherent difficulties in interpreting general configurations of lines in space, and considering the ubiquitousness of special line configurations in man-made environments and objects, we advocate the use of computational units tuned to the occurrence of special configurations. The organized use of these units circumvents the inherent difficulties in interpreting general configurations of lines. After a brief examination of the problem of interpreting general configurations of lines in space, a number of computational units are proposed which are naturally derived from angular relations. The process of propagation (which allows interpretation to spread over the image) is also advocated. Such computational units and processes, which are simple and efficient, can be conveniently organized in a rule-based framework where the occurrence of the various special configurations can be tested. The Multisensor Knowledge System provides such a framework.

1 Introduction

The rapid design of embedded electromechanical systems is crucial to success in manufacturing and defense applications. In order to achieve such a goal, it is necessary to develop design environments for the specification, simulation, construction and validation of multisensor systems. Designing and prototyping such complex systems involves integrating mechanical parts, software, electronic hardware, sensors and actuators. Design of each of these kinds of components requires appropriate insight and knowledge. This in turn has given rise to special computer-based design tools in each of these domains. Such Computer Aided Design (CAD) systems have greatly amplified the power and range of the human designer. To date, however, it is still extremely difficult to address overall system issues concerning how the components fit together, and how the complete system will perform.

It is crucial to develop a design environment in which these multiple facets of system design can take place in a coordinated way such that the description of one component can be easily interfaced to another component, even when they are radically different kinds of things (e.g., a control algorithm, a mechanical linkage and an actuator). The designer should have the freedom to try out ideas at different levels of detail; i.e., from the level of a sketch to a fully detailed design. The Multisensor Knowledge System provides part of the solution to developing such an environment.

Logical Sensor Specifications (LSS) were developed previously as a method to permit an implementation independent description of the required sensors and algorithms in a multisensor system. Figure 1 gives a pictorial description of the basic unit: a *logical sensor.* Sensor data flows up through the currently executing program (one of $program_1$ to $program_n$) whose output is characterized by the *characteristic output vector.* Control commands are accepted by the *control command interpreter* which then issues the appropriate control commands to the Logical Sensors currently providing input to the selected program. The programs 1 through n provide alternative ways of producing the same characteristic output vector for the logical sensor. The role of the *selector* is to monitor the data produced by the currently selected program and the control commands. If failure of the program or a lower level input logical sensor is detected, the selector must undertake the appropriate error recovery mechanism and choose an alternative method (if possible) to produce the characteristic output vector. In addition, the selector must determine if the control commands require the execution of a different program to compute the characteristic output vector (i.e., whether dynamic reconfiguration is necessary).

Logical Sensor Specifications are useful then for any system composed of several sensors, where sensor reconfiguration is required, or where sensors must be actively controlled. The principle motivations for Logical Sensor Specifications are the emergence of significant multisensor and dynamically controlled systems, the benefits of data abstraction, and the availability of smart sensors.

In previous papers we have explored several issues of multisensor integration in the context of Logical Sensor Specifications:

fault tolerance[12],

functional (or applicative) style programming[24],

features and their propagation through a network[23],

the specification of distributed sensing and control[8,9],

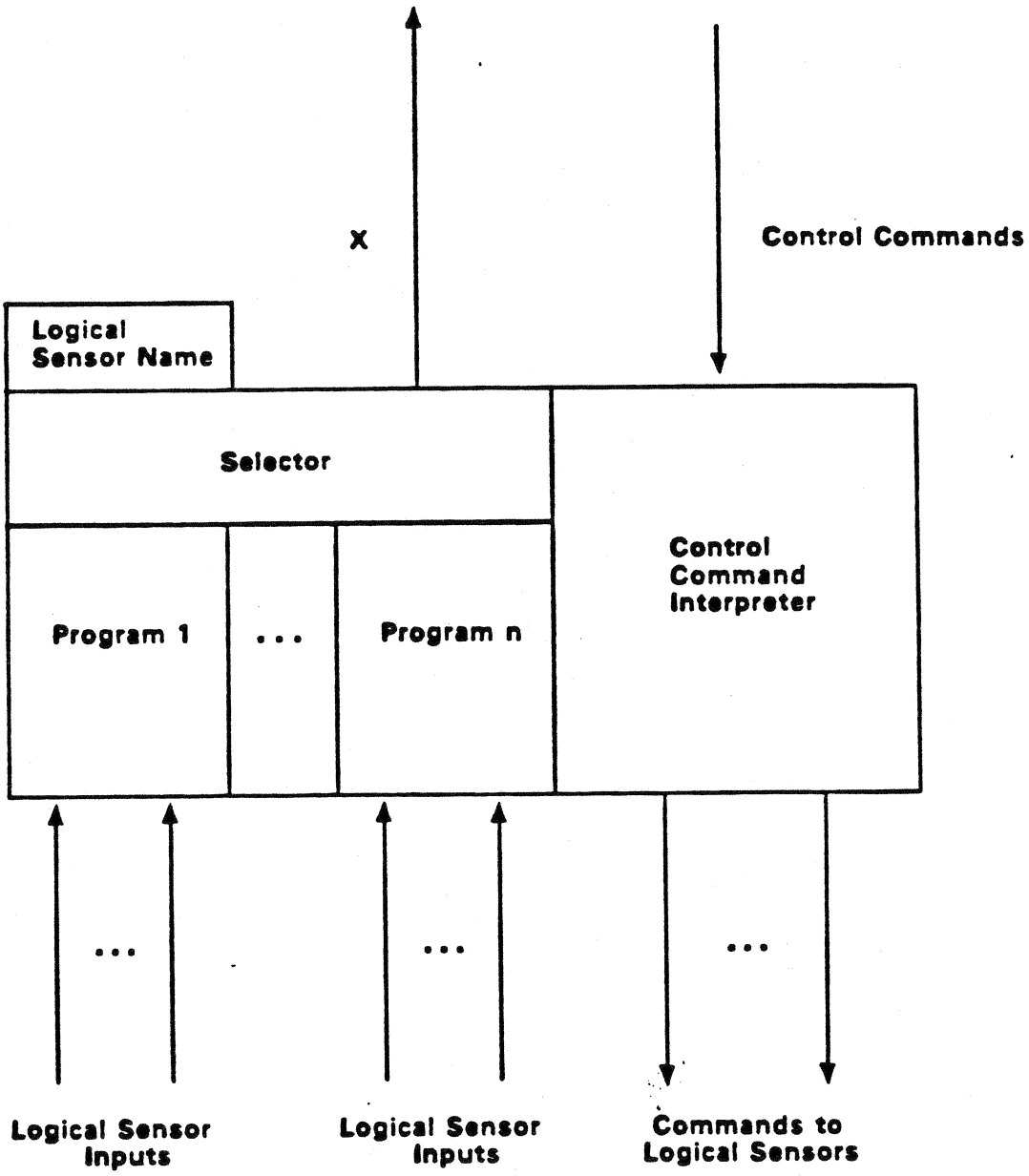

Figure 1: The Logical Sensor

the automatic synthesis of Logical Sensor Specifications for CAD/CAM applications[10,11].

Related work includes that of Albus[1] on hierarchical control, Bajcsy et al.[2] on the Graphical Image Processing Language, Overton[20] on schemas, and Chiu[3] on functional language and multiprocessor implementations. For an overview of multisensor integration, see[13,15].

In exploring these issues, we have found that the specification of multisensor systems involves more than just sensor features. It is true that knowledge must be available concerning sensors, but it is essential to also be able to describe algorithms which use the sensor data and the hardware on which they are executed. In the rest of the paper, we describe the components of an object-based approach to developing a knowledge system to support these requirements.

2 Multisensor Knowledge Systems

Much of our previous work on multisensor systems has concentrated on the specification of such systems and reasoning about their properties. It is necessary to be able to describe both the parameters and characteristics of individual components of multisensor systems, and to be able to deduce global properties of complete systems. Although it may be possible to deduce such properties (especially static properties like complexity, data type coercion, etc.), we believe that many interesting properties can only be determined by simulating the operation of the complete system.

Thus, we seek a representation that supports:

1. **multisensor system specification**: this describes the components and interconnection scheme of the particular system being designed,

2. **sensor, algorithm, processor and actuator knowledge representation**: this structures information about sensor characteristics (e.g., accuracy, hysteresis, dynamic range, etc.), algorithms (e.g., space and time complexity, amenity to parallel computation, stability, etc.) processors (e.g., cycle times, memory limits, address space, power requirements, etc.), and actuators (e.g., actuation principle, power requirements, etc.), and

3. **multisensor system simulation**: this permits one to monitor important parameters and to evaluate system performance.

Figure 2 shows the organization of the three capabilities within an object-oriented context. In the following subsections, we describe the Multisensor Knowledge System (*MKS*), an object-based approach to providing a unified answer to these three capabilities[6].

2.1 The Multisensor Knowledge Base

The multisensor knowledge base serves two main purposes:

1. to describe the properties of the system components (e.g., sensors, algorithms, actuators and processors), and

2. to provide class descriptions for the actual devices which are interconnected in any particular logical sensor specification.

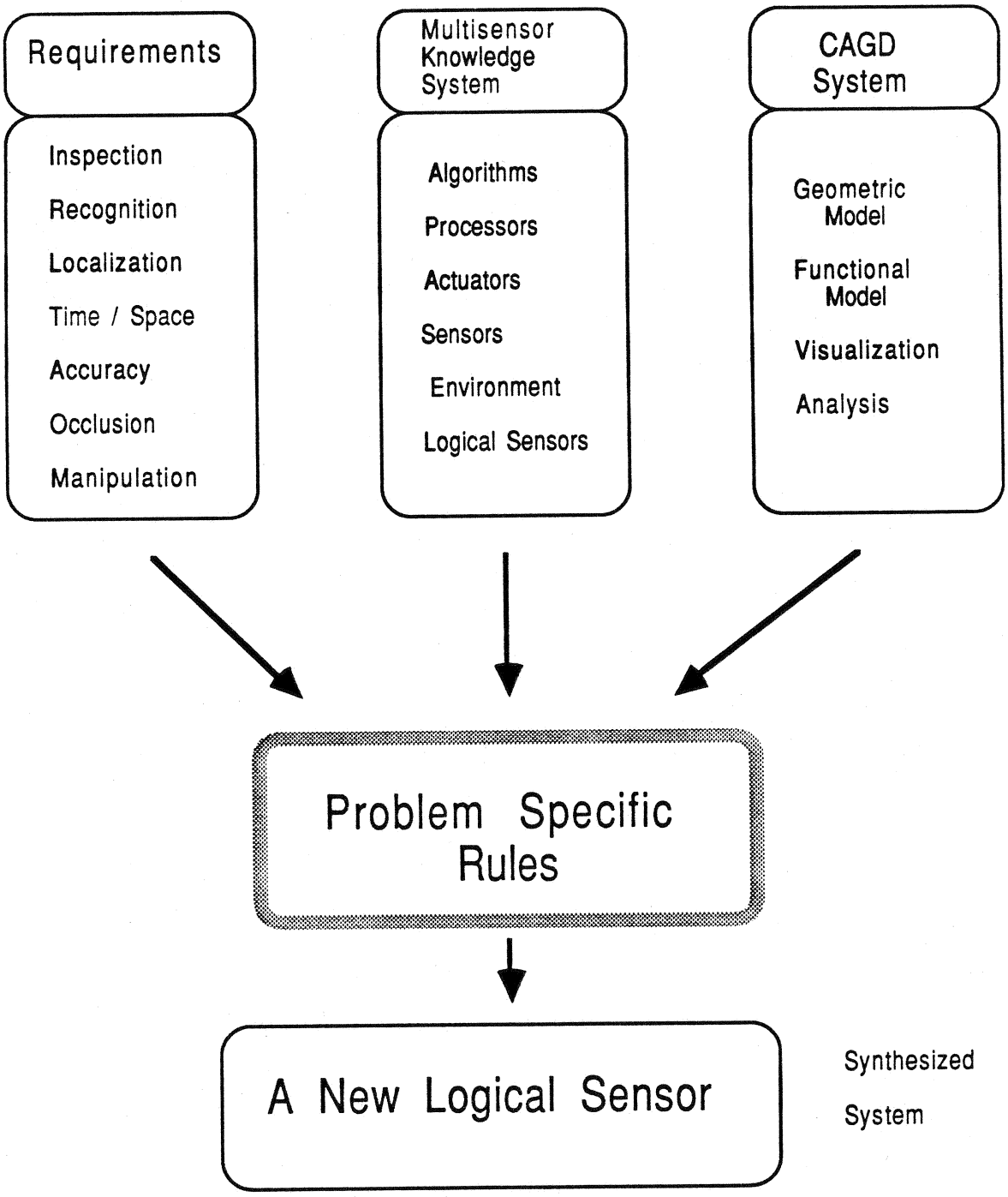

Figure 2: Multisensor Knowledge System

That is, the knowledge base must describe not only generic sensors (e.g., cameras), but specific sensors (e.g., Fairchild 9000, Serial No. 28753). It is then possible to reason about sensor systems at several levels. Moreover, it is possible that two distinct specifications require some of the same physical sensors. In such a case, it is the responsibility of the execution environment to resolve resource allocation conflicts.

We have chosen a frame-like knowledge representation. Frames relate very naturally to object-based descriptions, and, in fact, can be viewed as a class of restricted objects. It is straightforward to provide hierarchical descriptions of system components. For example, consider CCD cameras. The *CCD Camera* frame has two slots: element spacing and aspect ratio. These slots are specific to CCD cameras and as such do not appear as slots for *2-D cameras.* These latter have slots for scanning format, scan timing, resolution, output signal, and operating conditions. These slots are inherited by any instance of CCD camera. One level up, we find a frame for *Vision* sensors. This frame has specific slots for the spectral band and for the output type (e.g., 2-D byte array, multi-band, etc.). At the highest level of the hierarchy is the *Sensor* frame which has a slot for the physics of operation. This slot is used by any particular sensor to allow for an explanation of the physics underlying the workings of the sensor. In this way, it is possible to reason about the sensor.

Note that frames are themselves implemented as objects. Thus, actual devices are instances of some class of objects. This is very concise and conveniently exploits the similarities of frames and objects.

In previous work, we have described a set of generally applicable physical sensor features[7]. The manner in which physical sensors convert physical properties to some alternative form, i.e., their transducer performance, can be characterized by: error, accuracy, repeatability, drift, resolution, hysteresis, threshold, and range. These properties can be encoded in the appropriate slots in the frames describing the sensor.

2.2 Sensor Specification

An object-based style of programming requires that the logical sensor of Figure 1 be re-described in terms of objects and methods. We shall next give the general flavor of this style, but it must be remembered that any particular sensor is actually an instance of some object class, and, in fact, inherits properties from many levels up.

Thus, in order to get data from a logical sensor, the *characteristic output vector method* must be invoked. Likewise, to issue control commands to the sensor (e.g., camera pan and tilt parameters), the *control commands method* must be used. The role of the *selector* is still the same as in previous logical sensor implementations, however, it now, in essence, is invoked to produce the characteristic output vector.

Such a representation makes it very easy to design sensor systems. Moreover, such specifications allow for replacement of sensors and dynamic reconfiguration by simply having the *selector* send messages to different objects. Given current object-based programming technology, such systems can be rapidly developed and permit dynamic typechecking (on objects).

Figure 3 shows the Multisensor Knowledge Base, and below the dashed line, a set of particular instances of various algorithms, sensors, etc. (drawn as circles). A logical sensor specification (indicated as a blocked in subset of the circles) defines a grouping of algorithms, sensors, etc. This newly created logical sensor is an instance of the *logical sensor object* and can be sent messages. As mentioned above, there are two methods defined on logical sensors: the *characteristic output vector*

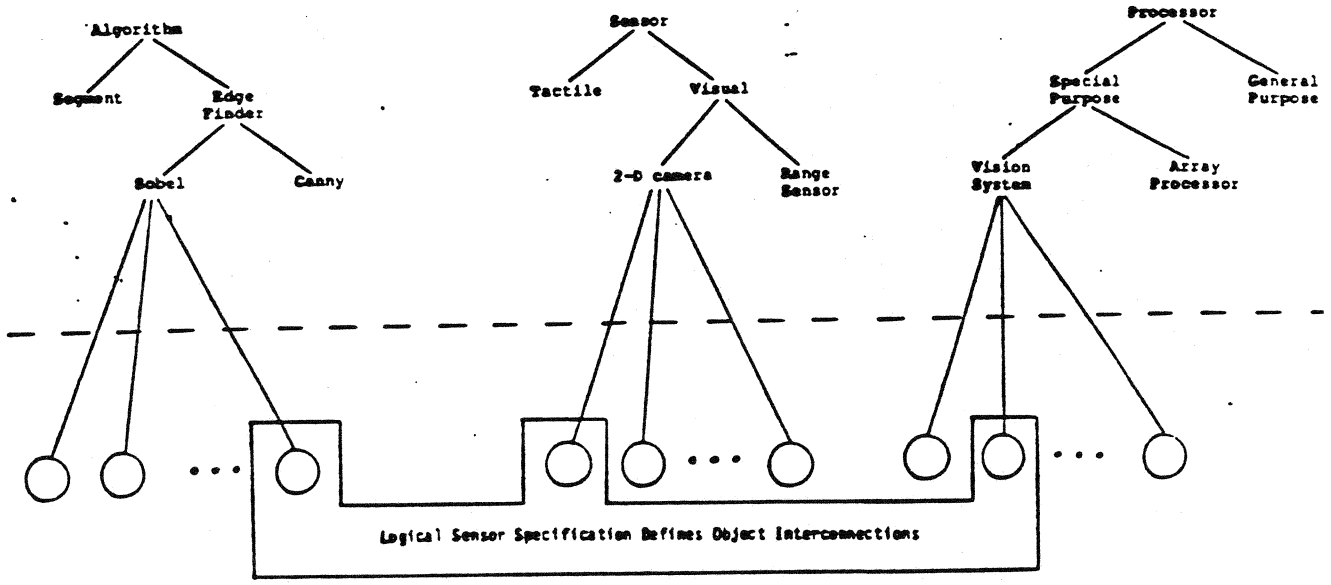

Figure 3: Logical Sensor Specification Using Object Instances

method and the *control commands method.* Thus, any logical sensor can be defined recursively in terms of other logical sensors (including itself).

3 Interpreting 3-D Structure

We now consider the application of the Multisensor Knowledge System to line-based interpretation of space. The use of lines for space interpretation is attractive because lines abound in man-made environments and it is often easier to extract their image than it is to extract a depth map or to compute optical flow.

There have been relatively few investigations involving line-based schemes[16,17,14]. These schemes require, in the general case, the observation of at least six distinct lines in at least three views, although a lesser number of observations is sufficient when special motions are involved (pure translations, pure rotations). Mitiche et al. proposed a rigidity-based approach which explicitly exploits the principle of angular invariance with respect to rigid motion[15]. Informally, the principle of angular invariance states that the angles between the lines of a set of lines in space do not change as a result of a rigid motion of this set of lines. The formulation is such that each pair of observed lines yields two equations, each containing the unknowns of the line's orientation and each line contributes one unknown per view. The resulting set of equations is solved for the orientation of the lines in space, from which the rotational parameters of the motions involved are recovered. The final step consists of computing, in a straightforward way, the translational parameters of the motion involved, and the position of the lines in space up to a scale factor.

Liu and Huang[14] suggested the intersecting planes formulation which translates directly the fact that a line in space is the result of three (projective) planes intersecting. In this formulation, each observed line yields one equation containing the six parameters of the rotational components of the motion involved. The resulting set of equations is solved for these parameters, from which the orientations of the observed lines in space are obtained. The final step, as for the angular invariance approach, consists of computing the translational parameters of the motions involved and the positions of the lines in space, up to a scale factor.

A number of problems are associated with the line-based schemes for solving the general problem:

1. A large number of observations must be made (at least six lines when three views are used). Also, establishing correspondence between three views for such a large number of lines can be a tough practical problem.

2. 'Desirable' input situations may be rare when a large number of line observations are required. A 'desirable' input situation is such that:

   (a) the projection of each line is as distinct as possible from one view to another (very different views), and

   (b) for each view, the projection of each line is as distinct as possible from the projection of other lines (very different lines).

3. A large number of required observations can exacerbate the problems usually associated with iterative schemes of computation (initial approximation, local minima, convergence).

However, two important observations can be made:

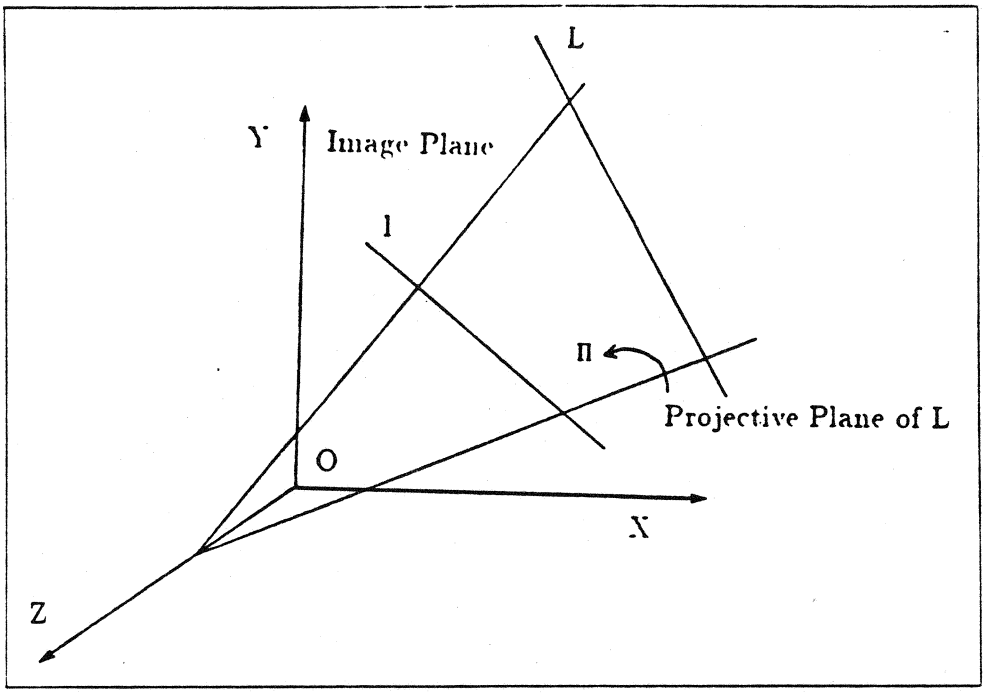

Figure 4: Viewing System Geometry

1. Special space configurations abound in man-made environments (for example, those confronted by a mobile robot). Commonly found special configurations invoving lines include known angular configurations in general, and parallel or perpendicular lines, in particular. In fact, it is difficult to observe a man-made environment where special configurations of this sort do not occur. Knowledge of the existence of such configurations simplifies dramatically the problem of 3-D interpretation. The usefulness of special, yet common, configurations has been recognized in the context of other studies (e.g., the use of planar surfaces, such as walls, in modeling a robot environment[4,5], or the use of 'base lines' such as the edge formed by the floor and a wall[25]).

2. Interpretations can be propagated. Indeed,when a part of space is interpreted, this can often be used to interpret other parts of space.

The exploitation of these points yields computational units (i.e., logical sensors) specialized in resolving particular, yet commonly found, configurations of space. These can be invoked when the existence of such configurations is either known or hypothesized. Propagation occurs when a part of space is interpreted. These computational units are being developed both in the context of the Multisensor Knowledge System at the University of Utah and 3D-Knobis (3D KNOwledge-Based Interpretation System) under development at INRS in Canada[17]. These logical sensors are designed in the spirit of dynamic sensing and provide the capability of recognizing or hypothesizing particular space configurations. A propagation module assesses the consistency of the resulting interpretations.

First, we present the scalar form of projective relations for lines. The viewing system is modeled as in Figure 4 (Cartesian reference system, central projection). Line L in space has projection l and (unit) orientation $\mathbf{A}(A_1, A_2, A_3)$. Then there exists a real number $\lambda > 0$ such that

$$A_1 = \frac{\lambda x_2 - x_1}{a}$$

$$A_2 = \frac{\lambda y_2 - y_1}{a} \qquad (1)$$
$$A_3 = \frac{(1-\lambda)f}{a}$$

where f is the focal length, (x_1, y_1) and (x_2, y_2) are *any* two distinct points on l and

$$a = \sqrt{(\lambda x_2 - x_1)^2 + (\lambda y_2 - y_1)^2 + (1-\lambda)^2 f^2}.$$

Equation (1) expresses the fact that line L is in (known) plane Π.

3.1 Angular Invariance: The General Case

Informally, the principle of angular invariance states that the angles between the lines of a set of lines in space do not change as a result of a rigid motion of this set of lines. In[16], lines were taken to be undirected, the assumption being that the cue of direction is absent in the image. For this reason, angular invariance for two (undirected) lines was translated by stating that the quantity $\cos^2\theta$ is preserved in rigid motion, θ being either of the two (supplementary) angles between the lines. However, since line correspondence is assumed known, one can also adopt the much stronger assumption of known correspondence between the directions of lines (i.e., if a line in one view is put in correspondence with a line in another view, then it is assumed that the direction of these lines can also be put in correspondence; most motions will allow this in practice). Therefore, angular invariance can be stated for directed lines as follows: for a rigid configuration of lines, the scalar product of the orientation of any two lines is preserved during motion. More precisely, for two views of $n \geq 2$ lines, the equations of angular invariance are:

$$\mathbf{A}^i \cdot \mathbf{A}^j = \mathbf{B}^i \cdot \mathbf{B}^j \qquad i = 1, \ldots, n \qquad (2)$$

where $\mathbf{A}^i$, $\mathbf{B}^i$ $(i = 1, \ldots, n)$ are defined as in Equation (1) above. This definition has the effect of simplifying the equations found in[16], their degree being reduced by a factor of two. This simplification leads to better results and much less sensitivity to the choice of initial approximation to the solution (the nonlinear equations are solved with an iterative algorithm). However, some of the problems mentioned above remain, indicating the need for a knowledge-based approach to 3-D space interpretation.

We have experimented with synthetic data as well as a real-world scene. Real scenes are the test for any algorithm, however, synthetic data allow fast and extensive testing. We first report results on synthetic data, then describe experiments with real data.

We simulated the use of a camera looking at a scene approximately $4m$ away. The scene is $3m$x$3m$x$1m$ as indicated in Figure 5. A large number of sets of six lines that cross the scene are generated randomly. Noisy lines are obtained by randomly moving two points on the image of each line in a 9x9 neighborhood. The inter-pixel distance for a central projection model of the camera is approximately $10^{-5}m$ as determined experimentally[22]. We used ZXSSQ from the IMSL library to solve the system of nonlinear angular invariance equations. The initial approximations to the unknowns (λ's) are all set to 1 arbitrarily (i.e., initially the lines are assumed parallel to the image plane). Computed (3-D) line orientations are compared with actual orientations by calculating the average error (over the six lines) in the angle between them.

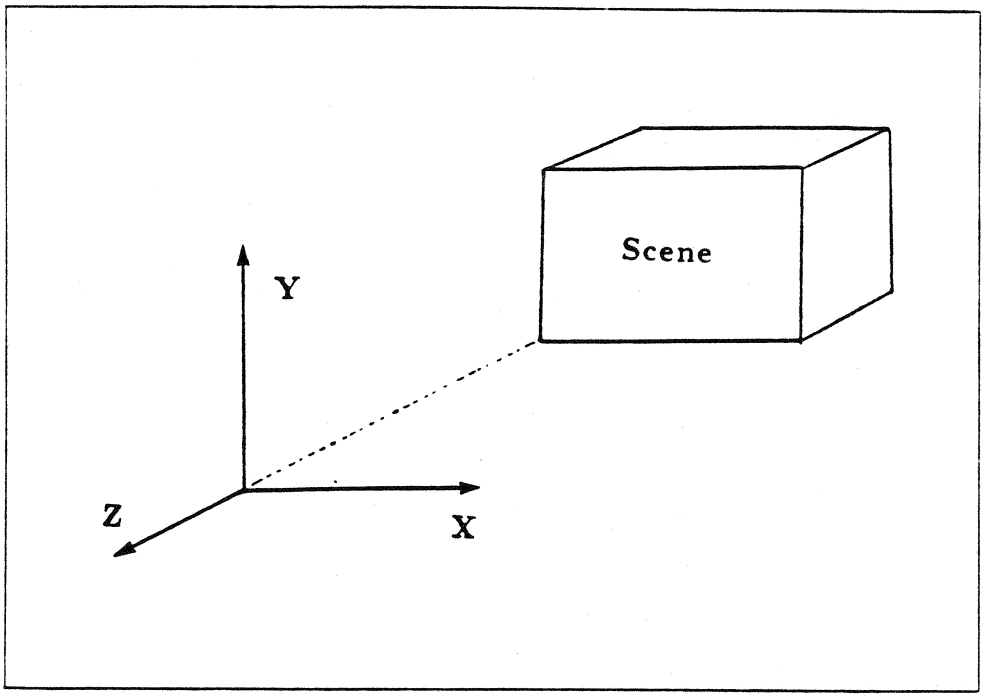

Figure 5: Reference System and Scene

In addition to noise, there are several factors which, in general, influence the results: relative position of cameras, position in space of the observed scene, and position and orientation of lines in the scene. We have experimented with some of these factors.

Figure 6 shows the X, Y, and Z distribution of error when the positions of the cameras differ by a translation of $2m$ along the X axes (camera 2 is $2m$ away from camera 1 in the X direction, camera 3 is $2m$ away from camera 2 in the X direction). When the amount of translation is reduced to $1m$, we note a certain decline in accuracy (see Figure 7). It is in general true (for any algorithm) that smaller displacements between the cameras are associated with less accuracy in the computed results.

An increase in accuracy is obtained, instead, when high-slant lines are not included in the experiments (see Figure 8). High-slant lines are, in general, associated with poorer accuracy.

Also, 'non-desirable' situations as discussed in the previous section (insufficiently different views, insufficiently different lines) cause poorer accuracy (for any algorithm).

In real scenes, one cannot rely on large displacements between views (as motions may not be controllable) or on the absence of 'non-desirable' situations (as a large number of views are required). This motivates the use of more accurate, specialized computational units that can resolve special, yet common, space configurations. A number of such units are now described.

3.2 Known Angular Configurations

We examine here the case of known angular configurations in general. Known angular configurations occur in a variety of man-made objects (e.g., architectural elements, familiar objects, CAD-modeled objects, etc.) of interest in important applications such as robotics (navigation or object recognition). An analogy can be made between the case of known angular configurations and the case of known point configurations used in successful experiments with videoconference images[22].

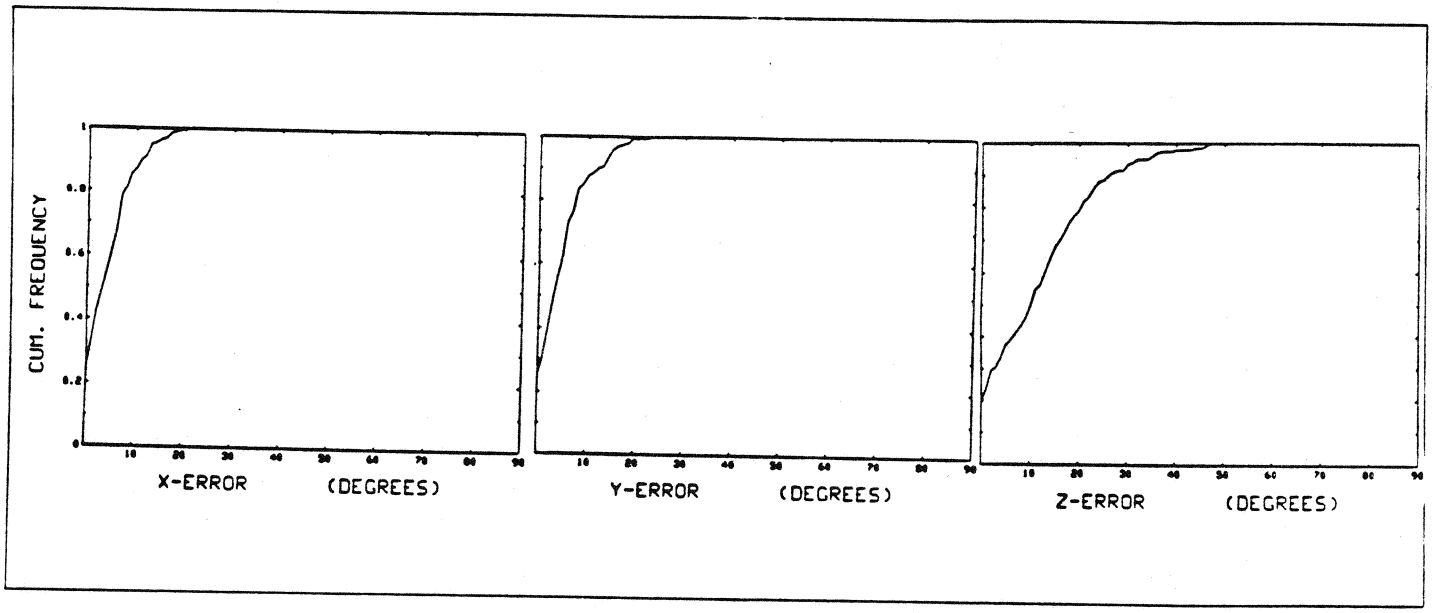

Figure 6: The General Case: $2m$ Translation

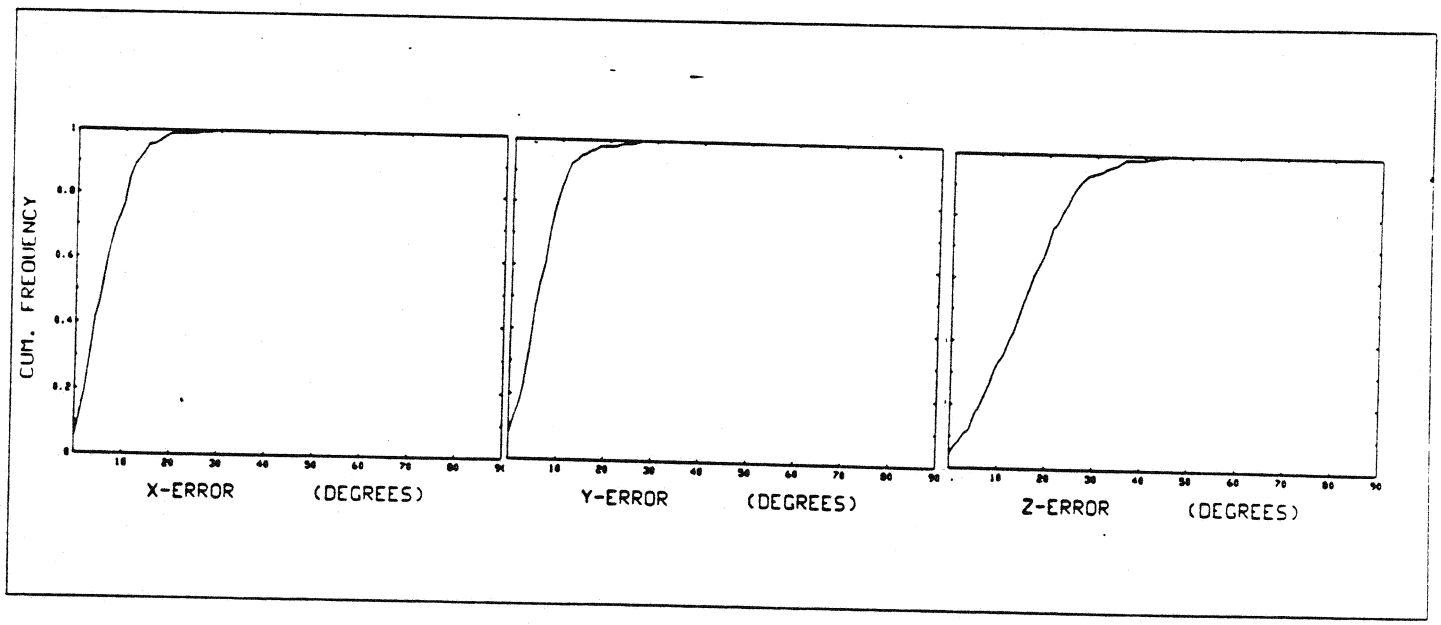

Figure 7: The General Case: $1m$ Translation

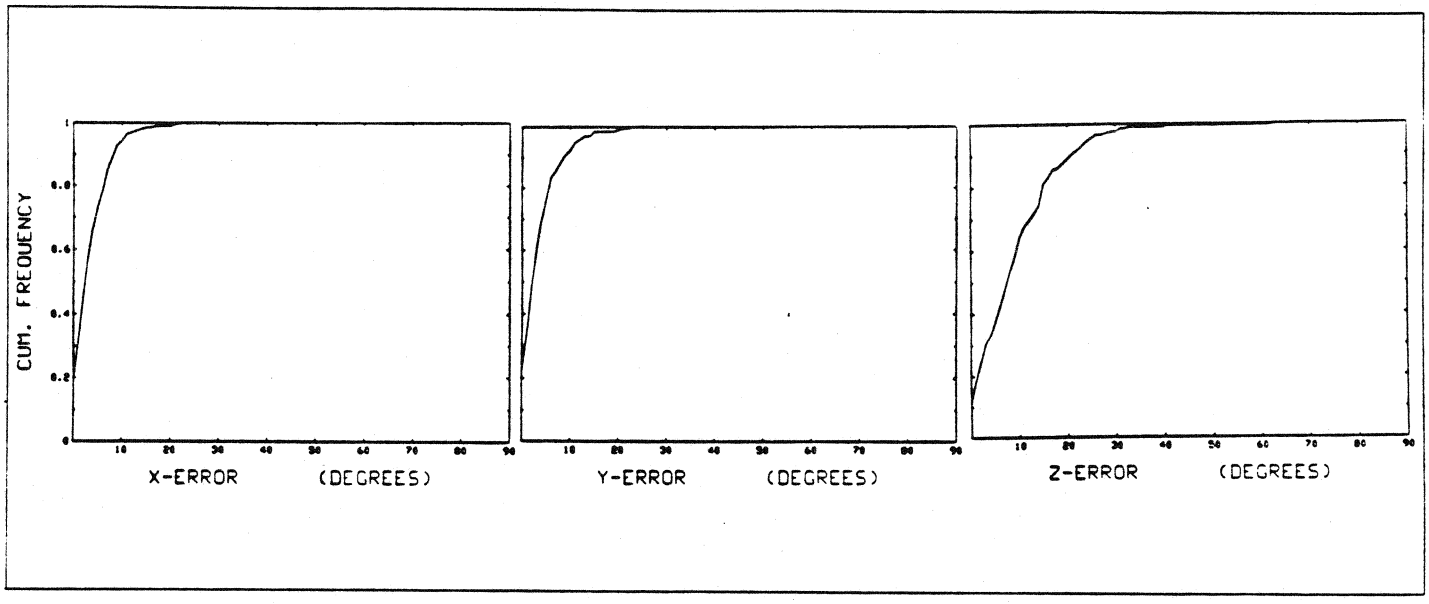

Figure 8: The General Case: No High-Slant Lines

With known angular configurations, we use only one view (in contrast with three views in the general case). The basic equation for line i and line j is:

$$\mathbf{A}^i \cdot \mathbf{A}^j = \cos(\theta_{ij}) \tag{3}$$

where $\mathbf{A}^i$, $\mathbf{A}^i$ are the orientations of lines i and j (their expression is as in Equation (1)) and θ_{ij} is the (known) angle between them.

Three lines will contribute three unknowns and three equations. Therefore, we need to use three or more lines. Results for three lines are shown in Figure 9. Results for three lines with high-slant lines eliminated are shown in Figure 10. With six lines, high-slant lines eliminated, we obtain the error curves of Figure 11.

3.3 Perpendicular Lines

Perpendicular lines are ubiquitous in man-made environments and objects as they occur at corners of walls-ceiling, walls-floor, corners of various objects such as tables, cabinets, and housing boxes of all sorts.

With perpendicular lines, only one view is necessary as in the case of known angular configurations of the previous section. The basic equation for lines i and j is:

$$\mathbf{A}^i \cdot \mathbf{A}^j = 0 \tag{4}$$

The expression of Equation (4) can be simplified by dropping the normalizing factors appearing in the denominator of the expressions $\mathbf{A}^i$ and $\mathbf{A}^j$ (see Equation(1)). In expanded form, the three equations for the three lines $i = 1, 2, 3$ is (superscripts in expressions below are not exponents, but indexes designating lines):

$$(\lambda^1 x_2^1 - x_1^1)(\lambda^2 x_2^2 - x_1^2) + (\lambda^1 y_2^1 - y_1^1)(\lambda^2 y_2^2 - y_1^2) + (1 - \lambda^1)(1 - \lambda^2) f^2 \quad = \quad 0$$

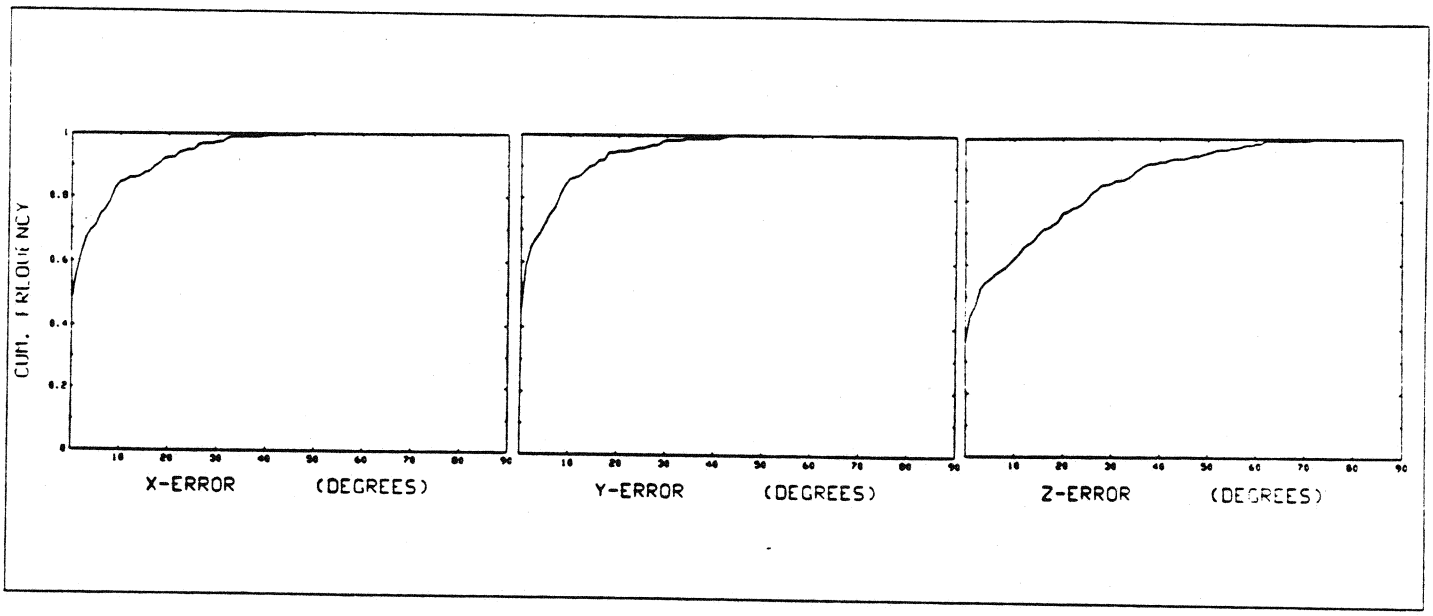

Figure 9: Known Angular Configuration: Three Lines

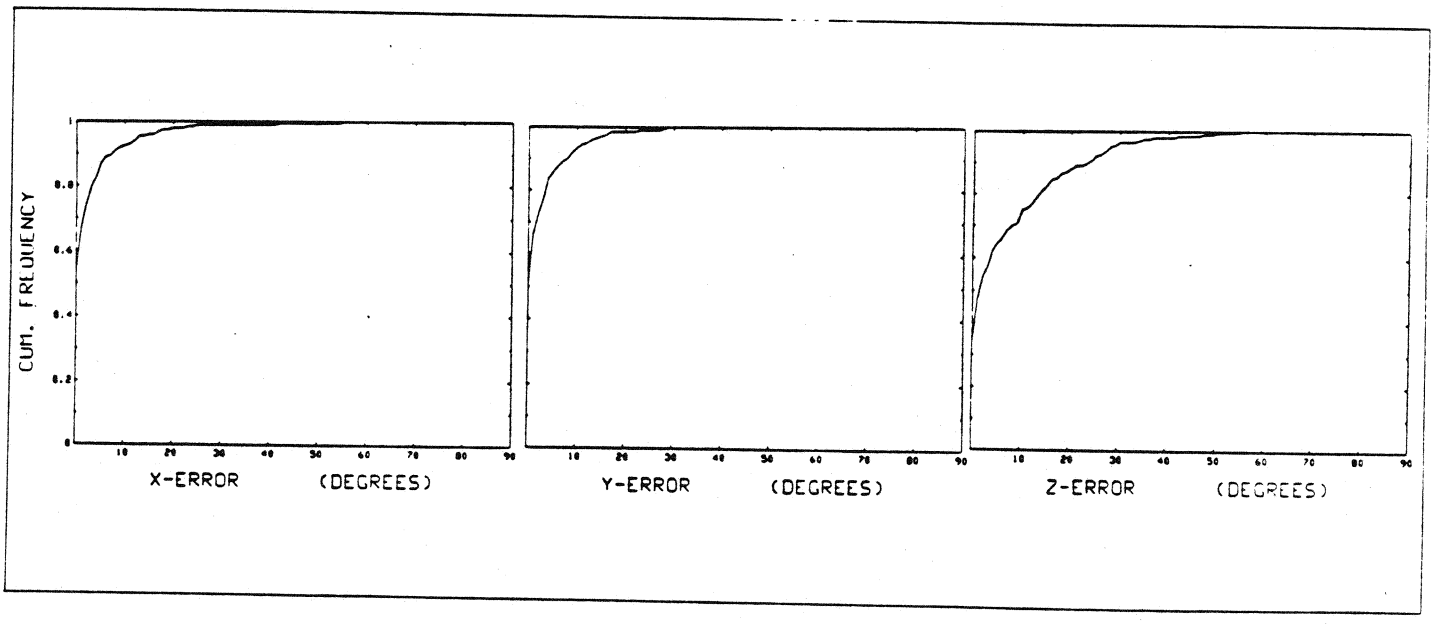

Figure 10: Known Angular Configuration: No High-Slant Lines

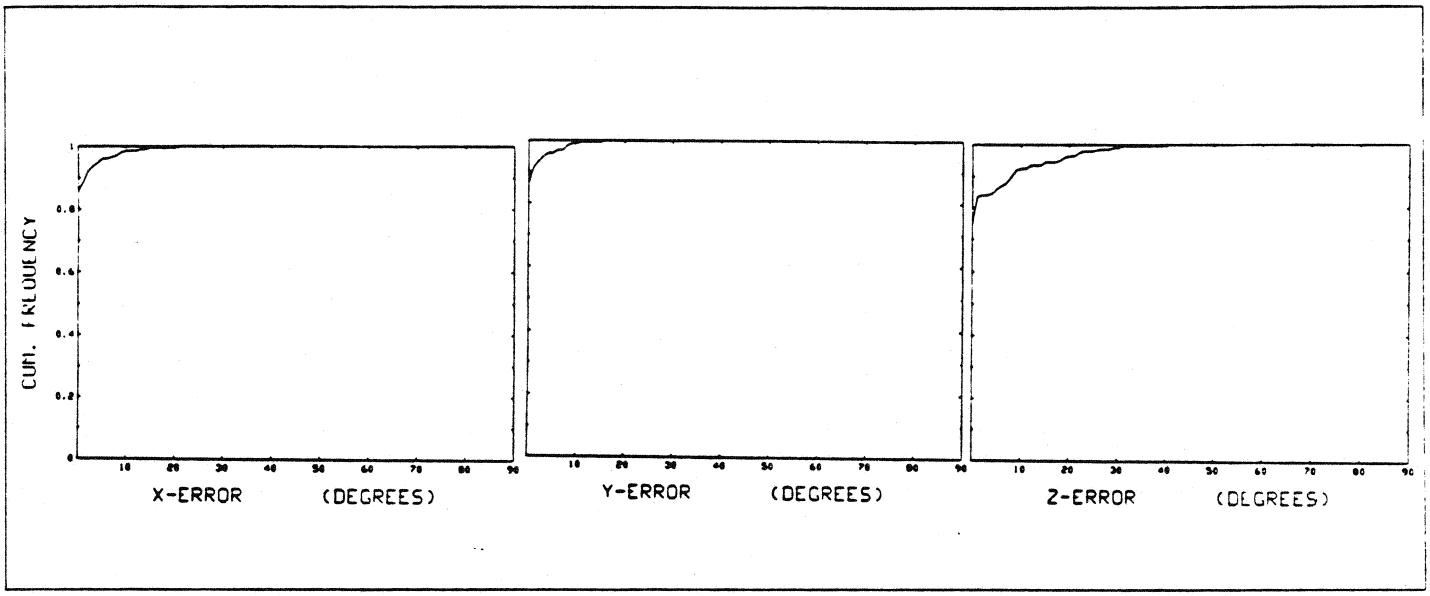

Figure 11: Known Angular Configuration: Six Lines

$$(\lambda^1 x_2^1 - x_1^1)(\lambda^3 x_2^3 - x_1^3) + (\lambda^1 y_2^1 - y_1^1)(\lambda^3 y_2^3 - y_1^3) + (1-\lambda^1)(1-\lambda^3) f^2 = 0 \quad (5)$$

$$(\lambda^2 x_2^2 - x_1^2)(\lambda^3 x_2^3 - x_1^3) + (\lambda^2 y_2^2 - y_1^2)(\lambda^2 y_2^3 - y_1^3) + (1-\lambda^2)(1-\lambda^3) f^2 = 0 \quad (6)$$

The system above can be solved analytically. If one expresses λ^2 in terms of λ^1 from the first equation of (5), expresses λ^3 in terms of λ^1 from the second equation, and then substitute the resulting expressions in the third equation, one obtains a second degree equation in λ^1. There are two solutions to this equation; there are therefore two solutions to our original problem; i.e., two possible interpretations of our perpendicular lines. In the absence of any other cue, these two interpretations are equally valid. To disambiguate the interpretation, one can use one additional line (or more) with known angular relation to the three lines (for instance, it is common to find a line parallel to one of the three perpendicular lines). In such a case, the additional line will yield three linear equations in one unknown (equations such as (5) with one of the variables already computed), for each possible solution. One can then select, for instance, the solution that yields the smaller least squares error when these three linear equations are solved. Another way to remove the ambiguity is to use a second view and propagation as explained earlier. Another useful heuristic is to impose bounds on the solution (λ's). Indeed, when the size of the objects appearing in the observed scene is within known bounds, the solution (λ's) is confined in a given interval (λ is related to the ratio of the depths of two points on the observed line). Results using one and two additional lines to disambiguate interpretation are given in Figures 12 and 13.

3.4 Parallel Lines

Parallel lines also abound in man-made environments. With a viewing system as in Figure 4, the direction of two (or more) parallel lines is the direction of the line through the center of projection and the intersection of the image of the lines (these images all intersect at the vanishing point or

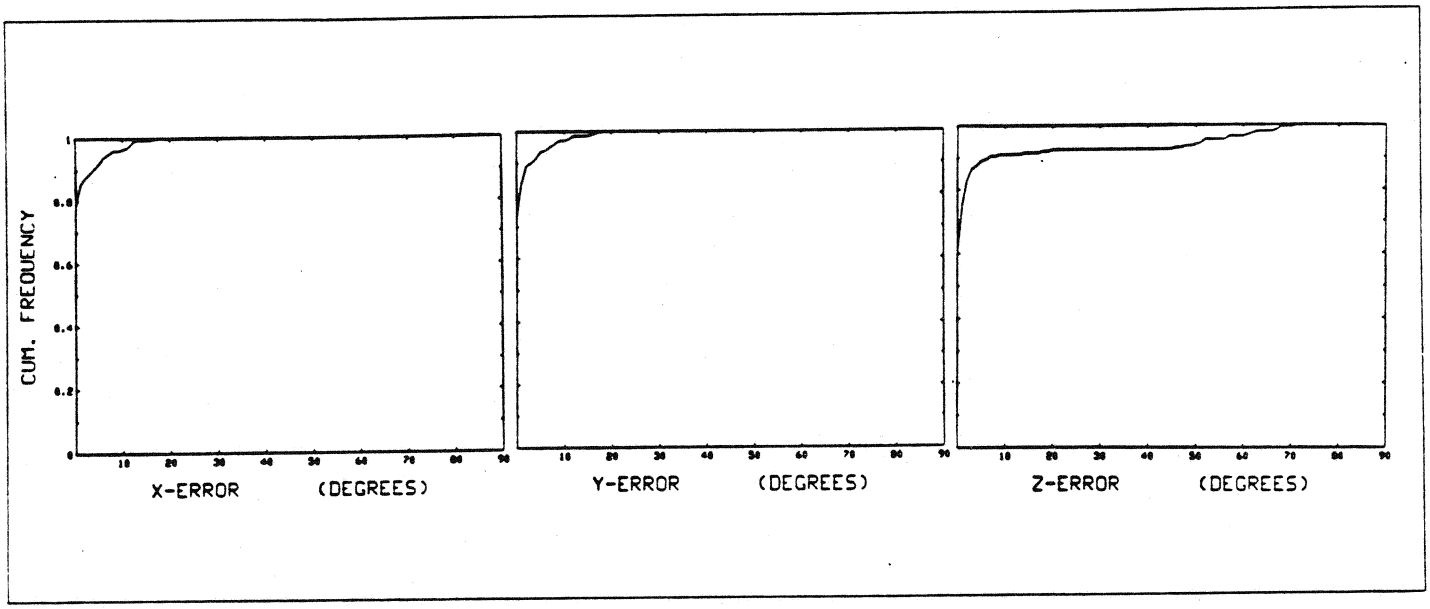

Figure 12: Perpendicular: One Additional Line

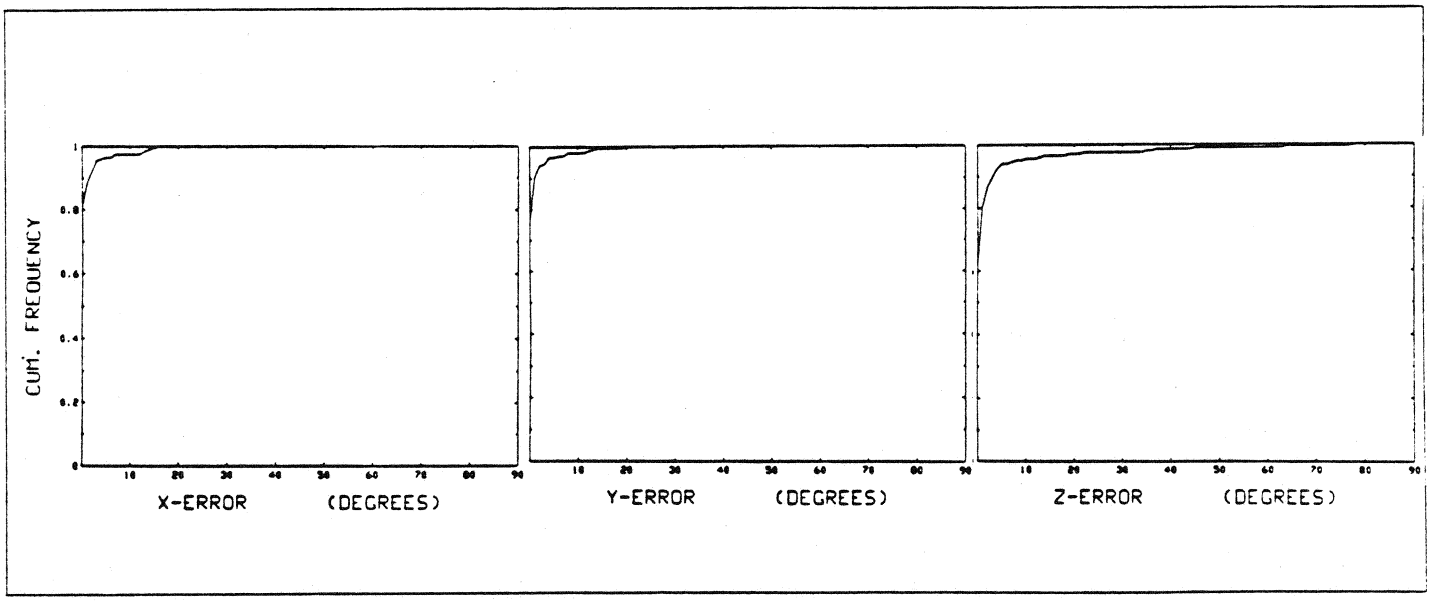

Figure 13: Perpendicular: Two Additional Lines

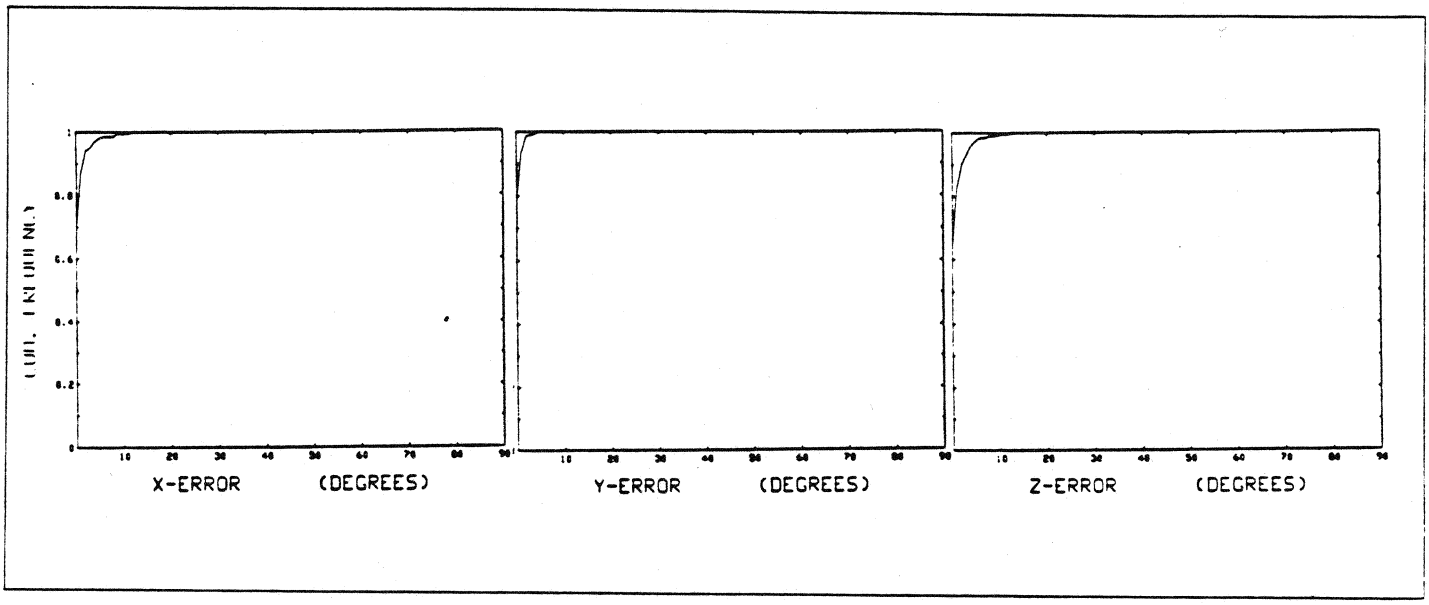

Figure 14: Parallel: Three Lines

focus of expansion; when the lines in space are parallel to the image plane, their orientation is that of their images). This direction can be computed quite robustly as is indicated in Figure 14.

3.5 Propagation

Propagation is the process by which interpretation achieved for part of space is used to achieve the interpretation of other (uninterpreted) parts of space. In the context of interpreting lines, when one has already determined the orientation of a number of lines (by any means), this knowledge can be used to determine in a straightforward way, and quite robustly in some cases, the orientation of other observed lines. We consider two cases:

1. **The interpreted part of space contains one line.** The angular invariance involving interpreted Line L^0 and uninterpreted Line L^i is:

$$\mathbf{A}^i \cdot \mathbf{A}^0 = \mathbf{B}^i \cdot \mathbf{B}^0 \tag{7}$$

   where $\mathbf{A}^i$, $\mathbf{B}^i$ are the orientations of L^i in the first and second view, respectively, and $\mathbf{A}^0$, $\mathbf{B}^0$ are those (known) of interpreted line L^0. With $m \geq 2$ views of $n \geq 2$ lines there are $(2n-3)(m-1)$ independent angular invariance equations. With 2 views, we have $2n-3$ equations; there would be $2n-2$ unknowns (we have n lines, one of which is known). Therefore, two views are not enough; we need to use at least three views. With three views, two uninterpreted lines and one interpreted line, there are four equations involving the known orientation (Equation (5)), and two equations involving only the uninterpreted lines, making a total of six equations. We also have six unknowns (one unknown per line per view). Therefore propagation can be applied given three (or more) views of one interpreted line and two (or more) uninterpreted lines.

2. **The interpreted part of space contains two or more lines.** Here, a count of unknowns and equations as above indicates that only two views are needed. Each uninterpreted line

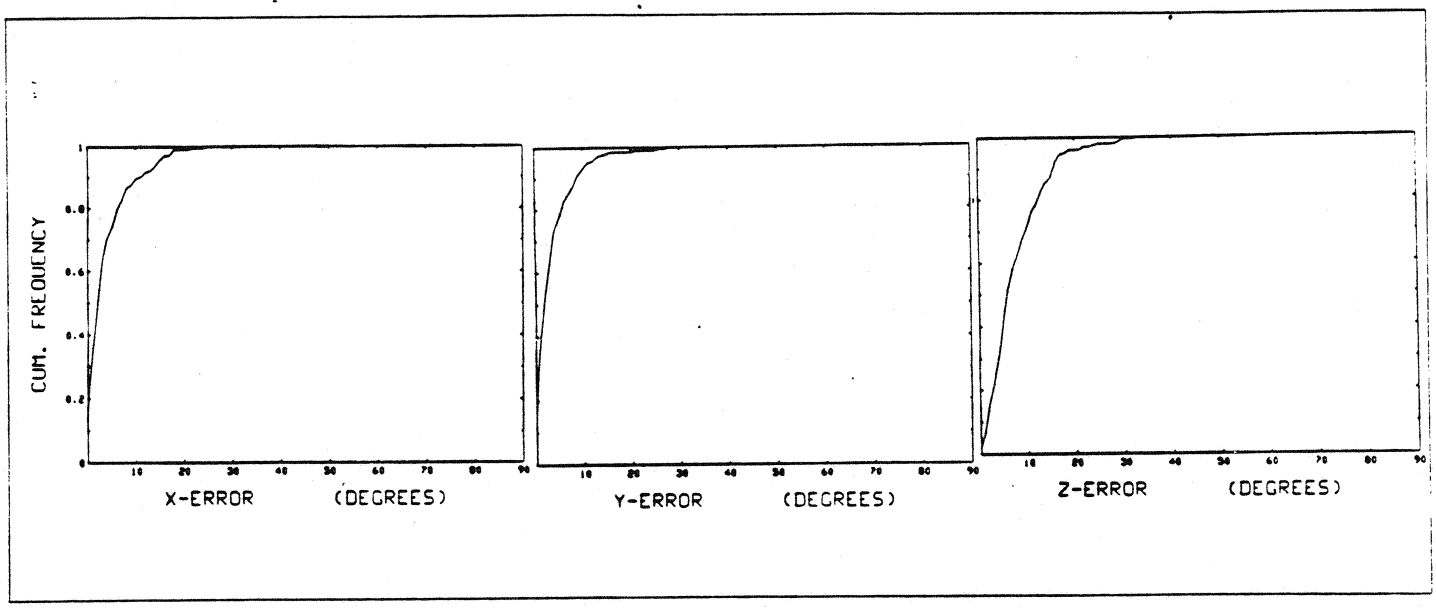

Figure 15: Propagation: From One Known Orientation

yields one equation (such as Equation (5)) for each interpreted line. With two or more interpreted lines, we have enough equations to solve for the orientation of the uninterpreted line. Experiments using one and two interpreted lines are illustrated in Figures 15 and 16, respectively.

We have mentioned that propagation can be used to disambiguate the case of perpendicular lines. Recall that we used one view for that case but had two possible solutions. If we have two views of three perpendicular lines, then given an additional line we can use propagation in each view to obtain two interpretations (for each view). Then angular invariance between the two views will disambiguate the interpretation as the actual line configuration will repeat from one view to the other, whereas, it is unlikely that the wrong interpretation does so.

4 Logical Sensors for Line Interpretation

4.1 Specifications

The Multisensor Knowledge System is defined in terms of FROBS (FRames + OBjectS), a Lisp package which supports both frames and objects [19]. It is a "best of both worlds" application, boasting the speed of Common Lisp objects and the benefits of frame world. FROBS is written in HP Common Lisp and also runs in PCLS [21]. FROBS come in two basic flavors, class FROBS and instance FROBS. Class FROBS serve as the templates from which multiple objects can be instantiated. They can inherit characteristics from one or more "parent" FROB classes. Figure 17 shows how an algorithm class FROB and a subclass FROB are defined.

The basic building block of the FROBS package is called a module. Modules consist of a class FROB and all of its associated methods. This provides for total method and data access hiding with no distinction between methods and slots. The organization of class FROBs can be viewed as a tree structure, although more complicated schema-type structures are possible through multiple

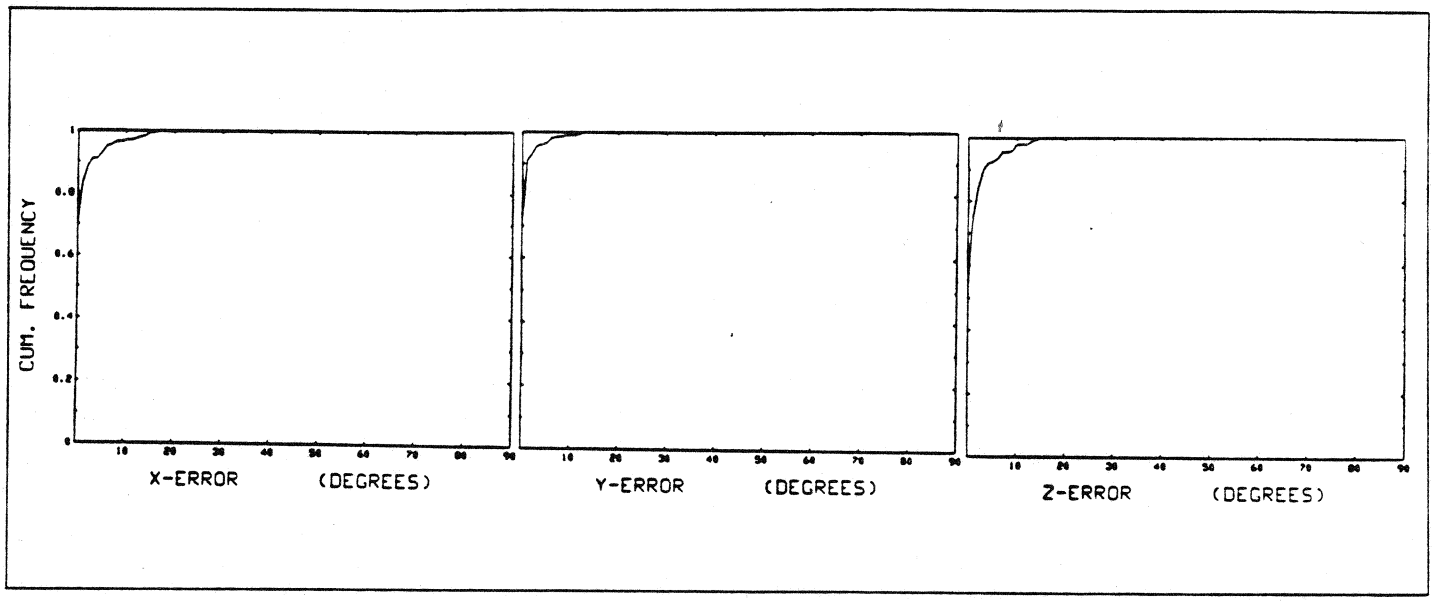

Figure 16: Propagation: From Two Known Orientations

```
(def-class algorithm nil
           :slots (name
                   size
                   language
                   machine
                   executable
                   args
                   input
                   input-type
                   output
                   output-type
                   e-function
                   machine-out
                   complexity
                   stability
                   robustness
                   p-factor))

(def-class feature-calculator ({class algorithm})
           :slots (feature-type
                   focus-type))
```

Figure 17: Example of FROB Class Definition

inheritance. FROB class instances are leaves of the tree. Figure 18 shows how FROB classes can be organized into a hierarchical structure with instances as leaves. FROBS are used to build both the knowledge-based vision system and the application system it synthesizes. This allows templates in the knowledge-based system to be directly used in the application system itself rather than performing a transformation to a target language as is done in a compiler. The concept of logical sensors is implemented easily using objects to form logical sensors. Class FROBS represent logical sensor templates to be instantiated for application system synthesis.

Most importantly the FROBS package provides forward chaining rules as well as slot daemons. Slot daemons are useful for automatic data consistency checking and hidden slot calculations. The forward chaining rules provide the mechanism needed to create the knowledge base.

4.1.1 System Support

The knowledge-based system must have utilities for supporting the networking of logical sensors and objects. These utilities provide the foundation from which the system is built. Higher level utilities are built on top of lower level ones for sophisticated system operations. The lowest level utility functions should have a maximal amount of flexibility since it is not known what or how more powerful constructs built upon them will be used. In the prototype system they are implemented as methods attached to FROB classes which define major components of the system. These classes and their methods form the templates from which application systems are synthesized. The application specific rules use knowledge of these templates to apply the line interpretation rules by relying on the transparent nature of the methods to handle lower level hardware or operating system specific tasks. An example is the FROB representing the class of cameras. Knowledge about operating this class of camera is represented in a "run" method which is local to the class. It executes operating system commands which are not of concern to the object using the method. A "run" method is also provided to other sensors which have other operating system commands which are transparent to the caller of the method.

4.1.2 Language Issues

Since the application system is created from FROBS in the same environment as the knowledge-based system, the application system runs in the Common Lisp environment. To require that all of the algorithms in the system be written in Common Lisp would be a severe restriction to its flexibility. The object-based approach allows algorithms written in any language to be incorporated into the system as an algorithm object. Methods are used to run the algorithm and provide it with the necessary I/O. Since the internal representation of the object is transparent to I/O from the outside, algorithms written in any language can be incorporated into the system as long as there are low level utilities in the system to support the methods which run them.

4.1.3 Object Communication Protocol

When designing a multisensor vision system using objects, there must be a well defined way for one object sensor to pass information to another. Logical sensors address this problem in an abstract sense, but a specific protocol must be chosen which has the flexibility to accept all kinds of data. The protocol is represented in the slots and methods of the logical sensor objects. There must be a way to pass information from machine to machine as well as an efficient way to pass information

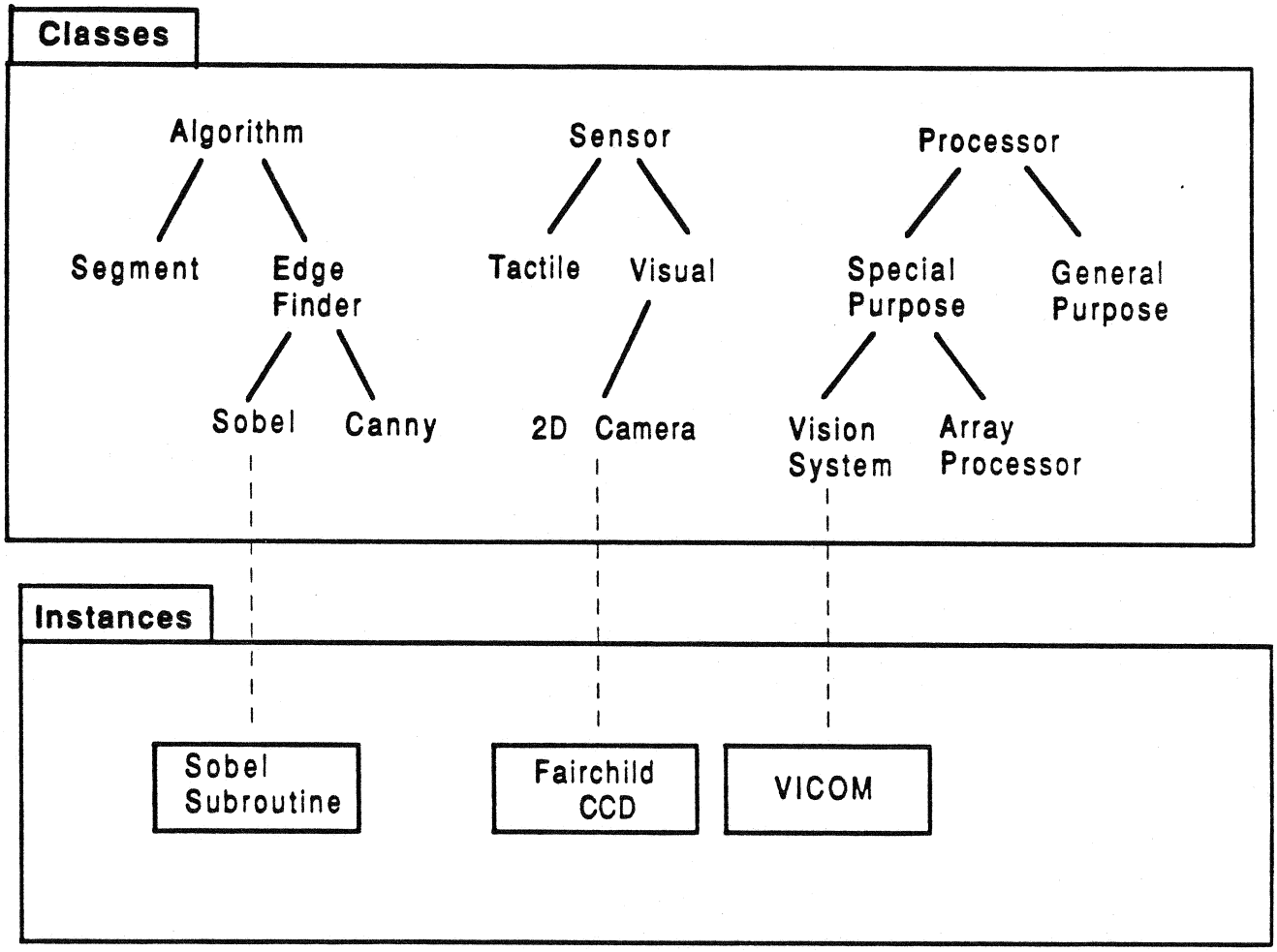

Figure 18: FROB Hierarchy

```
(def-rule select-lff
  :type ((?req requirements))
  :prem ((not (member 'lff (applications ?req)))
         (equal 'recognition (task ?req)))
  :conc ((assert-val ?req 'applications
                     (cons 'lff (applications ?req)))
 (make {class system-specs}
       :task 'recognizer
       :method 'lff
       :time (time ?req)
       :space (space ?req)
       :accuracy (accuracy ?req))))
```

Figure 19: A FROB Forward Chaining Rule

in the Lisp environment itself. We separate the two as different types of information passing, file piping and S-expression passing.

Passing S-expressions between objects is a trivial task. All that is required is a slot in the algorithm object which stores the expression to be passed. This slot is read by any object requiring the expression as input. To perform file piping on any host in the system, the simplest approach is to use the Unix pipe facility which allows executables to work as filters passing their output to the next program in the pipe. This is the easiest implementation since it is supported by the remote shell command "rsh" which is used to perform tasks on remote machines. It requires, however, that most programs written for the knowledge-based system be written in filter form on machine supporting Unix. This is not an unreasonable requirement since Unix is a widely supported operating system and it is good modular style to have a system designed with filters. Other programs can be run as well as filters although it is up to the user to supply names and flags in slots of the object which contains the program. Only filters are handled "automatically" by the piping method.

At some point in the multisensor system's operation, information from a remote machine will have to be read by an object in the Lisp environment or vice-versa. This requires some special processing on the part of the methods performing the pipe. To send the S-expression output of a Lisp algorithm to a non-Lisp algorithm, certain conventions must be adopted. The program receiving the S-expression must know that its input is in such a form. Each algorithm in the knowledge base must have information regarding what format it expects its input to be in and what format is produced as output. This is done with methods using slot information in the algorithm object. These methods determine what format conversions are necessary for information piped between algorithms. Information transfer between machines is performed when objects have slots indicating that their executables are on different machines. Figure 20 shows how a pair of objects in the multisensor application system use their methods and slot values to pipe input.

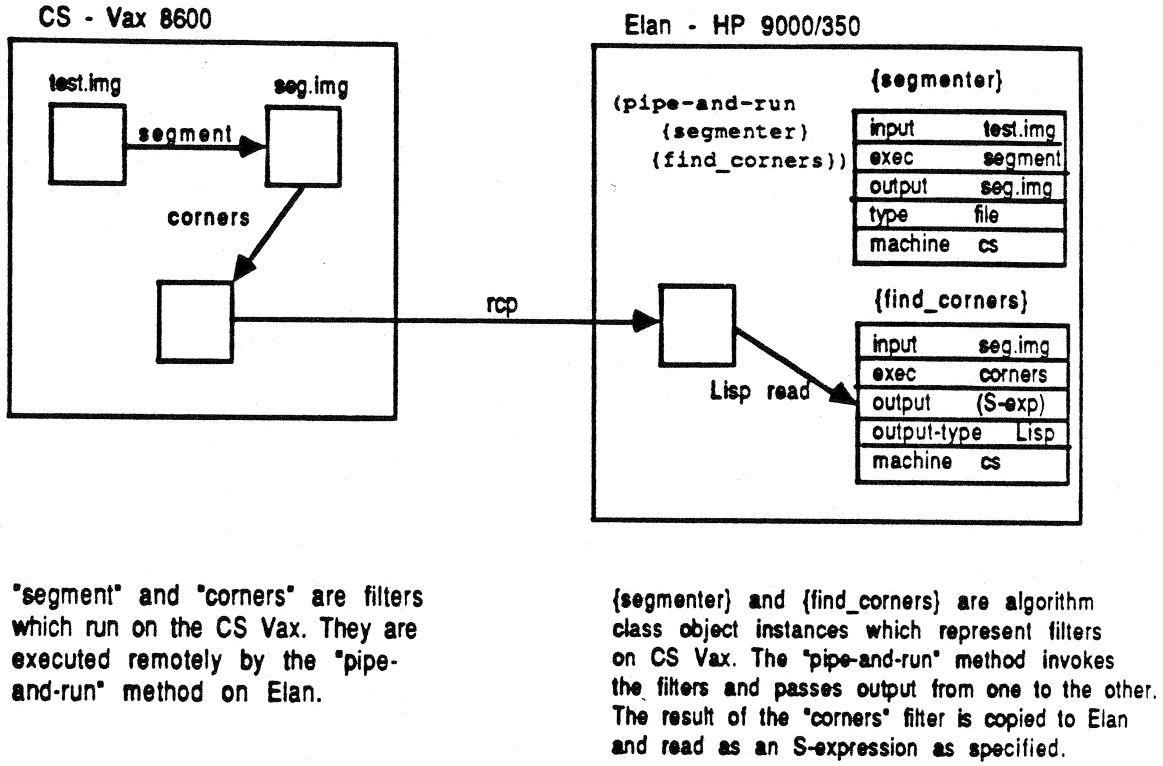

Figure 20: Piping of Information Between Objects

4.1.4 Line Interpretation FROBS and Rules

The line interpretation knowledge described earlier has been specified in MKS. Figure 21 shows the organization of the individual components. Detailed code is given in Appendix A.

Context variables are set when the rules fire or by known information. As processing takes place, new interpretations change the parameters, thus possibly firing new rules.

4.2 Experiments on Camera Data

We have run experiments with the *bookcase scene*, a view of which is shown in Figure 22. This observed scene is approximately three to four meters from the camera. The camera used is a portable HITACHI FP-15. It is equipped with a zoom which allows a focal variation between $14mm$ to $84mm$. Three views of the scene were taken. For convenient referencing, these are called the left view, the middle view and the right view (Figure 22 shows the middle view).

The left and right views were obtained by moving the camera from its position for the middle view approximately $1m$ to the left and $1m$ to the right, respectively, and then rotating by approximately 20 deg toward the scene (i.e., clockwise for left view and counterclockwise for right view) around the approximately vertical axis of the camera stand (the Y axis in our case). Figure 23 shows some of the lines considered. The lines have been numbered for easier reference.

The cases which required three views used all three views; the cases which required two views used the left view and the middle view; and finally, the cases which needed only one view used the left view. For each case, a number of experiments were run using various combinations of lines. The images were viewed on a TV monitor. The automatic computation of lines in the image being marginal to the main focus of the present study, the two (arbitrary) input points on each line were taken interactively. Also, no correction for geometric distortion of the images was attempted.

Table 1 summarizes the experiments. It indicates the average error in interpretation (average for each case over the several experiments for the case). The details of the experiment are given elsewhere[18]. The cases listed in the table are the following:

Case 1. Three views of six lines.

Case 2. Three lines of known angular configuration.

Case 3. Four lines of known angular configuration.

Case 4. Six lines of known angular configuration.

Case 5. Three perpendicular lines and one additional line to remove ambiguity.

Case 6. Three perpendicular lines and two additional lines to remove ambiguity.

Case 7. Three parallel lines.

Case 8. Propagation from one known orientation and two unknown orientations.

Case 9. Propagation from one known orientation and three unknown orientations.

Case 10. Propagation from two known orientations and two unknown orientations.

Overall, results indicate that specialized computational units perform well and that their organized use in a knowledge-based system is useful.

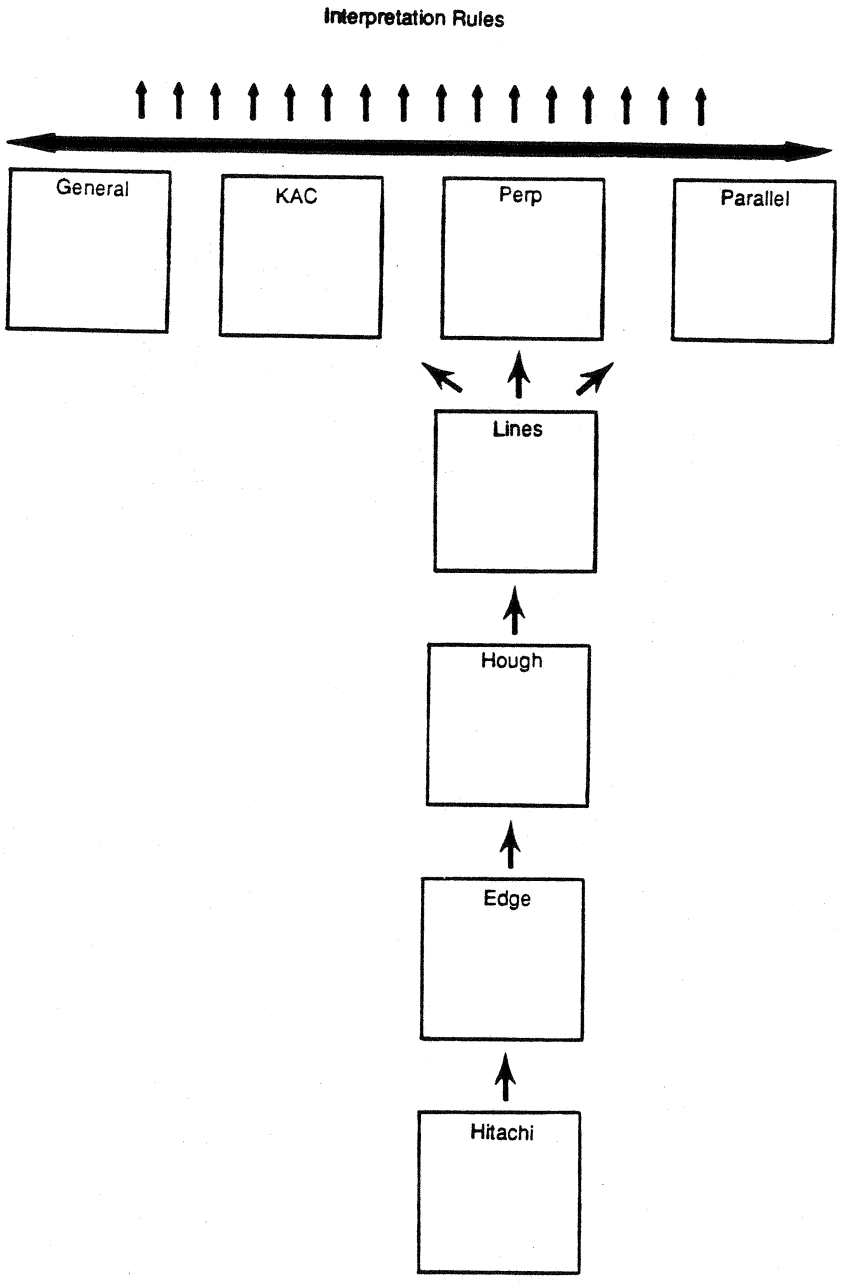

Figure 21: The Line Interpretation Logical Sensors

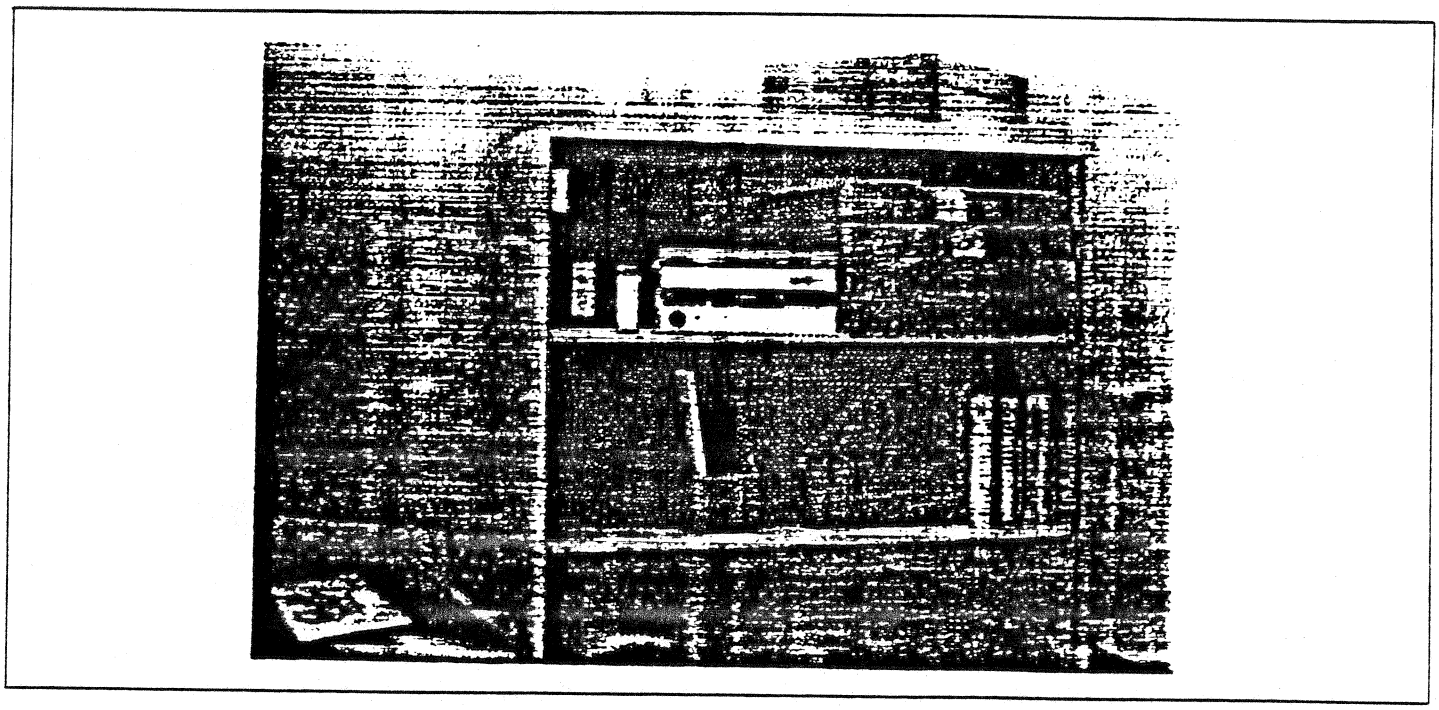

Figure 22: Bookcase Scene: Middle View

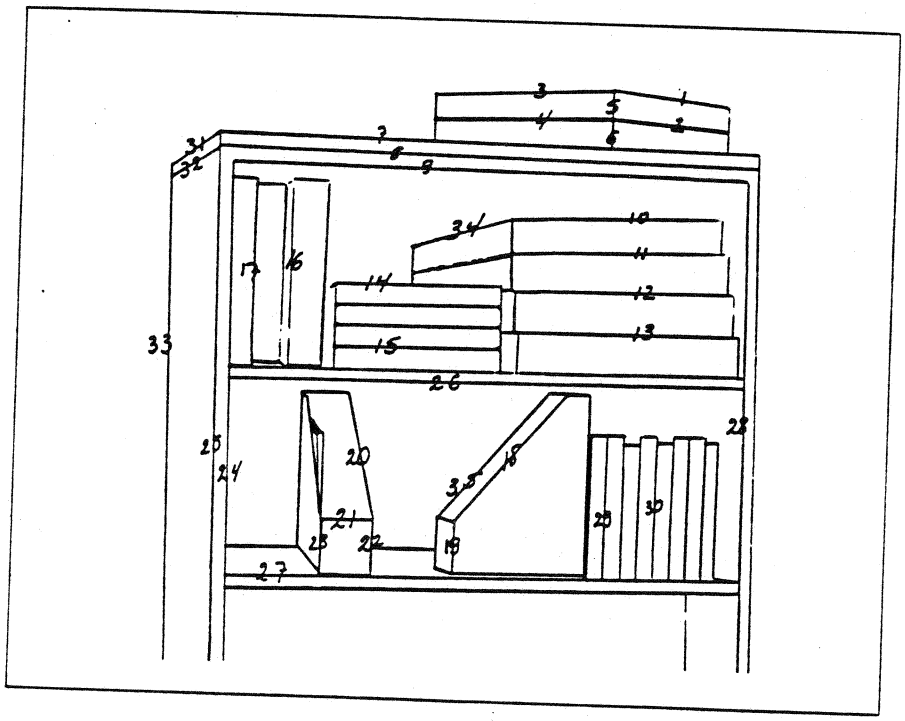

Figure 23: Lines from the Bookcase Scene

Case	X-Error	Y-Error	Z-Error
1	5.13	5.38	11.37
2	3.98	4.49	8.33
3	2.54	2.59	4.36
4	1.09	0.99	1.31
5	1.55	1.07	2.35
6	5.27	4.85	5.97
7	6.23	0.90	6.19
8	4.78	4.00	9.40
9	4.55	3.76	8.28
10	2.54	0.77	5.38

Table 1 Average error for experiments with the Bookcase Scene.

5 Summary and Future Work

The Multisensor Knowledge System offers many advantages for the design, construction, and simulation of multisensor systems. This framework is particularly useful if weak recognition methods are to be avoided through the use of specially tailored logical sensor objects. We have demonstrated the application of this approach to the interpretation of 3-D structure.

References

[1] J. Albus. *Brains, Behavior and Robotics.* BYTE Books, Peterborough, New Hampshire, 1981.

[2] R. Bajcsy. *GRASP:NEWS Quarterly Progress Report.* Technical Report Vol. 2, No. 1, The University of Pennsylvania, School of Engineering and Applied Science, 1st Quarter 1984.

[3] S.L. Chiu, D.J. Morley, and J.F.Martin. Sensor Data Fusion on a Parallel Processor. In *Proceedings of the IEEE Conference on Robotics and Automation*, pages 1629–1633, San Francisco, CA, April 1986.

[4] O.D. Faugeras, N. Ayache, and B. Faverjon. Building Visual Maps by Combining Noisy Stereo Measurements. In *Proceedings of the IEEE Conference on Robotics and Automation*, pages 1433–1438, San Francisco, CA, April 1986.

[5] O. Faugeras and F. Lustman. Identifying Planes for the Construction of the World Model of a Mobile Robot. In *Proceedings of the International Conference on Pattern Recognition*, pages 162–164, Paris, France, October 1986.

[6] T. Henderson and C. Hansen. *Multisensor Knowledge Systems.* Technical Report UUCS-TR-114, The University of Utah, Department of Computer Science, September 1986.

[7] T.C. Henderson and Esther Shilcrat. Logical Sensor Systems. *Journal of Robotic Systems*, 1(2):169–193, 1984.

[8] T.C. Henderson, C.D. Hansen, and Bir Bhanu. The Specification of Distributed Sensing and Control. *Journal of Robotic Systems*, 2(4):387–396, 1985.

[9] T.C. Henderson, Chuck Hansen, and Bir Bhanu. A Framework for Distributed Sensing and Control. In *Proceedings of IJCAI 1985*, pages 1106–1109, Los Angeles, CA, August 1985.

[10] T.C. Henderson, Chuck Hansen, Ashok Samal, C.C. Ho, and Bir Bhanu. CAGD-Based 3-D Visual Recognition. In *Proceedings of the International Conference on Pattern Recognition*, pages 230–232, Paris, France, October 1986.

[11] T.C. Henderson and Steve Jacobsen. The Utah/MIT Dextrous Hand. In *Proceedings of the ADPA Conf. on Intelligent Control Systems*, Ft. Belvoir, Va., March 1986.

[12] T.C. Henderson, E. Shilcrat, and C.D. Hansen. A Fault Tolerant Sensor Scheme. In *Proceedings of the International Conference on Pattern Recognition*, pages 663–665, August 1984.

[13] Thomas C. Henderson. *Workshop on Multisensor Integration for Manufacturing Automation.* Technical Report UU-CS-87-006, University of Utah, Department of Computer Science, Feb. 1987.

[14] Y. Liu and T.S. Huang. Estimation of Rigid Body Motion Using Straight Line Correspondences. In *Proc. of the IEEE Workshop on Motion, Charleston, SC*, pages 47–51, 1986.

[15] A. Mitiche and J.K. Aggarwal. An Overview of Multisensor Systems. *SPIE Optical Computing*, 2:96–98, 1986.

[16] A. Mitiche, S. Seida and J. Aggarwal. Interpretation of Structure and Motion from Line Correspondences. In *Proceedings of the International Conference on Pattern Recognition*, pages 1110–1112, Paris, France, October 1986.

[17] Amar Mitiche, Olivier Faugeras and J.K. Aggarwal. *Counting the Line.* Technical Report, INRS-Telecom, September 1987.

[18] Amar Mitiche and G. Habelrih. *The Angular Approach to the Interpretation of Straight Lines in Images.* Technical Report, INRS-Telecom, September 1987.

[19] E. Muehle. *FROBS Manual.* Technical Report PASS-note-86-11, University of Utah, October 1986.

[20] K. Overton. Range Vision, Force, and Tactile Sensory Integration: Issues and an Approach. In *Proceedings of the IEEE Conference on Robotics and Automation*, page 1463, San Francisco, California, April 1986.

[21] S. Shebs. *PCLS User Manual.* University of Utah PASS Group, 1985.

[22] P. Rives. *Caractérisation du Déplacement d'un Objet Tridimensionnel. Application au Domaine des Vidéoconferences.* Technical Report TR-82-10, INRS-Telecom, 1982.

[23] Esther Shilcrat, P. Panangaden and Thomas C. Henderson. *Implementing Multi-sensor Systems in a Functional Language.* Technical Report UU-CS-84-001, University of Utah, Department of Computer Science, Feb. 1984.

[24] Esther Shilcrat. *Logical Sensor Systems.* Master's thesis, University of Utah, Salt Lake City, Utah, June 1984.

[25] T. Tsukiyama and T.S. Huang. Motion Stereo for Navigation of Autonomous Vehicles in Man-Made Environments. In *Proceedings of the International Conference on Pattern Recognition*, pages 165–168, Paris, France, October 1986.

Appendix A. 3D Line Interpretation FROBS Code in MKS

FROB Rules

```
(def-rule select-general
  :type ((?req requirements))
  :prem ((environment ?req unknown)
 (occlusion ?req nil))
  :conc ((assert-val ?req 'views 3)
 (assert-val ?req 'lines 6)
 (run general-lines)))

(def-rule select-kac
  :type ((?req requirements))
  :prem ((environment ?req known)
 (occlusion ?req nil))
  :conc ((assert-val ?req 'views 1)
 (cond ((= (speed ?sys) 'fast)
(assert-val ?req 'lines 3))
       (T (assert-val ?req 'lines 6)))
 (run known-angles)))

(def-rule select-perp
  :type ((?req requirements))
  :prem ((environment ?req perp)
 (occlusion ?req nil))
  :conc ((assert-val ?req 'views
     (cond ((or more-lines bound-lambda) 1)
   (T 2)))
 (assert-val ?req 'lines
     (cond ((more-lines) 4)
   (T 3)))
 (run perpendicular)))

(def-rule select-parallel
  :type ((?req requirements))
  :prem ((environment ?req parallel)
 (occlusion ?req nil))
  :conc ((assert-val ?req 'views 1)
 (assert-val ?req 'lines 3)
 (run parallel)))

(def-rule use-3-views
  :type ((?req requirements))
```

```
  :premise ((interpreted ?req 1)
    (uninterpreted ?req 2))
  :conc ((assert-val ?req 'views 3)))

(def-rule use-2-views
  :type ((?req requirements))
  :premise ((interpreted ?req 2)
    (uninterpreted ?req 1))
  :conc ((assert-val ?req 'views 2)))
```

Class Definitions

```
(def-class logical-sensor nil
   :slots (name
   object-list
   selector
   program
                  out-vec
   message
   cci))

(def-method ({class logical-sensor} cov) ()
    (let* ((index (apply (selector $self) (message $self))))
      (eval (nth index (program $self)))))

(def-method ({class logical-sensor} com) (message)
    (setf (message $self) message))

(close-class)

(def-class algorithm nil
   :slots (name
   size
   language
   machine
                  executable
   args
   input
   input-type
                  output
   output-type
                  e-function
   machine-out
   complexity
```

```
   stability
                  robustness
   p-factor))

;
; Method for making new algorithm instance
;
(def-method ({class algorithm} make)
    (&key language name machine input-type output-type
  executable e-function robustness machine-out)
  (let ((inst (cond (name (new-instance* $self :name name))
    (T (new-instance* $self)))))
    (when e-function (setf (e-function inst) e-function))
    (when language (setf (language inst) language))
    (when name (setf (name inst) name))
    (when machine (setf (machine inst) machine))
    (when machine-out (setf (machine-out inst) machine-out))
    (when input-type (setf (input-type inst) input-type))
    (when output-type (setf (output-type inst) output-type))
    (when executable (setf (executable inst) executable))
    (when robustness (setf (robustness inst) robustness))
    inst))

;
; run for algorithm class produces (machine file) as output
;
(def-method ({class algorithm} run) ()
            (let* ((mach (machine $self))
                   (exec (executable $self))
   (args (args $self))
   (function (e-function $self))
   (genout (conc (conc "/tmp/" (string (name $self)))
 (string (gensym))))
   (out-type (output-type $self))
   (in-type (input-type $self))
   (genin "/tmp/stdin")
   (input (input $self))
   output)
              (cond ((def (output $self)) nil)
    ((def exec)
     (cond ((equal in-type 'file)
    (if (listp input)
      (setf input (cat-input-files mach input))))
   (T (put-file mach genin input)
      (setf input genin)))
```

```
    (system (cond ((def mach)
   (list "remsh" mach exec
 (if (def args) args)
 '" '< '" input '" '> '"
 genout))
  (T (list exec (if (def args) args)
   '< input '> genout))))
    (setf (output $self)
  (cond ((equal out-type 'file) genout)
(T (get-file mach genout)))))
                  (T (cond ((equal in-type 'file)
     (if (listp input)
(setf input (cat-input-files mach input)))
      (setf input (get-file mach input)))
     (T nil))
       (setf output (eval (cons function (quote-list input))))
       (setf (output $self)
     (cond ((equal out-type 'file)
    (put-file (machine-out $self)
      genout output)
    genout)
   (T output)))))))


(close-class)

(def-class image nil
   :slots (name
   size
   light
   camera-angle
   distance
   filter
   threshold
   segmenter
   objects
   features
   robust-features
   model
   machine
   file-name))


;
; Method for making new image instance
```

```
;
(def-method ({class image} make)
    (&key name size machine file-name)
  (let ((inst (cond (name (new-instance* $self :name name))
    (T (new-instance* $self)))))
    (when name (setf (name inst) name))
    (when size (setf (size inst) size))
    (when machine (setf (machine inst) machine))
    (when file-name (setf (file-name inst) file-name))
    inst))

(close-class)


(def-class sensor nil
   :slots (name
   type
   accuracy
   hysteresis
   dynamic-range
   hardware
   machine
   executable
   output-file
   output-type
   output))

;
; Method for making new sensor instance
;
(def-method ({class sensor} make)
    (&key name type machine executable output-file
  output-type)
  (let ((inst (cond (name (new-instance* $self :name name))
    (T (new-instance* $self)))))
    (when name (setf (name inst) name))
    (when type (setf (type inst) type))
    (when machine (setf (machine inst) machine))
    (when executable (setf (executable inst) executable))
    (when output-file (setf (output-file inst) output-file))
    (when output-type (setf (output-type inst) output-type))
    inst))

(def-method ({class sensor} operate) ()
    (let* ((mach (machine $self))
```

```
   (out (output $self)))
      (cond ((def out) out)
              (T (system (append (if (def mach) (list "remsh" mach))
       (list (executable $self))))
       (setf (output $self) (output-file $self))))))

(close-class)
```

Instance Definitions

```
(make {class sensor}
      :name 'ccd-camera
      :type 'visual
      :machine "cs"
      :executable "$frobs/run_camera.csh"
      :output-file "/tmp/cam_out.img"
      :output-type 'file)

(make {class sensor}
      :name 'hitachi
      :type 'visual
      :machine "cs"
      :executable "$frobs/run_hitachi.csh"
      :output-file "/tmp/hitachi_out.img"
      :output-type 'file)

(make {class algorithm}
      :language 'c
      :name 'display-on-elan
      :machine "elan"
      :input-type 'file
      :output-type 'file
      :executable "bin/put_frame -i -b -c")


(make {class algorithm}
      :name 'rle-to-gray
      :language 'c
      :machine "sp"
      :executable "/s/bin/lss/2D_frob_code/rle_to_gray"
      :output-type 'file)

(make {class image}
      :name 'hinge
      :size '(480 512)
      :machine "cs"
```

```
      :file-name "/u/weitz/vision/hinge_0.img")


(make {class algorithm}
      :name 'color-screen
      :language 'c
      :machine "sp"
      :executable "$frobs/color.csh"
      :input-type 'file)

(make {class algorithm}
      :name 'edge
      :language 'c
      :machine "sp"
      :executable "/s/IKS/bin/edge"
      :output-type 'file)

(make {class algorithm}
      :name 'hough
      :language 'c
      :machine "sp"
      :executable "/s/IKS/bin/hough"
      :output-type 'file)

(make {class algorithm}
      :name 'lines
      :language 'c
      :machine "sp"
      :executable "/s/IKS/bin/lines"
      :output-type 'file)
```