The "X-Vision" System: A General-Purpose Substrate for Vision-Based Robotics

Gregory D. Hager

Research Report YALEU/DCS/RR-1078
June 1995

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

The "X-Vision" System: A General-Purpose Substrate for Vision-Based Robotics

Gregory D. Hager
Department of Computer Science
Yale University, P.O. Box 208285
New Haven, CT, 06520

Phone: (203) 432-6432
Fax: (203) 432-0593
E-mail: hager@cs.yale.edu

Abstract

In the area of real-time vision there has been a shift away from large complex, geometry-based vision systems towards simpler, image-based, task-specific systems. One particular advantage of simplifying vision is that it becomes more accessible to users who are not experts in vision. Moreover, it is likely that making vision cheap and easy to use will accelerate both practical and theoretical advances in the field. However, building many task-specific applications will also place a greater emphasis on flexibility and reconfigurability of real-time vision systems.

Just as the use of graphics was accelerated by the development of the X-windows system, we believe that creating the proper type of programming environment around real-time vision, an "X-vision" system, will greatly accelerate the development of vision, both theory and practice. To this end, we have constructed a framework for visual tracking and hand-eye coordination out of modular software components. Over the past two years, we have used these components to construct several vision-based hand-eye and mobile robotic systems. This article describes the tracking system and some of its features, and briefly describes its use in several applications.

Submitted to the 1995 Workshop on Vision for Robotics

Keywords: vision-guided robotics, real-time vision

1 Introduction

Historically, applications using real-time vision resorted to specialized hardware in order to process the huge amount of information available in video images. This tended to limit work on real-time vision and related applications such as visual servoing to those who could afford such hardware and the support personnel required to maintain it. However, the speed of standard workstations continues to increase, prices continue to decrease, and multi-processor architectures are becoming more widely available and accessible. These advances anticipate the day when many real-time vision applications can be run on standard workstations or PCs outfitted with a simple framegrabber.

One area that is particularly amenable to software-based systems is tracking of spatially localized features in a sequence of images. Because the processing is local, large data bandwidth between the host and the framegrabber is not needed. Likewise, the amount of data that must be processed is relatively low and can be efficiently dealt with by off-the-shelf hardware.

The design of many task-specific visual tracking systems places a strong emphasis on modularity and reconfigurability. Providing this flexibility depends on having an intuitive, consistent framework for expressing vision programs. Just as user-interface design was spurred by the creation of portable graphics systems such as X-windows, we believe that vision research and application would be greatly accelerated by the construction of the "X-vision" system. To this end, we have constructed modular software-based system for experimental vision-based robotic applications [12]. The emphasis has been on flexibility and efficiency on standard scientific workstations and PC's. The system is intended to be a portable, inexpensive tool for rapid prototyping and experimentation for teaching and research. The model of computation is one of independent tracking agents that communicate information to one another in a hierarchical network. Such programs are amenable to implementation on either serial or MIMD processing hardware.

By adopting this viewpoint, not only does vision becomes simple and cheap to use, but we have found that many vision-based tasks are naturally and easily conceptualized within this framework. One reason for this is that our software abstraction of local tracking inherently *serializes* the vision process. That is, a complete image containing motion contains a great deal of complex dynamics: illumination changes, motion, occlusion and disocclusion, and so forth. Correctly anticipating and dealing with all of these changes in a coherent way is impossible in all but the most controlled circumstances. Moreover, since most processors for handling full-frame image sequences in real-time are SIMD, whatever operations are performed on an image sequence must be applied uniformly over entire images. This rigid structure severely limits the capabilities of such systems.

In contrast, localizing vision disregards much of the irrelevant dynamics of a scene. Several small image regions are naturally thought of as simple (in the sense of low-dimensionality) dynamic systems. These systems are loosely coupled and interact with one-another in specific, predictable ways. Furthermore, since the data associated with a single image region is relatively small, it is possible to utilize more flexible image processing algorithms that are adaptive to the local spatial and temporal properties of that region.

Our system is organized in such a way as to provide a simple, easily reconfigured abstraction for visual tracking. We have used this system for hand-eye coordination, eye-in-hand visual servoing, mobile robot navigation, and a variety of other tasks. The remainder of this article describes our system in more detail and provides experimental evidence that this approach is a useful tool in the construction of effective vision-based systems. The remainder of the article is organized into three sections: Section 2 describes our tracking system in some detail, Section 3 shows several examples of its use, and Section 4 discusses current and future research directions.

2 Tracking System Design and Implementation

It has often been said that "vision is inverse graphics." In many ways, our tracking system fits this analogy. Suppose, for example, that a graphics system is to display a complex object. Typically, an object-centered coordinate system will be defined, and the constituent parts of the object, *e.g.* the polyhedral faces of a polygon, are described in that coordinate system. This decomposition is carried out until primitive graphics objects, *e.g.* lines, are reached. These are translated to screen coordinates and displayed. Good graphics systems make defining these types of geometric relationships simple and intuitive.

Our tracking system provides this functionality *and* its converse. That is, it is possible to describe features or objects parametrically. Given the parameters of high-level objects, they are

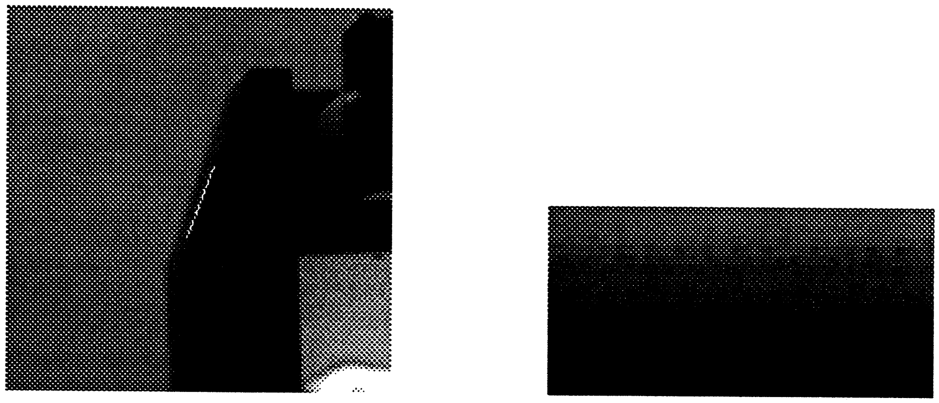


Figure 1. On the left, a sample image showing a reference line on the waist of the robot, and on the right the image associated with the window as it appears in window local coordinates.

broken down into their constituent components. However, instead of displaying the object, we now seek to locate the low-level features in the neighborhood of their expected location. Once found, we recompose the object for high-level feature from its components.

This motivates the two central ideas of our tracking architecture: window-based image processing, and state-based composition of networks of tracked features. Furthermore, just as graphics systems are often naturally described in an object-oriented way, we have found the object-oriented programming is well-suited to describing tracking systems. The use of object-oriented methods allows use to hide the details of how specific methods are implemented, and to interact with the system through a pre-specified set of generic interfaces. It also enhances the portability of the system.

2.1 Low-Level Feature Detection

A window is a rectangular area of an image defined by its size, sampling characteristics, position, orientation, in device (framebuffer) coordinates. All image processing operations within the window are defined *relative to the local window coordinate system.* To illustrate, suppose that we have an image with a window located about the white line in Figure 1(left). Then the image associated with that window (in window coordinates) is shown in Figure 1(right). Note that the line appears roughly horizontal in window coordinates.

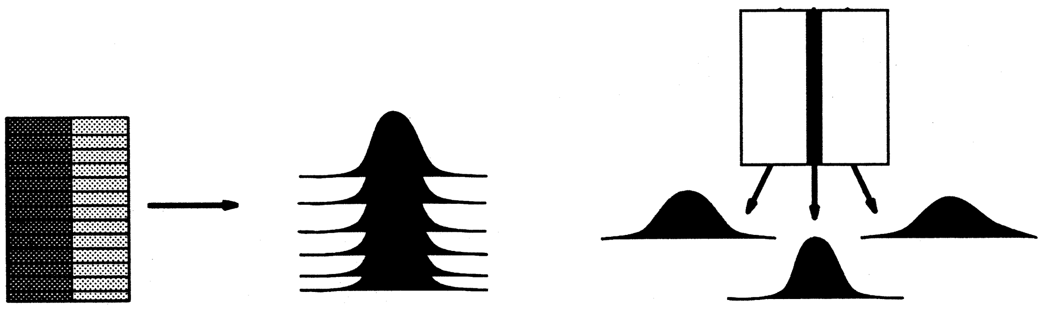


Figure 2. One dimensional convolution followed by superposition.

Tracking a feature means that a window maintains a fixed, pre-defined relationship to the feature. Hence, any operation such as line detection or feature matching can be implemented assuming the requisite feature only deviates slightly from a standard orientation and position in local coordinates. This makes image processing simple to implement, fast to execute, and easy to specialize. At the same time, acquiring windows at any position and orientation can be implemented quickly using ideas for fast rendering of lines and boxes borrowed from graphics [6]. Thus, the combination of movable oriented windows and image processing assuming a canonical configuration leads to fast feature tracking. The low-level features currently available in our system include solid or broken contrast edges detected using convolutions, and general grey-scale patterns tracked using SSD methods [2, 23, 13]. These basic features can be easily composed into a wide variety of more complex configurations.

2.1.1 Edges

We often utilize occluding contours and contrast edges as the basis for tracking applications. In most image processing systems, the majority of the time needed to perform edge detection is devoted to convolution. The key idea in fast contour localization is to use problem constraints and prior temporal information to simplify detection. The main observation is that, in window coordinates, detecting a contrast step edge in canonical position can be thought of as a series of one-dimensional detection problems as shown in Figure 2(left). Assuming the edge is vertical in the window, convolving each row of the window with a derivative-based kernel will produce an aligned series of response peaks. These responses can be superimposed by summing down the columns of the window. Finding the maximum value of this response function localizes the edge. If the edge is not correctly oriented, the response curve broadens. However, if the edge is symmetric, the maximum still correctly indicates the location of the edge in the window. Thus, for the price of a single pass with a one-dimension convolution and a series of summations we can localize the position of a contrast edge.

In order to compute orientation, we sum the response to the scalar convolution along image diagonals as indicated in Figure 2 (right). This closely mirrors the effect of performing the convolution at orientations close to nominal. The result is three response curves with different maximum values. The curve with the highest response is that closest in orientation to the underlying edge. By performing parabolic interpolation of the three curves, it is possible to predict the actual orientation of the underlying edge. In the ideal case, if the convolution template is symmetric and the response function after superposition is unimodal, the horizontal displacement of the edge should agree between all three filters. In practice, the estimate of edge location will be biased. For this reason, edge location is computed as the weighted average of the edge location of all three peaks. Assuming the convolution template can be expressed with integers, this entire operation can be performed with only integer addition and multiplies except for the interpolation step. If additional localization accuracy is required, a second derivative operator can be performed in the local neighborhood of the detected edge at the computed orientation, and the zero-crossing used to compute sub-pixel accuracy.

A particularly simple detector is the derivative of a triangle distribution. The convolution template consists of n -1's followed a 0 followed by n 1's [7]. This kernel is attractive because no multiplications are needed to compute the convolution. Furthermore, the time to compute the convolution can be made independent of the size of the kernel. This is accomplished by noting that the difference in response between one pixel and the next can be computed by four additions and subtractions of pixels at the transition points of the convolution template. This detector on a Sun Sparc 2 with an Imaging Technologies 100 series framegrabber requires 1.5ms for a 20 pixel line searching $\pm$ 10 pixels using a mask 15 pixels wide ($n = 8$).

Detectors based on these ideas have been found to have good rejection characteristics and good localization accuracy. Edges outside the window obviously do not affect their operation. Edges within the window, but oriented incorrectly do not provoke much filter response and are generally rejected. In addition, the value and/or sign of the response can be used to enhance the "tuning" of the filter for a particular contour. For isolated straight edges, localization accuracy has been found to be on the order of 0.1 pixels.

2.1.2 Correlation

SSD tracking relies on the *image constancy assumption* which states that two images separated by a brief time instant differ only by a geometric distortion. It is well-known that the geometric distortions of an image are well-approximated locally by an affine transformation [3]. Following [16, 19, 13], the general form of the image constancy constraint can be written as:

$$\beta I(\mathbf{A}\mathbf{x} + \mathbf{d}, t + \tau) + \gamma = I(\mathbf{x}, t), \quad \tau > 0,$$

where $\mathbf{A}$ is a 2×2 matrix, $\mathbf{d} = (dx, dy)^T$, $\mathbf{x} = (x, y)^T$, and β and γ are relative image contrast and brightness, respectively. Given an image with an identified region centered at $\mathbf{c} = (u, v)^T$ and a spatial extent represented as set of image locations, $\mathcal{W}$, the correspondence with a second image can be determined by minimizing

$$O(\mathbf{A}, \mathbf{d}) = \sum_{\mathbf{x} \in \mathcal{W}} (I(\mathbf{A}\mathbf{x} + \mathbf{d} + \mathbf{c}, t + \tau) - I(\mathbf{x} + \mathbf{c}, t))^2 w(\mathbf{x}), \quad \tau > 0, \tag{1}$$

where $w(\cdot)$ is a weighting function over the image region.

Two approaches to solving this problem have been proposed. A discrete optimization approach is described by [20]. However, their system considers only translations (no contrast or brightness compensation, and $\mathbf{A}$ fixed to be the identity matrix), and utilizes a special signal processor to perform the calculations.

Other authors solve the problem using continuous optimization [2, 19, 13]. This approach has the advantage of efficiently estimating all 8 parameters simultaneously, and can be modified to work in situations where not all parameters are fully determined by the gray-level structure of the image. The disadvantage is that, in practice this method will only work for motions of a fraction

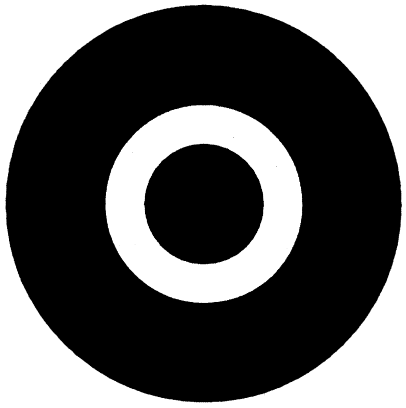


Figure 3. The bulls-eye pattern used for invariant-based target recognition.

of a pixel. Larger motions cannot be guaranteed to converge, although in practice they often do if the gray-level structure of the image is sufficiently simple [19].

In our tracking system, we adopt the continuous optimization approach. The full 8 parameter system is decomposed into smaller parameter groups, consisting of translation, brightness and contrast, scale, and orientation and shear.[1] We integrate interframe changes over time so that we are constantly computing the match between the initial and the current frame. These calculations are performed at an adaptive level of resolution, leading to a fast algorithm for tracking fast motions and a slower but more accurate algorithm for tracking slower motions. Tracking speeds on a Sun Sparc II range from $3 - 4$ milliseconds for pure translation of a 20×20 region at one-quarter resolution to $10 - 15$ milliseconds when both scaling and translation are calculated. Details can be found in [13].

2.1.3 Specialized Patterns

In many applications, it is useful to be able to detect fiducial marks for specialized patterns as a starting "seed" for initialization of more complex tracking. We provide such an initializer in the form of an invariant-based target recognizer. This method was inspired by [21].

Briefly, the method proceeds as follows. It is well-known that four points on a line define an value, the *cross-ratio,* that is invariant to projection [17]. We utilize this idea to detect the

[1]The last two have not yet been fully implemented and tested.

"bulls-eye" pattern shown in Figure 3. Across any line passing through the center, there are a total of 15 cross-ratios that can be computed (although not all are independent). For a particular configuration of circles, these cross-ratios index the pattern. Because it is circularly symmetric, this pattern has the same cross ratio no matter what direction it is viewed from.

Using the simple edge-detectors described above, we scan the image looking for a series of edges that have the correct sequence of bright/dark transitions, and which have the correct cross-ratio signature. If a candidate is found, a cross ratio calculation in the orthogonal direction is performed to verify the signature. The method has been found to work very reliably for targets of moderate image size (about 40 pixels minimum). It takes about 1.5 seconds to locate a target in a 480×640 image on a Sun Sparc II processor.

2.2 Networks of Features

Every feature or image property in our system can be characterized in terms of a state vector. For *basic features*—those that operate directly on images—the state of the feature tracker is usually the position and orientation of the feature (= that of its window) relative to the framebuffer coordinate system. We define *composite features* to be features that compute their state from other basic and composite features.

To illustrate the relationship between basic and composite features, consider computing the location of the intersection of two contours. The state of a single contour tracker in an image is the vector $L = (x, y, \theta)^T$ describing the location of a window centered on the contour and oriented along it. The low-level feature detection methods described above compute an offset normal to the edge, δt, and an orientation offset $\delta\theta$. Given these values, the state of the contour tracker is updated according to the following equation:

$$L^{+} = L^{-} + \begin{bmatrix} -\delta t \sin(\theta + \delta\theta) \\ \delta t \cos(\theta + \delta\theta) \\ \theta + \delta\theta \end{bmatrix} \tag{2}$$

Note that there is an aperture problem: the state vector, L, is not fully determined by information returned from feature detection. There is nothing to keep the window from creeping "along" the contour it is tracking.

This problem can be solved by defining a composite feature that is the intersection of two non-collinear contours. This feature has a state vector $C = (x, y, \theta, \alpha)^T$ describing the position of the intersection point, the orientation of one contour, and the orientation difference between the two contours. From image contours with state $L_1 = (x_1, y_1, \theta_1)^T$ and $L_2 = (x_2, y_2, \theta_2)^T$, the distance from the center of each tracking window to the point of intersection the two contours can be computed as

$$\begin{aligned}
\lambda_1 &= ((x_2 - x_1)\sin(\theta_2) - (y_2 - y_1)\cos(\theta_2))/\sin(\theta_2 - \theta_1) \\
\lambda_2 &= ((x_2 - x_1)\sin(\theta_1) - (y_2 - y_1)\cos(\theta_1))/\sin(\theta_2 - \theta_1)
\end{aligned}$$

The state of a corner $C = (x_c, y_c, \theta_c, \alpha_c)$ is calculated as:

$$\begin{aligned}
x_c &= x_1 + \lambda_1 \cos(\theta_1) \\
y_c &= y_1 + \lambda_1 \sin(\theta_1) \\
\theta_c &= \theta_1 \\
\alpha_c &= \theta_2 - \theta_1
\end{aligned} \tag{3}$$

Given a fixed intersection point, we can now *choose* "setpoints" λ_1^* and λ_2^* describing where to position the contour windows relative to the intersection point. With this information, the states of the individual contours can be adjusted as follows:

$$\begin{aligned}
x_i &= x_c - \lambda_i^* \cos(\theta_i) \\
y_i &= y_c - \lambda_i^* \sin(\theta_i)
\end{aligned} \tag{4}$$

for $i = 1, 2$. Choosing $\lambda_1^* = \lambda_2^* = 0$ defines a cross pattern. If the window extends h pixels along the contour, choosing $\lambda_1^* = \lambda_2^* = h/2$ defines a corner. Choosing $\lambda_1^* = 0$ and $\lambda_2^* = h/2$ defines a tee junction, and so forth.

A complete tracking cycle for this system would consist of first computing (4) to make the initial state of the contours consistent, then performing low-level feature detection, and finally computing (2) followed by (3).

More generally, we define a *feature network* to be a set of nodes connected by two types of directed arcs referred to as *up-links* and *down-links*. Nodes represent basic and composite features. Up-links represent the information dependency between a composite feature and the features used to compute its state. Thus, if a node is a source node with respect to up-links, it must be a basic feature. If a node has incoming up-links it must be a composite feature. If a node n has incoming up-links from nodes $m_1, m_2, \ldots, m_k$ the latter are called *subsidiary nodes* of n. Down-links represent the imposition of constraints or other high-level information on features. A node that is a source for down-links is a *top-level node*. If a node n has incoming down-links from nodes $m_1, m_2, \ldots m_k$, the latter are called *supersidiary nodes* of n. All directed paths along up-links or down-links in a feature graph must be acyclic. We also require that every top-level feature that is path-connected to some basic feature with respect to up-links must be path-connected to the same basic feature via down-links. For example, a corner is a graph with three nodes. The corner feature is a top-level feature. The two contours which compose it are subsidiary features. There are both up-links and down-links between the corner node and the feature nodes.

Given this terminology, we can now define a complete tracking cycle to consist of: 1) traversing the down-links from each top-level node applying state constraints until basic features are reached; 2) applying low-level detection in every basic feature; and 3) traversing the up-links of the computing the state of composite features. State prediction can be added to this cycle by including it in the downward propagation.

2.3 Feature Typing

In order to make visual constructions simpler and more generic, we have included polymorphic type support in the tracking system. Briefly, each feature, basic or composite, carries a type. This type essentially identifies the minimal information contained in the state vector of the feature. For example, there are *point features* which carry location information, *line features* which carry orientation information, and *fixed lines* which are line segments which are fixed at one endpoint.

Any visual construction is a mapping from input types to output types. So, for example, a line feature can be constructed from two point features by computing the line that passes through the feature. Conversely, a point feature can be computed by intersecting two line features. As long as the types match, it is irrelevant how the prerequisite state information is computed. That is, a point feature can be computed equally well by intersecting line features computed from direct tracking of edges in an image, or by intersecting line features constructed from other point features.

Feature typing is also polymorphic. For example, tracking two corresponding points in two images yields a stereo point feature. Likewise, tracking two line features in two images yields a stereo line feature. More generally, tracking two x's in two images yields a stereo x.

2.4 Programming Environment

We have constructed our tracking system as a set of classes in C++. Briefly, all features are derived from a base class called `BasicFeature`. Basic features are directly derived from this class, and are characterized by their state vector, and functions which compute state information and display the feature graphically. There are two types of composite feature which are also derived from `BasicFeature`. `CompFeature` describes a composite feature which has both upward and downward links. `FeatureGroup` is a composite feature with only upward links—that is, it does not impose any constraints on its subsidiary features. Any feature may participate in only one `CompFeature`, but many `FeatureGroups`.

Both `CompFeature` and `FeatureGroup` maintain and manage an internal queue of their subsidiary features. Information is propagated up and down the feature network using two functions: `compute_state` which computes a composite feature's state from the state of its subsidiary nodes, and `state_propagate` that adjusts the state of a subsidiary nodes based on the state of their supersidiary node. The default update cycle for a `CompFeature` is to call its own `state_propagate` function, to call the update function of the children, and then to call `compute_state`. A `FeatureGroup` is similar, except there is no `state_propagate` function. The tracking cycle is combined into a single function `track()` callable only from a top-level feature. Calling it sweeps information down the network to the set of basic features, updates of the state of all basic features, and sweeps updated state information back up the network.

We have found that this programming environment greatly facilitates the development of track-

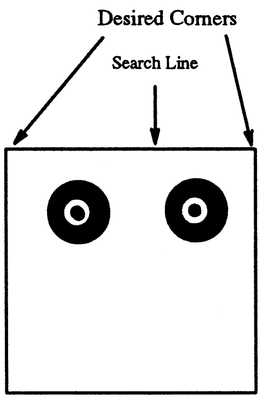


Figure 4. Schematic of the initialization example.

ing applications, and leads to clear compact program semantics. As an example, consider a simple program to locate and track the corners of the disk shown in Figure 4 using the fiducial marks located near one edge.

First, we declare all of the relevant components.

```
Video v(1);  Edge e; // Open a video device 1 and declare an edge detector
Target t1(sig1);     // Target 1 with signature 1 (not defined here)
Target t2(sig2);     // Target 2 with signature 2 (not defined here)
Line l1(&e,&v);       // Declare a contour tracker in video device 1
Line l2(&e,&v);       // Declare a contour tracker in video device 1
Corner c1(&e, &v,UL); // Declare an upper left corner tracker in video device 1
Corner c2(&e, &v,UR); // Declare an upper left corner tracker in video device 1
```

We then locate the two targets of interest.

```
if (!((t1.search() && t2.search()))  // Make sure we find both targets.
   exit(1);
```

We use the target locations to compute orientation, and the scale to choose an approximate edge location.

```
theta = atan2(t1.y() - t2.y(),t1.x() - t2.x());  // edge orientation
xoff = - t1.scale()*sin(theta)*OFFSET;          // where we guess
yoff = t1.scale()*cos(theta)*OFFSET;      // the line is in the image
l1.set_state(t1.x() + xoff, t1.y() + yoff, orientation);
l2.set_state(t1.x() + xoff, t1.y() + yoff, orientation+M_PI);
```

We then search for the left and right corners along the edge. Each `BasicFeature` has a default pattern recognizer that indicates the pattern is found.

```
If (!(l1.search(c1) && l2.search(c2)))
   exit(1);
```

Both corners can be added to a container `CompFeature` variable:

```
  CompFeature p;       // Declare a generic composite feature with no constraint functions
  p += c1;  p += c2;    // add features to an internal queue.
```

Once all features are properly initialized, the main loop of any tracking application is of the form:

```
  while (...) {
   p.track();
   ... other user code ...
}
```

Naturally, any other user code impacts the speed of tracking and so must be limited to operations that can be performed in a small fraction of a second. In most of our applications, this is a feedback control computation, or a broadcast of information to another processor.

3 Applications

In this section, we briefly describe several uses of the tracking system and report the performance we have been able to achieve. These results are excerpted from [24, 13, 10, 9, 8]. All experiments were performed on a Sun Sparc II workstation.

3.1 Pure Tracking

The following applications are intended to illustrate some of the capabilities of the tracking system. The first applications, computation of epipolar geometry, illustrates the use of tracking across two cameras while carrying out complex calculations. The second application illustrates how the modularity of the system makes it simple to perform comparative experiments.

Tracking Epipolar Geometry The epipolar geometry between two cameras is a useful tool in many visual applications involving moving cameras. The epipolar geometry as represented by the E matrix can be computed by locating $n \geq 8$ corresponding points in two images, compiling an n by nine matrix A, and selecting the eigenvector corresponding to the smallest eigenvalue of $A^T A$ [14]. The nine values of the eigenvector are the nine entries in E. Figure 5 describes the feature network used to implement corresponding epipolar line tracking. We rely on corners as defined in Section

$$\begin{tikzcd}
 & \text{Line} \arrow[dr] & & \text{Line} \arrow[dl] & & \text{Line} \arrow[dr] & & \text{Line} \arrow[dl] & \\
 & & \text{Corner} \arrow[drr] & & & & \text{Corner} \arrow[dll] & & \\
\cdots & \text{CorFeature} \arrow[drrr] & & & \text{CorFeature} \arrow[d] & & & \text{CorFeature} \arrow[dlll] & \cdots \\
 & \text{Corner} \arrow[dr] & & & \text{EMatrix} \arrow[dll] & & & & \\
 & & \text{Corline} & & & & & &
\end{tikzcd}$$

Figure 5. The feature network used to compute corresponding epipolar lines

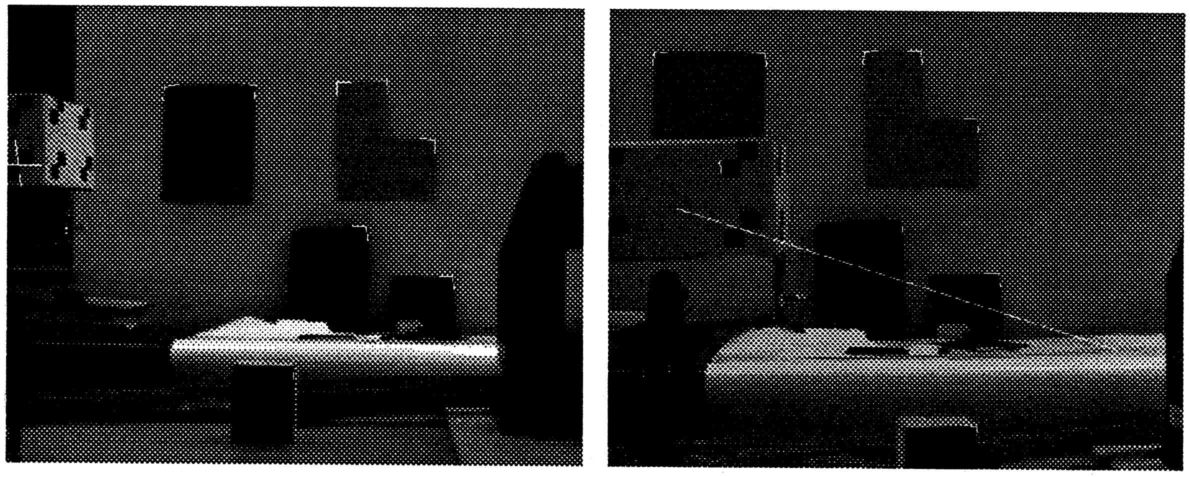


Figure 6. Left, the first image with a chosen feature (the cross at the far left) and right, the second image showing the corresponding epipolar line.

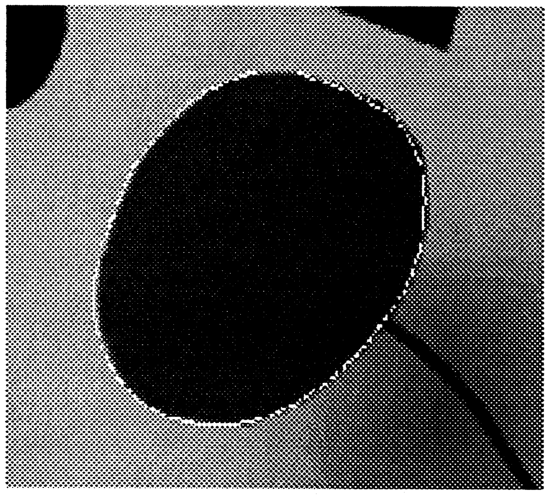
2.2. We define the class `StereoFeature` which is polymorphic `FeatureGroup` which manages two features in separate video images. Its state is the concatenation of the states of both features in their respective images. It imposes no constraints on the state of subsidiary features. Next, we define the class `EMatrix` to be a `FeatureGroup` that manages eight or more corresponding features. It contains a `compute_state` function that computes the E matrix as described above. Finally, we define a class `Cline` that manages a corner and an instance of an `Ematrix`. It has a `compute_state` function that computes the equation of the epipolar line in the second image. The display function is defined to show the line in the second video image.

Figure 6 shows two images from an experimental run of the system. Using nine corners composed of features 12 pixels long, we were able to reliably compute and track the corresponding epipolar line for a tenth feature point (the hash pattern on the lower right of the box) at a rate of approximately 10 Hz. The error in correspondence location was typically less than five pixels. It was interesting to note that the epipolar line equations are actually quite noisy. However, the computed line "swivels" about the corresponding feature point. Thus, while the line itself may swing as much as 20 or 30 pixels in some regions, it is extremely stable at the corresponding feature point.

Distraction Resistant Tracking The tracking package includes a contour tracker similar to the widely reported "Kalman Snake" algorithms [1, 5, 15, 22, 25]. The snake is tracked by localizing small edges as discussed in Section 2.1. The snake itself is a derivative of `CompFeature`. The upward propagation computes the state of the snake based on the state of edges using a simple spline fit. To determine the position of new search windows, we compute a weighted combination of temporal prediction and spatial interpolation to arrive at the line parameters, $\mathbf{L}$, of search windows for low level edge tracking.

Figure 7 displays images immediately after a contour tracker is initialized. Figure 7(a) shows the cubic spline determined by the knot points shown in Figure 7(b). The silhouette of the gray ovals are the intended targets for Figure 8.

One difficulty with the method as described is that it can be easily distracted as shown in Figure 8 and as observed by other authors [4]. In each instance, edges more prominent than the original edges enter into one or more search windows, effectively distracting them from their


(a)

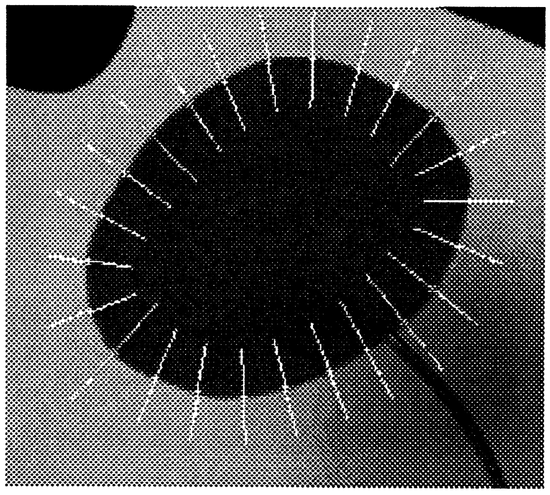

(b)

Figure 7. Initial tracking showing, (a) a cubic spline contour, and (b) the search windows. The short line segments represent the search window of each tracking component, with estimated edge location (and consequently, spline knot points) at their midpoints.

intended targets. Distraction occurs because individual components are simply high-gradient edge finders, and depend upon high-level models to correct them in the case that they stray. While many model- or template-based contour trackers, such as those described in [4], might not be distracted by the high-contrast edges in Figures (a) and (b), they would still fail in cases (c) and (d), where the change in the shape of the contour is small and/or gradual.

We have implemented an edge detector that combines elements of feature detection combined with some of the temporal correlation aspects of SSD tracking [24]. The same spline algorithm is easily instantiated using these edge trackers. Figure 9 illustrates the resulting distraction-free tracking. By simply maintaining some information about the kind of edges that are tracked (i.e., what intensities are observed inside the contour), tracking is improved greatly. Without recourse to any global models for shape, individual search windows can remain on target, even when what might be considered more "attractive" edges enter the search space.

Finally, in Figure 10, we show how a book remains accurately tracked even as significant distraction occurs in the background. We note that no specific dynamic models or shape models for the book have been used. The images were taken approximately 15-20 frames apart, at a tracking rate of 20Hz for 35 search windows of 40 pixel widths each. Similar results can be obtained no matter how quickly distractions in the background occur.

3.2 Hand-Eye Coordination

We have developed a system for hand-eye coordination as described in [10, 9]. The system relies on stereo vision to control the relative pose of an object in a robot end-effector and a static object in the environment.

The system uses what we refer to as a *skill based* organization. *Primitive skills* are vision-based regulators which provide feedback to attain a particular type of geometric constraint between the pose of the robot-held object and the target object. For example, a point-to-point positioning skill uses the projection of a point feature, *e.g.* a corner. Using information from both objects in two images, feedback is computed which moves the manipulator to place the feature of its object at the same location as the corresponding feature of the target object.

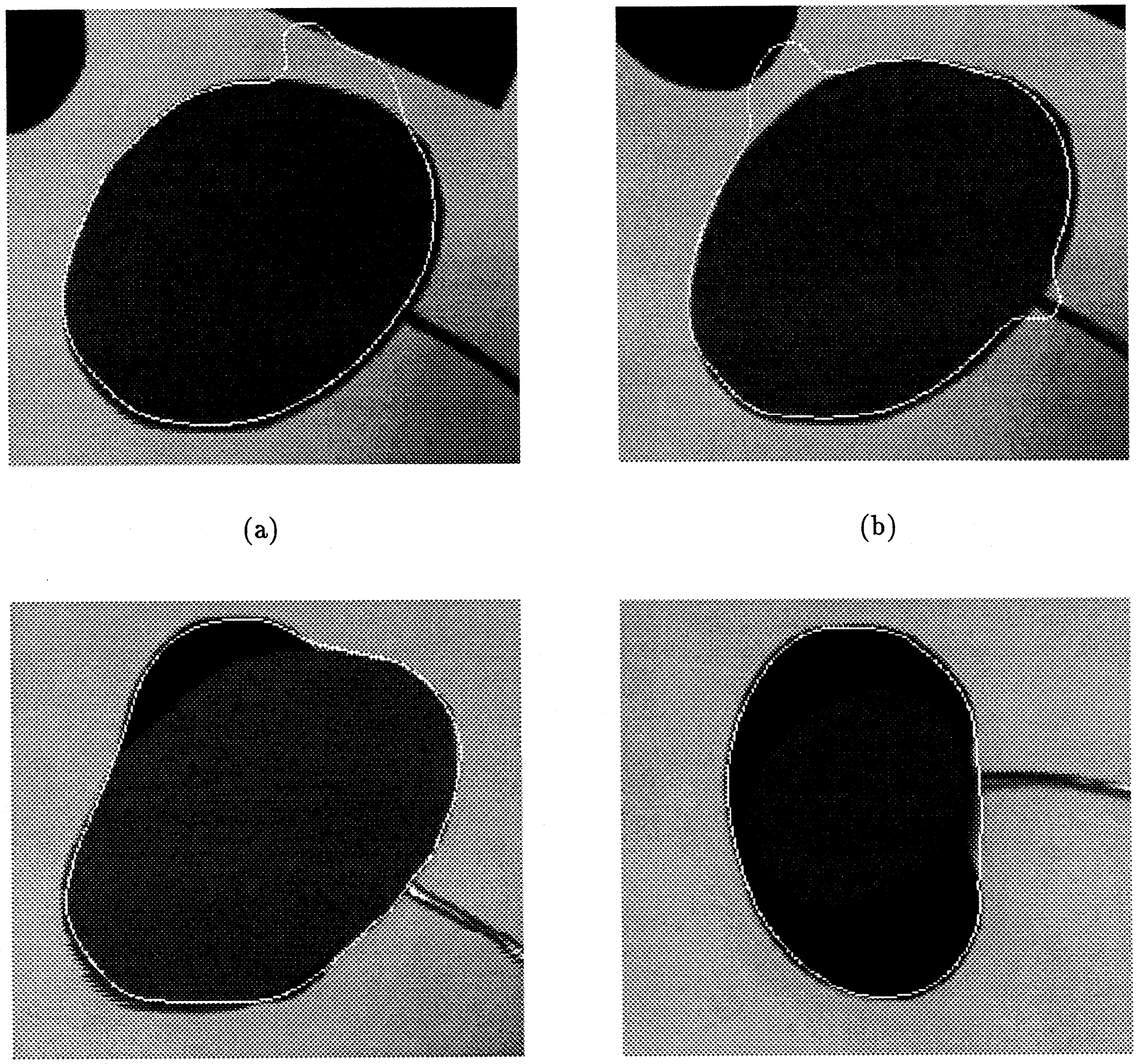


Figure 8. Instances of distraction and consequent mistracking.

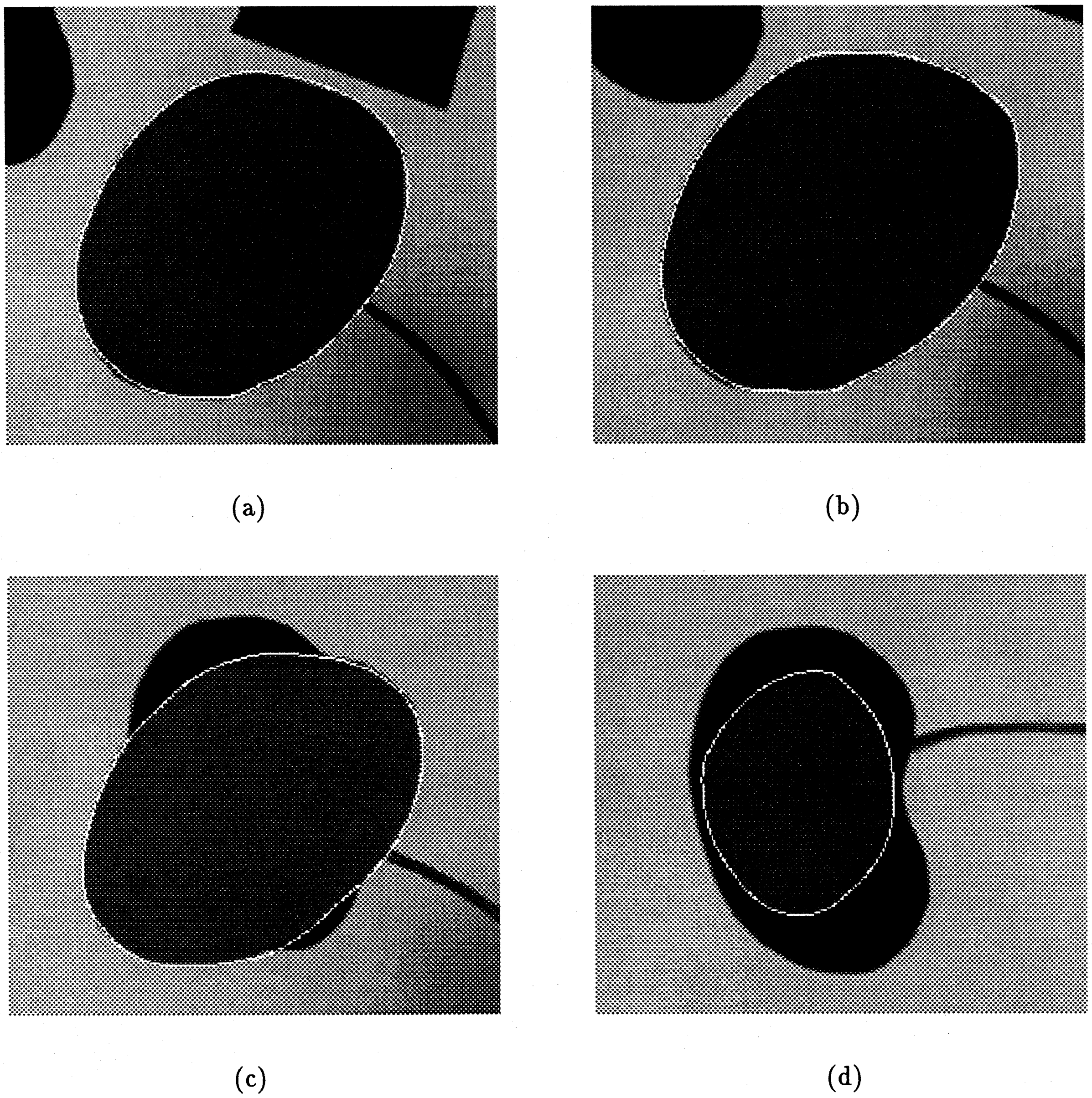


Figure 9. The fix. Some search windows include potential distractors, but they are ignored.

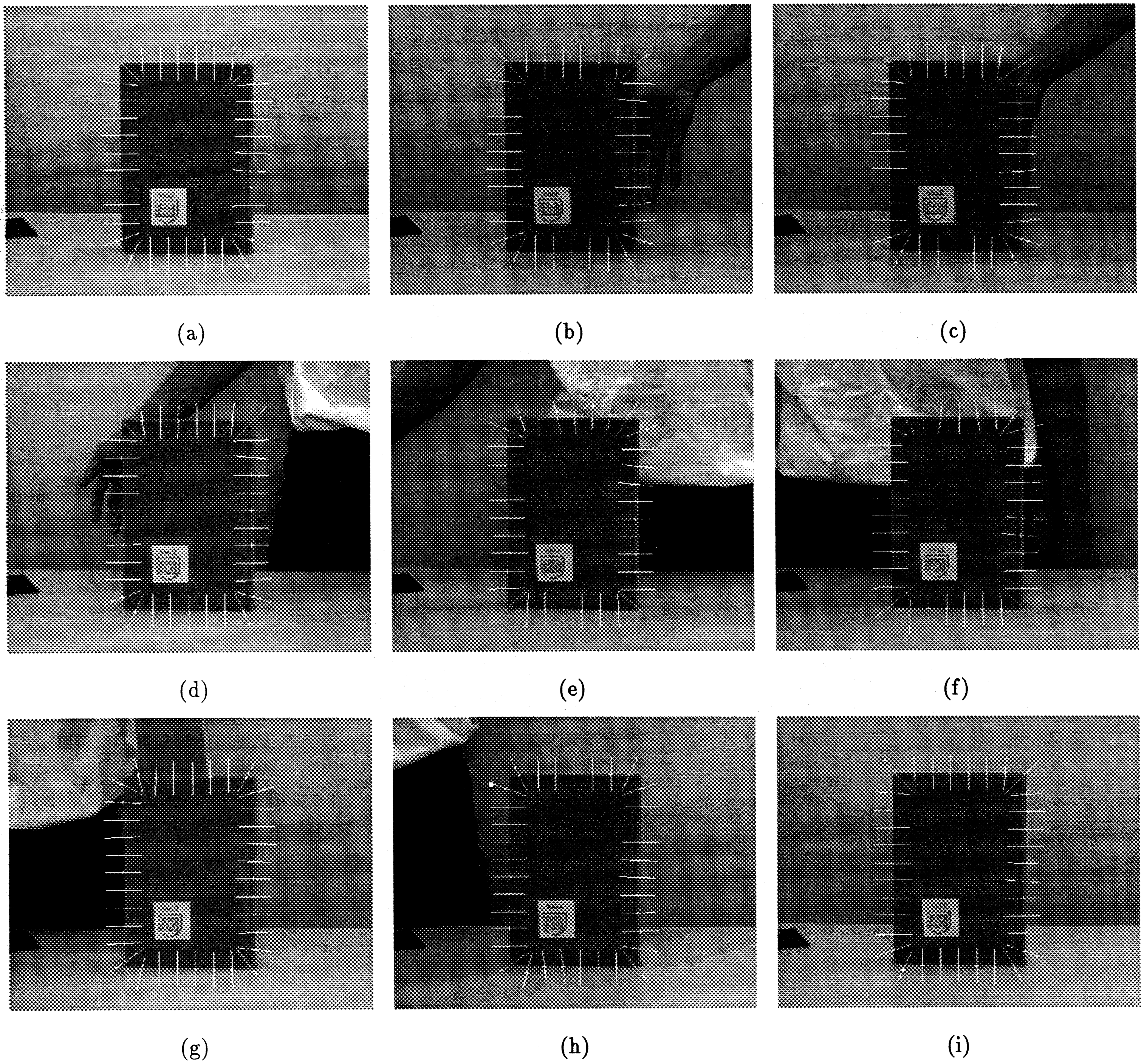


Figure 10. A contour of a book undistracted by movement of strong edges in background.

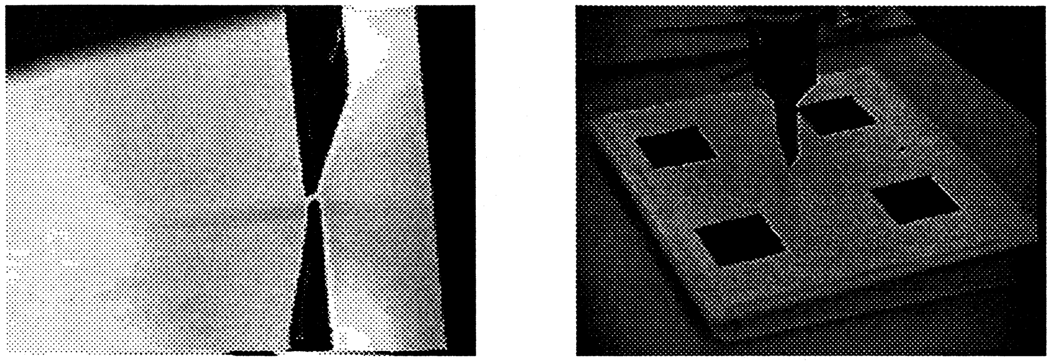


Figure 11. The results of performing point-to-point positioning to observable features (left) and to a setpoint defined in the plane (right).

Skill composition refers to ways of constructing more complex tasks from simple kinematic constraints. Two flavors of composition have been developed: kinematic composition, and geometric composition. The former refers to combining primitive kinematic constraints into more complex constraints. The latter refers to the combining visual information to arrive at more complex image-based constructions for achieving kinematic constraints.

Here, we illustrate two aspects of this composition: the use of typing to ensure program correctness, and the use of polymorphism to aid in the development of image constructions.

Positioning As noted above, point positioning uses an error term defined on two stereo points. The corresponding function is constructed as an `FeatureGroup` typed to manage two `StereoPoints`. In the simplest case, these points are constructed directly from point features such as corners that are directly apparent in an image. We have performed several hundred point-to-point positioning experiments of this type. With a camera baseline of approximately 30cm at distances of 80 to 100cm accuracy is typically within a millimeter of position. For example, Figure 11(left) shows the accuracy achieved when attempting to touch the corners of two floppy disks. For reference, the width of the disks is 2.5mm.

Positioning or orienting the robot in a plane is practically useful for systems which use a table or other level surface as a work-space. Planar positioning is also attractive because there are several image invariants that can be defined on planar points and lines [17]. In particular, given four planar

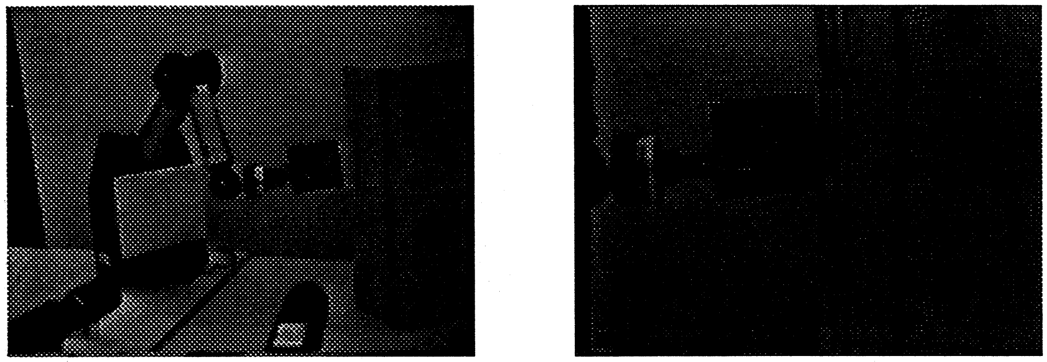


Figure 12. The robot inserting a disk into a disk drive. The slot is about 4mm in diameter, and the disk is 2.5mm wide.

points, no three of which are collinear, the coordinates of a fifth point can be constructed using ratios of determinants. This construction can be described as a `FeatureGroup` derivative which is typed as a point feature. As such it can be used as input to the same point positioning skill described above and as shown in Figure 11(right).

Positioning and alignment have been combined to perform the insertion of a floppy disk into a disk drive as shown in Figure 12. In this case, the operation was performed by moving a corner of the floppy to the edge of the slot, and simultaneously aligning the leading edge of the floppy to the slot by tracking the corners of the floppy and the outline of the slot in two images. This relies on information from two points and one fixed line. The relevant image information is supplied by trackers for the corners of the floppy and one for the macintosh slot. The floppy is tracked by two corner trackers which are point features. The floppy slot is tracked by locating one end of the slot and its edges. This tracker is typed as a fixed line.

Alignment of Surfaces of Revolution A locally cylindrical section of a surface of revolution has the property that the location of the "virtual" projection of the central axis can be computed from its occluding contours. It follows that an alignment operation using this virtual feature computed in two images can be used to align the central axis of a cylinder. This combination of image-level construction and hand-eye coordination has been used to perform the placement of a screwdriver on a screw (Figure 13). The setpoints are defined by tracking the sides of the screwdriver and

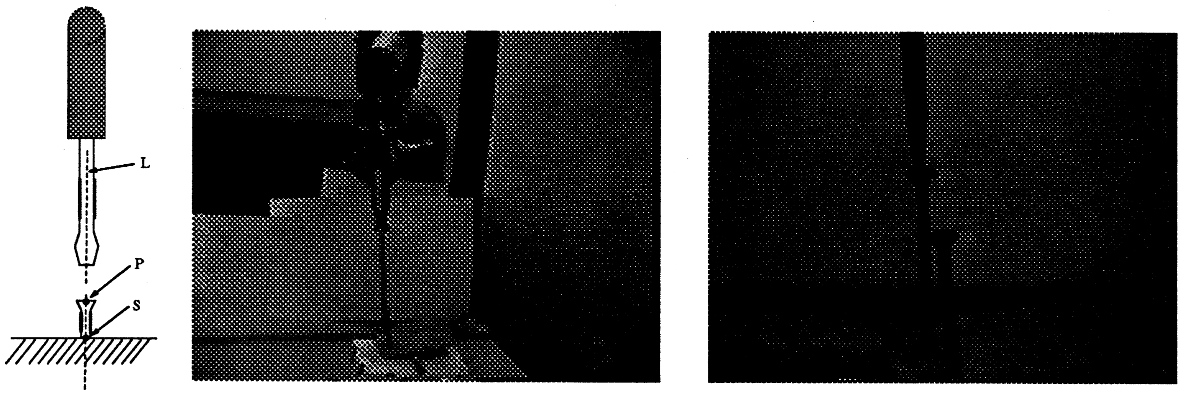


Figure 13. How tracking was used in aligning a screwdriver to a screw, and the results of a test run.

the sides of the screw to define their central axes. The tracker for the floppy slot defined above is reused to perform this tracking task. The accuracy of the resulting placement is typically within 2mm (Figure 13, right).

3.3 Mobile Robot Navigation

We are interested developing approaches to mobile robot navigation that utilize visual servoing. Our goal is to develop a system which can navigate through its environment by finding and memorizing trackable features. Motion through the environment will be expressed with respect to the motions and changes of the trackable features viewed by the system. Navigation and planning will be based on concatenating the visual records of motion paths and "replaying" them to move the robot toward a desired objective.

This is a challenging problem because it relies on feature selection as well as feature tracking. We have implemented feature selection for the SSD tracker as illustrated in Figure 14. However, this process is relatively slow and at first blush would seem to require global image processing.

We have developed a feature selection "tracker" that evaluates an image region for trackability. Feature selection within the tracker is then implemented as a `CompFeature` that maintains a `FeatureSelect` instance as well as one or more SSD trackers. The SSD trackers are used as "markers" for the best features found thus far. The tracking cycle of this special `CompFeature` is to first execute `FeatureSelect` on its current region of interest. If the region is found to be more trackable than the currently tracked features, one of the SSD trackers is initialized on the region. The SSD tracking cycles are then evaluated, and then the `FeatureSelect` instance is moved to the next region of interest. The result is that feature selection sweeps through the image *while tracking is occurring.* At the end of a sweep, the markers record the current location of the chosen features.

Currently, we have implemented both tracking and feature selection for this process, and have simple systems which record and replay motions using tracking information.

4 Conclusions

Our experience has shown that relatively simple tracking mechanisms combined with a well-defined notion of state composition makes the use of vision in robotics applications cheap, simple, and even fun. Both undergraduates and graduate students have used the system, and become proficient after a short "startup" period. Experts can easily develop and debug complex applications in a few hours time. By proper use of typing, more specialized notions of composition can be easily defined and supported within the system. The system has proved to be an ideal framework for comparative studies. It is straightforward to add other types of tracking primitives to the system, and benchmark them against existing methods on real images or canned sequences.

Clearly, simplified systems like this cannot perform every vision-based task. In particular, window-based processing of features will only succeed when the observed system is sufficiently well-behaved to be able to predict its motion through time. However, recent successes in domains such as juggling suggest that window-based techniques can accommodate highly dynamic systems [18].

Since the system is almost entirely software, it benefits from every increase in commercial processor speed. It is also extremely portable. The tracking system runs on an SGI Indigo with internal or external video, Sun systems equipped with a variety of framegrabbers, and PC compatibles equipped with digitizer boards. For example, the entire hand-eye system was recently ported to a robot at the DLR in Oberpfaffenhoffen Germany where it was used in experiments in space telerobotics [11]

In summary, we believe that this paradigm will have a large impact on real-time vision applica-

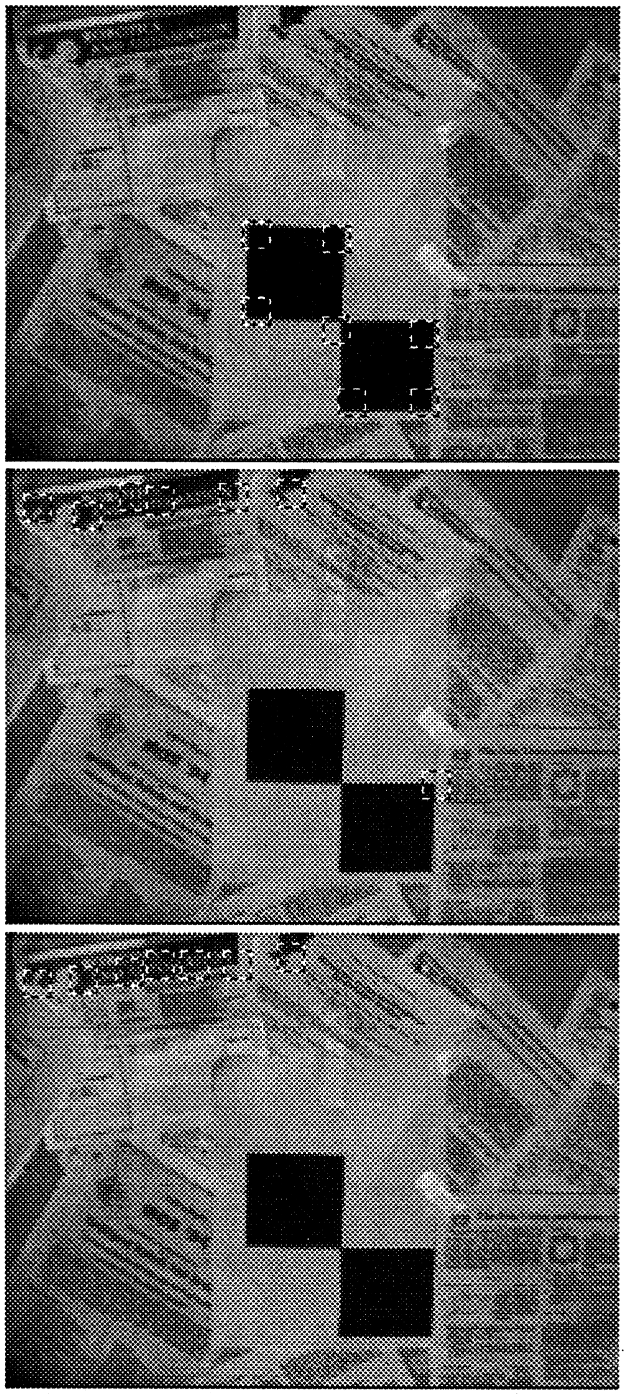


Figure 14. From top to bottom, the top seven features selected for pure translation, translation and one scale factor, and translation and two scale factors. As expected, strong corner-like features are chosen when only translation is important. These features are scale invariant, so the method chooses the more complex gray-scale patterns in the upper part of the image when both translation and scale are to be computed.

tions. We are currently continuing to advance the state of the art by considering how to build tracking methods that are faster [13] and more robust to occlusion and distraction [24]. We are also continuing to extend the capabilities of the system toward a complete vision-based programming environment. . Information on the current version is available at `http://www.cs.yale.edu/HTML/YALE/CS/AI/Vis`

References

[1] A.A. Amini, S. Tehrani, and T.E. Weymouth. Using dynamic programming for minimizing the energy of active contours in the presence of hard constraints. In *Proceedings, 2nd International Conference on Computer Vision*, pages 95–99, 1988.

[2] P. Anandan. A computational framework and an algorithm for the measurement of structure from motion. *Int. Journal of Computer Vision*, 2:283–310, 1989.

[3] A. Blake, R. Curwen, and A. Zisserman. Affine-invariant contour tracking with automatic control of spatiotemporal scale. In *Proc. Internal Conf. on Computer Vision*, pages 421–430. IEEE Computer Society Press, 1993.

[4] A. Blake, R. Curwen, and A. Zisserman. Affine-invariant contour tracking with automatic control of spatiotemporal scale. In *Proceedings, International Conference on Computer Vision*, pages 421–430, Berlin, May 1993.

[5] L.D. Cohen. On active contour models and balloons. *CVGIP: Image Understanding*, 53(2):211–218, March 1991.

[6] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics.* Addison Wesley, 1993.

[7] Gregory D. Hager. *Task-Directed Sensor Fusion and Planning.* Kluwer, Boston, MA, 1990.

[8] Gregory D. Hager. Real-time feature tracking and projective invariance as a basis for hand-eye coordination. In *Proc. IEEE Conf. Comp. Vision and Patt. Recog.*, pages 533–539. IEEE Computer Society Press, 1994.

[9] Gregory D. Hager. Six DOF visual control of relative position. DCS RR-1038, Yale University, New Haven, CT, June 1994.

[10] Gregory D. Hager, Wen-Chung Chang, and A. S. Morse. Robot hand-eye coordination based on stereo vision. *IEEE Control Systems Magazine*, February 1995. To Appear.

[11] Gregory D. Hager, Gerhard Grunwald, and Gerd Hirzinger. Feature-based visual servoing and its application to telerobotics. DCS RR-1010, Yale University, New Haven, CT, January 1994. To appear at the 1994 IROS Conference.

[12] Gregory D. Hager, Sidd Puri, and Kentaro Toyama. A framework for real-time vision-based tracking using off-the-shelf hardware. DCS RR-988, Yale University, New Haven, CT, September 1993.

[13] J. Huang and G. D. Hager. Tracking tools for vision-based navigation. DCS RR-1046, Yale University, New Haven, CT, December 1994. Submitted to IROS '95.

[14] T. S. Huang and C. H. Lee. Motion and structure from orthographic projection. *IEEE Trans. Pattern Anal. Mach. Intelligence*, 11(5):536–540, May 1989.

[15] M. Kass, A. Witkin, and D. Terzopoulos. Snakes: active contour models. *International journal of Computer Vision*, 1(1):321–331, 1987.

[16] B. D. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. In *Proc. Int. Joint Conf. Artificial Intelligence*, pages 674–679, 1981.

[17] J. Mundy and A. Zisserman. *Geometric Invariance in Computer Vision.* MIT Press, Cambridge, Mass., 1992.

[18] A.A. Rizzi and D.E. Koditschek. An active visual estimator for dexterous manipulation. Paper presented at the 1994 Workshop on Visual Servoing, 1994.

[19] J. Shi and C. Tomasi. Good features to track. In *Proc. IEEE Conf. Comp. Vision and Patt. Recog.*, pages 593–600. IEEE Computer Society Press, 1994.

[20] C.E. Smith, S.A. Brandt, and N.P. Papnikolopoulos. Controlled active exploration of uncalibrated environments. In *Proc. IEEE Conf. Comp. Vision and Patt. Recog.*, pages 792–795. IEEE Computer Society Press, 1994.

[21] C.J. Taylor and D.J. Kriegman. Algorithms for vision-based exploration. In K. Goldberg, D. Halperin, J.C. Latombe, and R. Wilson, editors, *The Algorithmic Foundations of Robotics.* A. K. Peters, Boston, MA, 1995.

[22] D. Terzopoulos and Szeliski. Tracking with kalman snakes. In A. Blake and A. Yuille, editors, *Active Vision.* MIT Press, Cambridge, MA, 1992.

[23] C. Tomasi and T. Kanade. Shape and motion from image streams: a factorization method, full report on the orthographic case. CMU-CS 92-104, CMU, 1992.

[24] K. Toyama and G. D. Hager. Distraction-proof tracking: Keeping one's eye on the ball. DCS RR-1046, Yale University, New Haven, CT, December 1994. Submitted to IROS '95.

[25] D.J. Williams and M. Shah. A fast algorithm for active contours and curvature estimation. *CVGIP: Image Understanding*, 55(1):14–26, January 1992.