STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM-214

STAN-CS-73-386

A CORNER FINDER FOR VISUAL FEEDBACK

BY

W. A. Perkins and T. O. Binford

SUPPORTED BY

ADVANCED RESEARCH PROJECTS AGENCY
ARPA ORDER NO. 457

August 1973

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM-214

AUGUST 1973

COMPUTER SCIENCE DEPARTMENT
REPORT NO. CS-386

A CORNER FINDER FOR VISUAL FEEDBACK

by

W. A. Perkins and T. O. Binford

Abstract: In visual-feedback work often a model of an object and its approximate location are known and it is only necessary to determine its location and orientation more accurately. The purpose of the program described herein is to provide such information for the case in which the model is an edge or corner. Given a model of a line or a corner with two or three edges, the program searches a TV window of arbitrary size looking for one or all corners which match the model. A model-driven program directs the search. It calls on another program to find all lines inside the window. Then it looks at these lines and eliminates lines which cannot match any of the model lines. It next calls on a program to form vertices and then checks for a matching vertex. If this simple procedure fails, the model-driver has two backup procedures. First it works with the lines that it has and tries to form a matching vertex (corner). If this fails, it matches parts of the model with vertices and lines that are present and then takes a careful look in a small region in which it expects to find a missing line. The program often finds weak contrast edges in this manner. Lines are found by a global method after the entire window has been scanned with the Hueckel edge operator.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract SD-183.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Goverment.

Reproduced in the USA. Available from the National Technical Information Service, Springfield, Virginia 22151.

A CORNER FINDER FOR VISUAL FEEDBACK

W. A. Perkins and T. O. Binford

ABSTRACT

In visual-feedback work often a model of an object and its approximate location are known and it is only necessary to determine its location and orientation more accurately. The purpose of the program described herein is to provide such information for the case in which the model is an edge or corner. Given a model of a line or a corner with two or three edges, the program searches a TV window of arbitrary size looking for one or all corners which match the model. A model-driven program directs the search. It calls on another program to find all lines inside the window. Then it looks at these lines and eliminates lines which cannot match any of the model lines. It next calls on a program to form vertices and then checks for a matching vertex. If this simple procedure fails, the model-driver has two backup procedures. First it works with the lines that it has and tries to form a matching vertex (corner). If this fails, it matches parts of the model with vertices and lines that are present and then takes a careful look in a small region in which it expects to find a missing line. The program often finds weak contrast edges in this manner. Lines are found by a global method after the entire window has been scanned with the Hueckel edge operator.

I. INTRODUCTION

Visual feedback is important in coordinating hand-eye manipulation. Because of the complexity and time involved in parsing a complete scene, it is often desirable to determine the location of a corner of a block or the position of the hand. Hence, there is a need for a visual corner-' finder. The early work at Stanford of **Wichman[1]** on a visual feedback system with primitive hardware and software support should be commended for its heroic effort in this new area. Shiral and **Inoue[2]** used visual feedback to place a block in a hole but found it necessary to keep the mechanical hand holding the block out of view so as not to complicate the visual analysis. Winston and **Lerman[3]** have described a corner-finding met hod which does not use lines as an intermediate step. Instead it uses correlations between **concentric** or osculating circles and they claim that this method has advantages for finding corners on textured surfaces. More recent work at Stanford in this direction has been done by **Sobel[4]** and **Tenenbaum[5]** culminating in Gill's **corner-**finding program **"GILEYE"[6].** Gill's program uses a contouring technique and thus can follow lines smoothly around the bend in a corner. (Edge operators tend , to have trouble in this region.) Gill's corner-finding

program was part of a visual-feedback system involving driver and arm-control programs. It was successful in precise positioning of blocks. However, the method used in finding corners has its drawbacks. It does not work well if there are several intensities inside the window. This means that window size must be small and it will not work well for cluttered regions. Further, it requires large contrast and most corners in the real world have three sides, whereas GILEYE can only find corners with two sides.

Because of these drawbacks we have written a new corner-finding program, called WALEYE. The program is written in SAIL[7] which is an extension of ALGOL. The objectives of this new program were: (1) ability to find corners with three sides as well as two; (2) ability to use windows of arbitrary size; (3) ability to find a particular corner in a cluttered region; and (4) high reliability even with marginal data and low contrast.

This program uses the Hueckel[8] edge operator for finding edge points in a scene. Then it finds lines with a global method that is particularly good for small windows and can work satisfactorily for arbitrarily large viewing windows. Vertices are formed by a local operator which uses simple heuristics. A model-driven program directs the

search. The model-driven program (1) calls for edge points and lines in the entire window; (2) eliminates all lines that cannot match any of the model lines; (3) calls for vertices to be formed; and (4) checks for a mat ching vertex.

The simple procedure outlined above works well for good corners in uncluttered regions but will often fail in the real world. The program was written with backup procedures to take care of these problems. Using its model of the corner, the program can destroy vertices and form test vertices which must satisfy vertex and model-matching criteria to be accepted. If this fails, the program uses the model to look for a missing line or lines. It does a careful scan over a small region and slightly relaxes its line-finding criteria.

Depending on the request, the program will return with the first acceptable corner that it finds or it will find all acceptable corners inside the window. If the location of a corner is poorly known, one has the option of using a large window or having the program look in a small window with an outward spiralling pattern of window centers.

From the results obtained so far the program seems to achieve the desired goals.

II. METHOD OF ANALYSIS

The problem of finding corners in scenes was separated into two major tasks: (1) finding edges, lines, and vertices from TV data and (2) using a given model of a corner to direct the search and make comparison tests with line and vertex data.

A. Line and vertex finding

The array of TV image intensities was first analyzed by application of an edge-finding operator. In this work we used an edge operator developed by Hueckel[8]. (See also Pingle and Tenenbaum[9]). When the operator was called to look for an edge at position (X,Y) it used the intensity points in a circle centered about (X,Y) and containing 26 points. If it found evidence of an edge anywhere in this circle, it recentered the circle on this edge and checked the intensity distribution in this new circle. If the edge operator found an edge, it reported back the X,Y location of the edge and the direction cosines of a vector perpendicular to the edge (pointing from lighter to darker intensities). In this work a specified window was scanned by applying the edge operator over a rectangular grid.

The next step was to determine which edge points defined lines and to find those lines. The method that we used to find lines has similarities to that discussed by Hough[10] and Griffith[11] and extended to normal parameters by Duda and Hart[12]. In the equation for a straight line

$$AX + BY + C = 0 \tag{1}$$

there are really only two free variables. Therefore we used the form

$$X\cos\alpha + Y\sin\alpha + C = 0. \tag{2}$$

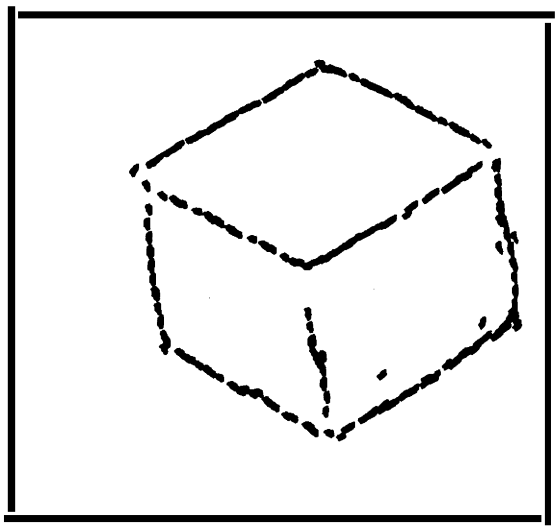
As noted by Duda and Hart every line in x-y space transforms into a point in the a-C plane. While they considered the edge points to be dots, our edge points (obtained with the Hueckel operator) are vectors each having an a and C value which makes this method even more attractive. The edge points are then stored in an array containing X,Y,α, and C for each one. The values of a vary over 2π (instead of π) in order to allow for intensity-direction information. Since edge points along a line should in principle all have the same a and same C values; they should transform into one point in a-C space. The edge

points from a parallelepiped (see Fig. 1 a) are shown as dots in a-C space (see Fig. 1 b). The edge points from each side do not form a point as in the ideal case but there is considerable clustering into short,' thick lines.

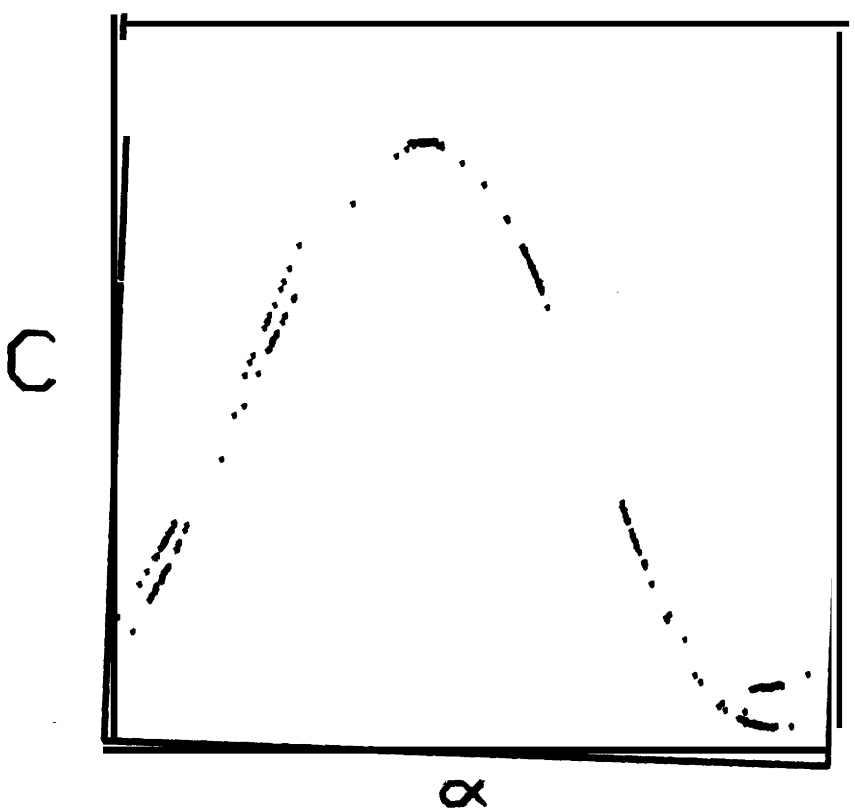
The scatter of points forming a line is not random because the error in C depends on the error in a. The uncertainty in C is caused mostly by the uncertainty in α and is given approximately by $dC = |\underline{n} \times \underline{R}| d\alpha$ where n is a unit vector perpendicular to the edge and $\underline{R}$ is a radius vector from the origin to the edge point. Thus edge points which are perpendicular to the radius vector and far from the origin have the largest uncertainty in C. Putting the origin at the center of the viewing window can reduce this uncertainty.

Note that two parallel lines with the same intensity direction lie close together in a-C space. By looking for sufficiently large groups of edge points as a function of a and then (for a particular a) as a function of C the average values of a and C for these clusters are determined.

In the early work a least squares fit was made to these average a and C values. This worked fine with good edge points. In this case the average values were close to the values determined by the least



(a)



(b)

Fig. 1. Clustering edge points to form lines. (a) Edge points of a parallelepiped in x-y space. (b) Same edge points in α-C space.

squares fit and there were no bad points to cause an erroneous least squares fit. The bad points can be eliminated from the fit by pruning, but this is tricky and requires a few cycles in determining the correct a and C values. Also this method is poor in principle since the acceptable range should not be a fixed parameter but should vary from line to line. With this method a long noisy line would often be found by the program as several overlapping lines. This was very unsatisfactory.

In the course of this work a much better method of using the residuals (the difference between the observed and calculated values) was found. Here

$$\text{Residual}(i) = X(i)\cos\alpha + Y(i)\sin\alpha + C. \qquad (3)$$

The most difficult part about a least squares fit is that one does not know a priori the correct line (α and C values) to fit the points to. However, it was discovered that the residuals for all the edge points that form a line would cluster even if they were fitted to the wrong line as long as the a and C values were reasonably close. Figure 2a shows a histogram of all the edge points from the parallelepiped of Fig. 1. The number of edge points in a bin are plotted versus their residual.

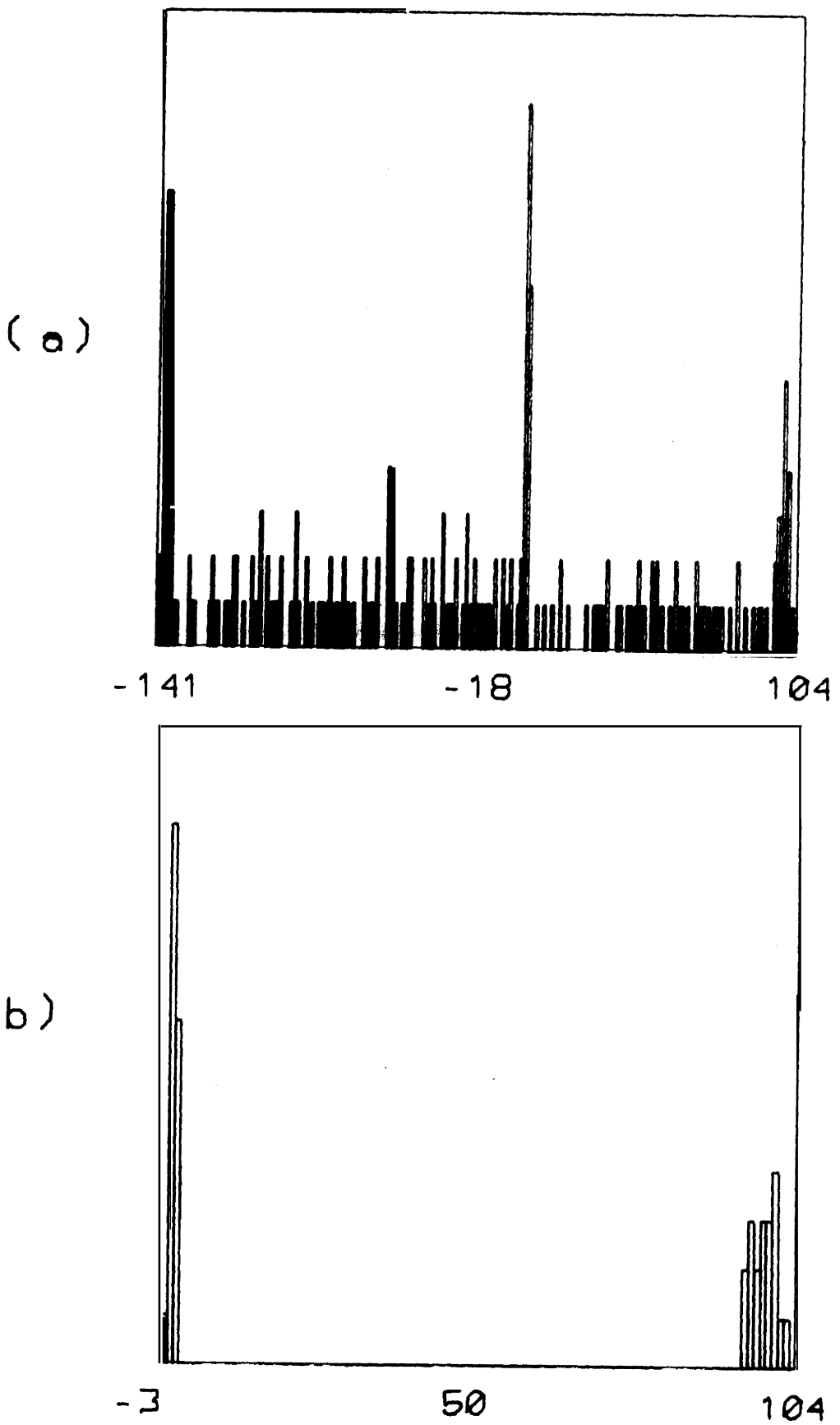


Fig. 2. Histogram of number of edge points versus their residual. (a) All edge points. (b) Edge points in a small a-C region.

. The three peaks correspond to three roughly parallel lines, one of which has the correct a and C values and is centered about a residual of zero (see Eq. 3). If we take only edge points from a small (about 10% of the a and C range) region of a-C space, we obtain the histogram shown in Fig. 2b. (Our a-C space window is large enough that very few good edge points are ever omitted in this step.) This contains edge points from two parallel lines which are close together in a-C space but far apart in residual space. Note that the edge points with residuals near 100 (see Fig. 2b) cluster although their residual is being computed for the wrong a and C values. This feature that one only needed to have approximate a and C values in order to eliminate unwanted points and to get the edge points forming a line to cluster made the calculations much easier. The errors in a and C determine the width of the cluster. If the initial values are far off this width can be reduced by iteration of the residual calculation with the new a and C values.

The points in each cluster were -examined further to separate points if large gaps were found. Five parameters were used in finding lines: two for the a-C space window, one for the least squares clustering criteria, two for the gap criteria, and one which was the

minimum number of good edge points for them to be accepted as a line. The results are not sensitive to the exact choice of the parameters. Each line is stored as a linear array with such quantities as a, C, $\sin\alpha$, $\cos\alpha$, number of edge points, chi value from the least squares fit, points along the line, gap information, and vertex information.

There is a vertex-forming procedure which uses local knowledge to extend lines and form vertices. It works very well in uncluttered scenes, but poorly in cluttered, complicated scenes. For typical scenes it forms the correct vertices in about 80% of the cases and does nothing at all or forms incorrect vertices for the other 20%.

The vertex former uses heuristics roughly as follows (at present it makes only one pass through all the lines). A circular region is formed at the end of an unended line (not centered about the end but starting at the end) in anticipation of a possible vertex near the center of the circle. First, the number of vertices and unended lines inside the circle is determined (lines just passing though the circle are ignored except for step 5 below). The following cases are considered:

1. If there is only one other line ending inside the circle, then three options are available. If their angles are close enough, then they are joined into one line. If the intersection is inside the circle, a vertex is formed. If neither of those situations apply, then the lines are left unchanged.

2. If there are two other lines ending inside the circle, a least squares fit is made of the three lines to a common intersection point. If all three lines pass close enough to that point, a three-line vertex is formed. If not, an attempt is made to form one line with either of the lines. If this fails, all lines are left unchanged.

3. If there is one vertex inside the circle, a test of the intersection points is made in an attempt to join this line to that vertex. If this fails, then the line is left unchanged.

4. If anything more complicated, such as three unended lines or two vertices is found inside the circle, the program balks and nothing is done.

5. If no ends of lines are found inside the circle, then the program looks for a line passing through the circle. If one is found and the intersection is inside the circle, then a junction is made.

With a model to guide the search these heuristics turned out to be' adequate.

B. Corner finding

The model of a corner that the program uses consists of:

1. the number of lines (1,2, or 3);

2. angular tolerance for the lines;

3. intensity direction if it is specified (i. e. which side of the line is darker);

4. unit vectors for each line pointing along the line toward the vertex.

(See Appendix A for more details.)

The search is perhaps best characterized by "top-down". The program is given a model as described above and a region or window in which to search. The corner-finding program first calls for a histogram of the intensity distribution inside the window to be sure there is something there to look at. If the intensity distribution is uniform, the search fails with a quick exit. If the intensity distribution is variable, the program calls for a uniform scan of the edge operator over the entire window. Next the program calls on the line-finder to find all the (straight) lines within the window using the edge data (as described in Section IIA).

Now for the first time the program has something which it can

compare with the model. Each line that was found is compared with all the model lines. If a line cannot fit any of the model lines, it is eliminated from the data. Two tests are applied at this stage:

1. If an intensity direction was given, it must match. This is easy to test for by using the angle of the model line (β) and the angle of the image line (α). These angles have a 2π variation and the choice of quadrant was determined by intensity information. For agreement it is simply required that,

$$|\alpha-\beta| < n/2. \tag{4}$$

2. The other test involves the angular tolerance. The tolerance (t) is given to the program as input and applies to all the lines. Lines are rejected if

$$|\cos(\alpha-\beta)| = |\cos\alpha\cos\beta + \sin\alpha\sin\beta| < 1 - t. \tag{5}$$

Typically we have used $\emptyset.1$ for t which corresponds to about 25 degrees angular difference. Corners are unique enough that this large angular tolerance is still satisfactory for typical scenes.

If the model is simply a line, the tests are just about over. If no

lines are left after the intensity and tolerance checks, the program exits with a failure. If more than one line is left, it returns the line which best matches the angle of the model line in the case that only one line is requested. If all lines are requested, it returns the lines in the order of best to worst angular match.

If we have a corner with two or three sides, the work has just begun. The program now calls on the vertex-former (described in Section IIA) to form all the vertices that it can. A direction test is now made for all vertices that were formed. Before vertices are formed, a horizontal model line extending to the right out of the vertex will match any horizontal line. However, after vertices are formed in the image, it will only match a line extending to the right out of a vertex. A "$\cos\beta$" and "$\sin\beta$" with direction toward the vertex are calculated for each model and each image line. Each image line in the vertex is now 'compared with all model lines. If

$$\cos\alpha \cos\beta + \sin\alpha \sin\beta < 1 - t \qquad (6)$$

the line is eliminated from the vertex. Vertices with zero or one line left after this test are destroyed. Separated lines are returned to their former state.

If a vertex has too few or too many lines after the direction test, it is ignored at this stage. If the vertex has the correct number of lines, it is examined further. Two image lines may have matched one model line so we now require each image line to match a different model line. If this matching works, the program succeeds and stores the pertinent information about the corner in a global array.

The search continues into a second phase if no matching corner is found or if all matching corners were requested. In this phase the program forms test vertices if there are enough lines. It sorts through all the lines to see if any combination can satisfy both the vertex criteria and match the model. It is in this second phase that the program corrects for any errors that the vertex former has made. The order of the tests was chosen so that wrong combinations should fail quickly. The program relaxes the vertex criteria slightly by changing the parameters. Corners found at this stage are not quite as definite as those found in the first stage. In the case that all matching corners are requested, the corners found in the second stage are listed after those found in the first stage.

If no matching corner is found in the second phase or if all matching corners were requested, a third phase is entered in which the program

starts looking for a missing line or lines. Consider the case of a three-sided corner. The program now checks all two-line vertices. Note that only matching vertices will be left at this stage as all non-matching lines were detached from their vertices in the direction-test of phase 1. The program matches the two lines in the vertex with the two model lines and thereby determines which line is missing. Now it does a fine-mesh scan of a small rectangular region with one end centered about the vertex, pointing in the direction of the missing model line. The difference parameter used by the edge operator is relaxed slightly to make it easier to find edges. The line-finding criteria are slightly relaxed also. The program now looks for lines with angles within the angular tolerance of the line that it is looking for. Any line thus found is matched against the model line it was looking for. If this match succeeds, it tries to form a three-line vertex as discussed above.

If the examination of two-line vertices fails to turn up a match (or if all matching corners were requested) the program examines unconnected lines. After discovering which model line matches the image line and checking to see if its vertex end is well inside the window, it does a scan at the vertex end of the line in the direction of one of the missing model lines. If a matching line is found after the first special scan, it makes another special scan looking for the other missing line.

III. RESULTS

We shall now discuss some results of this program in looking for typical corners. (The program can be run under the Stanford Hand-Eye System[13] and then it is logged in as one of several jobs. A driver program then runs the TV camera and gives this program the models of corners to find.) The first group of examples (Fig. 3-8) were chosen to illustrate general features of the program. The second group of examples (Fig. 9-16) were chosen to illustrate visual feedback in the mating of surfaces with straight edges.

Figure 3 shows a typical search for a three-sided corner. (These are some of the displays that appear on the console as the program searches for the corner.) Figure 3a shows the complete digitized picture of the scene. The window size in this example was 30X40 whereas the complete TV scene is 250x325. The model of the desired corner is shown in Fig. 3b. The intensity direction can be specified for any two lines of the corner and the arrows- point from lighter to darker intensities. The edge points that were found by the edge operator are shown in Fig. 3c. The edge points are indicated by short lines in the direction of the edge. The lines that were found from these edge points

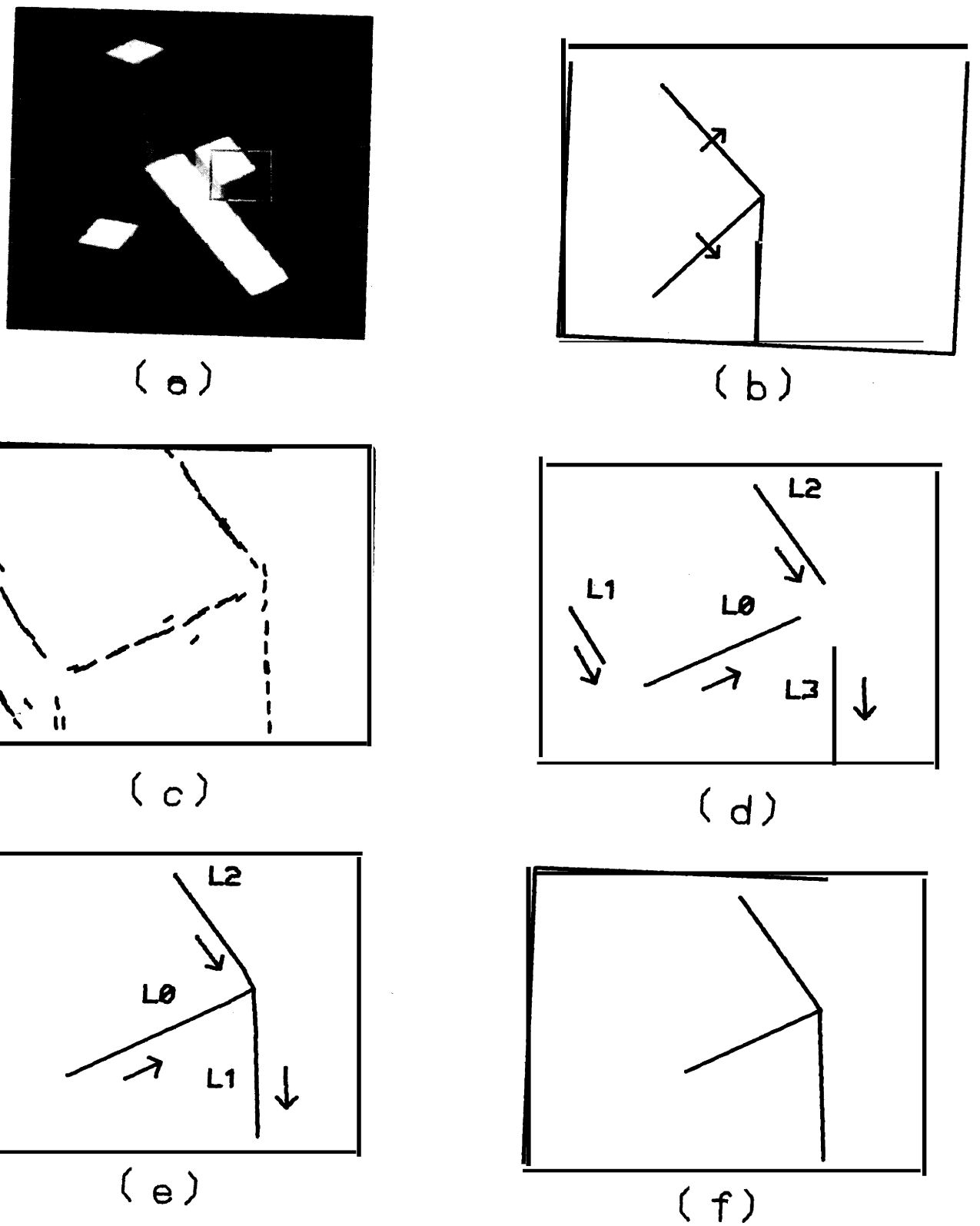


Fig. 3. Finding a three-sided corner. (a) Digitized picture showing window. (b) Model of corner with arrows pointing from lighter intensity region to darker region. (c) Edge points that were found. (d) Lines that were found. (e) Lines extended to form vertices. (f) Model of corner that was found. (Run time = 4 sec).

are shown in Fig. 3d. (Note the big gap that occurs near corners.) Line L1 is rejected and destroyed at this stage because it cannot match any of the model lines. Although L1 is parallel to L2, it has the wrong intensity direction. (Note that some lines are renumbered after lines are destroyed.) Next a local operator forms the vertex in Fig. 3e. The program finds a match with the model and stores the information describing the corner in a global array. In Fig. 3f the corner that was found is displayed from the information in the global array. The running time (without displays) to find this corner was 4 sec. Over half of this time was spent in finding edge points and half of the remaining time, in finding lines.

Figures 4 and 5 show the sequence involving a search for a three-sided corner in a cluttered scene. The window size here was set at 55X55 and we purposely made it difficult for the program by not indicating the intensity directions for any of the lines and by setting the tolerance at 0.3 so other nuisance lines were not rejected at an early stage. The edges and lines that the program found are shown in Fig. 4c and 4d respectively. The local operator forms some vertices as shown in Fig. 5a. (Note that lines sometimes have jogs in them as the program knows much more about lines than just their end points.) The

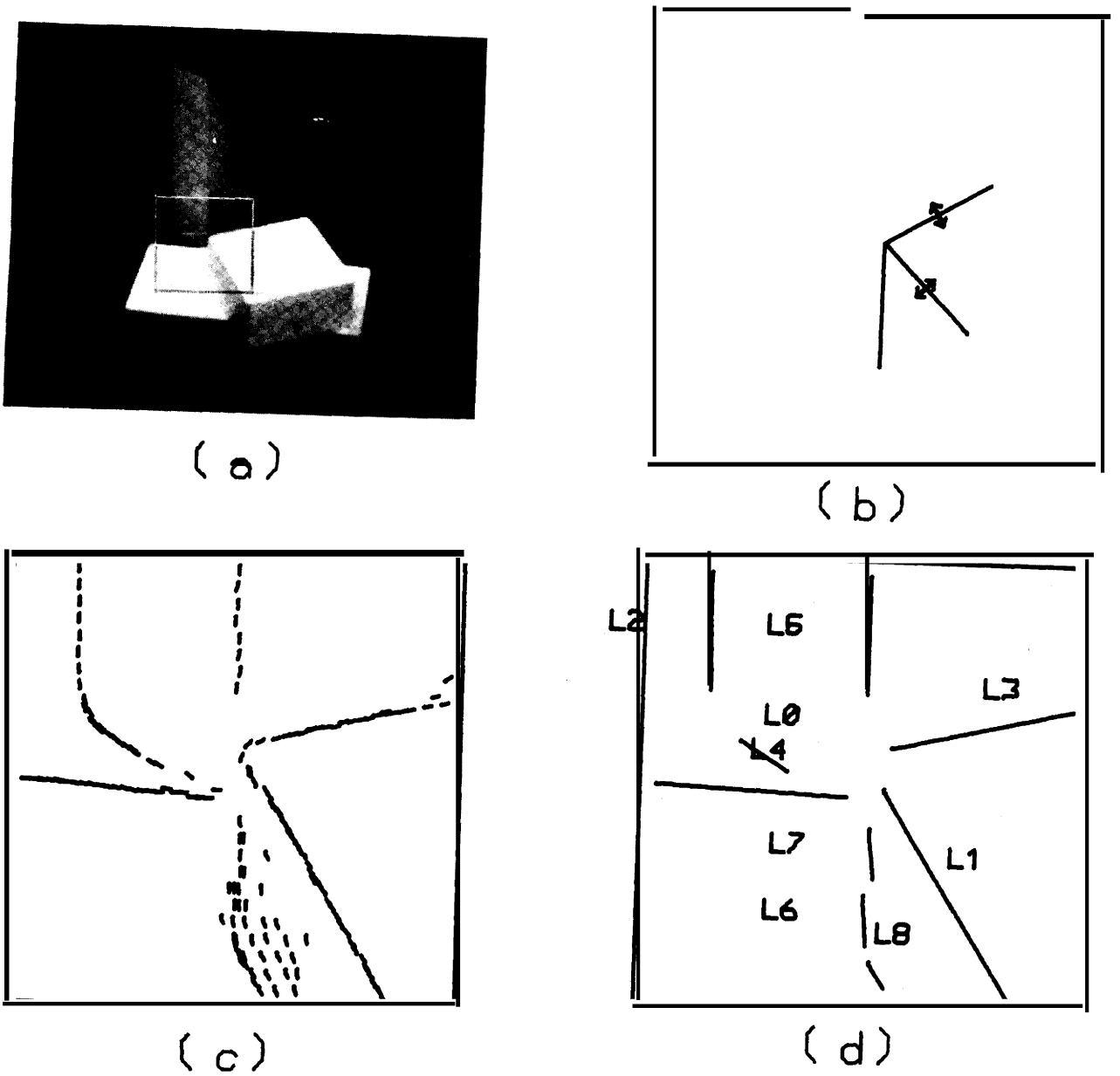


Fig. 4. Finding a three-sided corner in a cluttered region. (a) Digitized picture showing window. (b) Model of corner with double arrows indicating that either intensity direction is acceptable. (c) Edge points that were found. (d) Lines that were found.

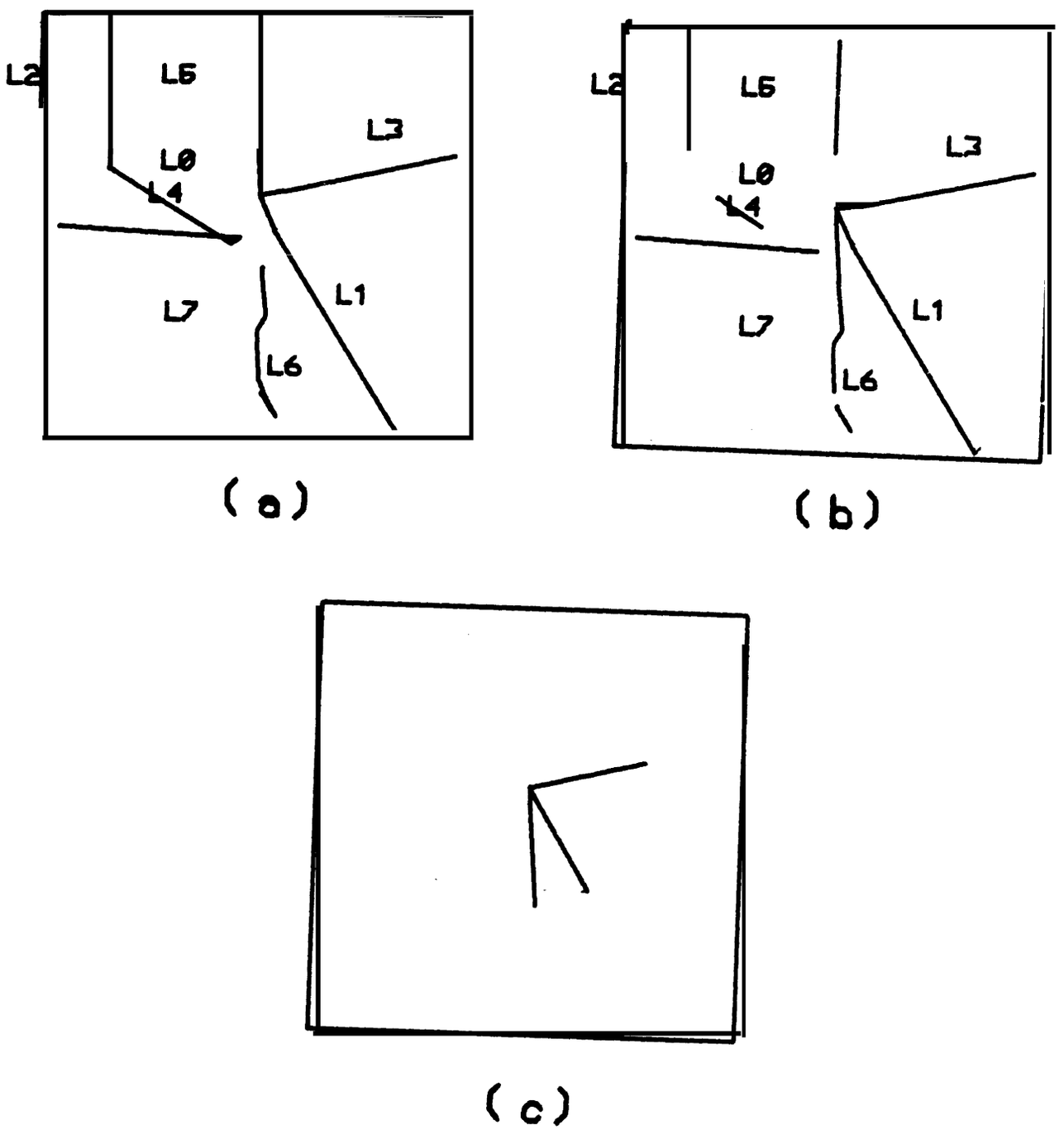


Fig. 5 Finding a three-sided corner in a cluttered region. (a) Lines extended and vertices formed. (b) Non-matching vertices broken and matching vertex formed. (c) Model of corner that was found. (Run time = 9 sec).

vertex-former failed badly here and line L5 rather than L6 is joined with L3 and L4. All vertex connections (except the one between L1 and L3) were broken by the direction test described in Section IIB. Mot having formed a matching vertex the program next sorted through all combinations of three lines. It found that lines L1, L3, and L6 satisfied both. the vertex criteria and the model-matching criteria and formed a new vertex as shown in Fig. 5b.

Figures 6 and 7 are another search for a three-sided corner and illustrate a problem that unfortunately occurs fairly often. There was a low contrast edge and the program found only two of the three lines that it was looking for as shown in Fig. 6e. Since no combination of three lines could form a correct vertex, it checked vertices with two lines. By matching the model lines with the vertex formed by L0 and L2 (see Fig. 6e) it determined which model line was missing and where that model line should be. The small rectangle in Fig. 7a shows the region in which the program did a more careful search for edge points and lines. Line L3 was found and a matching vertex was formed (see Figs. 7b,7c and 7d). In this special search the program ignores any lines which do not match the particular model line that it is looking for.

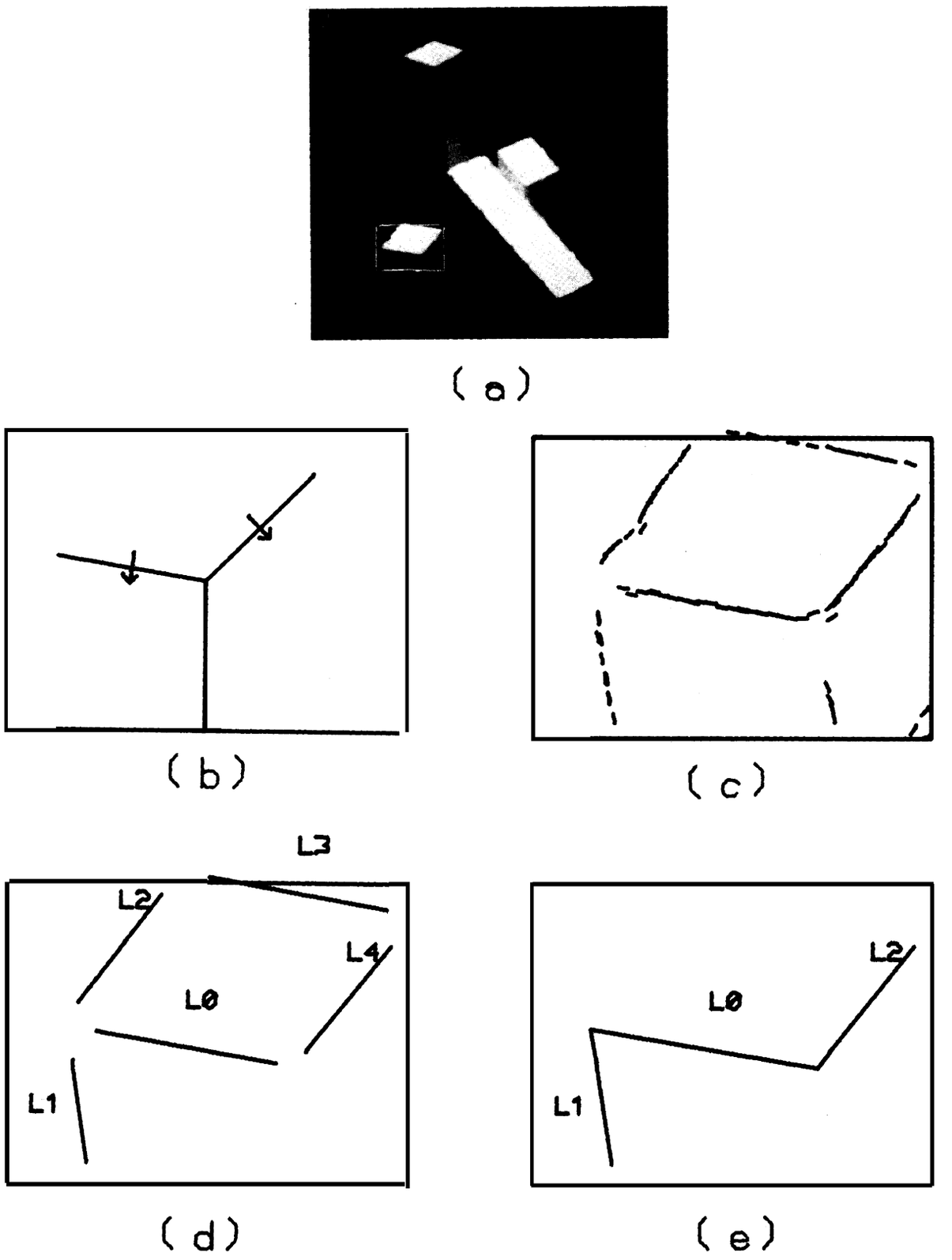


Fig. 6. Finding a three-sided corner with one low-contrast edge.
(a) Digitized picture showing window. (b) Model of corner.
(c) Edge points that were found. (d) Lines that were found.
(e) Lines extended to form vertices.

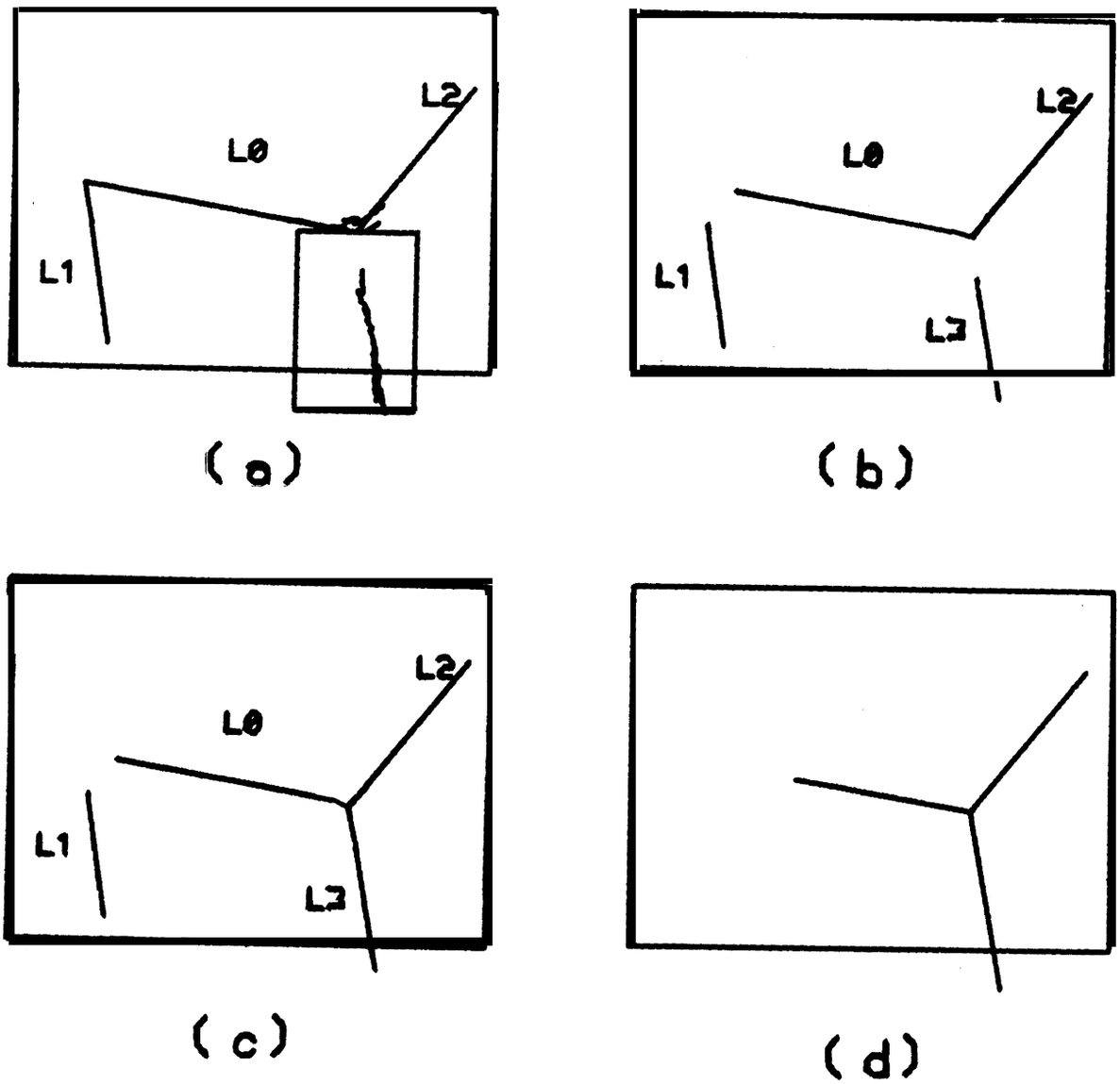


Fig. 7. Finding a three-sided corner with one low-contrast edge. (a) New edge points found In small window. (b) Lines. (c) Matching vertex formed. (d) Model of corner that was found. (Run time = 7 sec).

An example of the program finding a two-sided corner is shown in Fig. 8. Here again the program had difficulty finding a short, low-contrast edge, but a special search led to success.

One of the uses for a corner-finding program is visual feedback in the mating of surfaces with straight edges. The scene involving one block sitting on top of another block should be classified as a cluttered scene because six or more lines converge to the same vertex or nearby vertices. Earlier visual-feedback programs[1,2,6] stacked blocks successfully by requiring artificially high contrast scenes, ignoring interior edges, and/or remembering the position of the bottom block. In checking existing scenes and when the unexpected occurs, it may be necessary to have other capabilities. The present program differs from the earlier programs in that it can work with low contrast and interior edges, and in not requiring prior knowledge of the location of the bottom block.

This program is only intended as a corner-finder to be used in conjunction with another program which has information about the particular task for which the "hand" and "eye" are being coordinated. The program has no information about the semantics of blocks and shadows, for example. We took a set of seven pictures of typical combinations of one white block on top of another white block. In all seven cases we

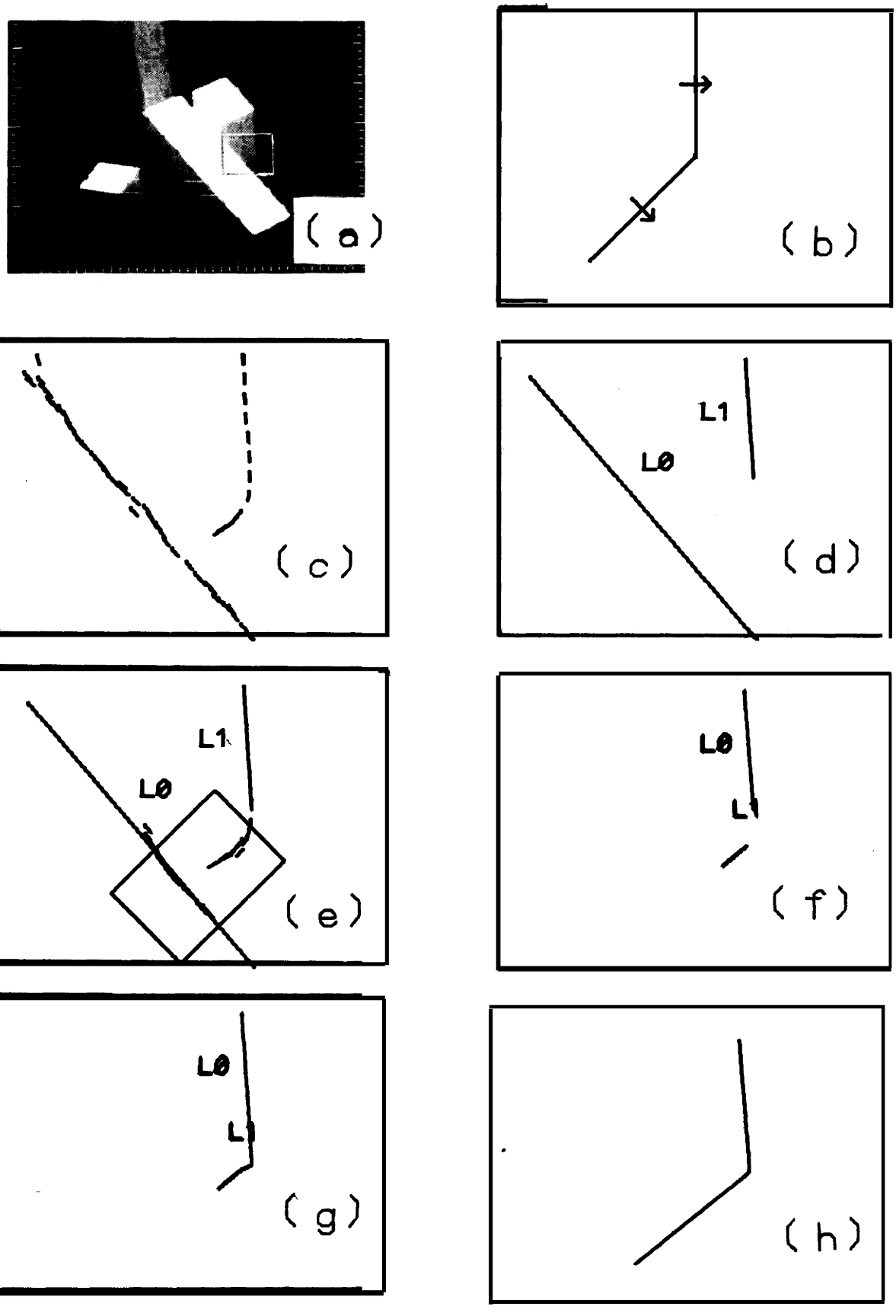


Fig. 8. Finding a two-sided corner. (a) Digitized picture showing window. (b) Model of corner. (c) Edge points that were found. (d) Lines that were found. (e) New edge points found in small window. (f) Lines. (g) Matching vertex formed. (h) Model of corner that was found. (Run time = 6 sec).

asked the program the same questions. We asked the program (WALEYE) to return all corners which matched the four model corners that are shown in Fig. 9. No intensity direction was specified in the models since this would vary with the alignment condition. However, intensity direction information is very useful as tops of block are bright and shadows are dark and WALEYE returned the intensity direction for each edge of the corner with a short arrow pointing from lighter to darker. This information could be very useful to a block-stacking program which called on WALEYE.

The digitized pictures of the blocks and the results obtained by WALEYE are shown in Fig. 10-16. The corner shown in the top left of Fig. 9 represented the bottom corner of the top block. Since this corner cannot be obscured by the presence of the bottom block, it should be found in all cases if the presence of the bottom block does not confuse the situation. (In all cases this corner was found, but in three cases an alternative corner was also found).

The view of the top corner of the bottom block depends greatly upon the position of the top block and if such a corner is found it may not be the correct corner of that object. Therefore, we look also for the two-sided- corners shown in the bottom of Fig. 9. This has

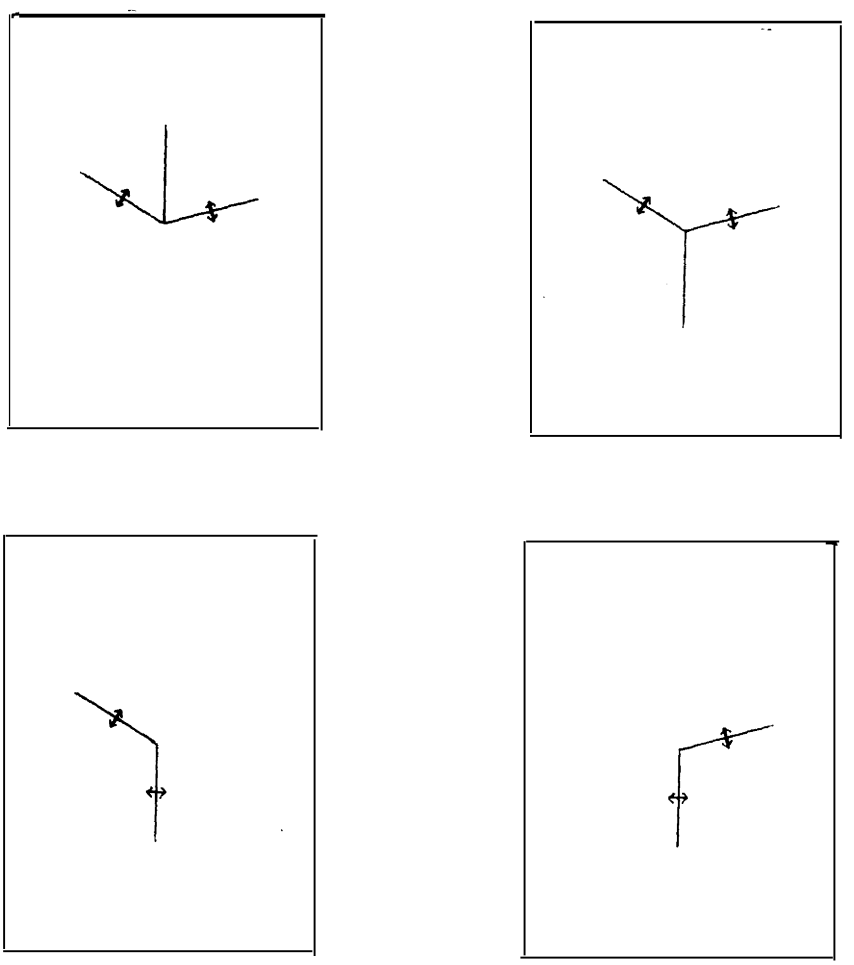


Fig. 9. Models of two three-sided corners and two two-sided corners used as input for the corner-finding program. The double arrows indicate that either intensity direction is satisfactory. The tolerance of angular match was set at 0.1.

advantages in that if one line of the corner is found, a careful search for the other line will be made (which was useful for two low-contrast edges in this set of seven pictures). In some cases the vertex found does indicate correctly the location of the bottom block.

In Fig. 10 and 11 the top and bottom corners that were found indicated correctly the location of the two corners. The two-sided corner data added further evidence that the correct corners were found. In Fig. 12 two three-sided corners are returned as candidates for the bottom corner of the top block. They differ only in the right- hand edge. One would naturally choose the higher edge so the location of the top block is determined. Although a three-sided bottom corner is not found, the location of the vertex and the orientation of the bottom block is clearly indicated by the two two-sided corners shown in the lower right of Fig. 12. In Fig. 13 the two edges on the right side are just slightly closer than the resolution limit of the edge operator. Again the program returns two choices for the bottom corner of the top block. Taking the corner with the higher left-hand edge results in the correct corner. The top corner of the bottom block was found correctly and this is confirmed by the two-sided corner data.

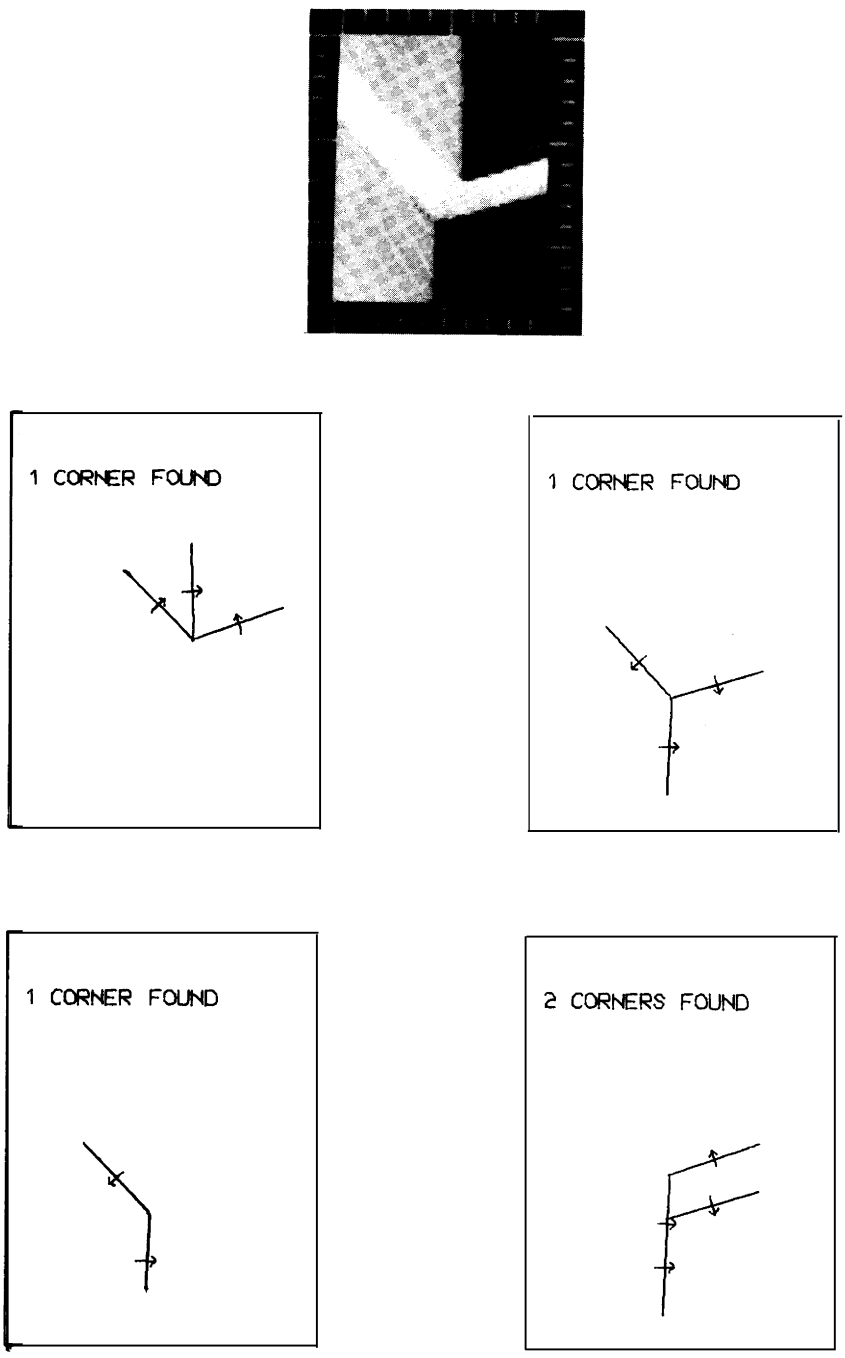
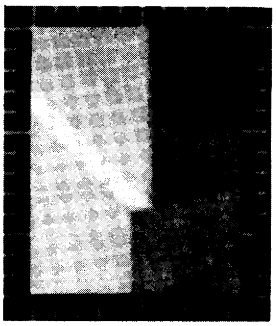


Fig. 10. Digitized picture of two blocks and the matching corners that the program found for the four model corners,



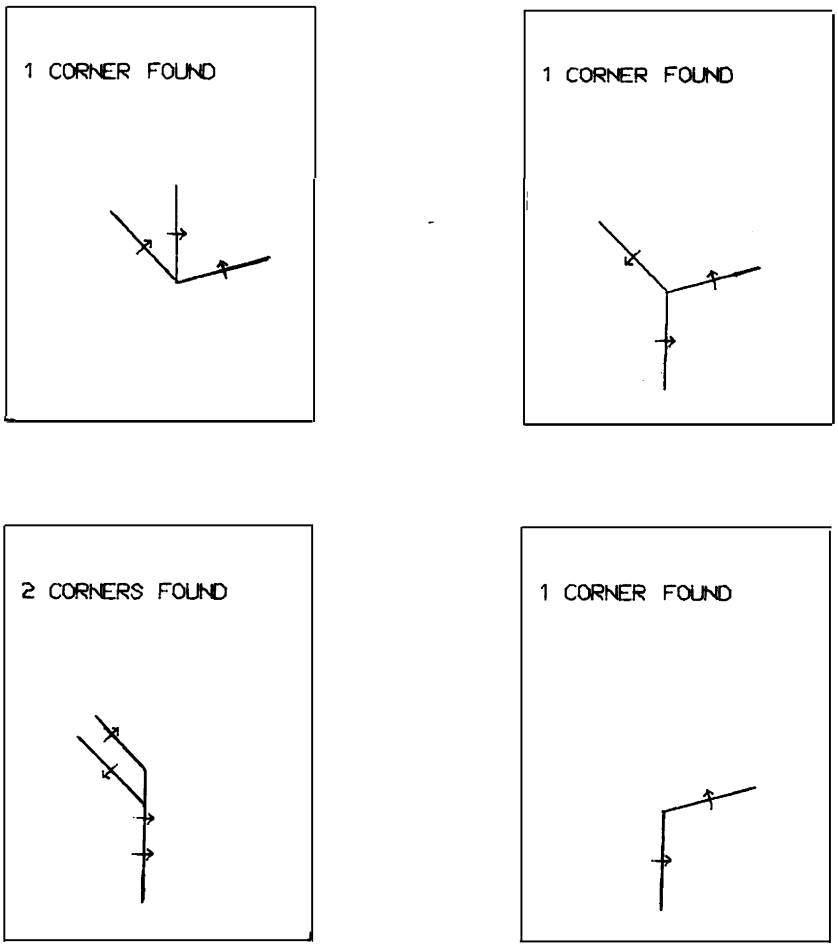


Fig. 11. Digitized picture of two blocks and the matching corners that the program found for the four model **corners.**

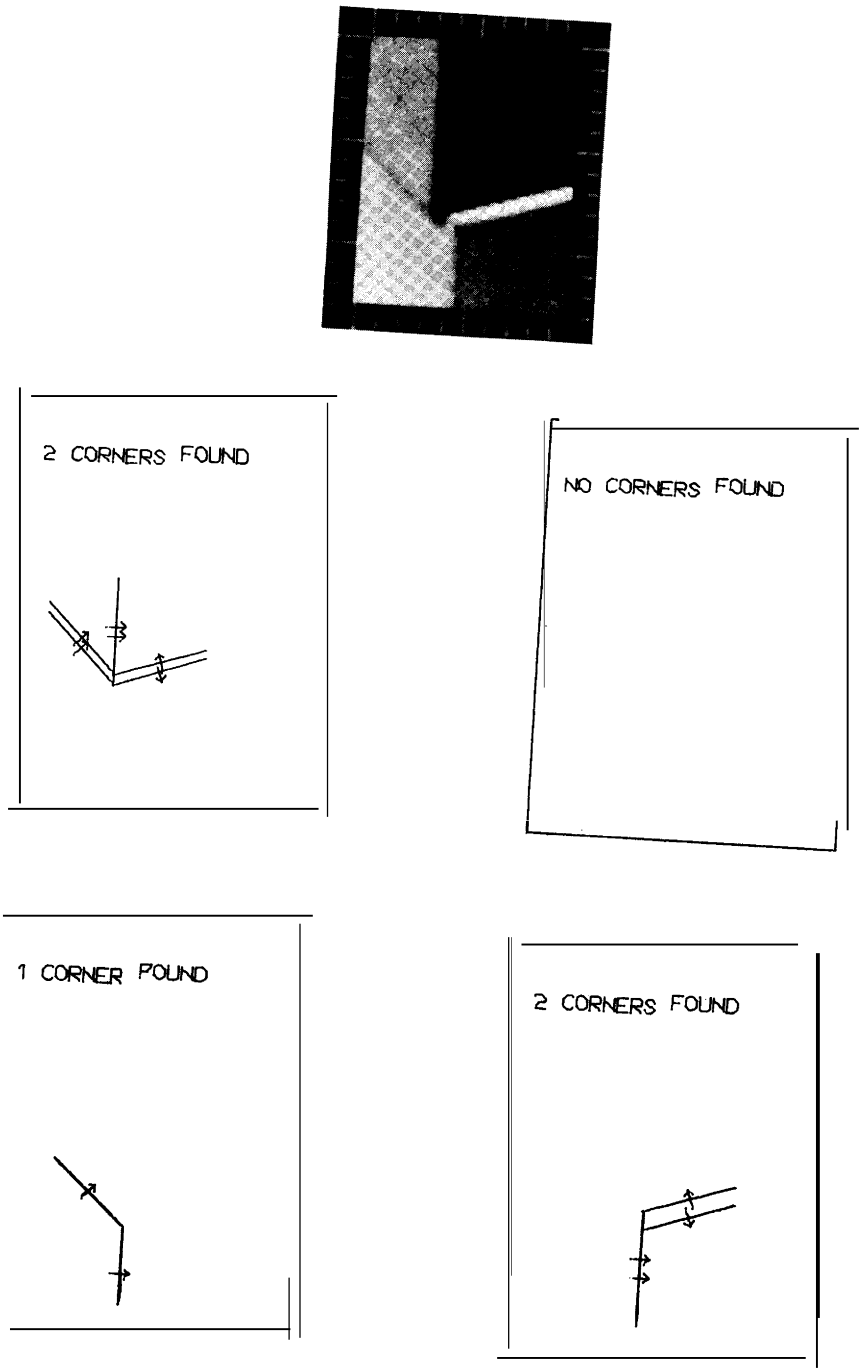
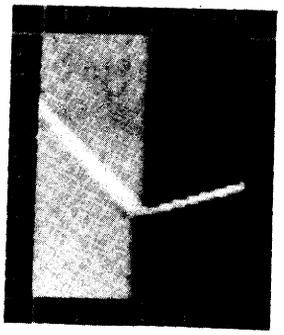


Fig. 12. Digitized picture of two blocks and the matching **corners** that the program found for the four model **corners.**



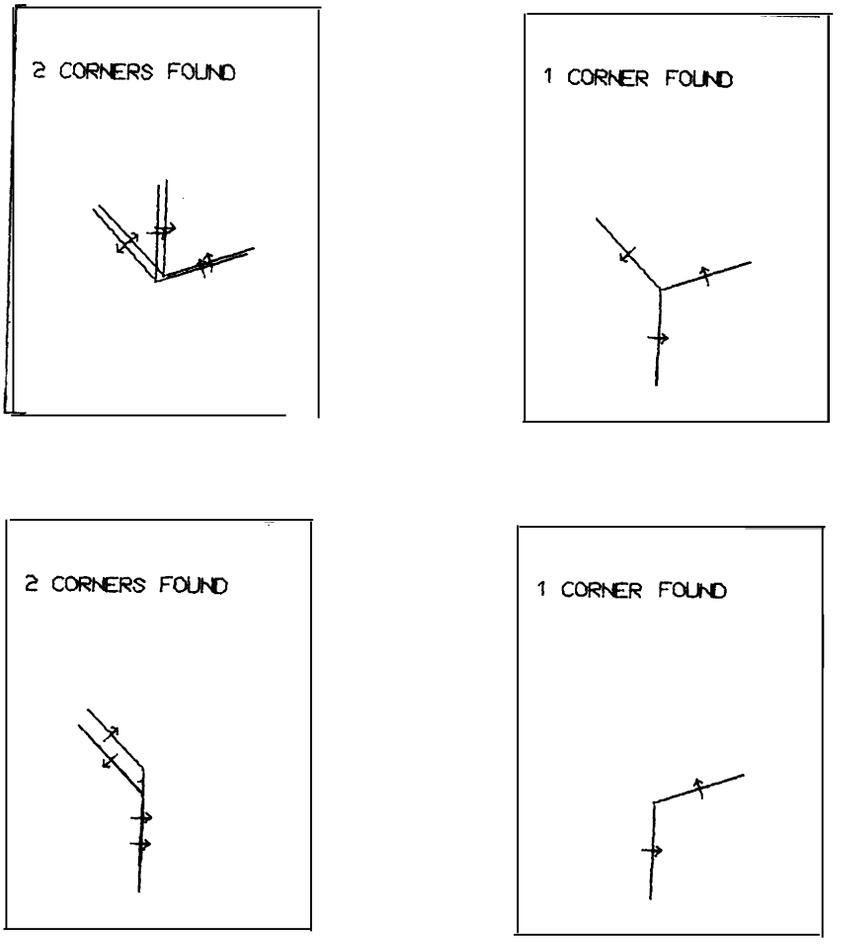


Fig. 13. Digitized picture of two blocks and the matching corners that the program found for the four model corners.

In Fig. 14 the bottom corner of the top block was found correctly . The top block obscured the right half of the lower block and neither a three-sided corner or a two-sided corner is found on the right half. However, the posit ion of the bottom block is correctly determined by the two-sided corner found in the lower left of Fig. 14. In Fig. 15 two choices for the right edge of the bottom corner of the top block are returned again. Taking the higher edge gives the correct locat ion of the top block. The program returned a consistent vertex locat ion for the top of the bottom block which would indicate a large uncovered area on its front-right edge. But tops of blocks are bright and this area is dark as indicated clearly by the intensity direction arrows in the lower-right of Fig. 15. Thus, the alternative explanation that the top block is extending far forward over the lower block causing shadows can emerge as the correct interpretation.

For the bottom corner of the top block in Fig. 16 the intersect ion of the three lines that the program finds (for- which only direction and vertex information is shown in the figure) is spread out farther than usual and the program labels this corner as a “POSSIBLE CORNER”. Nevertheless the program returns two three-sided corners for which the vertex

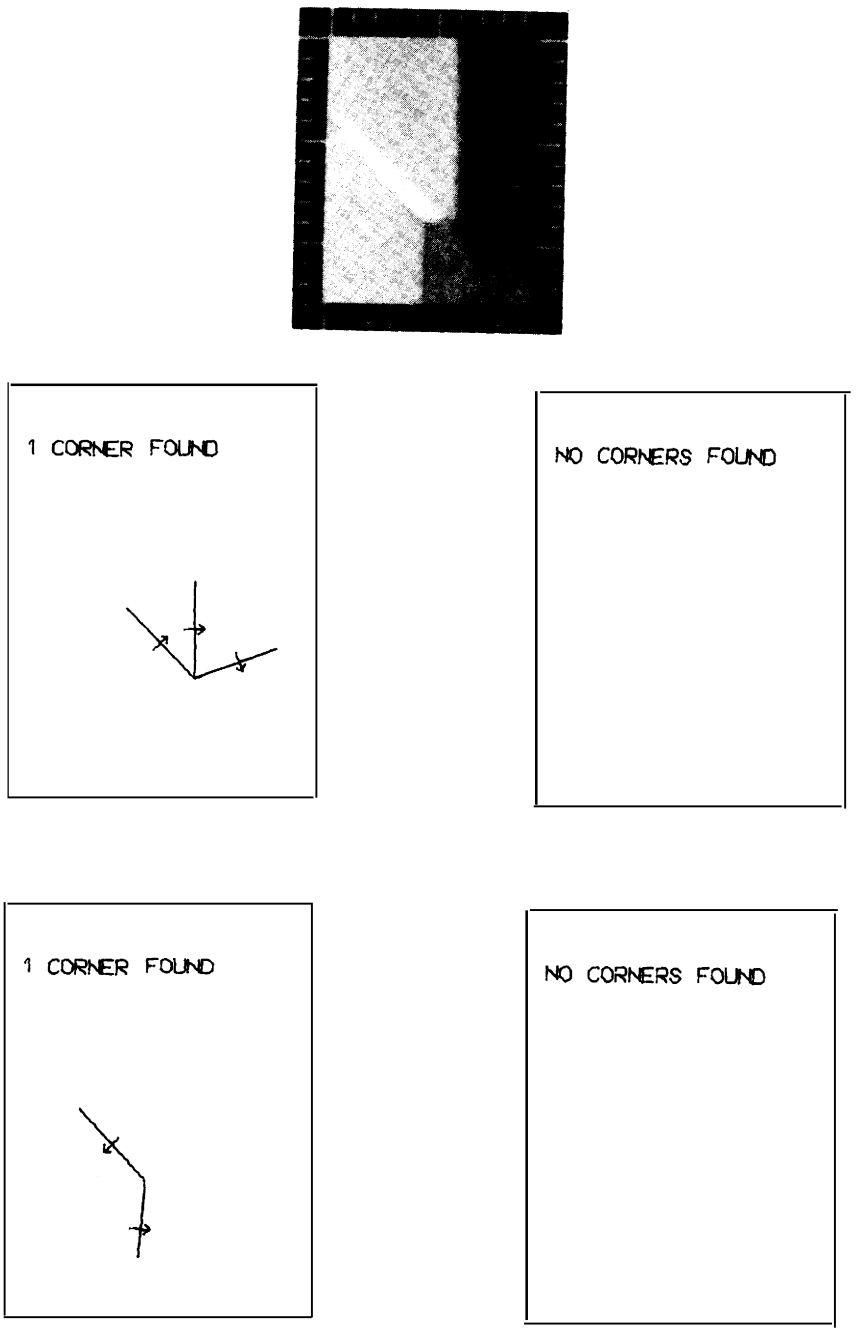


Fig. 14. Digitized picture of two blocks and the matching corners that the program found for the four model corners,

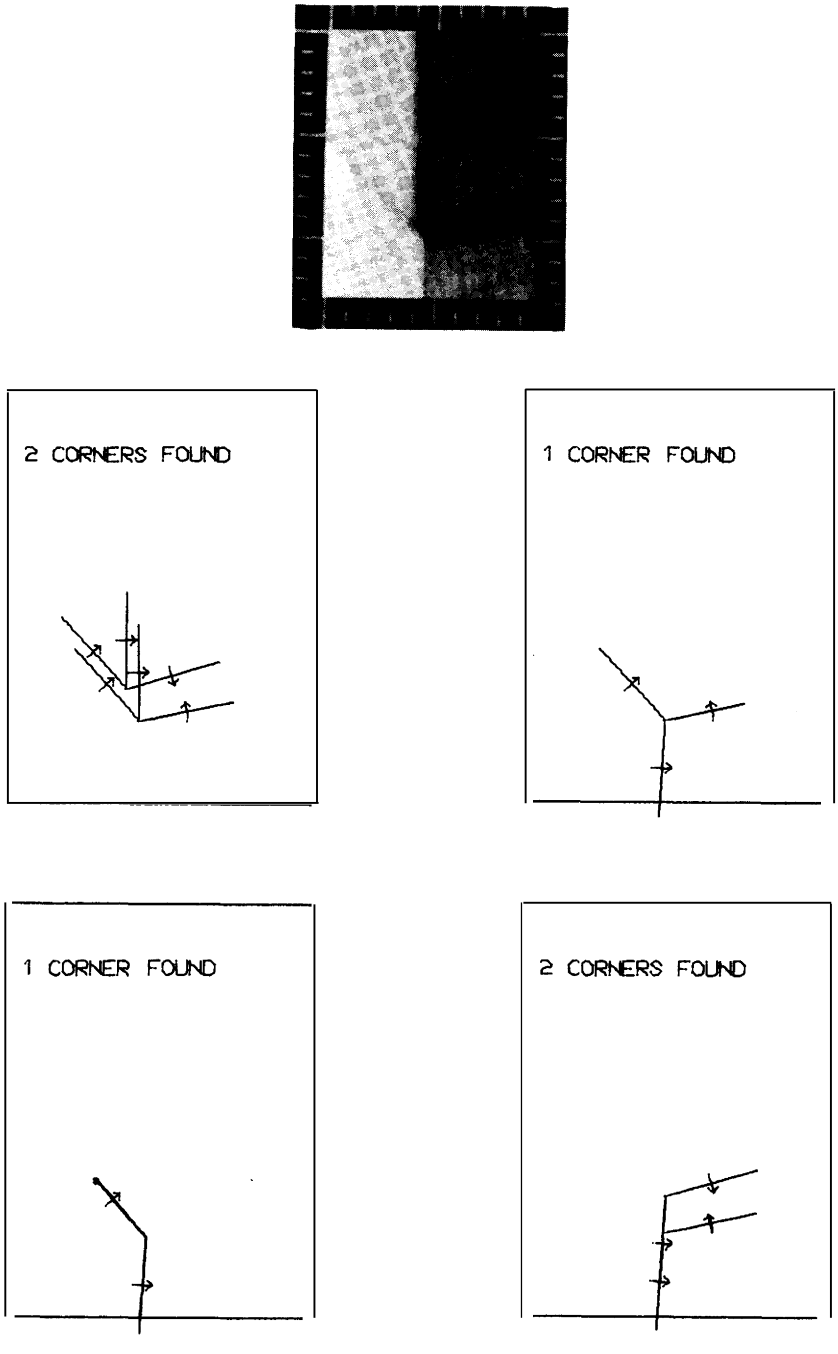
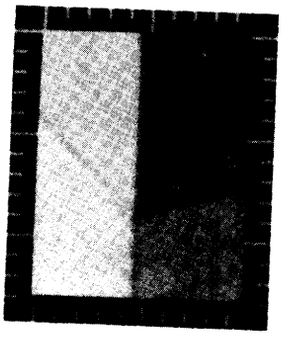
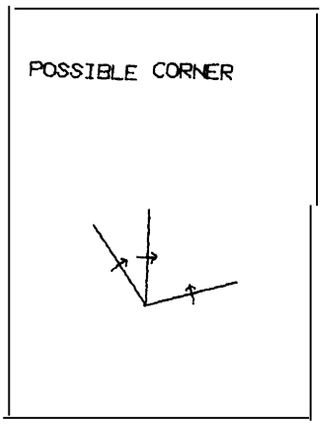
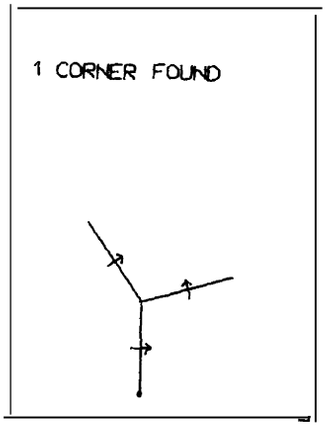
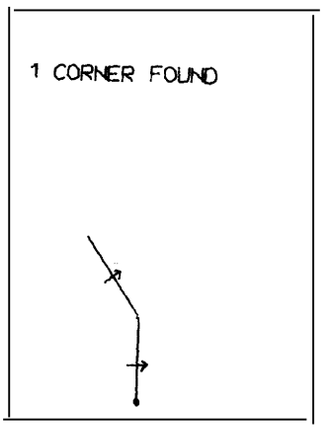


Fig. 15. Digitized picture of two blocks and the matching corners that the program found for the four model corners.









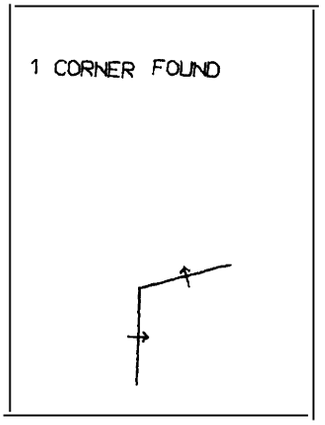


Fig. 16. Digitized picture of two blocks and the matching corners that the program found for the four model corners.

locations are very close together and which have two identical sides. (Note that the program has no trouble finding these thin line edges because the Hueckel operator can find line edges as well as step edges.) This strong evidence that the blocks are aligned is given support by the two-sided corners shown in the lower part of Fig. 16.

IV. CONCLUSIONS

A. Summary

Although robot arms can now do some assemblies blind[14], they will need feedback for difficult assemblies, inspections, and when unexpected problems occur. Some feedback' can be provided by tactile sensors[15,16] which at present are rather primitive. However, even if excellent tactile sensors were available, one would still want visual feedback for redundancy, to save time in cases for which visual feedback gives the answers more rapidly, and for information about objects which are in locations that are impossible to touch.

Although a program which analyzed a complete block scene[17] could be informative and would require less initial information about the

object that it is looking for, the programs which are presently available have some drawbacks. Firstly, they all require more time than is desirable for interactive manipulation. Secondly, even these programs require prior knowledge in order to make a 3-D determination of the location of a block. Thirdly, one would not like to turn them loose on a very complex scene such as one containing a large portion of the arm just to find the hand. Usually, one only wants to know the location of one or two objects and he wants the answer as quickly as possible. Hence visual-feedback programs tend to be specific in regard to the objects they are looking for and their location.

Earlier programs[2,6] were more restrictive in the degree of complexity that the scene could contain. The program discussed in this paper is far less bothered by background in the window and low contrast edges than the earlier ones.

The finite size of the edge operator limits the resolution of two parallel lines to about 3 times the spacing of intensity points (typically for objects at 1 meter and with a 25mm lens this is about a 1mm, but it can be made less if the TV camera zooms in on the region).

Being able to find three-sided corners is useful as most of the visible corners of blocks have three sides and a three-sided corner is

more nearly unique. In the cases tested (using a tolerance of 8.1) there was rarely an ambiguity in which two corners inside the window matched the model. When this is a problem, a higher level program can ask WALEYE to return all matching corners and then decide on the basis of higher level knowledge which of the corners is the one that it is looking for.

As discussed in Section IIB, there were two backup methods which became operative if the line-finder or vertex-former failed. If the vertex former failed, this was corrected by going through all combinations of lines and trying to form a matching vertex. If the line-finder failed, the program matched the vertices or lines that were found with the model and then did a more careful search for a missing line or lines. These backup procedures, especially the search for missing lines, turned out to be of great practical importance. Many times a low contrast edge was missed initially, but found later in the careful search of a small region.

In constructing this corner-finding program, we developed a line-finder. The initial evidence . for lines was obtained by a global method involving an α-C space projection. If there are only a few lines and a large number of random, noise-edge points (usually the number of background edge points is small) then the α-C space projection is perhaps the most efficient method of finding straight 'lines.

Given an initial guess for the α and C values of a line and a group of edge points, the residual clustering met hod will eliminate the background edge points and quickly give the correct a and C values. This method is also extendable to curves. Tests have shown that edge points of curves will cluster even if the initial guess is poor.

B. Suggestions for Future Work

The program does not fail very often, but when it does it is usually caused by the inability to find low-contrast edges. The program was tested with blocks that are painted white on all sides and therefore the variation of intensity across an edge is only caused by the difference in light striking adjacent faces. Some of the difficulty is caused by the poor dynamic range of digitized TV pictures and improved hardware would help here. Looking at digitized pictures, humans can still find the edges which the program missed, but apparently they do it by a global met hod rather than finding local evidence for the missing line. Program which use global methods can also find missing lines without local evidence[17]. A program with knowledge of the semantics of opacity, shadows, and physical models would certainly have some advantages.

However, to avoid the problems of present general scene analyzers for visual feedback, we need a program which can start in a prescribed part of the scene looking only for the designated object. If should use local knowledge if this is adequate to complete the task. If not, it should bring in larger areas of the scene and use global knowledge about complete objects (such as blocks) to help it find a corner.

By using WALEYE one can determine the orientation of a block by setting the tolerance t >1 and asking for all two-sided corners and three-sided corners inside a window which should encompass the block. With this information one could have another program figure out the orientation of the block. However, one could develop a better block finder by changing the model to simple and compound line features such as those used by Grape[17]. Almost all programs which analyze visual scenes work with planar objects such as blocks, and the program discussed in this paper is no exception. A useful extension would be for a visual-feedback program to find simple curved objects such as rods and holes.

ACKNOWLEDGMENTS

We would like to thank Karl Pingle, Robert Bolles, and Randy Davis for helpful discussions. The interest and encouragement of Professor Jerome Feldman is deeply appreciated.

APPENDIX A. Using the Corner-Finding Program.

The corner-finder exists as three separate programs: (1) WALDSK which finds corners in disk pictures; (2) WALEYE (similar to GILEYE) which is run as one job under the hand/eye monitor in conjunction the camera program; and (3) WALDRV (similar to PLAYGI) which is a simple driver program for WALEYE. The last two programs (WALEYE and WALDRV) and the camera program can be called by using the DO-FILE WAL. CMD[SYS,HE]. We shall first describe how to use these programs, then how one can use his own driver program instead of WALDRV to interact with WALEYE. WALDRV uses global arrays and one message procedure call to communicate with WALEYE, and one can easily incorporate these calls into his program which may move the hand, for example.

********** WALDSK **********

The following commands will cause this program to find the corner (with displays) that are shown in Fig. 3. After you log in, type the lower case commands and the computer will type back the upper case words. Comments are enclosed in curly brackets.

run waldsk[sys,he] <cr> (Type this at monitor level. }

SEGMENT LOGICAL NAME?

abc <cr> {Any 3 to 6 characters}

SEGMENT FILE NAME?

<cr>

DEVICE?

<cr>

INPUT FILE=

blk4. dat[lib,he] <cr>

DO YOU WANT CALCOMP PLOTS? Y OR N?

n

THE STANDARD OR PRESENT WINDOW SIZE IS:
WIDTH=40 HEIGHT=30 DO YOU WANT TO CHANGE IT? Y OR N?

n <cr>

ARE YOU LOOKING FOR A LINE,SIMPLE CORNER(2 SIDES),
OR PHYSICAL CORNER(3 SIDES)? [TYPE L,C, OR P]

P <cr>

ONE LINE WITH 5 NUMBERS THE FIRST TWO
DESIGNATES THE VERTEX IN SCREEN COORDINATES WHILE THE OTHERS GIVE
THE ANGLES OF THE LINES IN HOUR-HAND VALUES IN A CLOCKWISE MANNER
ABOUT THE VERTEX.

170 128 10.5 6 7.5 <cr>

ONE LINE WITH 3 WAL PARAMETERS.

FIRST IS INTENSITY:
-1 FOR DARKER OUTSIDE
0 FOR UNKNOWN
1 FOR LIGHTER OUTSIDE
SECOND IS SEARCH:
-1 FOR USE WINDOW ONLY
N≥1 FOR SEARCH IN AN AREA OF DIMENSION
N(INTEGER) ABOUT INITIAL WINDOW
THIRD IS TOLERANCE OF ANGLE MATCH:
USUALLY 0.1 BUT DOWN TO 0.02 WITH CARE.
IF TOLERANCE IS >1 THEN ANY CORNER
WILL BE ACCEPTED.

1-1 0.1 <cr>

(A model of the corner that you requested is displayed. See Fig. 3b.}

<cr> <cr>

CENTER OF THE WINDOW 170.0 128.0
ARE YOU SATISFIED WITH THE MODEL?
Y OR N?

y <cr>

DO YOU WANT ALL CORNERS THAT MATCH Y OR N?

n <cr>

(The program runs until a corner is found (or failure in some cases)
giving displays along the way. See Fig. 3. }

<cr> <cr>

(Program displays both input model and corner that was found. }

<cr> <cr>

DO YOU WANT TO CHANGE THE THREE WAL PARAMETERS ? --- Y OR N

n <cr>

DO YOU WANT TO CHANGE THE MODEL? --- Y OR N

n <cr>

DO YOU WANT A NEW INPUT FILE? --- Y OR N

n <cr>

END OF SAIL EXECUTION

********** WALEYE and WALDRV **********

The following is a typical sequence of operation using the TV camera in looking for a corner. Turn the Cohu or Sierra camera on and point it at some blocks on the table. (See PLAYGI. SAI[1,RCB] for a simple discussion of how to operate the camera.) The operator's commands are in lower case with the computer' s responses in upper case letters. Comments are again in curly brackets.

do wal. cmd[sys,he] <cr>

{ This is a short DO-FILE :
R HE↔
BB↔↔↔
:DISKIN WEC. CMD[SYS,HE]↔
:ECDRUN↔

and WEC. CMD is:
:DEFINE ECDRUN
EYE:LOG
CAM:LOG

```
                                        DRV:LOG
                                        EYE:RUN WALEYE[SYS,HE]
                                        CAM:RUN CAMERA[SYS,HE]
                                        DRV:RUN WALDRV[SYS,HE] }.


 SEGMENT LOGICAL NAME?
 SEGMENT FILE NAME?
 DEVICE?
 MON+RUN 29        {MON+ or EYE+ means that the MONITOR or EYE program
                     is talking to you. }
ECDRUN DEFINED
 END DISKIN
 EYE LOGGED IN AS JOB 30
 CAM LOGGED IN AS JOB 39
 DRV LOGGED IN AS JOB 48
                     (Job numbers will vary and sometimes you will
                      not get enough new job numbers as the job
                      capacity is exceeded, }
 END MACRO
EYE+SEGMENT LOGICAL NAME?
CAM+SEGMENT LOGICAL NAME?
DRV+SEGMENT LOGICAL NAME?

{You type) EYE;BB   {You must type upper case here!)

EYE-ACTIVATED

(You type} CAM;BB   (You must type upper case here!}

CAM+DATXFR: RETRIEVING DATA[SYS,HE]1
DATXFR: RETRIEVING DATA[SYS,HE]2
DATXFR: RETRIEVING DATA[SYS,HE]3
DATXFR: RETRIEVING DATA[SYS,HE]4
CAM-ACTIVATED

{You type) DRV;BB   (You must type upper case here!}

DRV+TRYEYE WAITING
    TRYEYE READY
  WHICH CAMERA ARE YOU GOING TO USE?
       1 FOR COHU --- 2 FOR SIERRA
```

```
1       <cr>

TYPE A <CR>
        WHEN YOU ARE SATISFIED WITH THE CAMERA POSITION

        <cr>

THE STANDARD OR PRESENT WINDOW SIZE IS:
        WIDTH=40 HEIGHT=30 DO YOU WANT TO CHANGE IT?   Y OR N?

y       <cr>

WIDTH AND HEIGHT
                EACH FOLLOWEO BY A CARRIAGE-RETURN

60      <cr>
60      <cr>

THE STANDARD OR PRESENT WINDOW SIZE IS:
        WIDTH=68 HEIGHT=60 DO YOU WANT TO CHANGE IT?   Y OR N?
n       <cr>

ARE YOU LOOKING FOR A LINE,SIMPLE CORNER(2 SIDES),
        OR PHYSICAL CORNER(3 SIDES)? [TYPE L,C, OR P]

P       <cr>

ONE LINE WITH 5 NUMBERS THE FIRST TWO
DESIGNATES THE VERTEX IN SCREEN COORDINATES WHILE THE OTHERS GIVE
THE ANGLES OF THE LINES IN HOUR-HAND VALUES IN A CLOCKWISE MANNER
                ABOUT THE VERTEX.

150 150 1.5 6 10.5      <cr>

ONE LINE WITH 3 WAL PARAMETERS.
        FIRST IS INTENSITY:
                -1 FOR DARKER OUTSIDE
                 0 FOR UNKNOWN
                 1 FOR LIGHTER OUTSIDE
        SECOND IS SEARCH:
                -1 FOR USE WINDOW ONLY
```

N≥1 FOR SEARCH IN AN AREA OF DIMENSION
N(INTEGER) ABOUT INITIAL WINDOW
THIRD IS TOLERANCE OF ANGLE MATCH:
USUALLY 0.1 BUT DOWN TO 0.02 WITH CARE.
IF TOLERANCE IS >1 THEN ANY CORNER
WILL BE ACCEPTED.

1 -1 0.1 <cr>

{A model of the corner that you requested is displayed by WALDRV. }

<cr> <cr>

CENTER OF THE WINDOW 150.0 150.0
ARE YOU SATISFIED WITH THE MODEL?
Y OR N?

y <cr>

DO YOU WANT ALL CORNERS THAT MATCH Y OR N?

y <cr>

79702100 MESSAGE TRACE: ORV EYE SRCH,IMAGE 3 1-1.1000 FAR

(The program runs until a corner is found (or failure in some cases)
giving displays along the way. If it displays a corner that it has
found, give 2 <cr> in order to go on. }

{If corner was found WALDRV displays input model again.
Type 2 <cr> in order to go on. }

DRV+DO YOU WANT TO CHANGE THE THREE WAL PARAMETERS ? --- Y OR N

n <cr>

DO YOU WANT TO CHANGE THE MODEL? --- Y OR N

n <cr>

DO YOU WANT TO CHANGE THE CAMERA POSITION? --- Y OR N

n <cr>

END OF SAIL EXECUTION

********** WALEYE and your own driver program **********

You can write your own driver program which asks WALEYE to find corners. The communication with WALEYE is done via global variables, global arrays and message procedures.

The global variables are:

DEB-EYE: Set DEB_EYE←0 (you do not want the debug mode).
DIS,EYE: Set DIS_EYE←-1 if you want displays. Otherwise set DIS_EYE←0 and WALEYE will not give any displays, but it will run faster.
EYEFLG: Set EYEFLG←0 (you do not want calcomp plots). EYEFLG will return with 0 if WALEYE succeeds or 10 if WALEYE fails completely.

The global arrays are:

integer array LOOK_AT[1:8]
Set LOOK_AT[1] ← TV camera number,
LOOK_AT[2] ← x coordinate of window center,
LOOK_AT[3] ← y coordinate of window center,
LOOK_AT[4] ← width (x-direction),
LOOK_AT[5] ← height (y-direct ion),
and you can ignore the other locations.

real array DIR_EYE[0:10,1:8]
Set DIR_EYE[0,1] ← 1 if you want WALEYE to return after finding the first corner that matches,

or DIR_EYE[0,1] ← -1 if you want WALEYE to find all corners that match.

The information about the corners that are found is returned in array DIR-EYE. For lines: (Let N stand for the Nth line.)

DIR_EYE[0,1] contains the number of lines that were found.
DIR_EYE[N-1,2] contains the x coordinate of the center of the line.
DIR_EYE[N-1,3] contains the y coordinate of the center of the line.
DIR_EYE[N-1,5] contains a.
DIR_EYE[N-1,6] contains C.
DIR_EYE[N-1,7] contains $\sin\alpha$.
DIR_EYE[N-1,8] contains $\cos\alpha$.

The parameters a and C determine a line [see Eq(2)]. From a one can determine the intensity direction of the line; see below.
The program stores a maximum of 10 lines.

For corners:

|DIR_EYE[0,1]| equals the number of corners that were found.
(If DIR_EYE[0,1] = -1 then this corner is questionable because there is a large gap between at least one of the lines and the vertex or the intersection of the three lines is farther from a single point than we usually accept. If any other matching corner is found, a questionable corner will not be returned.)
DIR_EYE[0,2] contains the x coordinate of the vertex.
DIR_EYE[0,3] contains the y coordinate of the vertex.
DIR_EYE[0,4] contains (for a three-sided vertex) the largest perpendicular distance between a line and the vertex.
DIR_EYE[0,5] contains a of the first line.
DIR_EYE[0,6] contains C of the first line.
DIR_EYE[0,7] contains t or $-\sin\alpha$ of the first line.
DIR_EYE[0,8] contains t or $-\cos\alpha$ of the first line.

The signs before $\sin\alpha$ and $\cos\alpha$ are such that these quantities define a unit vector along the line pointing toward the vertex.

DIR_EYE[1,1] contains a of the second line.
DIR_EYE[1,2] contains C of the second line.

DIR_EYE[1,3] contains t or -sinα of the second line.
DIR_EYE[1,4] contains t or -cosα of the second line.

If we have a three-sided corner the information about the third line is stored as follows:

DIR_EYE[1,5] contains a of the third line.
DIR_EYE[1,6] contains C of the third line.
DIR_EYE[1,7] contains + or -sinα of the third line.
DIR_EYE[1,8] contains + or -cosα of the third line.

If several corners are returned, the information about the second corner is stored in DIR_EYE[2,K] and DIR_EYE[3,K], the information about the third corner is stored in DIR_EYE[4,K] and DIR_EYE[5,K], etc. The program stores a maximum of five corners.

The following message procedure call is made:
ISSUE(7,"DRV","EYE",
MESSAGE SRCH_IMAGE(MODLINES, OIREC, SEARCH, TOLER, CORNMOD));
where,

MODLINES ← 1 for a line, 2 for a simple corner and 3 for a three-sided corner.

DIREC ← -1 if it is darker out side the angular region enclosed by the first and last line taken in a clockwise direction.
0 if the intensity direction is unknown.
1 if it is lighter outside.

SEARCH ← -1 for searching inside the window only.
N for searching in an area of dimension N times the original window size (N≥1).

TOLER ← t, tolerance of angular match for all model lines.
[t = 1 - arccos(angular difference)].

real array CORNMOD[1:3,8:13]

The first index refers to the line number and is 1,2, or 3. (For a

line edge only 1/3 of the array is used, and for a simple corner only 2/3 is used.) Suppose we have the three-sided corner shown in Fig. 17 with vertex at (XB,YB). We can choose any of the lines as Line 1, but we must label Line 2 and Line 3 in a clockwise manner about the vertex. If we choose the top line to be Line 1, then we have the labelling shown in Fig. 17 and inside means the angular sector from Line 1 to Line 3 in a clockwise' manner.

The quantity SIGN_DIRECTION is defined as follows:
SIGN_DIRECTION = -1 if it is darker out side of corner.
+1 otherwise (i.e. if it is lighter outside or
if no intensity direction is specified).

```
CORNMOD[ 1,11 ] ← -SIGN_DIRECTION.
CORNMOD[ 2,11 ] ← +SIGN_DIRECTION.
CORNMOD[ 3,11 ] ← +SIGN_DIRECTION.
```

(The first and last lines of a corner must have different signs as the difference between inside and outside reverses direction. The sign of Line 2 of a three-sided corner is unimportant as the intensity cannot be specified for that line.) Next we specify vectors which are perpendicular to the lines (pointing from lighter to darker if it is pertinent for that line).

```
CX1 = CORNMOD[1,11](YB -YC)
CY1 = CORNMOD[1,11](XC -XB)

CX2 = CORNMOD[2,11](YB -YD)
CY 2 = CORNMOD[ 2,11](XD -XB)

CX3 = CORNMOD[3,11](YB -YE)
```

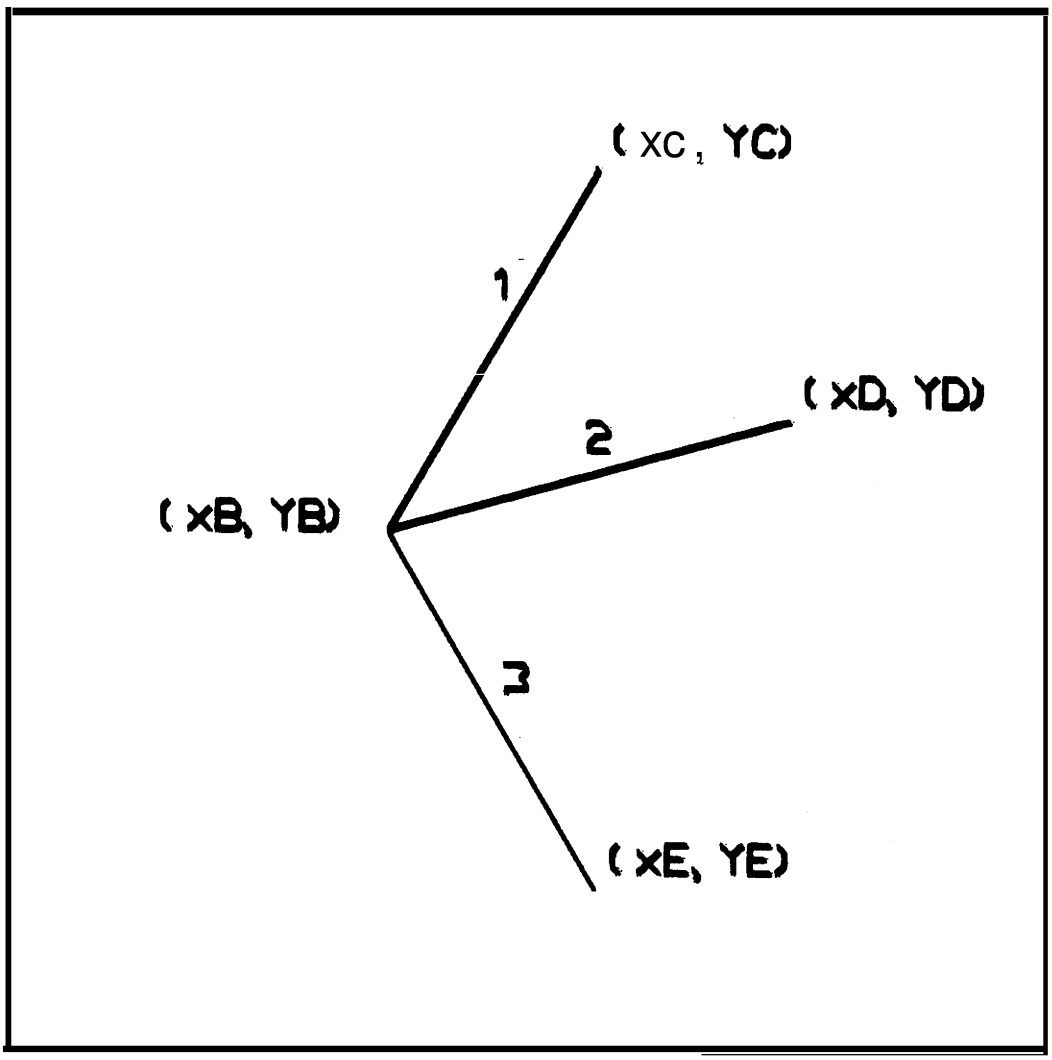


Fig. 17. Three-sided corner.

$CY3 = CORNMOD[3,11](XE-XB)$

From CX and CY we obtain the a-value by the following algorithm:

If $|CX| \geq 10\uparrow-9$ and $CX > 0$ then $a \leftarrow \arctan(CY/CX)$.
If $|CX| \geq 10\uparrow-9$ and $CX < 0$ then $a \leftarrow \arctan(CY/CX) + \pi$.
If $|CX| < 10\uparrow-9$ and $CY \geq 0$ then $a \leftarrow \pi/2$.
If $|CX| < 10\uparrow-9$ and $CY < 0$ then $a \leftarrow -\pi/2$.

Now we can fill out the rest of array CORNMOD. For tine N,

CORNMOD[N,8] ← a.
CORNMOD[N,9] ← $\sin\alpha$.
CORNMOD[N,10] ← $\cos\alpha$.
CORNMOD[N,12] ← CORNMOD[N,11]$\sin\alpha$.
CORNMOD[N,13] ← -CORNMOD[N,11]cosa.

CORNMOD[N,12] and CORNMOD[N,13] define a unit vector pointing along line N toward the vertex of the corner. The above definitions work for a simple corner with Line 3 absent. They also work for a simple line edge if we require the inside to be on the clockwise side of the line.

REFERENCES

1. W. M. Wichman, "Use of Optical Feedback in the Computer Control of an Arm," Stanford Artificial Intelligence Project, Memo No. 56 (Aug. 1967).

2. Y. Shiral and H. Inoue, "Visual Feedback of Robot in Assembly," ETL A. I. R. Group Memo No. 1 (1972).

3. P. H. Winston and J. B. Lerman, "Circular Scan," Artificial Intelligence Laboratory,Massachusetts Institute of Technology Vision Flash 23 (March 1972).

4. I. Sobel, "Camera Models and Machine Perception," Stanford Artificial Intelligence Project, Memo No. 121 (May 1970).

5. J. M. Tenenbaum, "Accommodation in Computer Vision," Stanford Artificial Intelligence Project, Memo No. 134 (Sept. 1970).

6. A. Gill, "Visual Feedback and Related Problems in Computer Controlled Hand Eye Coordination," Stanford Artificial Intelligence Project, Memo No. 178 (Oct. 1972).

7. See SAIL MANUAL, K. A. VanLehn, Editor, Stanford Artificial Intelligence Project, Memo No. 204 (June 1973).

8. M. H. Hueckel, "An Operator Which Locates Edges in Digitized Pictures," JACM Vol. 18, 113 (Jan. 1971).

9. K. K. Pingle and J. M. Tenenbaum, "An Accommodating Edge Follower," in The Proceedings of the Second International Joint Conference on Artificial Intelligence. p. 1 (London, Sept. 1971).

10. P. V. C. Hough, "Method and Means for Recognizing Complex Patterns," U. S. Patent 3,069,654 (Dec. 18,1 962).

11. A. K. Griffith, "Computer Recognition of Prismatic Solids," Ph. D. Th., Dept. of Math., Massachusetts Institute of Technology MAC TR-73 (June 1970).
"Edge Detection in Simple Scenes Using A Priori Information," IEEE Trans. Comput., Vol. C-22, p. 371 (April 1973).

'12. R. O. Duda and P. E. Hart, "Use of the Hough Transformation to Detect Lines and Curves in Pictures," CACM Vol. 15, p.11 (Jan. 1972).

13. J. A. Feldman and R. F. Sproull, "System Support for the Stanford Hand-Eye System," in The Proceedings of the Second International Joint Conference on Artificial Intelligence. p. 183 (London, Sept. 1971).

14. See for example, R. Paul, "Modelting, Trajectory Calculations and Servoing of a Computer Controlled Arm," Stanford Artificial Intelligence Project, Memo No. 177 (Nov. 1972).

15. H. A. Ernst, "MH-1 A Computer-Operated Mechanical Hand," Sc. D. Thesis, Massachusetts Institute of Technology (Dec. 1961).

16. T. Goto, K. Takeyasu, T. tnoyama, R. Shimomura, "Compact Packaging by Robot with Tactile Sensors," Proceedings of the 2nd. International Symposium on Industrial Robots, p. 149 (May 1972).

17. See for example, G. R. Grape, "Model Based (Intermediate-level) Computer Vision," Stanford Artificial Intelligence Project, Memo No. 201 (May 1973).