Computer vision of a moving square using a two-level data hierarchy

David Zokaites

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
COMPUTER VISION OF A MOVING SQUARE
USING A TWO-LEVEL DATA HIERARCHY

David Zokaites
April 1986

A Research Thesis submitted in partial
fulfillment of the requirements for
a Bachelor of Science Degree from
the Center for Imaging Science at the
Rochester Institute of Technology

Signature of the Author  David Z. Zokaites  1C-Apr-86
Imaging and Photographic Science

Certified by  Willem Brouwer
Thesis Advisor

Ronald Francis
Coordinator of Undergraduate Research
Abstract

An investigation of the relative speed and effectiveness of two computer vision algorithms has been conducted. One algorithm incorporates a two-level data hierarchy. The other incorporates a one-level hierarchy and serves as a relatively conventional basis for comparison. The computer vision algorithms, programmed in Fortran, detect and recognize a moving square. Both computer vision algorithms could readily be implemented in existing hardware. The two-level algorithm was found to be up to 90% faster than the one-level algorithm.

An analysis was made of elapsed CPU time variance as a function of time of day and user load. This was done to minimize the variance of results in comparing the above two algorithms. The mean and standard deviation of elapsed CPU time were both found to increase with system load, and system load was found to exhibit a midday peak.
Title of Thesis: Computer Vision of a Moving Square using a Two-Level Data Hierarchy

I __________________________ hereby grant permission to Wallace Memorial Library of RIT to reproduce my thesis in whole or part. Any reproduction will not be for commercial use of profit.

Date: 10-24-86
Acknowledgements

With their assistance and encouragement, many people helped bring this thesis to completion. The author wishes to thank the following people for their part in this endeavor: first and foremost, my wife Coni for her continued understanding, love, and encouragement; the members of this thesis committee (Dr. Willem Brouwer, Alexander Martens, and Paul Conlon) for their technical expertise and assistance; my parents for their love; and the many others who assisted me in any way.
# Table of Contents

Abstract .............................................. ii  
Thesis Release ................................. iii  
Acknowledgements ............................ iv  
Table of Contents .............................. v  
List of Tables ................................. vi  
List of Figures ................................. vii  
I. Introduction ................................. 1  
II. Experimental ................................. 13  
III. Results .................................... 37  
IV. Conclusions ................................. 46  
V. References ................................... 47  
VI. Appendix One: Program Listings ... 49  
VII. Appendix Two: Sample Output ..... 84  
VIII. Vita ......................................... 92
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Square Specification Variables</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>Standard Deviation and Mean CPU Time</td>
<td>38</td>
</tr>
<tr>
<td>3</td>
<td>Predicted CPU Time Savings</td>
<td>43</td>
</tr>
<tr>
<td>4</td>
<td>Average CPU Time Required for Two Algorithms</td>
<td>44</td>
</tr>
<tr>
<td>5</td>
<td>Comparison of Actual To Predicted Time Saved</td>
<td>45</td>
</tr>
</tbody>
</table>
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Data Hierarchies</strong></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Illustration of Data Hierarchies</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td><strong>Edge and Corner Detection</strong></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Number of Points Defining Edges of a Square</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>Clipping of Square at Scene Boundary</td>
<td>23</td>
</tr>
<tr>
<td>4</td>
<td>First Collinearity Test</td>
<td>26</td>
</tr>
<tr>
<td>5</td>
<td>Second Collinearity Test</td>
<td>27</td>
</tr>
<tr>
<td>6</td>
<td>Splitting Line Segments</td>
<td>28</td>
</tr>
<tr>
<td>7</td>
<td>Merge Check</td>
<td>29</td>
</tr>
<tr>
<td>8</td>
<td>Setting the Number of Test Points</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td><strong>CPU Variability Analysis</strong></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Log Standard Deviation Vs. Log Mean CPU Time</td>
<td>39</td>
</tr>
<tr>
<td>10</td>
<td>Mean CPU Time Vs. Load</td>
<td>40</td>
</tr>
<tr>
<td>11</td>
<td>Standard Deviation Vs. Load</td>
<td>41</td>
</tr>
<tr>
<td>12</td>
<td>System Load Vs. Time of Day</td>
<td>42</td>
</tr>
<tr>
<td>13</td>
<td>Mean Vs. Time of Day</td>
<td>42</td>
</tr>
<tr>
<td>14</td>
<td>Standard Deviation Vs. Time of Day</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td><strong>Conclusions</strong></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Time Saved Vs. Width</td>
<td>44</td>
</tr>
</tbody>
</table>
Contents

A. Relation to Previous Work ............... 2
  1. Overview .................................... 2
  2. CPU Time Variability Analysis ....... 2
  3. Edge Detection .............................. 4
  4. Data Hierarchies ......................... 5
  5. Corner Detection ......................... 6
  6. Pattern Recognition ...................... 7
B. Applications ................................. 8
C. Implementation ............................... 9
  1. General ...................................... 9
  2. Data Hierarchies ......................... 10
  3. Object and Scene ......................... 11
I. Introduction

A) Relation to Previous Work

1) Overview

This research presents a new approach to the implementation of data hierarchies. The difference in number of pixels (by area) of the various levels of data hierarchies is commonly four. The author arbitrarily chose a factor of sixteen. This data hierarchy was applied to the analysis of a moving square generated in software. Polygonal scenes (as analyzed here) were formerly popular subjects for analysis [Aggarwal and Duda 1975; Mitiche and Bouthemy 1985]. Generating images or scenes is rare in computer vision, but is quite common in computer graphics. As a preliminary to pattern recognition, a suboptimal corner detection algorithm was implemented [Pavlidis 1982]. The accuracy of this algorithm was improved considerably with three minor revisions. A pattern recognition algorithm was implemented which recognized a square given information derived from the location of its four corners.

2) CPU Time Variability Analysis

Accounting data measures the amounts of computer resources a user consumed. Some types of accounting data are elapsed CPU time, duration of user sessions, and amount of memory used [Knudson 1985; O'Neill and O'Neill 1980].
I. Introduction

The demands users place upon a computer vary from installation to installation. For example, one computer may be used mostly for many small programs, while another may be used mostly for text processing and larger programs. The configurations of large computers is often adjusted or "tuned" to peak performance under a given set of conditions [Ferrari, Serazzi, and Zeigner 1983]. For example, the page size of virtual memory systems may be adjusted, or a high speed printer may be added to a system to reduce the load on low speed printers.

The above-described field of computer performance evaluation and tuning is somewhat concerned with variability in accounting data. Davies' 1979 study examined elapsed CPU time for the same program under various conditions. He reported that elapsed CPU time increased with the load placed upon a computer [Davies 1979].

The author repeated this study using a VAX/VMS instead of a Decsystem 10/50. The author's results generally concur with those of Davies. The parameters of system load analyzed show a system and installation dependence.
3) Edge Detection

Edges of shapes usually correspond to intensity discontinuities in a digital image. Edge detection algorithms locate small spatial regions of large intensity change. Most edge detection algorithms locate boundary points which must be grouped together to form shape boundaries. A few techniques reduce the extent of this grouping by locating edges in terms of lines instead of points [Suk and Hong 1982].

Intensity discontinuities are found through the use of computational operators. A large variety of these operators are currently in use. Some of the most popular are templates [Kittler, Illingworth, and Paler 1983], Laplacian [Wiejak, Buxton, and Buxton 1985], and Hough Transforms [Brown 1983]. For a general discussion of these and other operators, see Ballard and Brown 1983.

The author implemented an edge operator which detected grey-level discontinuities. The digital images under analysis were binary, so detecting discontinuities was a fairly simple problem. The changes detected were from zero to one, and from one to zero [Capson 1984].
4) Data Hierarchies

Data hierarchies are also known as pyramid data structures. They contain N versions of an image, each at a different resolution level and pixel size. To create a data hierarchy, pixels of a high resolution view may be grouped together and averaged in software, resulting in a lower resolution view of the same scene.

![One-Level Hierarchy](image1)

![Two-Level Hierarchy](image2)

Figure 1: Illustration of Data Hierarchies

To increase speed and accuracy, data hierarchies may be incorporated into edge detection algorithms [Tanimoto and Klinger 1980; Spann and Wilson 1985; Harlow and Eisenbeis 1973]. For example, one level of a data hierarchy might be 512 pixels square (high resolution), and another level might be 64 pixels square (low resolution). The low resolution level could be useful for rapidly finding the outline of an apple, and the high resolution level for resolving this shape
Quadtrees are perhaps the most popular form of data hierarchy. In a quadtree, the high resolution image has four times as many pixels as the low resolution image. The author differed from this convention in choosing a high resolution image with sixteen times as many pixels.

5) Corner Detection

Numerous types of methods are currently employed to represent boundaries of shapes. One of the simplest is a region occupancy array. This is a array of the same size as the digital image. If a pixel is within the shape of interest, then the corresponding pixel in the occupancy array is given a value of one, otherwise is is given a zero.

A more common class of techniques is polygonal curve fitting. This method divides a boundary into sections, and fits a polynomial to these sections. Linear polynomials are perhaps most frequently used [Bezdek and Anderson 1986]. The endpoints of these boundary sections are called break points, polygon vertices, or feature points. (For an overview of these techniques, see Fischler and Bolles 1986.)

It is possible to optimize the locations of the break points [Dunham 1986]. Optimization minimizes both the number of break points and the error of polygonal approximations.
The disadvantage of optimization is higher computational cost. Some algorithms include optimization, and some do not.

The author implemented a suboptimal algorithm based on the description found in Pavlidis 1982. The performance of this algorithm was improved considerably by implementing a few minor revisions. For the extent of these revisions, see Corner Detection on page 29.

6) Pattern Recognition

Moving squares are relatively simple shapes to recognize, requiring neither a complicated model nor a complicated algorithm. There exist a wide variety of complex algorithms for pattern recognition. For an introductory discussion, see Ballard and Brown 1983.

The pattern recognition algorithm analyzed the following parameters of "squareness" which were calculated from the corner locations: the number of corners, lengths of the sides, and interior angles [Mitiche and Bouthemy 1985].
B) Applications

A possible application of the principles employed is tracking and identifying missiles. Other possible applications are: watching traffic to determine flow characteristics; tracking products on an assembly line to verify dimensional accuracy as well as presence or absence; identifying the region of text in a letter for automatic sorting; locating alignment marks for silicon wafer production; aligning and inserting components in printed circuit assembly; tracking motions of animals and people; measuring the flow of liquids; etc..
C) Implementation

1) General

This thesis presents a study of software techniques, particularly data hierarchies. These hierarchies were implemented in computer vision algorithms (software). This software implementation demonstrates the feasibility of hardware implementation.

The difficulties of hardware implemented were avoided to concentrate on the new aspects of the work done here. In keeping with this concentration, the software equivalent of a moving square was created. This was thought to be a speedier and more flexible approach than having a physical shape move. In addition, a square generated in software offered the advantage of a more controlled test ground.

Fortran was chosen as the language for the software because it is a popular language for scientific uses, it compiles efficiently, and it is the author's favorite language. The version of Fortran used was VAX-11 Fortran V4.3, a structured extension to Fortran-77. The machine used was VAXB of the Rochester Institute of Technology's VAXcluster. This is a Digital Equipment Corporation VAX/VMS model 11/785 computer.
I. Introduction

The computer vision algorithms implemented consisted of the following major sections: scene generation, edge detection, corner detection, and pattern recognition. The scene was a moving square on a stationary square background. The edge detection, corner detection, and pattern recognition algorithms were much more generalized than the scene was. These algorithms did not "know" a priori that the object being analyzed was a square. The edge and corner detection algorithms could find the edges and corners of almost any closed figure. The pattern recognition algorithm could determine if the shape found was a regular polygon with the specified number of sides. While these algorithms were designed to be very general, their utility was only tested with squares.

2) Data Hierarchies

A two-level data hierarchy was created. It contained two digital images, or arrays of pixels. These images correspond to the same scene, and differ in resolution. Digital images are usually 512 X 512 pixels. For the high resolution level of the data hierarchy, it was decided that a relatively small array (128 by 128) would most readily illustrate the computer vision algorithms implemented. For the low resolution level of the data hierarchy, an even smaller array (32 by 32) was chosen.
To implement a two-level data hierarchy, the scene was examined first in low resolution, then in high resolution. To implement a one-level data hierarchy, the scene was examined only in high resolution.

3) Object and Scene

The software equivalent of a moving square was created. The square had a constant brightness of one, while the background had a constant value of zero. (This square is described in more detail in the Object and Scene section of the Experimental on page 20.)

The creation of the moving square took into account the size of the digital image which mapped onto the scene. A high resolution view of the scene was created by mapping the scene onto a large digital image. A low resolution view resulted from mapping the scene onto a small digital image. As a result of the above procedure, it was not necessary to average the high resolution view to result in a low resolution view of the scene.

Initially, the square was allowed to move while the scene was being scanned. This resulted in motion blur. To eliminate the problem of motion blur, the square was allowed motion only between examinations of the scene. This condition could be achieved in hardware by synchronizing a strobe with the digitizing scan rate.
I. Introduction

The image digitizing system modeled here created two data files, one of which was the scene in low resolution, the other being the scene in high resolution.
II. Experimental

Contents

A. CPU Time Variability Analysis .......... 14
   1. Background ................................... 14
   2. Statistical Analysis ....................... 16
B. Modeling of CPU Time Saved .............. 17
C. Coordinate System ......................... 19
D. Object and Scene ............................. 20
   1. Choice of Object and Scene ............ 20
   2. Implementation ............................. 20
E. Edge Detection ............................... 22
   1. Background ................................... 22
   2. Implementation ............................. 23
F. Corner Detection ............................. 25
   1. Introduction ................................ 25
   2. First Collinearity Test .................. 26
   3. Second Collinearity Test ................ 26
   4. Split and Merge Aspects .................. 27
   5. Implementation ............................. 29
   6. Enhancements ............................... 29
G. Pattern Recognition ....................... 34
H. Data Hierarchies ............................ 35
I. Statistical Analysis ....................... 36
II. Experimental

A) CPU Time Variability Analysis

1) Background

Two general criteria are currently used to determine how much time a computer requires to complete a task. The first is elapsed time, as could be measured with a stop watch. The second is elapsed CPU time, or the amount of CPU time utilized. Note that in a timesharing environment (as occurred here), CPU time may readily be orders of magnitude lower than elapsed time.

The elapsed time required to replicate a task is known to vary greatly, being longest when the system load is the highest. One might expect that the CPU time needed to replicate a task would not vary at all. During the course of his experimentation, the author discovered that the CPU time required to replicate a task varied.

To characterize the relative speed of two algorithms, elapsed CPU time for both algorithms was measured. If this research had been done in an environment where CPU time showed great variation, then little faith could be placed in the results gathered.

The simplest method to reduce CPU variability would be to use a single-user machine. Another possibility would be to analyze CPU variability to isolate a low variability time
for processing. Due to the unavailability of a single-user machine with sufficient power, the latter option was chosen. CPU variability was found to be very low between midnight and 6:00 AM; consequently all data from the computer vision algorithms was collected during this time.

To analyze CPU variability, a program was written to utilize processor time in computing transcendental functions (sine, logarithm, etc.). Various amounts of processor time were used, ranging from 1 to 400 ms. The use of processor time was replicated 35 times. The program which used processor time in this manner was called CPU.FOR (a copy of which appears in Appendix One).

It was hypothesized that relationships existed among time of day, system load, and processor time. These hypothesized relationships appear in the Results section on page 38. To characterize these relationships, a command file (CPU.COM) was submitted to batch. This command file ran the above program (CPU.FOR) approximately once every hour and a half. Just before the program was run, the system load was determined by calling the local "nodeshow userload" utility. System load was defined as the average number of processes waiting for time or memory resources over the last fifteen minutes.
2) Statistical Analysis

A statistical analysis of the CPU time data was undertaken. A program (C.CPU3STATS.FOR) was written to calculate mean and standard deviation using the following formulas. The results of this statistical analysis can be found in the Results section of this report on page 38.

\[
\bar{X} = \frac{1}{N} \sum_{i=1}^{N} X_i \quad Sx = \sqrt{\frac{N}{N-1} \left( \sum_{i=1}^{N} X_i^2 - \left( \sum_{i=1}^{N} X_i \right)^2 / N \right)}
\]

Where
- \( \bar{X} \) = Mean
- \( Sx \) = Standard Deviation
- \( N \) = Number of Data Points
- \( X_i \) = ith Data Point
B) Modeling of CPU Time Saved

A mathematical model was developed to predict the relative speed of the two computer vision algorithms implemented.

The bulk of the CPU time utilized by the edge detection algorithms was thought to be spent examining the requisite number of pixels. A much smaller fraction of CPU time was thought to be used by the overhead portions (initializing variables, error trapping, etc.) of these algorithms.

The model analyzed the number of pixels each algorithm examined, the resultant relative savings of CPU time, and neglected the above-described overhead. Because the overhead was not included in the model, the model represents a best-case analysis, and the actual results are expected to be slightly less favorable.

The one-level algorithm analyzed only the high resolution view. This view is 128 pixels square, for a total of 16384 pixels. This was compared to the number of pixels examined by the two-level algorithm. The two-level algorithm first analyzed the entire low resolution view (32 pixels square) for 1024 pixels. The region of the shape in low resolution was isolated, then analyzed in high resolution for a variable number of pixels. This number of pixels depended on the width of the shape found.
A short computer program, MODEL.FOR, was written to implement this model. For more details on the modeling, see the Modeling of CPU Time Saved section of Appendix One on page 55. The results appear in the Results section on page 43.
C) Coordinate System

A coordinate system was arbitrarily chosen for the scene. The scene had a width of one, centered on the origin (0,0). The coordinates of top-right corner of the scene were (.5,.5); the bottom-left was denoted by (-.5,-.5).

A short subroutine called COORD (included in Appendix One under Coordinate System) implemented the above coordinate system. Arrays are accessed in terms of array indices, I and J for example. This routine translates from array indices to the above scene coordinate system. I and J, the array indices to be transformed, are passed to the routine and shifted so that they are centered about 0, then they are scaled so that they range from -.5 to +.5. X and Y in scene coordinates are returned.
D) Object and Scene

1) Choice of Object and Scene

The object and scene were chosen to be relatively simple: closed regular polygons; squares. This was done to eliminate unnecessary complications from the computer vision algorithms. Note that a more complex scene is not likely to change the relative performance of the two algorithms, but it would certainly make it much more difficult to achieve conclusive results.

The object and the scene it moves through were chosen to be coplanar squares. The moving square had a constant brightness of one, and the stationary background had a constant brightness of zero. The square was allowed to travel in a straight line at constant velocity. The direction of the square’s travel and its starting point were variable. The square may rotate about its center while it is translating. The direction and rate of rotation were variable. For an example of the generated square, see page 88 of Appendix Two.

2) Implementation

A function subprogram named SQUARE was written to implement the above choice of object and scene. It created the moving square during run-time. Input to this function
were the following variables to specify the movement of the square:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedr</td>
<td>speed of rotation</td>
<td>degrees/time</td>
</tr>
<tr>
<td>Speedt</td>
<td>speed of translation</td>
<td>scene widths/time</td>
</tr>
<tr>
<td>Width</td>
<td>width of square</td>
<td>scene width</td>
</tr>
<tr>
<td>Angleo</td>
<td>initial offset angle</td>
<td>degrees</td>
</tr>
<tr>
<td>Anglet</td>
<td>direction of translation</td>
<td>degrees</td>
</tr>
<tr>
<td>Xi, Yi</td>
<td>initial coordinates of square's center</td>
<td>scene units</td>
</tr>
</tbody>
</table>

To let the square rotate, the square's angle from the horizontal was incremented. To let the square translate, the square's definition stayed constant, and the coordinate system of the scene was moved in the opposite direction. This resulted in apparent motion in the desired direction.

The square was defined in polar coordinates to simplify letting the square rotate. The square's angle from the horizontal was simply incremented to let the square rotate. To create the scene, the array indices corresponding to each pixel were passed to the subroutine SQUARE. The array indices were transformed first into scene cartesian coordinates, then into polar coordinates. If the coordinates of the pixel's center lie within the square, then a one was returned. Otherwise, a zero was returned.
E) Edge Detection

1) Background

The edge detection algorithm was designed to be slightly more general than necessary to examine the scene. Even though only moving squares were analyzed, the edge detection algorithm was given the capacity to accurately find the edges of nearly any single convex polygon. As a result, it was not necessary to give this algorithm a priori knowledge of the shape it was analyzing.

To detect the edges of the as-yet unidentified shape, the scene was scanned horizontally, one row at a time. For most of the shape two points per line are sufficient to define the edges. However, on the top and bottom edges of the shape, a variable number of points are needed. See the figure below.

<table>
<thead>
<tr>
<th>The Square</th>
<th>Number of Boundary Points Per Row</th>
</tr>
</thead>
<tbody>
<tr>
<td>++++++++</td>
<td>8</td>
</tr>
<tr>
<td>+XXXXXX+</td>
<td>2</td>
</tr>
<tr>
<td>+XXXXXX+</td>
<td>2</td>
</tr>
<tr>
<td>+XXXXXX+</td>
<td>2</td>
</tr>
<tr>
<td>+XXXXXX+</td>
<td>2</td>
</tr>
<tr>
<td>+XXXXXX+</td>
<td>2</td>
</tr>
<tr>
<td>+XXXXXX+</td>
<td>2</td>
</tr>
<tr>
<td>++++++++</td>
<td>8</td>
</tr>
</tbody>
</table>

Key:
X  Interior Point
+  Boundary Point

Figure 2: Number of Points Defining Edges of a Square
As the square moved, sometimes parts of the square were clipped at the scene boundary. When this occurred, the edge detection algorithm found the edges of the unclipped portion of the square. Depending on how the square was clipped, the resulting shape could have anywhere from three to eight sides. See the below figure.

Key:
- Scene
- Square Inside Scene
- Square Outside Scene

Figure 3: Clipping of Square at Scene Boundary

2) Implementation

A subroutine, EDGE, was written to implement the above edge detection algorithm. The output from this subroutine includes two arrays containing the coordinates of the edge points, and a flag to note whether or not a shape was found. This subroutine calls the above-mentioned square generating function, creating the moving square during program execution.
To locate the square, the first and last pixel on each row with a value of one is found. In addition, a boundary condition may occur at the right-hand edge of the scene: If the edge of the square extends beyond the edge of the scene, then a one followed by a zero will not be found. This condition was trapped accordingly.
F) Corner Detection

1) Introduction

As a preliminary to pattern recognition, the corners of the square were found via a split-and-merge polygon fitting algorithm. Polygon fitting algorithms draw a polygon to approximate the boundary of closed figure. Split-and-merge algorithms split data into successively smaller segments when necessary, and new segments identified are merged when possible.

Due to quantization error, a theoretically smooth edge may become ragged or saw-toothed. As the width of a square becomes small in terms of pixels, the raggedness of theoretically smooth edges increases. (See page 89 for an example of a very ragged edge.) This phenomenon made it difficult to find the four corners of a square in the low resolution view of the scene. This problem occurred rarely in high resolution.

A significant portion of the polygon fitting algorithm are two collinearity tests. These tests determine whether a given set of points lie on the same line to within a given tolerance. The line is drawn directly from the first point to the last point. This line is probably not the optimal fit to the data, but it is relatively easy to calculate.
2) First Collinearity Test

The first collinearity test checks for a relatively rare occurrence which is illustrated below. Note that points B, C, and D are nearly collinear. Note also, that when viewed as an ordered sequence, these points are not collinear. The first collinearity test checks for this condition by computing and summing the length of the lines from point to point (AB, BC, CD, and DE). This sum is compared to the overall length (AE). If the sum divided by AE is less than 1.1, then the points are declared to be collinear; if the quotient is greater than 1.5, the points are not collinear.

![Diagram of collinear points A, B, C, D, and E]

Key:
- ——— Point to Point Lengths
- ——— Overall Length

Figure 4: First Collinearity Test

3) Second Collinearity Test

Ideally, every point under test will lie exactly on the line drawn, but this is rarely the case. The more perpendicularly distant points are away from the line drawn,
II. Experimental

the less the points are collinear. To evaluate this measure of collinearity, the second collinearity test calculates the distance from the line drawn to each point under test. The point furthest away from the line drawn is identified. By the above definition, this is also the least collinear point.

The largest perpendicular distance was scaled by the length of the line. If the result is less than a specified tolerance, the points are declared to be collinear. If not, then the number of points under test is reduced: the test points are divided into two groups at the most distant point.

Least Collinear Point

Figure 5: Second Collinearity Test

4) Split and Merge Aspects

Using the above two tests, a selectable number of the edge points were grouped together and tested for collinearity. If these points were found to be collinear to within a tolerance, they formed a line segment. If these
points were not collinear, then the number of points under test was reduced. The new endpoint was chosen to be the point furthest away from the original line segment. The new line was tested for collinearity and split further as necessary.

The above procedure identified a set of collinear points. Starting from the last element of the above set, the next group of collinear points was found as above. These two sets of collinear points form two line segments, which in turn form an interior angle. If this angle was equal to 180 degrees to within the specified tolerance, then the two line segments were merged.
II. Experimental

Key:

- Original Line Segments
- Merged New Line Segment

Note that only for clarity, only the endpoints of the above line segments are shown.

Figure 7: Merge Check

The indices to the points under test were then incremented, and the next group of points analyzed. This process continued until all the points describing the edges of the square were analyzed.

5) Implementation

The main subroutine for polygon fitting was named POLY, the subroutine which tests points for collinearity was named COLINE. Three other associated routines appear listed with the above in the Corner Detection section of Appendix One on page 76.

6) Enhancements

The polygon fitting algorithm was modified slightly in three ways to optimize its performance: 1) a merge check between the first and last corners found was added, 2) the collinearity tolerances were adjusted, and 3) the number of
test points was adjusted to be slightly larger than the expected width of the moving square.

The first edge point of the square was the starting point for corner detection. As a result, the first edge point was a default polygon vertex. The first edge point may have unnecessarily (and incorrectly) been declared a polygon vertex. This potential error was trapped by applying a merge check between the first and last polygon vertices: the second collinearity test was applied to the line segments bounded by the last, first, and second vertices.

There are two tolerances input to the polygon fitting algorithm: the first tolerance is the maximum separation of a point from the line it is thought to be collinear with; the second tolerance applies to the interior angle formed by two lines. If this angle equals 180 degrees within the second tolerance, then the two line segments are merged.

These two tolerances have a pronounced affect on the number of polygon vertices declared. If the above tolerances are set to low values, then a large number of polygon vertices will generally be found. If these tolerances are set high, then a small number of vertices will be declared. These tolerances were adjusted just to the point where four corners were found for a square in low resolution with a width equal to 40% of the scene width. Note that this square
was only 12 pixels wide. The tolerance for distance was set to 2.5 pixels, and the tolerance for angles was set to 45 degrees.

Corner detection was much easier to perform in high resolution than in low resolution. This is because quantization error and the raggedness of the squares' edges increased as the number of pixels representing a shape decreased. Enabling the corner detection algorithm to find four corners on a square with ragged edges required setting the two above tolerances to high values.

The corner detection algorithm started its analysis at the first edge point. Let us assume for the sake of discussion that the first edge point corresponded to the first corner of the square. Starting from the first edge point, a variable number of data points were grouped together and tested for collinearity. Let the number of points grouped together be denoted by "N". The value of N affected which edge points were declared to be corners. In the field, common values for N range from five to ten [Pavlidis 1982].

An imaginary line was drawn from the first corner to an edge point. The value of N determined which edge point formed the second endpoint of this line. For some values of N, the line was drawn from the first corner to a point near the second corner. This line potentially could have passed
the collinearity test with the tolerances specified.

A second line was drawn from the last endpoint of the above line to a new point. These two lines would not be collinear in this example, so the common endpoint of both lines would be declared to be a new polygon vertex. For small values of N (ten for example), the author discovered that this endpoint could be up to four pixels away from the true corner of the square. See the below figure.

To more accurately find the square's corners, the value of N was adjusted to be at least six pixels greater than the width of the largest square expected. N was set equal to the number of pixels on one row of the digital image. With this setting of N, edge points from more than one side of the square were grouped together to form the first line tested for collinearity. These points did not pass the collinearity test. The point furthest away from this line was found, and corresponded to an exact corner of the square. This line was then split at the exact corner of the square, and this corner declared a polygon vertex. This procedure, due to the adjustment of N, resulted in locating the corners of the square more precisely.
II. Experimental

Approximate Corners Found  Exact Corners Found

Key:
X Edge Point of Square
+ Edge Point Declared to be a Corner
- Imaginary Lines Drawn

Figure 8: Setting the Number of Test Points

Setting N to the above value required that another small change be made. After each polygon vertex was found, the indices to the points under test were incremented by N. The corner detection algorithm continued until the above indices were greater than the number of edge points. If the number of edge points was not an integer multiple of N, then some edge points would not have been analyzed using the above technique. This error was trapped by setting one of the above indices to the last edge point for the last pass through the algorithm.
G) Pattern Recognition

In most applications of computer vision, pattern recognition is a very important task. How can one track moving squares without first being certain that there are squares to track?

Three criteria were chosen to measure "squareness": the number of corners was tested for equality with four; the lengths of the sides were tested for being the same within a specified tolerance; and the interior angles were tested for being equal to ninety degrees within a tolerance. The number of corners, the lengths of the sides, and the interior angles were calculated as an integral part of the polygon fitting routine. The function subprogram written to implement pattern recognition was named RECOGN. Its text is listed in Appendix One on page 79.
H) Data Hierarchies

Taken as a whole, the above-described software implemented a one-level data hierarchy. The entire scene was scanned at low resolution to find the edges of the moving shape. From the edge data, the corners were found. From the corners, it was determined whether or not a square was found.

To implement a two-level data hierarchy, the author used the low resolution edge data to find the region of the moving shape. This region was then enlarged by a few pixels in all directions to allow for quantization error, then examined again in high resolution.

For the two-level data hierarchy, the corner detection could occur in either of three options: low resolution only, high resolution only, or combined low and high resolution.
I) Statistical Analysis

The two vision algorithms reported elapsed CPU time. To characterize the mean and standard deviation of elapsed CPU time, a program named [.VISION]STATS.FOR was written. Note that a different program was written to do similar statistical analysis for a different set of elapsed CPU time data (see CPU Time Variability). It was necessary to write a new program due to the variations in file format.
III. Results

Contents

A. CPU Time Variability Analysis ............... 38
   1. Introduction .......................... 38
   2. Standard Deviation Vs. Mean CPU Time .. 38
   3. Mean CPU Time Vs. Load ............... 39
   4. Standard Deviation Vs. Load .......... 40
   5. System Load, Mean, and
       Standard Deviation Vs. Time .......... 41
B. Modeling of CPU Time Saved .................. 43
C. Actual Time Saved .......................... 44
A) CPU Time Variability Analysis

1) Introduction

To analyze elapsed CPU time variability as functions of user load and time of day, system load was determined throughout one day at approximately one and a half hour intervals. Immediately following this, elapsed CPU time data was collected for replicates of various processes. These processes were designed to utilize various amounts of CPU time, from 1 to 400 ms.

2) Standard Deviation Vs. Mean CPU Time

For many processes, the standard deviation increases with the mean. The author expected the standard deviation of CPU time to increase with the mean, as it does. Note that the rate of increase changed throughout the day. See the below figure and table.

Table 2: Standard Deviation and Mean CPU Time

<table>
<thead>
<tr>
<th>Time</th>
<th>Standard Deviation (ms)</th>
<th>Mean Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:00 AM</td>
<td>.406</td>
<td>1.200</td>
</tr>
<tr>
<td></td>
<td>.490</td>
<td>4.371</td>
</tr>
<tr>
<td></td>
<td>.471</td>
<td>17.314</td>
</tr>
<tr>
<td></td>
<td>.725</td>
<td>75.657</td>
</tr>
<tr>
<td></td>
<td>1.132</td>
<td>183.314</td>
</tr>
<tr>
<td></td>
<td>1.721</td>
<td>432.743</td>
</tr>
<tr>
<td>3:00 PM</td>
<td>.632</td>
<td>1.200</td>
</tr>
<tr>
<td></td>
<td>.781</td>
<td>4.086</td>
</tr>
<tr>
<td></td>
<td>1.435</td>
<td>18.000</td>
</tr>
<tr>
<td></td>
<td>5.371</td>
<td>75.829</td>
</tr>
<tr>
<td></td>
<td>6.638</td>
<td>186.000</td>
</tr>
<tr>
<td></td>
<td>7.699</td>
<td>439.314</td>
</tr>
</tbody>
</table>
III. Results

![Log Standard Deviation vs. Log Mean CPU Time](image)

Figure 9: Log Standard Deviation Vs. Log Mean CPU Time

3) Mean CPU Time Vs. Load

This research was run in a timesharing environment. As the user load increased, the length of the slice of CPU time allotted to each user is hypothesized to have decreased. This resulted in each process being shuffled in and out of the CPU at an increasing rate. Moving a process in and out of the CPU requires CPU time. In other words, as the system load increases, the amount of CPU time spent in overhead increases. Considering the above, one might expect mean CPU time to increase with the system load, as it did indeed.
III. Results

4) Standard Deviation Vs. Load

User load in a timesharing environment is subject to large variations over small intervals of time. Processes heavily dependent on CPU resources are begun and ended at random intervals, resulting in variable demand upon CPU resources and large fluctuations in system load over the short term.

If the system load varies greatly, then the mean CPU time to replicate a task varies. This results in a high standard deviation. Using the above logic, the author expected the standard deviation to increase with the load, as did occur.
The CPU time data plotted below had a mean of approximately 76 ms. Note that this is a different group of data then plotted elsewhere in this section. This was necessary because the other data set was very noisy and did not exhibit a very clear relationship between standard deviation and load.

![Graph showing Standard Deviation Vs. Load](image)

**Figure 11: Standard Deviation Vs. Load**

5) System Load, Mean, and Standard Deviation Vs. Time

The computer centers for the system used are open from 8:00 AM to 11:00 PM daily. During the day when the computer centers are open, the system load is relatively high. As a consequence of high load, the mean CPU time and standard deviation are also relatively high. At night, the computer centers close and the system load is very low. Mean and standard deviation are consequently also very low. See the following three graphs.
III. Results

Figure 12: System Load Vs. Time of Day

Figure 13: Mean Vs. Time of Day

Figure 14: Standard Deviation Vs. Time of Day
B) Modeling of CPU Time Saved

The relative CPU time required by both algorithms was modeled. The two-level algorithm, as modeled, was much faster than the one-level algorithm.

Table 3: Predicted CPU Time Savings

<table>
<thead>
<tr>
<th>Width of the Square (% of Scene Width)</th>
<th>Predicted CPU Time Saved (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5</td>
<td>93.1</td>
</tr>
<tr>
<td>10</td>
<td>91.6</td>
</tr>
<tr>
<td>25</td>
<td>85.0</td>
</tr>
<tr>
<td>40</td>
<td>73.8</td>
</tr>
<tr>
<td>65</td>
<td>45.2</td>
</tr>
<tr>
<td>70</td>
<td>38.0</td>
</tr>
<tr>
<td>85</td>
<td>13.3</td>
</tr>
<tr>
<td>100</td>
<td>-6.3</td>
</tr>
</tbody>
</table>
III. Results

C) Actual Time Saved

The two-level edge detection algorithm was experimentally shown to be much faster than the one-level algorithm. The predicted results are quite close to the actual results. See the below figure and tables for more details.

Table 4: Average CPU Time Required for Two Algorithms

<table>
<thead>
<tr>
<th>Width of Square (% of Scene Width)</th>
<th>One-Level (ms)</th>
<th>Two-Level (ms)</th>
<th>Savings (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5</td>
<td>263</td>
<td>36</td>
<td>86</td>
</tr>
<tr>
<td>10</td>
<td>262</td>
<td>41</td>
<td>84</td>
</tr>
<tr>
<td>25</td>
<td>265</td>
<td>62</td>
<td>77</td>
</tr>
<tr>
<td>40</td>
<td>263</td>
<td>86</td>
<td>68</td>
</tr>
<tr>
<td>65</td>
<td>265</td>
<td>154</td>
<td>42</td>
</tr>
<tr>
<td>70</td>
<td>266</td>
<td>178</td>
<td>33</td>
</tr>
<tr>
<td>85</td>
<td>268</td>
<td>255</td>
<td>5</td>
</tr>
<tr>
<td>100</td>
<td>268</td>
<td>288</td>
<td>-8</td>
</tr>
</tbody>
</table>

Figure 15: Time Saved Vs. Width
### III. Results

Table 5: Comparison of Actual To Predicted Time Saved

<table>
<thead>
<tr>
<th>Width of Square (% of Scene Width)</th>
<th>Actual Savings (%)</th>
<th>Predicted Savings (%)</th>
<th>Error in Prediction (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5</td>
<td>86</td>
<td>93</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>84</td>
<td>92</td>
<td>8</td>
</tr>
<tr>
<td>25</td>
<td>77</td>
<td>85</td>
<td>8</td>
</tr>
<tr>
<td>40</td>
<td>68</td>
<td>74</td>
<td>6</td>
</tr>
<tr>
<td>65</td>
<td>42</td>
<td>45</td>
<td>3</td>
</tr>
<tr>
<td>70</td>
<td>33</td>
<td>38</td>
<td>5</td>
</tr>
<tr>
<td>85</td>
<td>5</td>
<td>13</td>
<td>6</td>
</tr>
<tr>
<td>100</td>
<td>-8</td>
<td>-6</td>
<td>-2</td>
</tr>
</tbody>
</table>
IV. Conclusions

In the two-level data hierarchy, the low resolution level located the region of the moving square. This region was then examined in high resolution. Note that the region of the moving square was often much smaller than the entire field. This is to be compared to the one-level data hierarchy in which the entire field was examined in high resolution to find the square.

With small shapes, a small section of the high resolution level was analyzed in the two-level hierarchy. This resulted in significant savings of time. If the shape was the same size as the scene (or nearly so), then the entire high resolution level was analyzed. In this case, the time spent analyzing the low resolution level was effectively wasted. The result is that the two-level data hierarchy implemented actually required more CPU time than the corresponding one-level hierarchy.

A comparison of actual to predicted time savings shows that there is a close correlation (see Table 4 on page 44). The author expected the predicted savings to be slightly higher than the actual savings. Note that this is the case, except for a square with a width of 100%.


V. References


Contents

A. CPU Time Variability Analysis .......... 50
   1. Command Files ....................... 50
   2. Program ............................. 50
   3. Statistical Analysis ............... 52
B. Main Vision Algorithm .................. 55
   1. Modeling of CPU Time Saved .......... 55
   2. Command Files ........................ 56
   3. Program ............................. 56
C. Coordinate System ....................... 64
D. Object and Scene ....................... 65
E. Edge Detection .......................... 67
F. Corner Detection ........................ 72
   1. Main Routine ......................... 72
   2. Collinearity Tests .................... 75
   3. Associated Subroutines .............. 76
G. Pattern Recognition ..................... 79
H. Statistical Analysis ..................... 81
A) CPU Time Variability Analysis

1) Command Files

$! file name is BATCH.COM
$! it submits CPU.COM to batch
$! written by David Zokaites Dec-85 to Jan-86
$ SUBMIT/RESTART/NOTIFY/QUE=VAXB$LARGE/LOG_FILE= -
 [DMZ5436.THEESIS.DEVELOP.CPU1CPU.LOG -
 [DMZ5436.THEESIS.DEVELOP.CPU1CPU]

$! file name is CPU.COM
$!Written by David Zokaites 16-Dec-85, 12-Jan-85
$!This file executes a series of commands to test cpu time
$!variability. These commands are repeated after a certain
$!delta time.
$!
$!initialize
$ SET NOON   !ignore all errors
$ WSO :== WRITE SYS$OUTPUT
$ SET DEF [DMZ5436.THEESIS.DEVELOP.CPU]
$ COUNT = 0
$ LOOP:
 $ WSO "count = ", COUNT
 $ NODESHOW USERLOAD VAXB
 $ RUN CPU
 $ RENAME CPUA.DAT 'COUNT'A
 $ RENAME CPUB.DAT 'COUNT'B
 $ COUNT = COUNT + 1
 $ WAIT 01:00   !HH:MM:SS.CC
 $ IF COUNT .LT. 24 THEN GOTO LOOP

2) Program

!Program name is CPU.FOR
!Written by David Zokaites
!8-Dec-85 to 13-Dec-85, revised 12-Jan-86
!It determines elapsed CPU time for both computation and I/O.
!Note that the elapsed time is determined repeatedly. This is
!to get an estimate of variability.

!initialize
!declare variables
    INTEGER*4 TIME,   !elapsed cpu time
    + MS,            !limit on do loop, roughly = CPU
    + REPS           !number of times cpu time is found
REAL ARG, !argument to math functions
+ COMP, IO, !scales MS for computation or i/o
+ COUNTER1, COUNTER2 !counters, could be integer

!LIB$INIT_TIMER and LIB$STAT_TIMER are part of system's
!run time library. The former initializes the count of
!elapsed CPU time, the latter determines elapsed time.

!open files
OPEN (NAME = 'CPUIN.DAT', TYPE = 'OLD', UNIT = 1)
OPEN (NAME = 'CPUA.DAT', TYPE = 'NEW', UNIT = 2)
OPEN (NAME = 'CPUB.DAT', TYPE = 'NEW', UNIT = 3)

!read and write
READ (1,*) REPS, COMP, IO
WRITE (2,*) 'Computation CPU time'
WRITE (3,*) 'I/O CPU time'
WRITE (2,5) REPS
WRITE (3,5) REPS
5 FORMAT (' in ms for repetitions of same thing '/
+ ' Number of repetitions = ', / I3/ )

!start of loop to read and compute
10 READ (1,*, END = 20) MS

!use cpu time in computation
!repeat this section to get an indication of variability
DO I = 1, REPS
  CALL LIB$INIT_TIMER !initializes timer

!use some time
  DO COUNTER1 = 1, (MS * COMP)
  DO COUNTER2 = 1, 1000
  VARIABLE = SIN(COUNTER1)*COUNTER1**COUNTER1-LOG(COUNTER1)
  END DO
  END DO

!determine elapsed time
  CALL LIB$STAT_TIMER (2, TIME)
  WRITE (2,*) TIME

  END DO
  WRITE (2,*) !skip a line

!use cpu time in I/O
!repeat this section to get an indication of variability
DO I = 1, REPS
  CALL LIB$INIT_TIMER !initializes timer

!use some time
  OPEN (NAME = 'JUNK.DAT', TYPE = 'NEW', UNIT=20)
IV. Appendix One: Program Listings

DO J = 1, (MS * I0)
WRITE (20,*) J
END DO
CLOSE (UNIT =20)

!determine elapsed time
CALL LIB$STAT_TIMER (2, TIME)
WRITE (3,*) TIME
END DO
WRITE (3,*) !skip a line

!end of loop to read and compute
GO TO 10

!end program
20 STOP ' NORMAL END OF CPU.FOR'
END

3) Statistical Analysis

! Program name is CPUJSTATS.FOR
! Written by David Zokaites 20-Dec-85, 12-Jan, 13-Jan-86
! This program characterizes mean, standard deviation, range.
! The input is the output from CPU.COM and CPU.FOR

!initialize
CHARACTER*7 FILE !cpu input data file to be opened
!initialize output file
OPEN (UNIT=4, NAME = 'STATS.DAT', TYPE = 'NEW')
WRITE (4,10)
10 FORMAT (' ELAPSED CPU TIME VARIABILITY ANALYSIS DATA '/'
+ ' time = time the data was collected '/'
+ ' users = number of users on the system, excluding batch '/'
+ ' load = avg sys load in last minute, # processes waiting'/'
+ ' process = process using cpu time, computation or I/O '/'
+ ' The statistical parameters calculated follow: '/'
+ ' mean (mean), standard deviation (sdev); range, from'/'
+ ' minimum (min) to maximum (max), for POINTS points. '/'
+ ' file = file containing elapsed cpu time '/'
+ ' TIME USERS LOAD PROCESS MEAN '
+ ' SDEV RANGE MIN MAX POINTS FILE SDEV/MAN ')

!call stats for two groups of file names
FILE(4:7) = '.DAT'
FILE (3:3) = 'A'
CALL STATS (FILE)
FILE (3:3) = 'B'
CALL STATS (FILE)
! end calling routine
STOP 'NORMAL END OF STATS.FOR'
END

! subroutine to do all the work
SUBROUTINE STATS (FILE)
! initialize
! declare variables
INTEGER
+ POINTS, ! number of numbers
+ DATA, ! input data
+ MIN, MAX, ! minimum and maximum values of the data
+ RANGE, ! max - min
+ SUM, ! sum of data
+ SUMSQ, ! sum of data * data
+ COUNTER, ! counter on which file was opened
+ LSDIGIT, ! least significant digit
+ MSDIGIT ! most

REAL
+ MEAN, ! mean of the input data
+ SDEV, ! standard deviation
+ SCALED ! scaled sdev = sdev / mean

CHARACTER*4 PROCESS, ! process that used cpu time
+ USERS, ! number of users on sys (excludes batch jobs)
+ LOAD ! system load in terms of jobs waiting
CHARACTER*7 FILE ! cpu input data file to be opened
CHARACTER*26 TIME ! TIME of data collection

OPEN (UNIT=1, NAME = 'CPU.LOG', TYPE = 'OLD') ! input
OPEN (UNIT=4, NAME = 'STATS.DAT', TYPE = 'OLD') ! output

! open loop to read files numbered 0, 1, 2, ...
COUNTER = -1
12 CONTINUE
COUNTER = COUNTER + 1
! read descriptive data from cpu.log
READ (1,15, END=40, ERR=40) TIME, USERS, LOAD
15 FORMAT ( / T27, A26, 5X, A3, 74X, A4 //)
! open CPU time input file
! compute lsdigit, msdigit
MSDIGIT = 10 * INT( COUNTER/10)
LSDIGIT = COUNTER - MSDIGIT
MSDIGIT = MSDIGIT/10
! compute FILE
FILE (2:2) = CHAR (LSDIGIT + 48)
IF (COUNTER .GE. 10) THEN
  FILE(1:1) = CHAR (MSDIGIT + 48)
ELSE
FILE(1:1) = ' '
END IF
!
OPEN (UNIT=2, NAME= FILE, TYPE = 'OLD')
!read initial data from file
READ (2,20) PROCESS, POINTS
20 FORMAT ( ' A3,///, I3 /)
!open loop to examine new sets of data in file
DO I = 1, 6
!initialize variables
MIN = 1000000
MAX = 0
SUM = 0
SUMSQ = 0
!compute statistics
DO J = 1, POINTS
  READ (2,*END=50) DATA
  IF (DATA .LT. MIN) MIN = DATA
  IF (DATA .GT. MAX) MAX = DATA
  SUM = SUM + DATA
  SUMSQ = SUMSQ + DATA * DATA
END DO
READ (2,*!skip over a blank line in a file

RANGE = MAX - MIN
MEAN = FLOAT (SUM) / POINTS
SDEV = POINTS * SUMSQ - SUM * SUM
SDEV = SDEV / ( POINTS * (POINTS - 1) )
SDEV = SQRT ( ABS (SDEV))
SCALED = SDEV / MEAN
!output data
WRITE (4, 30) TIME, USERS, LOAD, PROCESS, MEAN, SDEV, RANGE, + MIN, MAX, POINTS, FILE, SCALED
30 FORMAT ( ' ', A26, 3(4X, A4), 2F8.3, 4I7, A8, F8.3)
!close loop to examine new sets of data in file
END DO
WRITE (4,*)!skip line in output file
!close loop that read files
CLOSE (UNIT = 2)
GOTO 12
40 CLOSE (UNIT = 1)
!finalize
RETURN
50 STOP 'END OF FILE IN STATS.FOR'
END
B) Main Vision Algorithm

1) Modeling of CPU Time Saved

!Program name is MODEL.FOR
!Written by David Zokaites 4-Feb-86

!This program models the CPU time saved by a two-level data hierarchy as compared to a one-level data hierarchy. More specifically, the % savings of number of pixels examined is computed.

!initialize
REAL
  + LEVEL1, !number of pixels examined one-level hierarchy
  + LEVEL2, ! ... two
  + WIDTH,    !width of the square in % of scene width
  + WIDTHt,   !transformed width
  + SAVED     !% time saved

INTEGER
  + PIX1,   !number of pixels per row in low resolution
  + PIX2    !... high

PARAMETER (PIX1 = 32, PIX2 = 128)

OPEN (UNIT=1, NAME='MODELIN.DAT', TYPE='OLD')
OPEN (UNIT=2, NAME='MODELOUT.DAT', TYPE='NEW')

WRITE (2,*), ' % time saved with 2-level data hierarchy'
WRITE (2,*), WIDTH  % TIME

!compute
!read while not end of file
20 READ (1,*), END=100) WIDTH

!transform width
WIDTHt = WIDTH/100      !converts from % to decimal
!Detected square in low res could be 0 to 2 (avg of 1)
!pixels smaller than actual due to quantization error
!in generating the square. To correct for this, the
!region of interest was expanded by 2.5 pixels. Net
!avg expansion = 1.5 pixels.
WIDTHt = WIDTHt + 1.5/PIX1
IF (WIDTHt .GT. 1) WIDTHt = 1

!number of pixels examined
LEVEL1 = PIX2 * PIX2    !high resolution
LEVEL2 = PIX1 * PIX1    !low res
VI. Appendix One: Program Listings

LEVEL2 = LEVEL2 + (WIDTHt * PIX2) ** 2

!time saved
SAVED = (LEVEL1 - LEVEL2) / LEVEL1
SAVED = SAVED * 100          !converts to %

!output results
WRITE (2,40) WIDTH, SAVED
40 FORMAT (' ',2F9.3)

!end compute
GOTO 20

!end
100  STOP 'NORMAL END OF MODEL.FOR'
END

2) Command Files

$! file name is BATCH.COM
$! it submits VISION.COM to batch
$! written by David Zokaites Jan-86
$ SUBMIT/NOTIFY/QUE=VAXB$LATE/AFTER=TOMORROW/LOG_FILE= -
 [DMZ5436.DEVELOP]VISION.LOG [DMZ5436.DEVELOP]VISION

$! file name is VISION.COM
$! it collects data for my thesis
$!
$ set noon !ignore errors
$ set def [dmz5436.thesis.develop]
$ assign input.dat sys$input
$ assign sysout.dat sys$output
$ nodeshow userload 2
$ run vision
$ nodeshow userload 2
$ deassign sys$input
$ deassign sys$output

3) Program

******************************************************************************
* INTRODUCTORY COMMENTS
******************************************************************************
! Written by David Zokaites
! 11/19/85 - 1/86
! Program name is VISION.FOR
! This program implements the two computer vision algorithms
Appendix One: Program Listings

---

VI. Appendix One: Program Listings

---

! described in the author's research thesis, "Computer Vision of a Moving Square using a Two-Level Data Hierarchy."

* INITIALIZE

**INITIALIZE**

! declare variables

! View1 is a low res view of the scene PIX1 pixels square, ! View2 high PIX2

BYTE  INTEGER  ! integer data type, 1 byte of memory
  SQUARE  ! below function to create moving square

INTEGER
  + PIX1, ! number of pixels per row of VIEW1
  + PIX2, ! number of pixels per row of VIEW2
  + TIME, ! time
  + NSCANS, ! # of times scene is scanned
  + CPU, ! elapsed CPU time in milliseconds
  + SCANT, ! cpu time for calling EDGE in 2-level hierarchy
  + OVERALL, ! overall cpu time
  + OVERALL1, OVERALL2, ! overall cpu time for view1 or view2
  + XEDGE(512), YEDGE(512), ! boundary of shape 512 = 4 * pix2
  + Ncorners, ! number of corners of the shape found in POLY
  + CORNERS(10,2), ! X,Y locations of the corners
  + Nedge, ! # of points on boundary of shape
  + NTEST1, NTEST2, ! # of points tested for collinearity see POLY ! for VIEW1 and VIEW2

! boundary values for square location, VIEW1
  + HIJ, LOWJ, HII, LOWI,
  + IA, IB, JA, JB ! do loop indices in examining VIEW2

REAL
  + CONST1(2), CONST2(2), ! constants used in COORD
  + LOCATE1, LOCATE2, ! find search location for high res view
  + EXPAND, ! expand area of moving square in high res
  + IANGLES(10), ! interior angles of the polygon
  + LINET, ! tolerance for two lines being collinear
  + POINTT, ! tolerance for points on a line being collinear
  + TOLL, ! tolerance for lengths of sides in % from avg

! parameters of moving square, see line 50
  + SPEEDR, SPEEDT, XI, YI, WIDTH, ANGLE0, ANGLET,
  + SIDES(10) ! lengths of the sides of the polygon fitted

LOGICAL
  + FLAG, ! T if a shape has been found
  + MEDIUM, ! T if medium length output was chosen
  + LONG, ! long
  + RECOGN, ! below pattern recognition function
  + RECKON ! T if a square was recognized

CHARACTER*1  OUTPUT ! used to input length of output
!LIB$INIT_TIMER and LIB$STAT_TIMER are part of the system's run time library. The former initializes the count of elapsed CPU time, the latter determines elapsed time.

!miscellaneous initialization
!common blocks
COMMON /SQUAR/ SPEEDR, SPEEDT, Xi, Yi, !for SQUARE + ANGLET, ANGLE0, WIDTH, TIME
COMMON /CORD/ CONST1, CONST2 !COORD subprogram
COMMON /SCN/ Nedge !EDGE
COMMON /POLY/ LINET, POINTT !POLY and COLINE
COMMON /LINE/ XEDGE, YEDGE !POLY, LINE, EDGE
COMMON /REC/ TOLL !RECOGN
COMMON /OUTPUT/ MEDIUM, LONG !output in POLY, EDGE

!*** OPEN LOOP TO READ WHILE NOT END OF FILE
!input length of output
WRITE (6,20)
20 FORMAT (/ + ' Input B for brief output, M for med, and L for long.' )
READ (5,30, ERR=200, END=200) OUTPUT
30 FORMAT (Al)
IF (OUTPUT .EQ. 'l' .OR. OUTPUT .EQ. 'L') THEN
  MEDIUM = .FALSE.
  LONG = .TRUE.
ELSE
  IF (OUTPUT .EQ. 'M' .OR. OUTPUT .EQ. 'm') THEN
    MEDIUM = .TRUE.
    LONG = .FALSE.
  ELSE
    MEDIUM = .FALSE.
    LONG = .FALSE.
  END IF
END IF
!open output files
OPEN (UNIT = 1, NAME = 'OUTPUTA.DAT', TYPE = 'NEW')
IF (LONG) THEN
OPEN (UNIT = 3, NAME = 'OUTPUTC.DAT', TYPE = 'NEW')
OPEN (UNIT = 4, NAME = 'OUTPUTD.DAT', TYPE = 'NEW')
END IF
OPEN (UNIT = 11, NAME = 'OUTPUTT.DAT', TYPE = 'NEW')
!output length of output
WRITE (1,110) OUTPUT
110 FORMAT ( ' Chosen length of output was ', Al )
IF (LONG) WRITE(1,112)
112 FORMAT ( ' (B = brief, M = medium, L = long) ' )
!open loop to read
10 CONTINUE

!initialize global variables
DATA  PIX1, PIX2 /32, 128/
LOCATE1 = PIX2 / PIX1
LOCATE2 = LOCATE1 / 2 - .5
EXPAND = 1.001 + 1 / LOCATE1
HII = 0  !variables for square location in VIEW1
HIJ = 0
LOWI = PIX1 + 1
LOWJ = PIX1 + 1
CONST1(1) = PIX1/2 + .5  !used in COORD
CONST1(2) = PIX2/2 + .5
CONST2(1) = PIX1  !used in COORD to scale coordinates
CONST2(2) = PIX2
OVERALL = 0

!read variables from screen
!initial description
IF (LONG) WRITE (6,40)
   FORMAT (/ + 'One unit of time is needed to examine the scene. Scene'/*
   + 'has a width of one scene width, is centered about the '/*
   + 'origin. Please input the following variables in '/*
   + 'these units to define the squares motion.'/*)
!table of variable identification
IF (LONG) WRITE (6,50)
   FORMAT (/* + 'VARIABLE DESCRIPTION' UNITS '/*
   + 'SPEEDR  speed of rotation' degrees/time '/*
   + 'SPEEDT  speed of translation' scene width/time'/*
   + 'NSCANS  # of times scene is examined' counts '/*
   + 'Xi, Yi  initial coordinates' scene width '/*
   + 'WIDTH   width' scene width '/*
   + 'ANGLE0  initial offset angle' degrees '/*
   + 'ANGLT   direction of translation' degrees '/*
   + 'ANGLT is measured counter clockwise from horizontal X'/*
   + 'axis. ANGLE0 is measured counter clockwise from Y axis.'/*
   + 'Positive SPEEDR results in counter clockwise rotation,'/*
   + 'negative values result in clockwise rotation. Only '/*
   + 'Xi, Yi, and SPEEDR are allowed to be negative. NSCANS '/*
   + 'must be >= 1 '/*)
READ (5,*, ERR = 220, END = 220) SPEEDR, SPEEDT, NSCANS,
+ Xi, Yi, WIDTH, ANGLE0, ANGLT
!correct for bad input
SPEEDT = ABS (SPEEDT)
   IF (NSCANS .LT. 1) NSCANS = 1
   WIDTH = ABS (WIDTH)
   ANGLE0 = ABS (ANGLE0)
   ANGLT = ABS (ANGLT)
!echo print input
IF (LONG) WRITE (6,60) SPEEDR, SPEEDT, NSCANS,
+ Xi, Yi, WIDTH, ANGLE0, ANGLT
   FORMAT (/* 'INPUT' '/*
   + 'SPEEDR  SPEEDT NSCANS XI Yi WIDTH'/*
+ ' ANGLEO ANGLET'/
+ 2F8.3, I8, 5F8.3/
!input polygon fitting, variables
IF (LONG) WRITE (6,70)
70 FORMAT (+
+ ' Input the following variables which '/
+ ' affect the polygon fitting done here: '/
+ ' LINET, collinearity tolerance in degrees >=1 for 2lines'/
+ ' POINTT, collinearity tolerance in pixel units >=.3 for'
+ ' points on a line'/
+ ' NTEST1, number of points >=3 grouped together for VIEW1'/
+ ' NTEST2, number of points >=3 grouped together for VIEW2'/)
READ (5,*, ERR = 230, END=230) LINET, POINTT, NTEST1, NTEST2
!correct for bad input
IF (LINET .LT. 1) LINET = 1
IF (POINTT .LT. .3 ) POINTT = .3
IF (NTEST1 .LT. 3) NTEST1 = 3
IF (NTEST2 .LT. 3) NTEST2 = 3

!echo print input
IF (LONG) WRITE (6,80) LINET, POINTT, NTEST1, NTEST2
80 FORMAT ( ' INPUT '/
+ ' LINET POINTT NTEST1 NTEST2', /2F8.3, 2I8/)

!input pattern recognition variable
IF (LONG) WRITE (6,90)
90 FORMAT ( ' Input the following variable which '/
+ ' affect the pattern recognition done here: '/
+ ' TOLL, tolerance for lengths of sides in pixel units'
+ ' change from the average '/
+ ' tolerance for interior angles is calculated from TOLL'/)
READ (5,*, ERR = 240, END=240) TOLL
!correct for bad input
TOLL = ABS (TOLL)
!echo input
IF (LONG) WRITE (6,100) TOLL
100 FORMAT ( ' INPUT '/
+ ' TOLL', /F8.3/)

!initialize output files
!square generation variables
IF (LONG) THEN
WRITE (1,40)
WRITE (1,50)
END IF
WRITE (1,60) SPEEDR, SPEEDT, NSCANS,
+ XI, Yi, WIDTH, ANGLEO, ANGLET
!polygon fitting vars
IF (LONG) WRITE (1,70)
WRITE (1,80) LINET, POINTT, NTEST1, NTEST2
!pattern recognition vars

IF (LONG) WRITE (1,90)
WRITE (1,100) TOLL
!elapsed cpu time
WRITE (11,115)
115 FORMAT (/ 
+ ' Elapsed CPU time for executing major routine calls '/ 
+ ' Subroutine Called View CPU Time (ms) '/)

!reverse direction of rotation for a given input SPEEDR
SPEEDR = - SPEEDR

********************************************************************
* MAIN SECTION OF PROGRAM
********************************************************************
!repeat the below NSCANS times to allow the shape to move
DO TIME = 0, (NSCANS -1)

!examine VIEW1 to find a moving shape
!call edge detection algorithm (scan scene)
CALL EDGE (1, PIX1, 1, PIX1, 1, PIX1, FLAG, NTEST1, CPU)
!determine elapsed CPU time
OVERALL1 = CPU
SCANT = CPU
WRITE (11,120) 'edge', '1', CPU
120 FORMAT (2 A14, 114)

!if a shape was not found, go to end of this section
IF ( .NOT. FLAG) THEN
WRITE (1,*)
WRITE (1,*) ' A SHAPE WAS NOT FOUND IN VIEW1 ' 
WRITE (1,*) ' VIEW2 OMITTED '
OVERALL = OVERALL + OVERALL1
ELSE

!call corner detection (polygon fitting) algorithm
CALL LIB$INIT_TIMER !initializes CPU time
CALL POLY (PIX1,Nedge, Ncorners, CORNERS, IANGLES, 
+ NTEST1, SIDES, 1)
!determine elapsed CPU time
CALL LIB$STAT_TIMER (2,CPU)
OVERALL1 = OVERALL1 + CPU
WRITE (11,120) 'poly', '1', CPU

!call pattern recognition algorithm
CALL LIB$INIT_TIMER !initializes CPU time
RECKON = RECOGN (NCORNERS, SIDES, IANGLES)
!determine elapsed CPU time
CALL LIB$STAT_TIMER (2,CPU)
OVERALL1 = OVERALL1 + CPU
WRITE (11,120) 'recogn', '1', CPU

!find region of the moving shape in VIEW1
!find hi and low values for I and J
    CALL LIB$INIT_TIMER
    DO I = 1,Nedge
        IF ( YEDGE(I) .GT. HII)   HII = YEDGE(I)
        IF ( YEDGE(I) .LT. LOWI)  LOWI = YEDGE(I)
        IF ( XEDGE(I) .GT. HIJ)   HIJ = XEDGE(I)
        IF ( XEDGE(I) .LT. LOWJ)  LOWJ = XEDGE(I)
    END DO
!output results
    IF (LONG) WRITE (1,150) LOWI, HII, LOWJ, HIJ
150   FORMAT (' The region of moving square in VIEW1 is ' /
            ' I = ', I2, ' to ', I2, /
            ' J = ', J, ' to ', J, /)

!EXPAND expands VIEW1 locations by EXPAND pixels
    IA = LOCATE1 * (LOWI - EXPAND) -LOCATE2
    IB = LOCATE1 * (HII + EXPAND)  -LOCATE2
    JA = LOCATE1 * (LOWJ - EXPAND) -LOCATE2
    JB = LOCATE1 * (HIJ + EXPAND)  -LOCATE2
!check for and correct out of bounds errors
    IF (IA .LT. 1) THEN
        IA = 1
        END IF
    IF (IB .GT. PIX2) THEN
        IB = PIX2
        END IF
    IF (JA .LT. 1) THEN
        JA = 1
        END IF
    IF (JB .GT. PIX2) THEN
        JB = PIX2
        END IF
!output results
    IF (LONG) WRITE (1,160) IA, IB, JA, JB
160   FORMAT (' The region under analysis in VIEW2 is ' /
            ' I = ', I3, ' to ', I3, /
            ' J = ', J, ' to ', J, /)

!determine elapsed CPU time
    CALL LIB$STAT_TIMER (2,CPU)
    OVERALL1 = OVERALL1 + CPU
    OVERALL = OVERALL + OVERALL1
    WRITE (11,120) 'viewl total', '1', OVERALL1

!Examine VIEW2 to resolve square
!edge detection
    CALL EDGE (2, PIX2, IA, IB, JA, JB, FLAG, NTEST2, CPU)
!elapsed cpu time
    OVERALL2 = CPU
    SCANT = SCANT + CPU
WRITE (11,120) 'edge', '2', CPU
!corner detection
CALL LIB$INIT_TIMER
CALL POLY (PIX2, Nedge, Ncorners, CORNERS, IANGLES, + NTEST2, SIDES, 2)
!elapsed cpu time
CALL LIB$STAT_TIMER (2,CPU)
OVERALL2 = OVERALL2 + CPU
WRITE (11,120) 'poly', '2', CPU

!pattern recognition
CALL LIB$INIT_TIMER
RECKON = RECOGN (NCORNERS, SIDES, IANGLES)
!determine elapsed CPU time
CALL LIB$STAT_TIMER (2,CPU)
OVERALL2 = OVERALL2 + CPU
WRITE (11,120) 'recogn', '2', CPU
WRITE (11,120) 'view2 total', '2', OVERALL2
OVERALL = OVERALL + OVERALL2

!repeat edge detection, for a one-level data hierarchy
!instead of two-level
CALL EDGE (2, PIX2, 1, PIX2, 1, PIX2, FLAG, NTEST2, CPU)
!elapsed cpu time
OVERALL2 = CPU
OVERALL = OVERALL + CPU
WRITE (11,120) '1 level scan', '2', CPU
WRITE (11,120) '2 level scan', '1&2', SCANT
WRITE (11,120) 'delta time', '1&2', (CPU - SCANT)
WRITE (11,*)

!end of this major section
END IF
END DO
WRITE (11,120) 'TOTAL', '1&2', OVERALL

!END PROGRAM, end loop to read data
WRITE (1,180)
IF (LONG) THEN
WRITE (3,180)
WRITE (4,180)
END IF
WRITE (11,180)
180 FORMAT (/ ' ',80('*')/ ' end of one data set '/ )
GOTO 10

200 STOP 'EOF while reading OUTPUT'
220 STOP 'NORMAL END OF VISION.FOR (EOF reading square )'
230 STOP 'EOF while reading polygon fitting'
240 STOP 'EOF while reading TOLL'
END
C) Coordinate System

***************
* subroutine: translate from array indices (I,J)
* to scene coordinates (X,Y)
***************
! Arrays are accessed in terms of array indices, I and J for
! example. This routine translates from array indices to the
! coordinate system of the scene itself. Scene coordinates
! are centered at 0,0 and have a width and height of one.

! This subroutine returns X and Y.

SUBROUTINE COORD (I, J, N, X, Y)
COMMON /CORD/ CONST1, CONST2 !constants
!declare variables
   INTEGER
   + I, J !array indices
   + N !number of VIEW = 1 for low resolution, 2 for high
   REAL
   + X, Y, !position in terms of scene coordinates
   + CONST1(2), !used to shift coordinates
   + CONST2(2) !scales coordinates

!shift axis
   X = J - CONST1(N)
   Y = CONST1(N) - I
!scale axis
   X = X/CONST2(N)
   Y = Y/CONST2(N)

END
Appendix One: Program Listings

D) Object and Scene

************************************************************
FUNCTION: CREATE MOVING SQUARE

Written by David Zokaites 3/85
Create the software equivalent of a square which can
translate as well as rotate. The square has a brightness
of 1 on a background of brightness 0. If the coordinates
of the pixel lie within the square, then a 1 is returned.
Otherwise, a 0 is returned.

initialize

BYTE FUNCTION SQUARE (I, J, N)

INTEGER SCALE, TIME
COMMON /SQUAR/ SPEEDR, SPEEDT, Xi, Yi,
+ ANGLET, ANGLET, WIDTH, TIME

!increment THETA
!theta = the square's angle from horizontal
!change in THETA results in rotation of square
THETA = ANGLET + SPEEDR * TIME

!to let the square translate, the square's definition is
!constant, the coordinate system of the scene is transformed

!transform array indices
!convert from array indices (I,J) to scene coordinates (X,Y)
CALL COORD (I, J, N, X, Y)

!find change in scene coordinates: allow for square's motion
R = TIME * SPEEDT
DELTAX = R * COSD (ANGLET)
DELTAY = R * SIND (ANGLET)

!implement above change
X = X - Xi - DELTAX
Y = Y - Yi - DELTAY

!convert scene coordinates to polar coordinates
!phi = angle from center of square to center of pixel
PHI = ATAND (Y/X)
COSPHI = COSD( PHI )
IF (COSPHI .NE. 0) THEN
Rxy = X/COSPHI !may give division by zero error
ELSE
Rxy = Y/SIND( PHI ) !correction for error
END IF

!distance from center of square to center of pixel
Rxy = ABS( Rxy )

!calculate square's location
!compute effective angle, ANGLEE, note the below is a new
!definition of phi. draw a line from the square's center
!to a corner. Phi = angle
!between this line and a point on the edge of square
ANGLEE = THETA + PHI
TEMP = ABS( ANGLEE) + 45
SCALE = INT( TEMP/90)
ANGLEE = ABS( ANGLEE - 90*SCALE)
!Rs = distance from square's center to edge
Rs = 1/ (2* COSD(ANGLEE))
Rs = ABS( Rs * WIDTH)   !scales square to width

!determine if pixel is part of square: core of this subroutine
IF (Rxy .LE. Rs) THEN
   SQUARE = 1     !pixel is part of square
ELSE
   SQUARE = 0      !pixel not part of square
END IF

RETURN
END
E) Edge Detection

***************************************************************************
* SUBROUTINE: DETECT EDGES OF MOVING SHAPE
***************************************************************************
!Written by David Zokaites 10/85 - 1/86
!This subroutine scans the scene in either low or high res and
!finds the edges of the moving shape. The top right corner of
!the square is the first point stored in X, Y. The rest of the
!edge points are stored in clockwise progression. This routine
!returns X, Y, FLAG3, and CPU. CPU = the cpu time required
!to execute this routine. Note that the time required to call
!SQUARE is not included in CPU. This is done by storing the
!results of SQUARE in a temporary file, and reading this file.

!Initialize
SUBROUTINE EDGE (NVIEW, PIX, Ibegin, Iend, Jbegin, Jend,
+ FLAG3, NTESTP, CPU)

COMMON /SCN/ Nedge
COMMON /LINE/ X, Y
COMMON /OUTPUT/ MEDIUM, LONG !controls length of output

!declare variables
BYTE
+ SQUARE, !below function to create moving square
+ VIEW (128,128) !a view of the scene

INTEGER
+ NVIEW, !# of view, = 1 or 2
+ Ibegin, Iend, !area of VIEW under interest
+ Jbegin, Jend, !area of VIEW under interest
+ X(512), Y(512), !shape boundary in array indices
+ Xfirst(128), Yfirst(128), !first part of shape boundary
+ FIRSTj, LASTj, !locate square terms of array index
+ OLDFIRSTJ, OLDLASTJ, !locate square for old row of VIEW
+ PIX, !pixels in a row of VIEW
+ PIX2, !2 * PIX
+ PIX2P1, !PIX2 + 1
+ Nedge, !# of points on boundary of shape
+ Nedge2, !number of points in Xfirst
+ NTESTP, !see POLY, # of points tested for collinearity
+ ONE(150), !array with 1,2,3,4,5,6,7,8,9,0 for output
+ DELTAJ, !jend - jbegin + 1
+ START, !number of spaces skipped in output statement
+ ITEN, !number of I10 format specifier
+ LSDIGIT, !function: find least significant digit
+ INPUT, !input to statement function
+ CPU, !elapsed cpu time
+ LINES !number of lines to skip in read statement
LOGICAL !flags for locating shape
+ FLAG1, !T if first "1" on a row was found
+ FLAG2, !T if last
+ FLAG3, !T if shape was found
+ FLAG4, !T if first row was found
+ FLAG5, !T if bottom of shape has been found
+ WITHIN, !T if bottom of shape found before end of field
+ BEYOND, !T if shape extends beyond bottom of field
+ MEDIUM, !T if medium length output chosen
+ MEDLONG, !T if long length output was chosen

!declare statement function to find least significant digit
LSDIGIT (INPUT) = 10.0 * ( FLOAT(INPUT )/10.0 + .01 -
+ INT (INPUT/10))

!LIB$INIT_TIMER and LIB$STAT_TIMER are part of system's
!run time library. The former initializes the count of
!elapsed CPU time, the latter determines elapsed time.

OPEN (UNIT = 1, NAME = 'OUTPUTA.DAT', TYPE = 'OLD')
OPEN (UNIT = 10, NAME = 'SQUARE.DAT', TYPE = 'NEW')

!initialize VIEW
DO I = 1, PIX
DO J = 1, PIX
   VIEW (I,J) = SQUARE (I,J,NVIEW)
END DO
WRITE (10,20) (VIEW (I,K), K = 1, PIX)
20 FORMAT ('<PIX>I1)
END DO
CLOSE (UNIT=10)

!call routine to initialize cpu time
CALL LIB$INIT_TIMER

!set pointer to SQUARE.DAT
OPEN (UNIT = 10, NAME = 'SQUARE.DAT', TYPE = 'OLD')
LINES = IBEGIN - 2
IF (LINES .EQ. 0) READ (10,*)
IF (LINES .GT. 0) READ (10,30)
30 FORMAT ( '<LINES>'(/) )

!initialize variables
Nedge = 0
Nedge2 = 0
PIX2 = 2 * PIX
PIX2P1 = PIX2 + 1
FLAG3 = .FALSE. !square not found yet
FLAG4 = .FALSE. !end of square not found yet
FLAG5 = .FALSE.
MEDLONG = LONG .OR. (MEDIUM .AND. (NVIEW .EQ. 1))

!initialize one, start
IF (MEDLONG) THEN
  DO I = 0, 14
    DO J = 1, 9
      ONE(J + 10*I) = J
    END DO
    ONE(J + 10*I) = 0
  END DO
  START = 10 - LSDIGIT (JBEGIN)
  DELTAJ = JEND - JBEGIN + 1
  ITEN = (JEND - LSDIGIT(JEND)) - (JBEGIN - LSDIGIT(JBEGIN))
  ITEN = ITEN/10 - 1
END IF

!initialize output
!check for printer overflow error
IF (MEDLONG .AND. Jend - Jbegin + 6 .GT. 130) WRITE (1,40)
40 FORMAT (''80(''*)''/ 'More than 130 characters per record,' + ' requires special printer set up '/ '', 80 (''*)'')
!normal output
IF (MEDLONG) WRITE (1,60) NVIEW,
+ ( ONE (I), I = (JBEGIN/10 +1), (JEND/10) ),
+ ( ONE (I), I = JBEGIN, JEND )
60 FORMAT ( '/ ' This is the scene as the program saw' + ' it for VIEW',I1 / ' ROW ' + I<START>, <ITEN>I10, /, 5X, <DELTAJ>I1 )

!Examine VIEW to find square
DO I = Jbegin, Jend
!initialize variables for VIEW
!read VIEW
  READ (10,65) ( VIEW(I,J), J = JBEGIN, JEND)
65 FORMAT ( T<JBEGIN+1>, <JEND-JBEGIN+2> I1 )
!flags for locating square, square not found yet
FLAG1 = .FALSE.
FLAG2 = .FALSE.
!vars for locating square, if square not found, vars=< 0
OLDFIRSTJ = FIRSTJ
OLDLASTJ = LASTJ
FIRSTj = 0
LASTj = 0

!open do loop to examine one row of VIEW
DO J = Jbegin, Jend

!LOCATE SQUARE: find first and last pixel per row = 1
!find first pixel by finding first "1"
IF ((VIEW(I,J) .EQ. 1) .AND. .NOT. FLAG1 ) THEN
  FIRSTj = J !first pixel = J
  FLAG1 = .TRUE. !set flag to find only one first "1"
  FLAG3 = .TRUE. !square has been found
END IF
!find last pixel by finding first "0" after first "1"
IF ((VIEW(I,J).EQ.0).AND..NOT.FLAG2.AND.FLAG1)THEN
  LASTj = J - 1 !last pixel = ...
  FLAG2 = .TRUE. !set flag to find only one last pixel
END IF
!last pixel: boundary at right edge
IF ((VIEW(I,J) .EQ. 1) .AND. (J .EQ. Jend)) THEN
  LASTj = Jend !last pixel = Jend
END IF

!finish examination of one row of VIEW
END DO
!fill array of shape's boundary
!first row of moving shape
IF ( FLAG1 .AND. .NOT. FLAG4 ) THEN
  DO K = LASTJ, (FIRSTJ+1), -1
    NEDGE2 = NEDGE2 + 1
    Xfirst(NEDGE2) = K
    Yfirst(NEDGE2) = I
  END DO
  FLAG4 = .TRUE.
END IF
!middle section of moving shape
IF (FLAG1) THEN
  NEDGE = NEDGE + 1
  X(NEDGE) = LASTJ
  Y(NEDGE) = I
  NEDGE2 = NEDGE2 + 1
  XFIRST(NEDGE2) = FIRSTJ
  YFIRST(NEDGE2) = I
END IF
!last row of moving shape
!see if shape extends beyond the bottom row of VIEW
BEYOND = (LASTJ .NE. 0) .AND. (I .EQ. Iend)
!see if bottom of shape has been found within VIEW
WITHIN = FLAG4 .AND. (FIRSTJ .EQ. 0)
WITHIN = WITHIN .AND. .NOT. FLAG5
IF BEYOND I = I + 1
  IF (BEYOND .OR. WITHIN) THEN
    IMINUS1 = I - 1
    DO K = (OLDLASTJ -1), OLDFIRSTJ, -1
      NEDGE = NEDGE + 1
      X(NEDGE) = K
      Y(NEDGE) = IMINUS1
    END DO
    DO K = (NEDGE2 - 1), 2, -1
      NEDGE = NEDGE + 1
      X(NEDGE) = XFIRST(K)
      Y(NEDGE) = YFIRST(K)
    END DO
    FLAG5 = .TRUE.
  END IF
END IF
!output data
IF (MEDLONG)
   + WRITE (1,70) I, (VIEW(I,K), K = Jbegin, Jend)
70 FORMAT (' ', I3, ' ', <PIX>I1)
   close outer do loop
END DO
!output data
IF (LONG) WRITE (1,80) NEDGE, NVIEW, (K, X(K), Y(K), K=1,Nedge)
80 FORMAT(/,I3, ' points describe shape boundary in VIEW',I1, /
   + ' Point #    X and Y in Array Indices '/
   + 1000(3I8/))
!end subroutine
   CALL LIB$STAT_TIMER(2,CPU)
   CLOSE (UNIT=10)
END
F) Corner Detection

1) Main Routine

*******************************************************************************
* SUBROUTINE: POLYGON FITTING
*******************************************************************************
Written by David Zokaites 11/85
This subroutine implements a polygon fitting or corner detection algorithm. For more details, see the author's thesis, "Computer Vision of a Moving Square using a Two-Level Data Hierarchy".

This subroutine returns Ncorners, CORNERS, IANGLES, and SIDES.

initialize
SUBROUTINE POLY (PIX, Nedge, Ncorners, CORNERS, IANGLES, + NtestP, SIDES, NVIEW)

COMMON /POLY/ LINET, POINTT
COMMON /LINE/ X,Y !for LINE subroutine
COMMON /OUTPUT/ MEDIUM, LONG !controls output

INTEGER
! boundary of moving shape, coordinates of the data points
+ X(512), Y(512),
+ FIRSTp, ! index to first point of collinearity test group
+ LASTp, ! index of the last
+ NTESTP, ! number of points above group normally contains
! usually 5 to 10, must be >3 here. bigger for VIEW2 than VIEW1 because larger squares should be broken into bigger segments for polygon fitting.
+ VERTEX(10), ! last polygon vertex declared
+ Ncorners, ! number of corners found in POLY
+ CORNERS(10,2), ! vertices
+ LINEAR, ! true if points are collinear
+ MAXp, ! index of point where max error (MAXe) found ! in a collinearity test
+ PIX, ! number of pixels in one row of VIEW
+ Nedge, ! # of points in X and Y boundary of shape
+ NVIEW ! # of view under test, = 1 or 2

REAL
+ ANGLE, ! function to calculate angle between adjacent sides of the fitted polygon
+ LENGTH, ! function to calculate length of polygon sides
+ IANGLES(10), ! interior angle among adjacent edges of polygon
+ TESTA, ! interior angle under test for merging
+ SIDES(10), ! lengths of the sides of the polygon
+ LINET, !collinearity tolerance between lines
   !ranges from 5 to 30?, minimum value here is 20?
+ POINTT !used in COLINE not POLY

LOGICAL
+ MEDIUM, !true if medium length output was chosen
+ LONG   !true if long ...

!open output files
OPEN (UNIT = 1, NAME = 'OUTPUTA.DAT', TYPE = 'OLD')
   IF (LONG) THEN
OPEN (UNIT = 3, NAME = 'OUTPUTC.DAT', TYPE = 'OLD')
OPEN (UNIT = 4, NAME = 'OUTPUTD.DAT', TYPE = 'OLD')
END IF
!initialize debugging output
IF (LONG) WRITE (3,10) NVIEW
10   FORMAT (/ ' Variables in COLINE for VIEW' ,Il, /
      + ' FIRSTp LASTp MAXp MAXe LINEAR ' )
IF (LONG) WRITE (4,20) NVIEW
20   FORMAT (/ + ' The angle between two lines as found in POLY for VIEW',Il/
      + ' The lines are described by three points./')
      + ' ANGLE POINT1 POINT2 POINT3 ')

!initialize variables
VERTEX(1) = 1
CORNERS(1,1) = X(1)
CORNERS(1,2) = Y(1)
Ncorners = 1
FIRSTp = 1
LASTp = NTESTP

!find vertices
DO WHILE (LASTP .LE. NEDGE)

CALL COLINE (FIRSTp,LASTp,MAXp,LINEAR)

IF (LINEAR) THEN
   IF (VERTEX(Ncorners) .EQ. FIRSTp) THEN
      !line from VERTEX(Ncorners) to FIRSTP=
      !line from FIRSTP to LASTP
   ELSE
      !determine if line from VERTEX(Ncorners) to FIRSTP
      !can be merged with line from FIRSTP to LASTP
      !compute angles
      IANGLES(Ncorners + 1) = ANGLE (VERTEX (NCORNERS),
         FIRSTP, LASTP)
      IF (LONG) WRITE (4,30) IANGLES(Ncorners + 1),
         VERTEX(Ncorners), FIRSTP, LASTP
      30   FORMAT (F9.3, 319)
      IF (VERTEX(Ncorners) .EQ. LASTP) THEN
         IF (LINEAR) THEN
            WRITE (4,30) IANGLES(Ncorners + 1),
               VERTEX(Ncorners), FIRSTP, LASTP
         30   FORMAT (F9.3, 319)
         ELSE
            WRITE (4,30) IANGLES(Ncorners + 1),
               VERTEX(Ncorners), LASTP, FIRSTP
         30   FORMAT (F9.3, 319)
         END IF
      END IF
   END IF
END IF
TESTA = 180 - IANGLES (NCORNERS + 1)
IF ( TESTA .LE. LINET ) THEN
!merge line segments; line from VERTEX(Ncorners) to
!FIRSTP = line from VERTEX(Ncorners) to LASTP
ELSE
NCORNERS = NCORNERS + 1
VERTEX(Ncorners) = FIRSTP
CORNERS(Ncorners,1) = X(VERTEX(Ncorners))
CORNERS(Ncorners,2) = Y(VERTEX(Ncorners))
END IF
END IF
!increment indices to points under test
FIRSTp = LASTp         !step ten
LASTp = LASTp + NTESTp
!check for LASTP being too big
IF ((LASTP .GT. NEDGE) .AND. ((NEDGE-FIRSTP) .GE. 1))
  LASTP = NEDGE
!split last line into 2 segments at MAXp
ELSE !continue if started
  LASTp = MAXp
END IF
END DO !end do while started
!end of subroutine
!IANGLES(1) was not found yet
IANGLES(1) = ANGLE (VERTEX( NCORNERS), VERTEX(1),VERTEX(2))
IF (LONG) WRITE (4,30) IANGLES(Ncorners + 1),
+ VERTEX(Ncorners), FIRSTP, LASTP
!perform merge check on first and last corners
TESTA = 180 - IANGLES (1)
IF ( TESTA .LE. LINET ) THEN    !merge
NCORNERS = NCORNERS - 1
DO I = 1, NCORNERS
DO J = 1, 2
CORNERS (I,J) = CORNERS ( (I+1), J )
VERTEX (I) = VERTEX (I+1)
END DO
END DO
!recompute interior angle
IANGLES(1) = ANGLE (VERTEX( NCORNERS), VERTEX(1), VERTEX(2) )
IF (LONG) WRITE (4,30) IANGLES(Ncorners + 1),
+ VERTEX(Ncorners), FIRSTP, LASTP
END IF
!compute sides
DO I = 1, (NCORNERS -1)
SIDES(I) = LENGTH ( VERTEX(I), VERTEX(I+1) )
END DO
SIDES (NCORNERS) = LENGTH ( VERTEX( NCORNERS), 1)
!output data
WRITE (1,40) Ncorners, NVIEW, (K, IANGLES(K), SIDES(K),
+ (CORNERS (K,L), L=1,2), K=1,NCorners)
40   FORMAT (' ', Il, ' Corners of the moving shape were found')
+ ' in VIEW',/Il,/ + Interior Side / + ' Corner # Angle Length X and Y in array indices' + / 10(I9, 2F8.3, 2I8 /) / )

END

2) Collinearity Tests

************************************************************
* SUBROUTINE: TEST POINTS FOR COLLINEARITY
************************************************************
! This subroutine returns MAXp and LINEAR.

!initialize
SUBROUTINE COLINE (FIRSTp,LASTp,MAXp,LINEAR)

COMMON /POLY/ LINET, POINTT
COMMON /LINE/ X,Y
COMMON /OUTPUT/ MEDIUM, LONG

INTEGER + I, !do loop index + X(512), Y(512), !coordinates of the data points + LINEa(2), !line from FIRSTp to LASTp, test collinearity + FIRSTp, !index of first point of group tested + LASTp, !index of the last point ... + LINEAR, !true when points of LINEa are collinear + MAXp, !index of point where max error (MAXe) found !in a collinearity test

REAL + DET, !determinate of X and Y from firstp to lastp + LEN, !length of LINEa + LENA, !sum point(i) to point(i+1) lengths + ERROR, !error at a point + POINTT, !collinearity tolerance for points on a line !minimum value here is 1? + LINET, !used in POLY not COLINE + MAXe, !error at MAXp, MAXp = point of max error + TEMP1, TEMP2 !intermediate variables

LOGICAL + MEDIUM, !true if medium length output was chosen + LONG !true if long

IF (LONG) OPEN (UNIT = 3, NAME = 'OUTPUTC.DAT', TYPE='OLD')

!initialize variables
MAXp = FIRSTp
MAXe = 0.0
LINEAR = .TRUE.

!evaluate LINEa
DET = Y(LASTp)*X(FIRSTp) - Y(FIRSTp)*X(LASTp)
CALL LINE (FIRSTp, LASTp, LINEA)
LEN = LINEA(1) * LINEA(1) + LINEA(2) * LINEA(2)
LEN = SQRT ( LEN )

!first collinearity test: compare LENA to LEN
!compute LENA
LENA = 0
DO I = (FIRSTp + 1), LASTp
  TEMP1 = ( X(I) - X(I - 1) ) * ( X(I) - X(I - 1) )
  TEMP2 = ( Y(I) - Y(I - 1) ) * ( Y(I) - Y(I - 1) )
  LENA = LENA + SQRT ( TEMP1 + TEMP2 )
END DO
LENA = LENA / LEN

!test collinearity
IF ( LENA .LT. 1.1 ) THEN
  ! linear = .true.
  IF (LONG) WRITE (3, 65) FIRSTp, LASTp, MAXp, MAXe, LINEAR
  RETURN
END IF
IF ( LENA .GT. 1.5 ) LINEAR = .FALSE.

!find MAXe
DO I = (FIRSTp+1), (LASTp-1)
  ERROR = DET - LINEa(2)*X(I) + LINEa(1)*Y(I)
  IF ( ABS(ERROR) .GT. MAXe ) THEN
    MAXe = ABS(ERROR)
    MAXp = I
  END IF
END DO
MAXe = MAXe / LEN

!compute LINEAR
IF (MAXe .GE. POINTT) LINEAR = .FALSE.
! if maxe > pointt linear = .true.

!end of subroutine
!output variables
IF (LONG) WRITE (3, 65) FIRSTp, LASTp, MAXp, MAXe, LINEAR
65 FORMAT( ' ', 3I8, F3.3, L8)
!end
END

3) Associated Subroutines

**********************************************************************
* MISC SUBROUTINES AND FUNCTIONS
**********************************************************************
! compute angle between adjacent edges of the fitted polygon
! using the law of cosines: \( a \times a = b \times b + c \times c - 2 \times b \times c \times \cos(A) \)

REAL FUNCTION ANGLE (VERTEX, FIRSTP, LASTP)

INTEGER ! pointers to X, Y endpoints of adjacent sides
+ VERTEX, FIRSTP, LASTP

REAL VTOF, ! distance from vertex to lastp,
+ TEMP, ! intermediate variable
+ FTOL, VTOL, ! distance firstp to lastp, vertex to lastp
+ LENGTH, ! below function
+ SMALL ! small correction for round off error

PARAMETER (SMALL = .000003)

VTOF = LENGTH (VERTEX, FIRSTP)
FTOL = LENGTH (FIRSTP, LASTP)
VTOL = LENGTH (VERTEX, LASTP)

ANGLE = VTOL * VTOL - VTOF * VTOF - FTOL * FTOL
TEMP = -2 * VTOF * FTOL
! check for potential division by 0 error
IF ( TEMP .EQ. 0 ) THEN
  ANGLE = 90
RETURN
END IF

ANGLE = ANGLE / TEMP
! check for abs(angle) > 1 due to round off error
IF ( ABS (ANGLE) .GT. 1 ) THEN
  ANGLE = ANGLE - SIGN (SMALL, ANGLE)
END IF

ANGLE = ACOSD (ANGLE)
RETURN
END

! compute the length of one side of the fitted polygon
! this function returns length
REAL FUNCTION LENGTH (BEGIN, END)

INTEGER
+ BEGIN, END, ! pointers to X, Y endpoints of the line
+ DELTA(2)

CALL LINE (BEGIN, END, DELTA)
LENGTH = DELTA(1) * DELTA(1) + DELTA(2) * DELTA(2)
LENGTH = SQRT ( LENGTH )

RETURN
END

! compute delta X and delta Y for the line from Xbegin, Ybegin
!to Xend, Yend. This subroutine returns DELTA()

SUBROUTINE LINE (BEGIN, END, DELTA)

COMMON /LINE/ X,Y

INTEGER
+ BEGIN, END, !indices to ends of the line
+ X(512), Y(512), !boundary of shape
+ DELTA(2) !parameters of the line

DELTA(1) = X(END) - X(BEGIN) !delta x
DELTA(2) = Y(END) - Y(BEGIN) !delta y

END
G) Pattern Recognition

******************************************************************************
* FUNCTION: PATTERN RECOGNITION
******************************************************************************
!Written by David Zokaites 11/85 - 12/85

!This function determines if a shape is square. If it is !square, then a value of .TRUE. is returned. Otherwise, .FALSE. !is returned.

!initialize
LOGICAL FUNCTION RECOGN (NCORNERS, SIDES, IANGLES)

!declare variables
REAL
+ IANGLES (10), !interior angles
+ SIDES (10), !lengths of the sides of the polygon
+ AVERAGE, !average of the sides
+ ERROR, !average deviation of one side from the average
+ ANGLEREF, !reference interior angle
+ CHANGE, !an interior angle - angleref
+ TOLL, !tolerance for lengths of sides change from avg
+ TOLA, !tolerance for interior angles in degrees
+ TOLAIN !input tolerances

INTEGER
+ NCORNERS, !number of corners found
+ NSIDES !number of sides the shape should have

DATA NSIDES, ANGLEREF, AVERAGE /4, 90, 0/

OPEN (UNIT=1, NAME = 'OUTPUTA.DAT', TYPE = 'OLD')

!determine if shape is a square
!first criteria: number of corners
IF (NCORNERS .NE. NSIDES) THEN
RECOGN = .FALSE.
WRITE (1,20) NCORNERS, NSIDES
20 FORMAT (' The shape has the wrong number of corners: ', + IL, ' not ', IL)
WRITE (1,30)
30 FORMAT (' Therefore the shape is NOT a square. '/)
RETURN
END IF
!second criteria: lengths of sides
!compute AVERAGE
AVERAGE = 0
DO I = 1, NCORNERS
    AVERAGE = AVERAGE + SIDES(I)
END DO
AVERAGE = AVERAGE / NCORNERS
! compare sides to average
DO I = 1, NCORNERS
    ERROR = SIDES(I) - AVERAGE
    IF ((TOLL - ERROR) .LT. 0) THEN
        RECOGN = .FALSE.
        WRITE (1,40) I, SIDES(I), (ERROR - TOLL)
        40 FORMAT ('Side #', I1, ' of length ', F8.3,
            + ' is out of tolerance by ', F8.3, ' % ')
        WRITE (1,30)
    END IF
END DO

! third criteria: interior angles
! compute TOLA from TOLL
    TOLA = ATAND( TOLL/ AVERAGE )
! make the test
DO I = 1, NCORNERS
    CHANGE = ABS(IANGLES(I) - ANGLEREF)
    IF (CHANGE .GT. TOLA ) THEN
        RECOGN = .FALSE.
        WRITE (1,50) I, IANGLES(I), (CHANGE - TOLA)
    END IF
50 FORMAT ('Interior angle #', I1, F8.3,
        + ' is out of tolerance by ', F8.3, ' degrees. ')
WRITE (1,30)
RETURN
END IF
END DO
! if all criteria pass
RECOGN = .TRUE.
WRITE (1,60)
60 FORMAT ('The shape passes all criteria and ' + 'therefore IS a square.'/)

RETURN
END
H) Statistical Analysis

Program name is [VISION]STATS.FOR
Written by David Zokaites 15-Jan-86
This program characterizes mean, standard deviation, range.
The input is OUTPUTT.DAT from VISION.FOR

!initialize
!declare vars
INTEGER
  + POINTS, !number of numbers
  + DATA(3), !input data
  + MIN(3), MAX(3),!minimum and maximum values of the data
  + RANGE(3), !max - min
  + SUM(3), !sum of data
  + SUMSQ(3) !sum of data * data

REAL
  + MEAN(3), !mean of the input data
  + SDEV(3), !standard deviation
  + DELTA !% change from mean(1) to mean(2)

LOGICAL NEWSET !T if read from new data set

CHARACTER*5 TEST !test string from input data file
CHARACTER*7 LABEL(3) !labels output

!misc initialization
!open files
OPEN (UNIT=1, NAME = 'OUTPUTT.DAT', TYPE = 'OLD') !input
OPEN (UNIT=3, NAME = 'STATSB.DAT', TYPE = 'NEW') !output
OPEN (UNIT=4, NAME = 'STATSA.DAT', TYPE = 'NEW') !output
!initialize output files
WRITE (3,10)
WRITE (4,20)
!set LABEL
  DATA LABEL /'1 level','2 level','delta'/

!open loop to read and examine new sets of data while
!not end of file
30 CONTINUE
! initialize vars
  NEWSET = .FALSE.
  POINTS = 0
  DO I = 1, 3
    MIN(I) = 1000000
    MAX(I) = -1000000
    SUM(I) = 0
    SUMSQ(I) = 0
  END DO
!open loop to examine old data set
40 CONTINUE
READ (1, 50, END = 100) TEST
!increment sums if ...
   IF ( TEST .EQ. 'view2') THEN
!increment sums
   READ (1,55) DATA
   POINTS = POINTS + 1
   DO I = 1, 3
      IF ( DATA(I) .LT. MIN(I) ) MIN(I) = DATA(I)
      IF ( DATA(I) .GT. MAX(I) ) MAX(I) = DATA(I)
      SUM (I) = SUM(I) + DATA(I)
      SUMSQ(I) = SUMSQ(I) + DATA(I) * DATA(I)
   END DO

!if end of data set...
ELSE IF (TEST .EQ. '*****') THEN
!compute statistics
   NEWSET = .TRUE.
   IF (POINTS .GE. 2) THEN
      DO I = 1, 3 !compute statistics
         RANGE (I) = MAX(I) - MIN(I)
         MEAN (I) = FLOAT ( SUM(I) ) / POINTS
         SDEV(I) = POINTS * SUMSQ(I) - SUM(I) * SUM(I)
         SDEV(I) = SDEV(I) / ( POINTS * (POINTS - 1) )
         SDEV(I) = SQRT ( ABS (SDEV(I)))
      END DO
      DELTA = 100 * MEAN(3) / MEAN(1)
!output data
      WRITE (3, 70) MEAN, DELTA
      WRITE (4, 80) ( MEAN(I), SDEV(I), RANGE(I), MIN(I),
                      + MAX(I), LABEL(I), POINTS, I = 1,3)
   END IF

END IF
!close loop to examine new sets of data in file
IF (NEWSET) THEN
   GOTO 30
ELSE
   GOTO 40 !goto examine old data set
END IF

!FORMAT STATEMENTS
!initialize output files
10 FORMAT ( mean '/
   + ' 1 level 2 level delta delta% ')
20 FORMAT ( + ' Statistics calculated in [.VISION]STATS.FOR'
   + ' from data in OUTPUTT.DAT')
+ ' mean (mean), standard deviation (sdev); range, from'/
+ ' minimum (min) to maximum (max), for POINTS points. '/
+ ' MEAN SDEV RANGE MIN MAX DATA TYPE POINTS'/
!read test
  50 FORMAT (T4, A5)
!read data
  55 FORMAT (3(T35, I8, '/))
!output statistics
  70 FORMAT (4 F10.3)
  80 FORMAT (3(', 2F8.3, 3I7, A10, I8/))

!finalize
 100 STOP 'NORMAL END OF STATS.FOR'
 110 STOP 'END OF FILE IN STATS'
    END
VII. Appendix Two: Sample Output

Contents

A. CPU Time Variability Analysis ............... 85
   1. Output from BATCH.COM and CPU.COM ... 85
   2. Output from CPU.FOR ...................... 85
   3. Output from CPUJSTATS.FOR ............. 86

B. Main Vision Algorithm ....................... 87
   1. Modeling of CPU Time Saved .............. 87
   2. Main Output ............................... 87
   3. Elapsed CPU Time Data ................... 90
   4. Statistical Analysis ....................... 90
      a. First Output File .................... 90
      b. Second Output File .................. 91
A) CPU Time Variability Analysis

1) Output from BATCH.COM and CPU.COM

Note that the following output file was shortened to fit the available space.

count = 0
Userload on VAXB.
Tuesday, January 14, 1986 1:32 PM is: 43 users, 1 batch job,
Free memory is: 2686 pages, System load average is: 3.58 6.58

NORMAL END OF CPU.FOR
count = 1
Userload on VAXB.
Tuesday, January 14, 1986 3:23 PM is: 46 users, 1 batch job,
Free memory is: 3144 pages, System load average is: 8.40 9.32

NORMAL END OF CPU.FOR
count = 2
Userload on VAXB.
Tuesday, January 14, 1986 6:09 PM is: 30 users, 1 batch job,
Free memory is: 7094 pages, System load average is: 4.01 3.53

2) Output From CPU.FOR

Note that the following output file was shortened to fit the available space.

Computation CPU time
in ms for repetitions of same thing
Number of repetitions = 35

1
0
2
1
1
1
2
2
2
1
2
1
3) Output from [.CPU]STATS.FOR

ELAPSED CPU TIME VARIABILITY ANALYSIS DATA

time = time the data was collected
users = number of users on the system, excluding batch
load = avg system over last minute in # processes waiting
process = process using cpu time, computation or I/O

The statistical parameters calculated follow:
mean (mean), standard deviation (sdev); range (range), from
minimum (min) to maximum (max), for POINTS points.

file = file containing elapsed cpu time

<table>
<thead>
<tr>
<th>TIME</th>
<th>USERS</th>
<th>LOAD</th>
<th>PROCESS</th>
<th>MEAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>January 14, 1986 1:32 P 43 3.58 Com 1.257  --&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>January 14, 1986 1:32 P 43 3.58 Com 4.829  --&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| January 14, 1986 1:32 P 43 3.58 Com 18.314 -->
| January 14, 1986 1:32 P 43 3.58 Com 76.829 -->
| January 14, 1986 1:32 P 43 3.58 Com 185.886-->
| January 14, 1986 1:32 P 43 3.58 Com 437.629-->
| January 14, 1986 3:23 P 46 8.40 Com 1.200  --> |
| January 14, 1986 3:23 P 46 8.40 Com 4.086  --> |
| January 14, 1986 3:23 P 46 8.40 Com 13.000 --> |
| January 14, 1986 3:23 P 46 8.40 Com 75.829 --> |
| January 14, 1986 3:23 P 46 8.40 Com 186.000-->
| January 14, 1986 3:23 P 46 8.40 Com 439.314-->
| January 14, 1986 6:09 P 30 4.01 Com 1.200  --> |
| January 14, 1986 6:09 P 30 4.01 Com 3.886  --> |
| January 14, 1986 6:09 P 30 4.01 Com 17.371 --> |
| January 14, 1986 6:09 P 30 4.01 Com 76.400 --> |
| January 14, 1986 6:09 P 30 4.01 Com 185.571-->
| January 14, 1986 6:09 P 30 4.01 Com 433.914-->

The above arrows note where the above file was shortened to fit on this page.
B) Main Vision Algorithm

1) Modeling of CPU Time Saved

% time saved with 2-level data hierarchy

<table>
<thead>
<tr>
<th>WIDTH</th>
<th>% TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.500</td>
<td>93.080</td>
</tr>
<tr>
<td>10.000</td>
<td>91.593</td>
</tr>
<tr>
<td>25.000</td>
<td>84.937</td>
</tr>
<tr>
<td>40.000</td>
<td>73.780</td>
</tr>
<tr>
<td>65.000</td>
<td>45.187</td>
</tr>
<tr>
<td>70.000</td>
<td>37.968</td>
</tr>
<tr>
<td>85.000</td>
<td>13.312</td>
</tr>
<tr>
<td>100.000</td>
<td>-6.250</td>
</tr>
</tbody>
</table>

2) Main Output

Note that the following output file was shortened to fit the available space.

Chosen length of output was M

INPUT

SPEEDR  SPEEDT  NSCANS  Xi  Yi  WIDTH  ANGLEO  ANGLET
20.000  0.100  2  0.000  0.000  0.400  0.000  0.000

INPUT

LINET  POINTT  NTEST1  NTEST2
45.000  2.500  16  32

INPUT

TOLL
2.400

This is the scene as the program saw it for VIEW1

ROW  1  2  3
1  12345678901234567890123456789012
2  00000000000000000000000000000000
3  00000000000000000000000000000000
4  00000000000000000000000000000000
5  00000000000000000000000000000000
6  00000000000000000000000000000000
7  00000000000000000000000000000000
8 00000000000000000000000000000000
9 00000000000000000000000000000000
10 00000000000000000000000000000000
11 00000000000000000000000000000000
12 00000000000000000000000000000000
13 00000000000000000000000000000000
14 00000000000000000000000000000000
15 00000000000000000000000000000000
16 00000000000000000000000000000000
17 00000000000000000000000000000000
18 00000000000000000000000000000000
19 00000000000000000000000000000000
20 00000000000000000000000000000000
21 00000000000000000000000000000000
22 00000000000000000000000000000000
23 00000000000000000000000000000000
24 00000000000000000000000000000000
25 00000000000000000000000000000000
26 00000000000000000000000000000000
27 00000000000000000000000000000000
28 00000000000000000000000000000000
29 00000000000000000000000000000000
30 00000000000000000000000000000000
31 00000000000000000000000000000000
32 00000000000000000000000000000000

4 Corners of the moving shape were found in VIEW1

<table>
<thead>
<tr>
<th>Corner #</th>
<th>Angle</th>
<th>Length</th>
<th>X and Y in array indices</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>90.000</td>
<td>11.000</td>
<td>22 11</td>
</tr>
<tr>
<td>2</td>
<td>90.000</td>
<td>11.000</td>
<td>22 22</td>
</tr>
<tr>
<td>3</td>
<td>90.000</td>
<td>11.000</td>
<td>11 22</td>
</tr>
<tr>
<td>4</td>
<td>90.000</td>
<td>11.000</td>
<td>11 11</td>
</tr>
</tbody>
</table>

The shape passes all criteria and therefore IS a square.

4 Corners of the moving shape were found in VIEW2

<table>
<thead>
<tr>
<th>Corner #</th>
<th>Angle</th>
<th>Length</th>
<th>X and Y in array indices</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>90.000</td>
<td>51.000</td>
<td>90 39</td>
</tr>
<tr>
<td>2</td>
<td>90.000</td>
<td>51.000</td>
<td>90 90</td>
</tr>
<tr>
<td>3</td>
<td>90.000</td>
<td>51.000</td>
<td>39 90</td>
</tr>
<tr>
<td>4</td>
<td>90.000</td>
<td>51.000</td>
<td>39 39</td>
</tr>
</tbody>
</table>

The shape passes all criteria and therefore IS a square.

This is the scene as the program saw it for VIEW1

ROW 1 2 3
12345678901234567890123456789012
1 00000000000000000000000000000000
2 00000000000000000000000000000000
The shape passes all criteria and therefore IS a square.

The shape passes all criteria and therefore IS a square.
3) Elapsed Cpu Time Data

Note that the following output file was shortened to fit the available space.

<table>
<thead>
<tr>
<th>Elapsed CPU time for executing major subroutine calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subroutine Called</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>edge</td>
</tr>
<tr>
<td>poly</td>
</tr>
<tr>
<td>recogn</td>
</tr>
<tr>
<td>view1 total</td>
</tr>
<tr>
<td>edge</td>
</tr>
<tr>
<td>poly</td>
</tr>
<tr>
<td>recogn</td>
</tr>
<tr>
<td>view1 total</td>
</tr>
<tr>
<td>1 level scan</td>
</tr>
<tr>
<td>2 level scan</td>
</tr>
<tr>
<td>delta time</td>
</tr>
<tr>
<td>edge</td>
</tr>
<tr>
<td>poly</td>
</tr>
<tr>
<td>recogn</td>
</tr>
<tr>
<td>view1 total</td>
</tr>
<tr>
<td>edge</td>
</tr>
<tr>
<td>poly</td>
</tr>
<tr>
<td>recogn</td>
</tr>
<tr>
<td>view1 total</td>
</tr>
<tr>
<td>1 level scan</td>
</tr>
<tr>
<td>2 level scan</td>
</tr>
<tr>
<td>delta time</td>
</tr>
<tr>
<td>TOTAL</td>
</tr>
</tbody>
</table>

4) Statistical Analysis

a) First Output File

Note that the following output file was shortened to fit the available space.

Statistics calculated in E.VISION1STATS.FOR from OUTPUTT.DAT mean (mean), standard deviation (sdev); range (range), from minimum (min) to maximum (max), for POINTS points.

<table>
<thead>
<tr>
<th>DATA TYPE</th>
<th>MEAN</th>
<th>SDEV</th>
<th>RANGE</th>
<th>MIN</th>
<th>MAX</th>
<th>POINTS</th>
</tr>
</thead>
</table>
VII. Appendix Two: Sample Output

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>262.667</td>
<td>1.496</td>
<td>5</td>
<td>260</td>
<td>265</td>
<td>1 level</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>35.667</td>
<td>2.289</td>
<td>9</td>
<td>30</td>
<td>39</td>
<td>2 level</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>227.000</td>
<td>2.928</td>
<td>10</td>
<td>222</td>
<td>232</td>
<td>delta</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>261.933</td>
<td>1.624</td>
<td>6</td>
<td>259</td>
<td>265</td>
<td>1 level</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>40.667</td>
<td>1.633</td>
<td>6</td>
<td>37</td>
<td>43</td>
<td>2 level</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>221.267</td>
<td>1.668</td>
<td>6</td>
<td>218</td>
<td>224</td>
<td>delta</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>265.133</td>
<td>3.067</td>
<td>12</td>
<td>262</td>
<td>274</td>
<td>1 level</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>61.667</td>
<td>2.845</td>
<td>10</td>
<td>55</td>
<td>65</td>
<td>2 level</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>203.467</td>
<td>3.739</td>
<td>15</td>
<td>197</td>
<td>212</td>
<td>delta</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

b) Second Output File

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>mean</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 level</td>
<td>2 level</td>
<td>delta</td>
<td>delta%</td>
<td></td>
</tr>
<tr>
<td>262.667</td>
<td>35.667</td>
<td>227.000</td>
<td>86.421</td>
<td></td>
</tr>
<tr>
<td>261.933</td>
<td>40.667</td>
<td>221.267</td>
<td>84.474</td>
<td></td>
</tr>
<tr>
<td>265.133</td>
<td>61.667</td>
<td>203.467</td>
<td>76.741</td>
<td></td>
</tr>
<tr>
<td>263.267</td>
<td>85.533</td>
<td>177.733</td>
<td>67.511</td>
<td></td>
</tr>
<tr>
<td>264.867</td>
<td>153.667</td>
<td>111.200</td>
<td>41.983</td>
<td></td>
</tr>
<tr>
<td>266.400</td>
<td>177.667</td>
<td>88.733</td>
<td>33.308</td>
<td></td>
</tr>
<tr>
<td>267.733</td>
<td>254.600</td>
<td>13.133</td>
<td>4.905</td>
<td></td>
</tr>
<tr>
<td>268.000</td>
<td>288.467</td>
<td>-20.467</td>
<td>-7.637</td>
<td></td>
</tr>
<tr>
<td>263.133</td>
<td>40.600</td>
<td>222.533</td>
<td>84.571</td>
<td></td>
</tr>
<tr>
<td>262.800</td>
<td>66.400</td>
<td>196.400</td>
<td>74.734</td>
<td></td>
</tr>
<tr>
<td>262.533</td>
<td>128.600</td>
<td>133.933</td>
<td>51.016</td>
<td></td>
</tr>
<tr>
<td>264.933</td>
<td>253.067</td>
<td>11.867</td>
<td>4.479</td>
<td></td>
</tr>
<tr>
<td>265.000</td>
<td>284.000</td>
<td>-19.000</td>
<td>-7.170</td>
<td></td>
</tr>
<tr>
<td>267.000</td>
<td>290.800</td>
<td>-23.800</td>
<td>-8.914</td>
<td></td>
</tr>
<tr>
<td>266.600</td>
<td>290.467</td>
<td>-23.867</td>
<td>-8.952</td>
<td></td>
</tr>
</tbody>
</table>
David Zokaites was born and raised in Pittsburgh, PA. He attended North Catholic High School and is a graduating senior at Rochester Institute of Technology, majoring in Imaging and Photographic Science.

His interests include Aiki Jujutsu, camping, and carpentry. Mr. Zokaites is married and resides in Rochester, NY with his lovely wife, Coni, and their two cats, Jonathan and Harry.