1972

TAG: A Terminal for Animated Graphics. Its design and simulation

Wayne Charles Dowling
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Part of the Graphics and Human Computer Interfaces Commons

Recommended Citation
https://lib.dr.iastate.edu/rtd/5902

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
DOWLING, Wayne Charles, 1938-
TAG: A TERMINAL FOR ANIMATED GRAPHICS. ITS
DESIGN AND SIMULATION.

Iowa State University, Ph.D., 1972
Computer Science

University Microfilms, A XEROX Company, Ann Arbor, Michigan

© 1972

Wayne Charles Dowling

ALL RIGHTS RESERVED

THIS DISSERTATION HAS BEEN MICROFILMED EXACTLY AS RECEIVED.
TAG: A Terminal for Animated Graphics. Its design and simulation

by

Wayne Charles Dowling

A Dissertation Submitted to the Graduate Faculty in Partial Fulfillment of The Requirements for the Degree of DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames Iowa

1972

Copyright © Wayne Charles Dowling, 1972. All rights reserved.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>ABSTRACT</th>
<th>viii</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>A. Background</td>
<td>1</td>
</tr>
<tr>
<td>B. Review of Literature</td>
<td>5</td>
</tr>
<tr>
<td>C. Statement of the Problem</td>
<td>10</td>
</tr>
<tr>
<td>D. Facilities Employed</td>
<td>11</td>
</tr>
<tr>
<td>II. TERMINAL DESIGN</td>
<td>13</td>
</tr>
<tr>
<td>A. Overview</td>
<td>13</td>
</tr>
<tr>
<td>1. Re-statement or goals</td>
<td>13</td>
</tr>
<tr>
<td>2. Primary techniques</td>
<td>16</td>
</tr>
<tr>
<td>3. Secondary techniques</td>
<td>16</td>
</tr>
<tr>
<td>4. General structure</td>
<td>18</td>
</tr>
<tr>
<td>B. Pictorial Processing</td>
<td>19</td>
</tr>
<tr>
<td>1. Substructure manipulation</td>
<td>19</td>
</tr>
<tr>
<td>2. Projection to picture plane</td>
<td>20</td>
</tr>
<tr>
<td>3. Windowing</td>
<td>22</td>
</tr>
<tr>
<td>a. Points</td>
<td>23</td>
</tr>
<tr>
<td>b. Lines</td>
<td>26</td>
</tr>
<tr>
<td>C. Storage and Instruction Structure</td>
<td>30</td>
</tr>
<tr>
<td>1. General format of instruction</td>
<td>30</td>
</tr>
<tr>
<td>2. Control transfer instructions</td>
<td>31</td>
</tr>
<tr>
<td>3. Frame counter related instructions</td>
<td>34</td>
</tr>
<tr>
<td>a. Frame counter actions</td>
<td>34</td>
</tr>
<tr>
<td>b. Conditional actions</td>
<td>35</td>
</tr>
<tr>
<td>c. Other actions</td>
<td>37</td>
</tr>
<tr>
<td>4. Matrix manipulation</td>
<td>38</td>
</tr>
<tr>
<td>5. Draw instructions</td>
<td>41</td>
</tr>
<tr>
<td>a. Function</td>
<td>41</td>
</tr>
<tr>
<td>b. Bit matrix structure</td>
<td>42</td>
</tr>
<tr>
<td>c. Lightpen actions</td>
<td>42</td>
</tr>
</tbody>
</table>
6. Other instructions
   a. No op
   b. Interrupt
   c. Stop and expansion
   d. Matrix storage

D. Communications with the Computer
   1. Messages to the terminal
   2. Messages to the computer
   3. Coding of messages

E. Theoretical Evaluation of the System

III. SIMULATION OF THE SYSTEM

A. General Simulation Considerations
   1. Levels of simulation
   2. Simulation of communicating automata
   3. Preliminary design of the simulation
      a. Choice of language and facilities
      b. Structure of the program

B. Simulation of the Communications Channel
   1. Message structure
   2. The communications monitor LINK

C. Simulation of the Terminal
   1. The interface TERM
   2. The refresh controller RUN
   3. The pictorial processor PICT
      a. Point processing
      b. Line processing
      c. Establishment of new origin

D. Simulation of the Display Hardware
   1. The plotting system PLOT
   2. The timer TIMER
   3. The graphic output system

E. Simulation of the Operator
F. Computer Support

1. Message packing support
   a. Word types
   b. Transmission routines
   c. Return information from terminal
2. Assembly routines
3. Other facilities

IV. DEMONSTRATIONS OF THE SYSTEM

A. A General Example
B. Various Types of Views
   1. A long focal length lens
   2. An axonometric view
   3. A three-point perspective
   4. A two-point perspective
   5. Multi-view and oblique drawings
C. Animation
   1. An independent animation
   2. Animation with computer assistance

V. OPPORTUNITIES FOR FURTHER DEVELOPMENT

A. Continuation of Present Design
B. Improvements to Simulation Routines
C. Design Modification
D. Other Design Alternatives
E. The Visibility Problem
F. An Efficient Drawing System

VI. SUMMARY

A. Evaluation of the Tests
B. Cost Considerations

VII. LITERATURE CITED
# VIII. ACKNOWLEDGEMENTS

# IX. APPENDIX A - THE ASSEMBLER

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. Control Instructions</td>
<td>99</td>
</tr>
<tr>
<td>1. Assembly control</td>
<td>99</td>
</tr>
<tr>
<td>2. Transmission control</td>
<td>101</td>
</tr>
<tr>
<td>B. Assembly Instructions</td>
<td>102</td>
</tr>
<tr>
<td>1. Words</td>
<td>102</td>
</tr>
<tr>
<td>2. Matrices</td>
<td>103</td>
</tr>
<tr>
<td>a. Bit matrices</td>
<td>103</td>
</tr>
<tr>
<td>b. Substructure matrices</td>
<td>104</td>
</tr>
<tr>
<td>c. Camera matrices</td>
<td>106</td>
</tr>
<tr>
<td>d. Window matrices</td>
<td>107</td>
</tr>
<tr>
<td>3. Control</td>
<td>108</td>
</tr>
<tr>
<td>4. Frame and conditional</td>
<td>108</td>
</tr>
<tr>
<td>5. Draw</td>
<td>109</td>
</tr>
<tr>
<td>6. Matrix manipulation</td>
<td>110</td>
</tr>
<tr>
<td>7. Other</td>
<td>110</td>
</tr>
<tr>
<td>C. Source Code for Examples</td>
<td>111</td>
</tr>
<tr>
<td>D. Compile-time Processing</td>
<td>118</td>
</tr>
<tr>
<td>E. Index of Macro Instructions</td>
<td>128</td>
</tr>
</tbody>
</table>

# X. APPENDIX B - LISTING OF SIMULATION PROGRAMS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. The Hexadecimal Conversion Routine</td>
<td>132</td>
</tr>
<tr>
<td>B. The Matrix Generation Routines</td>
<td>134</td>
</tr>
<tr>
<td>C. The Assembly Routines</td>
<td>138</td>
</tr>
<tr>
<td>D. The Listing and Transmission Routines</td>
<td>149</td>
</tr>
<tr>
<td>E. The Terminal Routines</td>
<td>156</td>
</tr>
<tr>
<td>F. The Display Routines</td>
<td>164</td>
</tr>
<tr>
<td>G. The Operator Routine</td>
<td>169</td>
</tr>
<tr>
<td>H. Index of Entry Names</td>
<td>169</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Structure of the TAG system</td>
<td>18</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Calculation for substructure manipulation</td>
<td>19</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Perspective geometry</td>
<td>21</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Pyramid of vision</td>
<td>23</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Window geometry - points</td>
<td>24</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Line windowing techniques</td>
<td>26</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Simple windowing techniques</td>
<td>27</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Window geometry - lines</td>
<td>28</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Window clipping - multiple crossing cases</td>
<td>29</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Instruction format</td>
<td>31</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Branch instruction format</td>
<td>32</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Procedure head format</td>
<td>32</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Procedure head application</td>
<td>33</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Frame counter instruction formats</td>
<td>35</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Conditional action examples</td>
<td>36</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Camera and timer instruction format</td>
<td>37</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Matrix fetch format</td>
<td>40</td>
</tr>
<tr>
<td>Figure 18</td>
<td>Matrix correction format</td>
<td>40</td>
</tr>
<tr>
<td>Figure 19</td>
<td>Draw instruction format</td>
<td>41</td>
</tr>
<tr>
<td>Figure 20</td>
<td>Bit matrix structure</td>
<td>42</td>
</tr>
<tr>
<td>Figure 21</td>
<td>Matrix storage format</td>
<td>45</td>
</tr>
<tr>
<td>Figure 22</td>
<td>Message codes</td>
<td>49</td>
</tr>
<tr>
<td>Figure 23</td>
<td>Example of LINK listing</td>
<td>58</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>24.</td>
<td>Example of TERM listing</td>
<td>60</td>
</tr>
<tr>
<td>25.</td>
<td>Example of RUN listing</td>
<td>61</td>
</tr>
<tr>
<td>26.</td>
<td>Example of #LISTA listing</td>
<td>73</td>
</tr>
<tr>
<td>27.</td>
<td>Example of #LIST listing</td>
<td>74</td>
</tr>
<tr>
<td>28.</td>
<td>General example</td>
<td>77</td>
</tr>
<tr>
<td>29.</td>
<td>Long focal length lens</td>
<td>78</td>
</tr>
<tr>
<td>30.</td>
<td>Axonometric projection</td>
<td>79</td>
</tr>
<tr>
<td>31.</td>
<td>Three-point perspective</td>
<td>80</td>
</tr>
<tr>
<td>32.</td>
<td>Two-point perspective</td>
<td>81</td>
</tr>
<tr>
<td>33.</td>
<td>Multi-view drawing</td>
<td>82</td>
</tr>
<tr>
<td>34.</td>
<td>Oblique drawing</td>
<td>83</td>
</tr>
<tr>
<td>35.</td>
<td>Animation without computer assistance</td>
<td>84</td>
</tr>
<tr>
<td>36.</td>
<td>Animation with computer assistance</td>
<td>86</td>
</tr>
<tr>
<td>37.</td>
<td>Source code for examples</td>
<td>111</td>
</tr>
<tr>
<td>38.</td>
<td>Compile-time instructions</td>
<td>118</td>
</tr>
<tr>
<td>39.</td>
<td>Generated PL/1 source code</td>
<td>123</td>
</tr>
<tr>
<td>40.</td>
<td>The hexadecimal conversion routine</td>
<td>133</td>
</tr>
<tr>
<td>41.</td>
<td>The matrix generation routines</td>
<td>134</td>
</tr>
<tr>
<td>42.</td>
<td>The assembly routines</td>
<td>138</td>
</tr>
<tr>
<td>43.</td>
<td>The listing and transmission routines</td>
<td>149</td>
</tr>
<tr>
<td>44.</td>
<td>The terminal routines</td>
<td>156</td>
</tr>
<tr>
<td>45.</td>
<td>The display routines</td>
<td>164</td>
</tr>
<tr>
<td>46.</td>
<td>The operator routine</td>
<td>169</td>
</tr>
</tbody>
</table>
ABSTRACT

This study presents a design for a computer graphic terminal capable of displaying real-time animations while requiring only limited communications with the supporting computer. A communications rate of 2.4 kilobaud, which may be achieved over voice-grade phone lines, is selected as a design limit. The information structure and processing is given. The details of hardware implementation are not developed.

The proposed design achieves its objective by storing three-dimensional coordinates of points and performing scaling, rotation, translation, and projection during refresh. Orthographic, oblique, and perspective projection is supported. Substructure manipulation, a subroutine capability, and a linear translation feature further reduce required communications and thus increase the display capabilities with the limited communications.

The feasibility of the design is evaluated through a computer-based simulation. Several possible improvements are suggested.
I. INTRODUCTION

This dissertation describes TAG, a Terminal for Animated Graphics, a design providing substantial real-time animation capabilities while requiring only limited communication with a computer. The feasibility of the design is demonstrated by a computer-based simulation.

A. Background

Computer input and output devices may be classified as character or graphic. A character device is restricted to strings of characters (frequently arranged in a two-dimensional array or page) selected from a character set. Many terminals have the ability to accept new characters into their sets, but the total number available at one time is always limited. Though such devices may draw "typewriter pictures", they are quite restricted in communicating other than the character-string language for which they were designed. By contrast, a graphic terminal is capable of presenting a more general two-dimensional display. TAG is a graphic device.

Communications between a graphic device and the computer may take several forms. In elementary systems the computer instruction repertoire includes instructions to display an element. Such instructions must be re-executed periodically to refresh the display, thus requiring considerable time from
the computer. More commonly, a separate display instruction sequencer will execute display instructions. These instructions may share the computer memory with the computer instructions, but for remote applications a separate display memory is usually provided. A communications channel permits the computer to load instructions into this display memory. The form of the channel is determined by the relative locations and the volume of communications required. Most real-time graphic devices require sufficiently voluminous communications that a location far from a computer is prohibitively expensive. Most non-graphic terminals are capable of operating over a voice-grade telephone line, and may therefore be located at great distances at relatively low cost.

Graphic output devices may be classified according to the mechanism used to draw the display. Most off-line devices are plotters, which physically move a pen over the drawing surface (or move the surface under the pen). Because of their slow speed, they are poorly suited to on-line applications. They will be further considered only for output for the simulation.

Most real-time output devices employ an electron beam as the writing instrument, deflecting it through the desired pattern on either the face of a cathode ray tube (CRT) or a sensitized media. CRT displays, except for the storage-tube types, require periodic refresh of the display for viewing.
Consequently, either the local display control or the computer must be continuously transmitting the picture to the display. Though the storage tube display will retain a static display without refreshing, the entire picture must be redrawn after an erasure. No savings is accomplished for a dynamically changing display.

The usual mechanism for graphic refresh consists of a display memory from which the X and Y coordinates of successive points are extracted. In addition to the coordinates of the points, the memory must also contain information on whether a line is to be drawn to the point, or whether the electron beam is to be blanked, establishing the point as the origin for the next line to be drawn. An instruction to transfer control for refresh is usually included also, together with a timer to delay refresh for "short" display lists, to prevent burning of the CRT phosphor. Changes in a display are made by changing the coordinates of the points, and by adding to or deleting from the list. These techniques are tolerable for moving a few points. But in applications for computer animation, common operations such as zooms, pans, and dollies involve moving every point of the display.

CRT displays normally employ one of two scan techniques. The more common (and employed in TAG) is the vector scan, in which the beam traces out each line of the drawing. Vector scans usually offer a resolution of 1024 x 1024, and produce
light lines on a dark background. The other technique, the raster scan, employs a beam which sweeps the face in a manner similar to that employed in television. Most systems employ a 512 line raster, for a resolution of 512 x 512. Such a system can employ a television monitor or receiver as an auxiliary display. Note, however, that the video signal to such a display has a considerably greater bandwidth than can be passed over a voice-grade phone line. Such displays may produce either light-line or dark-line drawings, and are capable of displaying several levels of gray.

Graphic input on a CRT terminal is usually accomplished by a lightpen, which consists of a probe-mounted photocell. The lightpen detects the passage of the electron beam through its aperture, and signals that the position currently being refreshed is its location. The recognition of which line was being drawn when the strike occurred is relatively straightforward in a vector display. For a raster display, however, usually only the coordinates on the display face can be determined, and the computer must then determine what lies in that area to know what is being selected. Drawing by the operator may be supported by a tracking star, a pattern which is programmed to follow the lightpen.

Computer animation is the use of computer graphics to produce moving displays. Such displays are used in the production of computer-generated motion pictures, and as a means
of real-time communication between computer and user. For the former application, the display is usually generated in a non-real-time manner and recorded on film for later projection at the desired speed. Debugging of a program for producing such a film is a slow process, because of the long turn-around time required for exposure and processing of the film. Systems permitting interaction with the program while observing the display in real time are valuable for such film production, and indispensable for real-time communications applications. Such systems are available, but are expensive because of the necessity of close-coupling between terminal and computer, and because of the large demands upon the supporting computer.

It is the object of this design project to investigate the feasibility of a solution to this need.

B. Review of Literature

An excellent introductory paper is Lewin's "An introduction to computer graphic terminals" [1]. The paper is tutorial in intent, and gives a clear and organized discussion of the various techniques used to accomplish the objectives of a computer graphics terminal, including the advantages and disadvantages of the various designs. It points out the impracticality of raster scan, a technique which has since proven very useful for at least a limited set of applications. (This is noted as a point of interest rather than as a criti-
While the discussion of the hardware display devices was very adequate, the discussion of the data structure and processing techniques was largely on a block diagram level.

Meyer and Sutherland's "On the design of display processors" [2] is also tutorial in intent. It relates the authors' observations from the experience of selecting and/or designing an on-line display system. Their principal observation is that with the argument "For just a little more money we could do this," the design process can become iterative, and will result in a hierarchy of computers from a time-shared main computer which is called only for the most complex tasks through a display control unit which drives the display device. It identifies various existing systems as examples of different points in this cyclic design process. The paper concludes that the optimum choice for the computer-terminal interface is dependent upon the relative locations of the two and upon the type of processing being done.

An excellent tutorial paper in data structures for graphic applications is Williams's "A survey of data structures for computer graphics systems" [3]. It explains the differences between sequential, random, and list structures with examples, and presents strong arguments for the use of list structures.
These three papers, [1], [2], [3] describe the state of the art in their respective areas at their respective times of writing. Techniques reported were originally reported in earlier papers which are not reviewed here. Any more recent papers which reported re-implementations of these techniques without substantial changes are regarded as insignificant and are not reviewed.

Ophir, Rankowitz, Shepherd, and Spinrad report the implementation of "BRAD: The Brookhaven Raster Display" [4], a system using a TV type scan of the full screen. Advantages are very low cost (particularly in multiple installations), an extremely rich display, and the capability of using dark lines on a bright field. Its disadvantages include the complexity of the computer task of modifying the display, and the complexity of the decoding of information from a pointer. These disadvantages are inconsequential when the application demands the rich display, and are minimal in consequence in applications requiring relatively static displays. A further report [5] describes the use of the BRAD system for stereo displays.

In "Computer animation - an exciting new tool for educators" [6], Weiner presents a summary of the current state of the art for computer animation. This tutorial paper indicates the advantages of using on-line real-time animation, and also the limited availability and high cost of this tool.
Three systems operating with limited communications are described.

In "A simple display for characters and graphics" [7], Miller and Wine report the design and implementation of the MODEL T system, which is designed as a minimal cost remote terminal capable of operation over a voice-grade (2.4 kilo­baud) phone line. The system employs no local computer. To save local storage requirements (Refresh over the phone line would produce an unacceptable flicker.), a direct-view stor­age tube is used. The system, even without the hardware character generator option, can draw alphanumericic characters faster than a teletype. Estimate of cost is under $2000. The system is poorly suited for applications requiring fre­quent small erasures, as the entire drawing must be re­transmitted after each erasure.

Another system employing a storage tube is the G101 [8]. It employs a local PDP-8L computer which can expand the lim­ited communications, and operates over a 110 baud phone line.

A limited communications system with variable-length code is reported by Hause and Weller [9]. The described code appears to be primarily for use between refresh memory and display, and is intended for an incremental display capable of moving only one increment per instruction. The code pre­sented appears to have little merit except under these re­strictions.
None of the limited-communications systems appeared capable of supporting real-time animation.

Several systems are capable of supporting real-time animation. Most require a dominant use of a large local computer to generate the changing display. Two systems reported significant advances to relieve this computer burden.

In "A head-mounted three dimensional display" [10], Sutherland describes a system employing dedicated hardware to relieve the computer of much of this load. The three-dimensional coordinates of the points are passed to a hardware matrix multiplier which calculates the coordinates in a sight-point reference system. By means of this transformation during refresh, the system generates a view for a moving sight point in real time. A local computer is employed to calculate the continuously changing matrix for the transformation. No attempt to operate without local computer support is described. A clipping divider which is employed in the windowing for this system is described in a companion paper by Sproull and Sutherland [11].

Hagan, Nixon, and Schaefer describe "The Adage graphics terminal" [12] which employs a similar dedicated-hardware coordinate transformation unit, but implemented in hybrid circuitry. Here, also, a local computer is provided, and no possible operation without is discussed.
The optimum design of a graphics system is discussed by Foley [13]. His techniques for evaluation of a design consist of queue analysis on a model. The response is plotted versus the cost for several proposed designs, and an optimum selected. All models employed incorporate a local computer. A general conclusion is made that the best additional investment is usually in a wider band communications channel, up to 2.4 kilobaud. Beyond this point line costs rise rapidly, and investments in reducing communications will give a greater return.

C. Statement of the Problem

The goals of this project are two: the design of the terminal, and the simulation of the design.

The terminal will support the usual techniques of animation in real time. It will require no more communications with a computer during display of this animation than can be provided by a voice-grade phone line. It shall not require the support of a local computer. The visibility problem need not be solved. The design shall specify the information structure and desired processing. Details of the hardware to implement the desired processing need not be specified.

The simulation will show the feasibility of the design by demonstrating the ability of the terminal to display animated pictures with a volume of communications within the stated limits. The simulation need not run in real time.
Software and other support programming for the computer need be developed only to the extent justified for convenience in the preparation of the demonstrations. The purpose of the program is to simulate to demonstrate feasibility, not to produce drawings. No attempt will be made to create an efficient drawing system for the host computer, as such could be better done without the restriction of simulating the hardware design.

Though the stated goals formally exclude several requirements, the philosophy of "Never pass up anything free," together with an uncertainty as to the potential later value of easily-implemented features has resulted in the inclusion of some capabilities in these excluded categories. An elementary visibility capability (requiring operator assistance with the light pen) has been included in the design, and the programs developed present a usable (though expensive) threedimensional drawing system for the host computer.

D. Facilities Employed

The simulation was performed at the Iowa State University Computation Center. The host computer was an IBM 360/65. A CalComp 565 plotter was used for output. Programs were written in PL/1, and utilized the SIMPLOTTER routines for the production of graphical output.
The SIMPLOTTER routines are a set of problem-oriented routines for plotting two-dimensional graphs of data. Output may be taken on the line printer, which gives a low-resolution picture of the graph with no increase in turn-around time over non-plot jobs. When the program is debugged, the output may be directed to the CalComp incremental plotter which produces a plot with 0.010" resolution. Details on the use of SIMPLOTTER may be found in Scranton and Manchester [14].
II. TERMINAL DESIGN

The design of the TAG terminal presented here is primarily one of an information structure capable of supporting the graphic capabilities desired. The processing performed on this information is presented to explain the functioning of the system. Some further detail on the intended implementation of this processing is given in chapter III. on the simulation of the design. But a full detail design of the hardware is beyond the scope of this project.

A. Overview

1. Re-statement of goals

Three requirements have been stated for the terminal design: the lack of local computer support, the limited communications with the remote computer, and the achievement of real-time animation. All need further amplification, to serve as design specifications.

Quite obviously, the terminal designed will have many characteristics of a computer. The requirement of no local computer will be interpreted to mean that no computational facility will be provided except for that explicitly incorporated for the specific function of the system. This specification was selected so that the amount of computational facility required would be apparent. No attempt has been made
to compare the design with that which might be possible by building around a local mini-computer.

The limitation of voice-grade communications will mean that the rate of communications shall not exceed 2.4 kilobaud. The design shall include the encoding scheme for sending the messages.

Real-time animation is the most open of the specifications. With 1024 bit x 1024 bit frames occurring at a rate of 24 per second, it would be possible to display information at a rate in excess of 25 megabaud. That no viewer could possibly accept information at this rate should be obvious. Unless there is a great deal of redundancy in the information content of successive frames, the resultant film will be confusing.

In live cinematography, it would be possible to have many camera actions and many subject actions occurring simultaneously. Camera actions which may be required include pans (change of direction of sight from same camera location), zooms (change only of focal length of lens), dollies (change of camera location), and tilts (rotation of camera about lens axis, so that picture spins on screen). In general, more than two concurrent camera actions would usually be confusing, and one would be a more normal upper limit.

Subject actions may include movement of points and movement of groups of points or substructures. Motions may in-
clude rotations and translations, and occasionally enlargements and reductions. Motion of a group of points or substructure will be considered as one subject action. For the types of subject matter for which computer animation is practical, the number of independent subject actions would seldom exceed three.

A measure of the success of the system will be the number of different actions, both camera and subject, which may be performed concurrently in real-time. If two can be achieved, the design will be considered acceptable. Three will be considered very acceptable.

A standard practice in animation is double-framing, that is the practice of exposing two consecutive frames of film of identical content. The picture then changes only every two frames, or twelve times per second. This has been found to produce acceptable pictures. As with single-framing, the film is projected at 24 frames per second. Double-framing will be permitted in the evaluation of the system performance.

One might logically consider exposing three identical frames, or triple-framing. This would produce a picture

---

1 Most projectors use double- or triple-interruption shutters, opening two or three times for each frame. Thus the shutter operates at 48 or 72 times per second, to reduce flicker.
change only eight times per second. Experience has shown that this produces a jerky picture, and is therefore not commonly employed in animation. Though not desirable, triple framing will be considered, if needed to achieve the desired animation. It is assumed that under such circumstances, triple-framing would be used for a real-time check on the animation, and that the film would be produced later using single- or double-framing in a non-real-time mode. If triple-framing is needed to achieve the goals, the success will be considered less substantial.

2. Primary techniques

The terminal memory stores the three-dimensional coordinates of each point. This three-dimensional description of the subject matter may then be used to generate views for many different camera situations. The projection to two dimensions for display is performed by the terminal during refresh, using projection specification information.

The computer need communicate only for changes. For camera changes, only the projection specification (camera matrix) need be changed.

3. Secondary techniques

A substructure manipulation capability permits a substructure to be modified (translated, rotated, or scaled) during refresh. A substructure action may then be effected
by changing only the substructure modification specification (substructure matrix), rather than moving every point of the substructure.

A translation capability is built into both the projection and substructure specifications, permitting a variable offset proportional to the frame number. This facility operates without computer communication, and will support linear substructure translation and linear camera translation (dolly) actions.

A subroutine capability is built into the display memory fetch mechanism, permitting a procedure to be called from several places in a program. There are no parameters in the procedure call, but the various specification matrices may be used to modify the results of the call. For example, a procedure may be written to draw a common substructure, and then be called several times, each time with a different substructure matrix. The result will be multiple copies of the substructure, each of size and position specified by the substructure matrix for one of the calls. The combination of substructure manipulation and subroutine facilities eliminate the need for relative coordinate plotting instructions. The subroutine facility would also be useful in preparing multi-view drawings, the camera matrix for each view being established before calling a procedure to draw the subject.
4. General structure

For economy in the dedicated hardware of the terminal, all calculations are performed fixed-point. Standard word length is sixteen bits, and for non-integer quantities, ten bits are considered fractional.

The terminal consists of an interface, responsible for handling communications with the computer; a refresh controller, responsible for fetching and executing instructions from the display memory; a pictorial processor, to further process display instructions; and a display unit to display the resultant pictures. The display unit in turn includes a display processor, a display, a lightpen, and a camera. This structure is illustrated in Figure 1.

Figure 1. Structure of the TAG system
B. Pictorial Processing

Pictorial processing in the TAG terminal is in three steps: substructure manipulation, projection, and windowing. For each step, a theoretical derivation of the calculations performed will be presented first, followed by the simplified calculation actually implemented.

1. **Substructure manipulation**

Rotation may be accomplished by multiplying the vector of coordinates \([X, Y, Z]\) by a 3 x 3 rotation matrix. If enlargement or reduction is also desired, this matrix may be multiplied by the appropriate scale factor. If the coordinate vector is augmented by the elements 1.0 and the frame number, and the matrix augmented by two rows containing the offset and the incremental offset, the result will be a vector of modified coordinates \([X', Y', Z']\), effecting rotation, scaling, translation, and incremental translation, as shown in Figure 2.

\[
\begin{bmatrix}
X & Y & Z & 1.0 & FR
\end{bmatrix}
\begin{bmatrix}
R & * & SF
\end{bmatrix}
\begin{bmatrix}
OFFSET
\end{bmatrix}
\begin{bmatrix}
\Delta OFFSET
\end{bmatrix}
\begin{bmatrix}
X' & Y' & Z'
\end{bmatrix}
\]

**Figure 2. Calculation for substructure manipulation**
The substructure specification is this 5 x 3 matrix. It may be loaded from memory under program control, permitting changes to be made as desired. (For details see Matrix Manipulation, p. xx.)

2. **Projection to picture plane**

Projection to the picture plane is conceptually broken into two steps: movement to camera coordinates, and projection. The latter step varies according to the type of view (perspective, orthographic, or oblique) desired.

Movement to camera coordinates may be accomplished by re-augmentation of the coordinates with 1.0 and the frame number, and multiplication by another 5 x 3 matrix, as in the substructure manipulation. The camera coordinate system is a right-hand system, with the positive X' axis to the right, the positive Y' axis up, and the positive Z' axis pointing behind the camera. The origin is at the camera. All points within the picture, then, will have negative values for Z'.

Perspective views are generated by dividing the X' and Y' coordinates by -Z':

\[
\begin{bmatrix}
X'''
Y'''
\end{bmatrix}
= \begin{bmatrix}
X'
Y'
\end{bmatrix}
/ -Z''
\]

This corresponds to the use of a picture plane at Z'' = -1.0. This is implemented by carrying out this division in hardware, following the matrix multiplication. The division is actually performed following the windowing. If the points (in camera coordinates) are:
A'' = [-1, -1, -4] and
B* = [+0.75, +2, -2],
the calculated perspectives will then be:
A'''' = [-1, -1] / -(-4) = [-0.25, -0.25] and
B'''' = [0.75, 2] / -(-2) = [0.375, 1].
This is illustrated in Figure 3, which is a profile view of
the geometry.

This calculation corresponds to a "normal" lens. A lens
of a different focal length could be created by scaling this
calculation. However, instead, we choose to scale Z'', as
this may be accomplished by dividing the third column of the
camera matrix by the relative focal length.
If an orthographic (axonometric) projection is desired, \( Z'' \) is replaced with the negative of the desired scale factor and the same post-matrix-multiply division by \(-Z''\) is used. This corresponds to first projecting the subject to the plane \( Z'' = -SF \) and then projecting a perspective view of this plane. It is implemented by replacing the third column of the camera matrix by the vector \([0, 0, 0, -SF, 0]\). Thus the pictorial processor need not know whether a camera matrix specifies a perspective or an orthographic projection.

An oblique view may be achieved by modifying the matrix to add suitable components of the \( Z'' \) coordinate to each of the \( X'' \) and \( Y'' \) coordinates, and then modifying the matrix as for an orthographic view.

3. **Windowing**

In a two-dimensional refresh memory, only points within the picture being displayed would normally be stored. However the three-dimensional subject specification is to be usable for many different possible views, and will therefore frequently contain points outside the picture area. Some of the out-of-frame points will come from small \( Z'' \) values (for example, points near the side of the camera), and consequently the \( X'''' \) and \( Y'''' \) coordinates for these points may be very large. Since fixed-point calculations are employed, significant bits will frequently be lost on the division, and points will appear on screen that should be far outside. Therefore
some technique for eliminating such points before the division by $Z''$ is desirable.

   a. Points Each point is checked against each side of the window frame to determine whether it is inside or outside. The test is performed in three-dimensional space, by testing the coordinates $[X'', Y'', Z'']$ against the plane through the edge and the camera point. This pyramid of vision is illustrated in figure 4, and corresponds to an angle of vision of $53^\circ$.

Figure 4. Pyramid of vision
The equations for the four bounding planes are:

\[
\begin{align*}
(0) & \cdot X + (+1) \cdot Y + (-0.5) \cdot Z = 0 \quad \text{(bottom)} \\
(+1) & \cdot X + (0) \cdot Y + (-0.5) \cdot Z = 0 \quad \text{(left)} \\
(0) & \cdot X + (-1) \cdot Y + (-0.5) \cdot Z = 0 \quad \text{(top)} \\
(-1) & \cdot X + (0) \cdot Y + (-0.5) \cdot Z = 0 \quad \text{(right)}
\end{align*}
\]

The left side of one of these equations evaluated for the camera coordinates of a point is the window function for that point and boundary. If positive, the point is inside the boundary; if negative, it is outside. The geometry for the top and bottom boundaries is illustrated in Figure 5.
Window values for the points A'' and B'' are:

\[
\begin{array}{cc}
A'' & B'' \\
(bottom) & +1 & +3 \\
(left) & +1 & +1.75 \\
(top) & +3 & -1 \\
(right) & +3 & +0.25 \\
\end{array}
\]

Thus we see that point A'' is inside all boundaries and point B'' is outside of the top.

The equation coefficients for the window equations are arranged in two matrices similar to those employed for substructure and camera specifications. The first window matrix contains in its first line the X and Y coordinates of the center point and the number of window equations to be checked. The X''' and Y''' calculated from the perspective calculations are coordinates from this center of vision. It will normally be at the center of the screen, but may be moved elsewhere, as would be done photographically by using a rising front or a swinging or tilting back on a view camera. This permits two-point perspectives with horizon other than at the center, and one-point perspectives with the vanishing point off-center. The remaining four lines of this matrix normally contain the equation coefficients for the four edges of the screen. These share a matrix, as they must usually be changed when the center is moved.
The second window matrix will accept up to five additional frame lines. Additional frame lines may be used for composite (collage) pictures, and for wipe transitions.

b. Lines Discarding points outside the picture is not adequate. As each line is drawn from point to point, a discarded point would leave a line connecting its two neighbors. What is necessary is that each line passing outside the frame be clipped at the frame boundary. This is shown in Figure 6.

![Figure 6. Line windowing techniques](image)

First, two special cases (actually very common) may be considered. If both end points are inside all frame boundaries, then the line lies entirely within the picture area, and may be drawn with its original end points. If both end points are outside of the same frame boundary, the line lies entirely outside the picture area, and may be discarded. These are illustrated in Figure 7.
In all other cases, further checking is required. The line may be described parametrically. The parameter value of 0 is assigned to the origin, and 1 to the terminus. All intermediate points correspond to values between these two limits. In the window calculation for each end point (and each frame side), a window value was calculated (positive = inside, negative = outside). Because X, Y, and Z are linear functions of the parameter, and the window function is a linear function of X, Y, and Z, the value of the parameter (between 0 and 1) corresponding to each window value being zero (edge of frame) can be calculated from the equation

\[ P = \frac{W(\text{orig})}{(W(\text{orig}) - W(\text{term}))} \]

The line A'' B'', shown in Figure 8, crosses only the top frame boundary. This crossing occurs at a parameter value of

\[ P = \frac{3}{(3) - (-1)} = \frac{3}{4} = 0.75. \]
On each frame edge for which $W_{\text{orig}} < 0 < W_{\text{term}}$ the line crosses inwards. The parameter value at which the crossing occurs is determined, and the maximum of these values corresponds to the last entry crossing. This is the value of the parameter at which the line may enter the picture. On each frame edge for which $W_{\text{orig}} > 0 > W_{\text{term}}$ the line crosses outwards. The parameter value for this crossing is also determined. The minimum of these values represents the first exit crossing, at which the line may leave the picture. Figure 9 shows these relations.
If the entry parameter is greater than the exit parameter, the line misses the window and should be rejected. If the entry parameter is less than the exit parameter, then the line crosses the picture. The entry point (new origin) and the exit point (new terminus) are calculated from the corresponding parameter values with the equation:

\[
[X,Y] = P \times [XT,YT] + (1-P) \times [XO,YO]
\]

These sets of coordinates are passed to the perspective division and the line is plotted.
C. Storage and Instruction Structure

The coordinates processed by the pictorial processor must be obtained from the display memory. The selection and execution of plotting and other instructions is dependent upon the structure of this memory, and this structure in turn must be designed to accommodate the desired instruction set.

1. General format of instruction

The instruction storage structure is sequential. The additional storage required for links in a list structure would increase the volume of data to be sent from the computer, and more hardware at the terminal would be required to fetch instructions from a list structure. The subroutine facility provides some substructure capability.

The display memory is organized into sixteen-bit words. The sixteen bits are frequently subdivided into four groups of four bits each. Such a group of four bits, a half byte, is called a nybble. The value of a nybble will be represented in hexadecimal: 0-9, A-F.

Instructions are of varying length. The first word of each instruction is called the instruction head. The first nybble of the instruction head is the op code, and specifies the type of instruction to be performed. The next two nybbles are the variant. Their meaning depends upon the type of instruction specified by the op code. The last nybble of the head specifies the length of the instruction. Its value is
the number of subsequent words in the instruction, not including the head. Thus a length nybble of 0 means that there are no subsequent words and the head is the entire instruction. A length nybble of F represents fifteen subsequent words, or a total of sixteen, which is the maximum instruction length. Figure 10 illustrates this format.

![Figure 10. Instruction format](image)

2. **Control transfer instructions**

Following the execution of an instruction, the next instruction is normally located by adding the length (length nybble plus one) to the address of the current instruction. A few instructions are capable of altering the instruction fetch sequence. The branch and procedure head instructions will be discussed here. Conditional branch and several interruptable instructions will be discussed later.

The branch instruction has an op code of 1. It is two words long, the second word specifying the address to which control is to be transferred. The variant bits are ignored. Its format is shown in Figure 11.
Upon execution, in addition to transferring control to the specified location, it will cause the address of the next sequential instruction to be stored in a revert register. This may be used for return from a subroutine call. The same branch instruction then serves for either a GO TO or a CALL instruction. Which of these is effected is determined by the coding at the location to which the branch is made.

The procedure head does not alter sequential control at the time it is executed, but its function is so closely related to the control sequence that it is presented here. The procedure head is an instruction three words long. The op code is 2, and the variant nybbles are ignored. The second word is always a head for a branch instruction. (That is, the second and third words constitute a branch instruction, nested within the procedure head instruction.) A procedure head is illustrated in Figure 12.

![Figure 11. Branch instruction format](image)

![Figure 12. Procedure head format](image)
The only function of the procedure head execution is to store the revert register in the third word of the instruction. Thus the instruction located at its second word becomes a branch instruction to the instruction following that from which call was made. To effect return from a procedure, a branch to this second word of the head is made, from which a branch is made back to the calling instruction sequence. This technique does not permit recursive calls, but procedure calls may be nested to any depth, so long as the memory is adequate to hold all the procedures involved. Figure 13 demonstrates the use of this subroutine linkage.

Figure 13. Procedure head application
Frame counter related instructions

A variety of loosely related functions are incorporated into the instruction with op code 3. These relate principally to the frame counter and actions that are logically performed at the time of frame counter modification. The option for each group of actions is selected independently by proper variant specification.

a. Frame counter actions

The first variant nybble specifies one of four frame counter actions. Only the two least significant bits of the nybble are observed, so values 0-3 are interpreted modulo 4.

If the nybble is 0, no frame counter action ensues. This provides for independent execution of the options specified by the second variant nybble. Such an instruction is always of length one.

If the first variant nybble is 1, the frame counter is set to the value specified by the second word of the instruction. Length is always two words.

If the first variant nybble is 2, the frame counter is incremented by the value specified by the second word of this two-word instruction.

If the first variant nybble is 3, the frame counter is incremented by the second word value, and is then compared with the value specified by the third word. If the frame counter after incrementing (possibly by 0) is less than or equal to the test value, then the test is said to be low.
it is greater than the test value, then the test is said to be high. Figure 14 shows this group of instructions.

Figure 14. Frame counter instruction formats

b. **Conditional actions**  If the frame counter test is performed and the result of the test is low, then the next instruction will be found at the fourth word of the current (test) instruction. One or more low option instructions, with a maximum aggregate length of thirteen words, may be placed here. They will be executed only on a low test. These words are considered a part of the test instruction, and are included in the length specification of its instruction head. This is the reason for the thirteen word limit, since two words have already been used for increment and test values. If more than thirteen words of low option instructions are required, a branch instruction may be included. In
the absence of a branch, execution following the low option group will proceed sequentially with the next instruction. In the event of a low test, all actions specified by the second variant nybble will be skipped.

If the frame counter test is high, or if no test was specified by the instruction, certain options specified by the second variant nybble will be performed. The next instruction will be located from the length specification of the current instruction, which will cause the low option instructions to be skipped. No instructions will be executed only on the high test, unless a branch is included in the low option instructions. The sequence of control for both high and low tests is indicated for examples in Figure 15.

Figure 15. Conditional action examples
c. **Other actions**  
Other actions performed by this instruction, if specified by the second variant nybble, include camera actions and timer checking. These actions are skipped if a low test occurs, but may be specified again in the low option instructions, if desired. The first bit of this nybble is ignored. The second and third bits specify camera action, and the fourth specifies the timer check, as shown in Figure 16.

![Figure 16. Camera and timer instruction format](image)

Camera bits of '11' (nybble = 6 or 7) turn the camera on. That is, filming is started by opening the shutter. Note that a picture is recorded only if drawn while the shutter is open. The camera unit has no phosphor to retain a static picture which can later be snapped, but sees only a moving beam which traces out the desired pattern.

Camera bits of '10' (nybble = 4 or 5) turn the camera off by closing the shutter. No further film will be recorded until a camera on instruction is executed, at which time a new frame will be started. The display at the operator's console will continue, however, even when the camera is off.
Camera bits of ‘01’ (nybble = 2 or 3) advance the film if the camera is on. This instruction has no effect if the camera is off.

Camera bits or ‘00’ have no action.

If the timer bit is ‘1’ (nybble odd), the timer is checked. The purpose of the timer is to prevent burning of the CRT phosphor by too-frequent refresh, as might occur with a very short display program. The timer will cause a pause to occur until a preset time has elapsed since the last timer instruction. The minimum time is 25 milliseconds, permitting a maximum of forty refreshes per second. Fewer, of course, may occur with a very long program, in which case the timer will cause no pause. It may be set to 41.67 milliseconds (1/24 second) to ensure operation at the standard frame rate.

One other condition could endanger the phosphor — a looping program (intentional or unintentional) with no timer test. To protect against this, an instruction counter counts instructions executed since the last timer instruction. If this count exceeds a pre-determined limit, an interrupt to the computer occurs.

4. Matrix manipulation

Various matrices specify the manner of operation of the terminal. These frequently need to be changed. For example: substructure matrices must be changed to manipulate substructures, or for calls on detail procedures to produce mul-
tiple copies of the detail; the camera matrix must be altered to change the view, or for multi-view drawings; the window matrix must be changed for composite (collage) displays, for change of center, or for wipe transitions; and the bit matrix (which will be more fully discussed under draw instructions, p. 41) must be changed to arm or disarm certain options for drawing, lightpen actions, and interrupts.

These capabilities are provided in three similar instructions, op codes 5, 6, and 7. For all three types the first variant nybble specifies which matrix is to be manipulated, as follows:

0 Bit
1 Substring
2 Camera
3-4 Window
5-? User defined

A matrix fetch action is specified by op code 6. This instruction is always two words long, the second word being the address of memory from which the matrix is to be obtained. The number of words to be loaded into the matrix is specified by the second variant nybble. The maximum is F or 15, the full matrix. The format appears in Figure 17.

---

\(^{1}\)Number of matrices, to a maximum of 16 (0 - F), is implementation defined.
Matrix store action is specified by op code 5. Like the fetch, it is two words long, with the second word specifying the memory location, and the second variant nybble specifying the number of words to be stored.

A matrix correction action is specified by op code 7. This instruction is useful for small changes in a matrix. It differs from the fetch in that the words to be loaded are obtained from the instruction, and that the loading may begin at any word of the matrix. The second variant nybble specifies the position in the matrix at which correction is to begin, and the length nybble specifies the number of words to be corrected. This, of course, is equal to the number of subsequent words in the instruction. This instruction format is illustrated in Figure 18.
5. **Draw instructions**

The draw instruction, op code 4, calls upon the pictorial processing to construct the picture. It is always of length four, the three subsequent words containing the X, Y, and Z coordinates of the point being plotted. The two variant nybbles are treated as a variant bit string of length eight. Figure 19 shows the format of the draw instruction.

```
| 4 | VARIANT 3 | X COORDINATE | Y COORDINATE | Z COORDINATE |
```

Figure 19. Draw instruction format

**a. Function**  
The variant bit string is ANDed with the draw bit string from the bit matrix. If any corresponding bits are both '1's, then a line is drawn from the previous point to the new one (or whatever portion of such a line falls within the window frame). If none of the variant and draw bits are matched '1's, then no line is drawn, but the point still becomes the origin for the next line to be drawn. A draw instruction with 00 variant, which can never draw a line, is a traverse instruction.

If a drawn line on the display passes through the aperture of the lightpen, a lightpen strike occurs. The response to the strike will be determined by the variant bits and several additional bit strings from the bit matrix. These actions will be discussed under lightpen actions.
b. **Bit matrix structure**  The bit matrix holds numerous miscellaneous items. Since this is the first place where it is extensively used, it will be discussed here.

Eight bit strings, each of length eight, are contained in the first four words of the matrix. They are, in order, the draw, change, set, clear, terminal interrupt, computer lightpen interrupt, instruction interrupt, and spare bit strings. The fifth word contains a terminal interrupt address. The sixth through tenth words are unused. The eleventh through fifteenth words contain a working vector used in the pictorial processing. Word fourteen must always contain 1.0, and word fifteen is the frame counter. This structure is shown in Figure 20.

![Figure 20. Bit matrix structure](image)

---

c. **Lightpen actions**  When a lightpen strike occurs, the response is determined by the variant bit string and several of the bit strings from the bit matrix.

If any variant bit and the corresponding change bit are both '1's, then changes in the variant bits of the instruc-
tion may be made. Each '1' set bit will set the correspond­
ing variant bit to '1', and each '1' clear bit will set the
 corresponding variant bit to '0'. If neither set nor clear
 bit is '1', the corresponding variant bit remains unchanged.
 If both set and clear bits are '1's, the result is undefined.
 The lightpen ability to modify variant bits, which in turn
 can control whether a line is drawn, permits selective dele­
tion of lines, and thereby an elementary form of operator-
 controlled visibility determination. Note, however, that en­
tire lines must be added or deleted. The visibility thus
displayed cannot change in the center of a line.

If any variant bit and the corresponding terminal inter­
rupt bit are both '1's, then control will be transferred to
the address specified by the terminal interrupt address in
the bit matrix. If the instruction there is a procedure
head, control may be subsequently returned to the instruction
following the draw instruction. This is useful in menu type
activity, giving the operator selection without computer com­
munication.

If any variant bit and the corresponding computer light­
pen interrupt bit are both '1's, then the display is stopped
and a message is sent to the computer. This provides menu or
line selection capabilities with the computer responding.
6. **Other instructions**

   a. **No op** Instructions with op code 0 have no effect except to cause the next instruction to be fetched from the location determined by the length nybble of the instruction.

   b. **Interrupt** The instruction with op code 8 causes an optional interrupt of the computer. This instruction is always one word long. If any variant bit and the corresponding bit of the instruction interrupt bit string in the bit matrix are both '1's, then the terminal will stop and a message will be sent to the computer.

   c. **Stop and expansion** Instructions with op codes 9 through F will cause the terminal to stop and a message will be sent to the computer. This interrupt is not conditioned upon any bit matching. Only one of this group of instructions is needed as a stop instruction. The remainder are intended for possible later expansion of the instruction set.

   d. **Matrix storage** A section of memory may be reserved for storage of a matrix with either a no op or a stop instruction with the length nybble specifying the number of words reserved. The matrix fetch and store instructions require as address the address of the head reserving the storage space. The words fetched or stored will start in the first word following this head, as may be seen in Figure 21.
Though either no op or stop instructions may be used, the stop instruction is preferred. A block of no op matrices will permit sequential instruction execution to skip through memory, should an erroneous branch transfer control to them, while the stop matrix head will halt execution and notify the computer, terminating the invalid sequence earlier. The no op form may be used, if space is desired in the middle of an instruction sequence, without having to branch around it.

D. Communications with the Computer

The terminal serves as a communications link between the operator and the computer. Though it is capable of performing many simple tasks autonomously, it can initiate these only if instructed to do so by the computer. The operator interacts with the terminal primarily to send messages to the computer, frequently requesting it to direct the terminal in its display activity. Though the need for communications has been limited, efficient coding is still necessary to meet the requirement of voice-grade communications.
A variety of messages need to be passed between the computer and the terminal. Most messages are valid in only one direction, and will therefore be listed according to the direction of transmission.

Two way teletype communications is provided between operator and computer. This would be the primary form of non-graphic communications, and will not require the support of a display program in the terminal.

1. Messages to the terminal

   The computer may send four classes of messages to the terminal.

   A data load instruction provides the terminal with data for loading into memory and information on its disposition. A LONG LOAD form with greater capabilities, and a SHORT LOAD form requiring less directive information, are provided.

   A DATA REQUEST asks the terminal to send the contents of specified locations of the display memory. It is provided to allow the computer to analyze the status of the terminal.

   A start instruction directs the terminal to start a display. There are three forms: a START, for which a starting address must be supplied; a RESTART, which always starts at location zero; and a RESUME, which starts at the instruction following that last executed. There is also a SHORT LOAD AND RESUME command, which loads memory and then resumes execution of the display instructions.
Finally, there is a HALT instruction, which causes the terminal to cease display operations.

2. Messages to the computer

Three classes of messages may be returned by the terminal.

When the display is stopped for any reason, a message is sent indicating the reason for the stop. The termination messages are STOP INSTRUCTION executed, INTERRUPT INSTRUCTION executed, LIGHTPEN INTERRUPT occurred, INSTRUCTION COUNT LIMIT, and HALT RESPONSE.

A DATA RETURN message is sent, containing the requested data, in response to a DATA REQUEST.

Two error messages may be sent by the terminal, one indicating the receipt of an INVALID COMMAND from the computer, and the other indicating receipt of INVALID DATA.

3. Coding of messages

The messages are byte oriented. Of the 256 possible bytes, 239 are reserved for use as teletype characters and one as the idle character, leaving only sixteen for terminal communications. Communications at four bits of information per byte transmitted would represent only 50 percent efficiency. Therefore a two-mode communications channel has been designed.
In the command mode, the teletype and idle character reservations apply and the sixteen remaining codes are command headers for terminal functions. The code 00 indicates the idle state for the communications channel. Codes 01 through EF are reserved for teletype use. Codes F0 through F7 are message headers for terminal to computer messages, and F8 through FF are for computer to terminal messages.

In the data mode, the teletype is inhibited, and all eight bits of the byte may be used to transmit information. The data mode is terminated normally only by reaching a pre-specified length. Following most command headers are a fixed number of bytes giving further information such as location in memory and amount of data to be communicated. If data is to be communicated (as on a load or a request response), it follows this information, also in data mode. Data is communicated using two bytes for each word.

In addition to the normal termination of the data mode at the pre-specified length, there is one abnormal termination. In data mode, nearly every possible code or sequence of codes is valid. There is one exception. Data will be considered invalid if more than fifteen consecutive words are zero. There is no reason for loading the terminal memory with more than fifteen consecutive zeros, as a matrix may contain only fifteen words before another matrix head, which must have at least a non-zero length specification. This
situation is interpreted as either a loss of communications or an error in the length specification. Loading will be stopped, to prevent wiping out much of memory, and an INVALID DATA error message will be sent to the computer.

Coding of the sixteen messages is given in Figure 22.

<table>
<thead>
<tr>
<th>Message Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0</td>
<td>Location</td>
</tr>
<tr>
<td>F1</td>
<td>Location</td>
</tr>
<tr>
<td>F2</td>
<td>Location</td>
</tr>
<tr>
<td>F3</td>
<td>Location</td>
</tr>
<tr>
<td>F4</td>
<td>Location</td>
</tr>
<tr>
<td>F5</td>
<td></td>
</tr>
<tr>
<td>F6</td>
<td></td>
</tr>
<tr>
<td>F7</td>
<td>Data</td>
</tr>
<tr>
<td>F8</td>
<td>Location Length</td>
</tr>
<tr>
<td>F9</td>
<td>Location Length Data</td>
</tr>
<tr>
<td>F A</td>
<td>Location Len Data</td>
</tr>
<tr>
<td>F B</td>
<td>Location Len Data</td>
</tr>
<tr>
<td>F C</td>
<td></td>
</tr>
<tr>
<td>F D</td>
<td></td>
</tr>
<tr>
<td>F E</td>
<td>Location</td>
</tr>
<tr>
<td>F F</td>
<td></td>
</tr>
</tbody>
</table>

Figure 22. Message codes
The location is usually sixteen bits, which is adequate to address a 64K word memory, should the terminal be provided with one this large. The length specification is usually eight bits, and specifies the length of the data in words, to a maximum of 255. This may mean that several data load messages will need to be sent to load a new program. But with four bytes of non-data for up to 510 bytes of data, the creation of longer messages is not needed. Rather this length maximum serves as a protection against a communications error in the length specification tying up the line for an extended period while loading garbage into the terminal memory.

For the SHORT LOAD, and also the SHORT LOAD AND RESUME, the location specification is only twelve bits, and thus can address only the first 4K words of memory. The length specification is only four bits, and so the maximum data communicated is fifteen words. For short corrections such as one matrix or less, which are common for animation changes, the savings in communications over the long load form is sufficient to justify this special short form.

E. Theoretical Evaluation of the System

Once loaded with a display description, the proposed TAG terminal will be able to generate animations with any number of linear substructure translations and with a linear dolly with no computer assistance. Most other activities desired will require communications with the remote computer.
Communications at 2.4 kilobaud corresponds to a rate of 300 bytes per second. Since the frame rate is twenty-four per second, this allows 12.5 bytes per frame. Using the SHORT LOAD AND RESUME, three bytes are required for header and directive information. The terminal will return three bytes for each stop, giving a total of six bytes overhead per communication. Thus the computer could load three words (six bytes) during one frame, nine words (25 - 6 = 19 bytes) during a double frame, or fifteen words (37.5 - 6 = 31.5 bytes) during a triple frame. A complete matrix (camera, substructure, etc.) could be changed in a triple frame. The nine words possible in a double frame would change the first three rows of a matrix. This corresponds to rotation about an axis through the point [0, 0, 0]. The three words possible during a single frame could change the fourth row of a matrix, achieving a repositioning of the substructure or camera.

If the communications rate could be increased to 2.9 kilobaud, a double frame would suffice for 12 words, which would be adequate for most matrix corrections, since the incremental offset would seldom be used when the computer was modifying the matrix. It thus appears that with 2.9 kilobaud communications and using double-framing, the terminal can support an unlimited number of linear translations and one other action. With 2.4 kilobaud, it could support this with triple-framing, and could support this with double-framing
providing the other action were a rotation about the origin or some other action requiring nine elements or less of a matrix to be changed.

It remains to be demonstrated that the terminal can generate the displays described. The actual volume of communications employed by such displays will be observed during the simulation, to verify the above estimates as reasonable.
III. SIMULATION OF THE SYSTEM

The simulation presented here has as its primary goal the demonstration that a graphical terminal with the TAG information structure is capable of generating real-time animations with a communications volume within the specified limit. Full source listings for the programs used are included in Appendix B, p. 132.

A. General Simulation Considerations

1. Levels of simulation

Three levels of simulation were considered in the design of the program: detail, structural, and functional.

Detail simulation, that showing the exact bit manipulation and detail hardware structure of the system, was not employed. The design being simulated is not a detailed design.

Structural simulation is that in which the information structure and processing are simulated, but with free substitution of processing techniques efficient in a program for those efficient in hardware. It was used in much of the terminal simulation. No implication is made that the same techniques (e.g. do loops, use of pointers and based variables) would be used in the hardware implementation.

Functional simulation was employed in those portions of the system which were not new innovations to this design, and for which a structural simulation did not appear justified or
even possible. These include the remote computer, the display hardware, and the operator.

A fourth situation exists, though it cannot really be classified as a simulation. There is no reason to require a timer pause in the simulation, as the simulated CRT phosphor (the CalComp plotter) could not be damaged by too-frequent refresh of the display. So the timer "simulation" must be classified as non-functional.

2. Simulation of communicating automata

The total system consists of at least three automata: the computer, the terminal, and the operator. Since the terminal, when implemented in hardware, will be doing parallel processing, it, in turn, may be regarded as an automaton consisting of several sub-automata.

Each automaton (or simulated automaton) will operate independently until it receives a message (input) from one of the others. If the time of arrival of the message is non-sequential, the automaton is time-independent. An example of such a system would be one in which all inputs are filed in a queue and later accepted under "local" control. A system of time-independent automata could be simulated by simulating each automaton in sequence until its input queue was empty.

But the system here is time-dependent. The result of a message from the computer to change the display arriving after one frame is generated and the result of the same message
arriving after several frames are clearly different. The system could be simulated by operating concurrent simulations of all automata as tasks. But unless the rate of each simulation were directly proportional to the rate of the simulated hardware, the simulation would not be fully valid. The hardware terminal may be capable of generating several frames between messages from the computer, while the simulated terminal might only generate one between corresponding messages from the simulated computer.

A valid simulation of communicating time-dependent automata then requires a time-scaled simulation. Without detailed design of the terminal processing, this was clearly beyond the scope of this project. Further, such a simulation was not necessary to demonstrate the required performance.

The system as simulated will have only one automaton active at any time. Following a message transmission, the computer will wait for a response, and following a response, the terminal will wait for another message from the computer. Such a "ping-pong" system is easily simulated, by having one automaton represented by a subroutine called by the other. The computer was assigned the calling role, and the terminal that of the subroutine.

3. Preliminary design of the simulation

a. Choice of language and facilities The IBM 360/65 was selected as host computer because it was the only readily
available system of suitable size for such a simulation. No other system was considered.

The programming language PL/1 was selected because of the ease with which the information structures of the system could be described. APL would have been better for many portions of the processing, but was rejected because a compiler was not available on the host computer.

The SIMPLOTTER routines were used for graphical output, as they were the only routines available from PL/1. The only alternatives would have been to write directly on the plot tape (which would probably cause difficulties for the ISU accounting system), or to write on an intermediate data set which would be read and processed in a subsequent step under Fortran G or H which can call the CalComp routines. Even if reasonable alternatives had existed, the conveniences offered by SIMPLOTTER would make it a very desirable choice.

b. **Structure of the program** The simulation program was written as a series of external procedures, to permit corrections to be made without requiring re-compilation of the entire program. In addition to the parameter lists, these procedures communicate by means of several external variables. Except for MESSAGE, the same external names are not used in routines for the computer simulation and those for the terminal. This avoids inadvertant communication between the two not accounted for in the message stream.
B. Simulation of the Communications Channel

1. Message structure

The communications between the computer and the terminal in the hardware version of the design would be a continuous stream of bytes (the idle byte 00 being present whenever no communications is needed). A detailed simulation of this would require a call to pass each byte. Since procedure prologues are time consuming in PL/1, this appeared an economically prohibitive plan. The bytes were then grouped into messages, with one call per message. The length of the message is variable. Each message starts with a message head, which is a byte in command mode, and includes all subsequent bytes in data mode, both specification and data. The message is transmitted through the external variable-length character string MESSAGE.

The receiving unit determines the length of the data mode communications from the header and specifications, as in the hardware version. This length is then compared with the actual message length received, and if they differ, the error condition is raised. Though it is not possible to have an error in length specification cause a header to be read in data mode or data to be read in command mode, the error situation is detected and reported. The hardware version would attempt to continue, and the results of such an undetected error are quite unpredictable. A reliable communications
channel has been assumed. No attempt has been made to simulate the results of errors in data transmission. In the hardware version, an operator would normally be monitoring the terminal's graphic output, and could intervene should such an error occur.

2. **The communications monitor LINK**

The routine LINK is responsible for monitoring all communications between computer and terminal. Rather than having the computer call the terminal, it calls LINK which then calls the terminal. On both call and return, it lists the number of bytes in the message and a copy of the message (in hexadecimal). An example appears in Figure 23.

![Figure 23. Example of LINK listing](image)

C. Simulation of the Terminal

1. **The interface TERM**

The interface is responsible for receiving the messages from the communication channel, directing teletype codes to the teletype printer, and responding to message headers according to their types. It is also responsible for assembling and transmitting return messages to the computer. As previously mentioned, the received message length is checked against the specified length.
Teletype codes (which may occur only in command mode — i.e. at the head of a message) are listed to report their occurrence. Since the operator simulation (which shall be discussed later) does not have the capability of returning teletype codes, this is not simulated. There would be little difficulty in allowing the operator to send such messages or in transmitting them to the computer. But the problem of decoding them once received at the computer is considerable and unnecessary to the goals of this project.

LOAD commands will cause the ensuing data to be placed in the specified location of the refresh memory MEM. A variable pointer DATAP is used to mark the start of the data string for efficiency in unpacking.

DATA REQUEST commands cause a return message to be sent, consisting of a DATA RETURN header followed by the contents of the specified locations of the refresh memory MEM.

START, RESTART, and RESUME commands will cause the starting address to be set to the appropriate value (specified location for START, zero for RESTART, or previous stop value for RESUME) and will call the RUN routine to start display refresh.

When the display is stopped, control will return from RUN and the terminal will assemble a message consisting of a header which specifies the cause of the termination and the location of the next instruction to be executed. The TERM
routine then returns, to communicate this message to the computer.

In the simulation, the HALT command is non-functional, since the computer will never send a message to the terminal when the terminal is operating. It will cause a HALT RESPONSE to be returned, consisting of a header and the location at which the display was last stopped. TERM will list all operations performed, as Figure 24 illustrates.

**Figure 24. Example of TERM listing**

2. **The refresh controller RUN**

The routine RUN is responsible for obtaining instructions from the refresh memory MEM, decoding them, and performing the functions specified. Each successive instruction location is calculated from the current instruction location LOC and the length specification L in the instruction head.

Graphical instructions (draw and traverse) call on the routine PICT for execution. Camera instructions (camera on, camera off, and frame) call on various entries of the routine PLOT (which is also called by PICT) for execution, and timer instructions call the routine TIMER. All other instructions are executed by the routine RUN.
Control is not returned from RUN until the display is stopped. A code $CODE is returned to indicate the reason for termination.

As each instruction is fetched from MEM, the instruction counter ICT is incremented, and if it exceeds the pre-set limit the display is stopped.

RUN will produce a hexadecimal listing of the instructions executed. At present, this trace option may not be disabled. A trace output is shown in Figure 25.

```
0127 60F1 0007 TIME = 210356830 ICT = 0
0129 3060 TRAN = 1
013A 60F1 0007 TIME = 210356830 ICT = 0
013C 60F1 0007 TIME = 210356830 ICT = 0
013E 1001 0012
0077 2002 1001 0000
0075 1021 0283
0083 7002 1001 026C
0290 61F1 001E
0008 60F3 0000 0300 0300
02FC 4011 0003 0000 0200
0090 4013 0033 0000 0700
```

Figure 25. Example of RUN listing

3. **The pictorial processor PICT**

The routine PICT receives from RUN the three coordinates of the point to be plotted and a bit specifying whether a line is to be drawn from the last point. The coordinates are
passed in the bit array as the eleventh through thirteenth elements. The fourteenth and fifteenth elements are the value 1.0 and the frame counter.

a. **Point processing**  This five element vector is multiplied by the substructure matrix, and the result stored back in the coordinate elements. It is then multiplied by the camera matrix, which gives the point location in camera coordinates. This vector is then multiplied by the transposed window matrix to generate a vector of window values, positive for each frame line for which the point is inside, and negative for each outside.

If the bit passed from RUN specifies no line to be drawn, line processing is skipped.

b. **Line processing**  From here, processing must involve lines rather than just points. The coordinates and window values are available for both the current point (terminus) and the previous point (origin). These are used, as described in Windowing, p. 22, to determine if any of the line is to be drawn, and to determine the coordinates of the appropriate windowed origin and terminus if drawing is needed. If a line is to be drawn, these windowed coordinates are passed to the PLOT routine. This is accomplished with two calls, one to PLOTS (which moves without drawing) with the origin coordinates, and one to PLOT (which draws the line) with the terminus coordinates. This would correspond very
much to the input which would be received from a two-dimensional display memory in a conventional vector graphic terminal.

c. Establishment of new origin Following either rejection or plotting of a line, the coordinates (before windowing) and window values for the current point are stored as origin for the next line, and control is returned to RUN for execution of the next instruction.

D. Simulation of the Display Hardware

The display hardware need be no different than that of a conventional vector graphic display employing a two-dimensional coordinate memory. The purpose of the display simulation, then, is primarily to support the simulation of the unique portions of the terminal.

1. The plotting system PLOT

The PLOT routine is primarily responsible for collecting the points into lists of connected points, both to minimize calls on the SIMPLOTTER routines, and to keep the number of superpositions below the maximum allowed. Whenever the sequence of points breaks (i.e., coordinates for PLOTS differ from those for preceding PLOT call), or whenever the chain accumulates more than 99 points, PLOT calls GRAPHS to plot the line. GRAPH will be called first with four register (+) marks, if this is the first line in the frame.
PLCT will also check for possible lightpen strikes, by noting whether the line passes within the aperture size of the lightpen coordinates. If a strike occurs, an external bit variable LPSTR is set, so that on return, RUN will be able to properly process the strike.

Camera control instructions call on entries to PLOT: CAMCN, CAMOFF, and FRAME. These entries cause the preceding frame to be terminated (that is, any assembled chain of lines will be passed to GRAPHS). The actual output simulates the display output, rather than the camera. There would be no point in simulating the display with the camera off, as no output would be observed to confirm proper operation. The status of the camera is noted in a label on each frame. This corresponds to having a camera on indicator on the display console. If a blank frame is generated with the camera off, it is deleted. With the camera on, it will be generated as it would on a film from the camera. The frame number is also displayed as a label on the plot. Each time the frame or camera entries are called, the plotter will start a new plot, and the operator routine OPER will be called to determine the status of the lightpen for the next frame. The routine PLOT has two additional entries, START and FINISH which are used for initialization and finalization respectively. These calls are considered to be outside the simulation.
2. **The timer TIMER**

The timer routine TIMER is called by RUN whenever a timer instruction is executed. It will reset the instruction counter to zero, and makes a timer check.

The function of the timer in hardware would be to cause a pause until a preset time after the previous timer check, to protect the CRT phosphor. This is unnecessary in the simulated version, and in fact quite undesirable. For the simulation, the timer check has been used instead as a protection against a loop containing a timer instruction consuming excessive time. It will check a clock, and if past a preset time limit will call FINISH to terminate the simulation and save the outputs before the operating system's step timer terminates the step in a less favorable manner.

The timer must be initialized with a call to TIMERRST with the desired instruction count limit and the desired time limit. The time limit is in seconds, and refers to clock time, rather than CPU time. This initialization call is also considered to precede the simulation.

3. **The graphic output system**

Graphic output is handled by the SIMPLOTTER routines from the library. Entries used include GRAPH which establishes a new graph, GRAPHS which superimposes a new curve on the previous graph, and ORIGIN which is used to delete the axes and to advance the paper between plots. Further infor-
mation on these routines has been given under Facilities Em­ployed, p. 11.

E. Simulation of the Operator

A true simulation of the operator without an on-line graphic output is obviously not possible. But neither is it essential to the goals of the simulation. As the lightpen requires positioning that would come from the operator in the hardware version, and operator simulating routine OPER is provided to establish this information.

The routine OPER is called at the start of each frame by PLOT. The function is to establish LPON, a bit variable specifying whether the lightpen is in use, LPC, a two-element array with coordinates of the lightpen, and APPER, a variable specifying the size of the area about the coordinates within which a line may be detected by the pen. These items are all external, for communications to PLOT. They are read from cards using data-directed input. Each card starts with the frame number at which it is to be used. The routine will read this number and then wait until the frame number reaches or exceeds that specified before reading the data-directed information.

F. Computer Support

Though no general computer support was included in the specifications of the project, some is obviously desirable
for ease in preparing the demonstrations of the system. This can be divided into three categories: support of packing information into messages for the terminal and sending them, support for specification of display instructions, and other facilities.

1. **Message packing support**

   The computer maintains a copy $\delta_{MEM}$ of the terminal memory $MEM$, into which it may assemble programs before transmission. It also maintains two bit arrays $\delta_{MEMC}$ and $\delta_{MEHF}$ which record one of four states for each word, from which the need for transmission may be determined.

   **a. Word types**

   There are four possible states for a word of memory: DON'T CARE, CHANGED, SENT, and INITIAL.

   The DON'T CARE state is indicated by both bits being '0'. This is the usual initial state, and indicates that the program doesn't care (or know) what is in the particular word of memory at the terminal. This type of word is generated by several instructions. An example would be the third word of a procedure head, into which the return address will be planted during execution. In listings, this type of word is indicated by overprinting with XXXX.

   The second type of word is a CHANGED word. This is a word which has been changed in the computer copy, but not yet transmitted to the terminal. It is indicated by both bits being '1', and is listed without special notation.
The third type is a SENT word. This is one which has been sent to the terminal, and which the terminal will not change. Only a SENT word can be guaranteed to have the same values in both terminal and computer copies of the memory. As this word is already in agreement, it need not be transmitted again. Having such a word flagged can save a great deal in communications when few changes have been made in the memory from the previous frame. A SENT class word is flagged with $\texttt{SMEMC} = '1'$ and $\texttt{MEMF} = '0'$, and is underscored in listings.

The final type is for words which may be changed by the terminal (and therefore must not be flagged as SENT) but for which an initial value should be transmitted. INITIAL class words are flagged with $\texttt{HEMC} = '0'$ and $\texttt{MEMF} = '1'$, and are enclosed in parentheses in listings. The INITIAL class differs from the CHANGED class in that following transmission it is not converted to the SENT class.

b. Transmission routines Two routines are used for sending messages to the terminal. $\texttt{LOAD}$ is responsible for selecting a portion of memory to be transmitted in one message, and $\texttt{SEND}$ is responsible for transmitting it.

$\texttt{LOAD}$ receives as parameters the addresses of the first and last words of the memory to be sent. It will determine the need to transmit each word in this range from $\texttt{MEMF}$, though it will include a group of under four words that do
not require transmission rather than split the stream into two messages. It will break the message at the maximum length. Though the design specifies a maximum length of 255 words, the present implementation will limit to 128 words. This change was made to test the message dividing function without having to generate long memory loads, and has been retained simply because there was inadequate need to justify the recompilation needed to change.

If a group of sixteen or more words is to be sent, @SEND is called to assemble and transmit the message using the long load form. If fifteen words or less need to be sent, the short form load is specified by a call to @SENDS, which is another entry to @SEND.

@LOAD also has another entry @CORR which will follow the load with a resume. If any message of the correction is a short form, it will be saved until last and sent as a short load and resume, by calling @SENDTR. If no short load is encountered, the loading will be followed by a call to @RESUME.

@SEND, @SENDS, and @SENDTR receive as parameters the addresses of the first and last words of memory to be included in a single message. This routine is responsible for assembling a message, including the header, the location and length specifications, and the data. It is also responsible for transmitting this message by calling LINK, and for reflagging CHANGED type words as SENT.
Other entries, @START, @RESTART, and @RESUME provide for messages of these types, which will cause display to commence when received by the terminal.

A call to the entry @REQ will generate a data request message. On return, the requested data will be placed in @MEM.

c. Return information from terminal The return message following a transmitted one which specifies operation of the display will contain information on the reason for termination and the location. These will be placed in the external variables RETURN and NEXT for use by the calling program. Except for the values of these items which can be checked only by the calling program, the validity of the return messages is checked by @SEND. An error is reported if the message is not as expected.

2. Assembly routines

The goal on the assembly routine was to design an easily-used, easily-implemented routine, even if the cost of use was extreme. The design was passible on the first two items and excelled on the third.

The compile-time capabilities of PL/1 were used to allow the user to write macro instructions which are converted into PL/1 instructions to generate the desired terminal program. All macro instructions contain the character #, and all names used by the generated PL/1 instructions contain the character
which minimize the problems of the user inadvertently using a name used by the system as a variable or label in his coding.

Since instructions in a program must be able to refer to instructions which occur later in the program, some technique for obtaining these addresses is needed. It should be noted that the labels for which addresses are desired are labels in the terminal program, not in the computer program which generates the terminal program, and that the addresses required are addresses in the terminal memory (or the computer's copy of the terminal memory), not addresses in the generating program.

A one-pass assembler could be made by making all address references indirect and loading a reference table as true references were determined. This would require indirect reference use at execution, which was not in the terminal design. Two methods of two-pass assembly were considered. In the first, the program would be assembled with missing addresses on the first pass, and these addresses would be planted on the second pass. In the second, the first pass would only determine addresses for labels and the second pass would assemble the program. The last method was selected.

The source code, after compile-time processing, is a sequence of PL/1 instructions. In order to pass this twice, a do loop was used. On the first pass, time need not be spent
actually generating terminal code, so the do-loop counter is tested and only the location counter @LOC is incremented. Labels are integer variables, and are merely assigned the current value of the location counter when encountered. The do-loop is started with an #ASSM macro instruction and is terminated with an #ENDA. The assembly must be terminated before the assembled program can be transmitted to the terminal, though many #ASSM - #ENDA groups may appear in the generating program.

The macro instructions to generate terminal code are converted to calls on subroutines to generate the respective types of instruction code, with the parameter list specifying the additional data to be used. Details on the individual macros and their generating code are given in Appendix A, p. 103. These routines in turn call on a routine @STORE to place this generated code into consecutive locations of @MEM. @STORE will set the @MEMC and @MEMF flags as specified. If a CHANGE type word agrees with the previous content of the word in @MEM and this previous word was marked SENT, it will remain flagged as SENT, since the terminal is known to be in agreement. Thus if a new matrix is generated to replace a similar old one, even though the entire matrix may be regenerated in the computer, only those elements which have been changed will be sent to the terminal.
3. Other facilities

Two facilities are provided for listing the assembled program. If the variable \#LISTA is set to '1'B, then the assembly routines will list the code as assembled. It will be listed in hexadecimal, one instruction to a line, and will be flagged to show the type of word generated. No words will be flagged as SENT, however, as no SENT words are ever generated by the assembly routines. An example is shown in Figure 26.

```
0000 0001 0002 0003 0004 0005 0006 0007 0008 0009 000A 000B 000C 000D 000E 000F
0010 0011 0012 0013 0014 0015 0016 0017 0018 0019 001A 001B 001C 001D 001E 001F
0020 0021 0022 0023 0024 0025 0026 0027 0028 0029 002A 002B 002C 002D 002E 002F
0030 0031 0032 0033 0034 0035 0036 0037 0038 0039 003A 003B 003C 003D 003E 003F
0040 0041 0042 0043 0044 0045 0046 0047 0048 0049 004A 004B 004C 004D 004E 004F
0050 0051 0052 0053 0054 0055 0056 0057 0058 0059 005A 005B 005C 005D 005E 005F
0060 0061 0062 0063 0064 0065 0066 0067 0068 0069 006A 006B 006C 006D 006E 006F
0070 0071 0072 0073 0074 0075 0076 0077 0078 0079 007A 007B 007C 007D 007E 007F
0080 0081 0082 0083 0084 0085 0086 0087 0088 0089 008A 008B 008C 008D 008E 008F
0090 0091 0092 0093 0094 0095 0096 0097 0098 0099 009A 009B 009C 009D 009E 009F
00A0 00A1 00A2 00A3 00A4 00A5 00A6 00A7 00A8 00A9 00AA 00AB 00AC 00AD 00AE 00AF
00B0 00B1 00B2 00B3 00B4 00B5 00B6 00B7 00B8 00B9 00BA 00BB 00BC 00BD 00BE 00BF
00C0 00C1 00C2 00C3 00C4 00C5 00C6 00C7 00C8 00C9 00CA 00CB 00CC 00CD 00CE 00CF
00D0 00D1 00D2 00D3 00D4 00D5 00D6 00D7 00D8 00D9 00DA 00DB 00DC 00DD 00DE 00DF
00E0 00E1 00E2 00E3 00E4 00E5 00E6 00E7 00E8 00E9 00EA 00EB 00EC 00ED 00EE 00EF
00F0 00F1 00F2 00F3 00F4 00F5 00F6 00F7 00F8 00F9 00FA 00FB 00FC 00FD 00FE 00FF

Figure 26. Example of \#LISTA listing
```
The macro instruction LIST will cause the specified portion of $\text{MEM}$ to be listed. This listing will be sixteen words to a line, with no attention to instruction boundaries. Words will be flagged with any of the four classes, as in Figure 27.

The variable #SCALE may be set to scale all spacial dimensions in the subject. The fixed binary coding used can represent values only from -32 to +31.999+. If the subject had coordinate values up to 300, one could set #SCALE = 10, and the coordinates would be scaled down by a factor of 10, to a maximum value of 30. The user can thus use any size dimensions and any units, with the only restriction being that he be consistent.

A variety of routines are provided to generate matrices for substructure and camera use. They do not store the matrix, but return it as an argument. This permits successive routines to modify a matrix to obtain that desired. Routines

Figure 27. Example of #LIST listing

The variable #SCALE may be set to scale all spacial dimensions in the subject. The fixed binary coding used can represent values only from -32 to +31.999+. If the subject had coordinate values up to 300, one could set #SCALE = 10, and the coordinates would be scaled down by a factor of 10, to a maximum value of 30. The user can thus use any size dimensions and any units, with the only restriction being that he be consistent.

A variety of routines are provided to generate matrices for substructure and camera use. They do not store the matrix, but return it as an argument. This permits successive routines to modify a matrix to obtain that desired. Routines
to store such a matrix, either directly as a substructure matrix or modified as a camera matrix, are included in the assembly routines available.
IV. DEMONSTRATIONS OF THE SYSTEM

The TAG terminal simulation was used to demonstrate the proper operation of the system on several examples. The basic description of the example, the volume of communications required, and an illustration of the output produced will be presented in this chapter. The full source code for the demonstrations is included in Appendix A, p. 111. A partial listing of the code generated by the compile-time instructions and of the listing and trace outputs produced during the simulation is included on pages 123 and 61. These are both excerpts from the examples of this chapter.

A. A General Example

In the first example, a box is presented as a subject. Its lid is treated as a substructure, so that it may be manipulated independently. It is described in the closed position, but placed in an open one by a substructure matrix. A substructure consisting of a block with a pyramid roof is also created, and two copies of it are placed in the primary subject, using different size, orientation, and location. The substructure matrices are set up to have the box lid open at 30°, and to have one of the roofed blocks inside the box and one outside. A camera position is selected and a line of sight directed toward the subject. A camera matrix is specified for a perspective using a normal focal length lens.
The communications required to establish this entire drawing, starting with unknown terminal memory, was 624 bytes or about 2.1 seconds of communications. This would be typical of that required for set-up or for extensive changes.

B. Various Types of Views

In the next group of examples, the camera matrix is changed to create a variety of different views of the same subject. These demonstrations were simulated in sequence following the general example just presented, so only corrections to the display memory were required.
1. A long focal length lens

In this example, the only change from the first was that the focal length of the lens was changed to 4.0. The resultant view shows the proper functioning of windowing, for the subject exceeds the field of view.

![Figure 29. Long focal length lens](image)

Communications required for the correction before this example amounted to 23 bytes. An additional three bytes were returned at the termination of the previous example, making a total of 26 bytes between the two frames. This is one byte over that which is possible per double-frame at 2.4 kilobaud.
2. **An axonometric view**

In this example, the camera matrix was changed to produce an orthographic projection using the same axis of vision as in the previous examples. As before, 23 bytes were required to transmit changes, plus three returned from the terminal.

![Figure 30. Axonometric projection](image-url)
3. **A three-point perspective**

The camera matrix was again changed, this time to create a perspective with a focal length of 0.5, and to move the camera to a closer position so that the subject would better fill the picture. The axis of vision was tilted down in order to better display the three-point convergence. This time 25 + 3 bytes of communications were required.

![Figure 31. Three-point perspective](image-url)
4. A two-point perspective

In this example, the axis of vision was changed to a horizontal position, so that the picture plane would be vertical and no convergence of vertical lines would occur. The camera was lowered to reduce distortion, and the center of vision was repositioned placing the horizon above the picture center. Since this required changes in both camera and window matrices, communications requirements increased to 61 + 3 bytes.

Figure 32. Two-point perspective
5. **Multi-view and oblique drawings**

Next the terminal memory was modified to produce a multi-view drawing. Since this involved calling the drawing subroutine three times, once with the camera matrix for each view, it required more extensive changes than the simple matrix modifications in the prior examples.

![Figure 33. Multi-view drawing](image)

The modified program also included a camera matrix for an oblique drawing, and a call on the drawing subroutine with it loaded, but on a separate frame.
Figure 34. Oblique drawing

The communications required to set up was 229 + 3 bytes, or about 0.77 seconds. No additional communications were required between the multi-view frame and the oblique one.

C. Animation

Two animation simulations were executed. In both cases, 33 frames were generated, and these have been photographed on film for evaluation at the normal speed. Only every fourth frame is shown in the examples included in this report, since the small changes between individual frames is of no value unless viewed as a motion picture.
1. **An independent animation**

The first animation was intended to show the capabilities of the terminal which do not depend upon computer support during the animation. One substructure was given a linear translation, and the camera executed a linear dolly toward the subject.

![Figure 35. Animation without computer assistance](image-url)
It should be noted that the frame counter is only a reference number. Though it could be incremented by one to show the true frame number, it is easier to have the total sequence be 1024 counter frames, and to then select the increment according to how many frames of film were desired.

A total of 79 + 3 bytes were required to set up this example, after which it ran without computer communications.

2. Animation with computer assistance

The second animation used computer assistance during the animation to modify the lid substructure matrix so that the lid would open. A linear translation caused one of the roofed blocks to move upwards. This did not use computer support. It would have been possible to give a linear translation to each substructure (even the box lid) and a linear dolly to the camera with no increase in computer communications. The only reason that this was not done was that the result would have been confusing, rather than a good illustration of the terminal capabilities.
Figure 36. Animation with computer assistance

To set up this animation, 39 + 3 bytes were required. Thereafter between frames 3 bytes were returned from the terminal and 19 were sent to change the substructure matrix and resume, for a total communications of 22 bytes. This is within the 25 byte limit for double-framing.
V. OPPORTUNITIES FOR FURTHER DEVELOPMENT

A. Continuation of Present Design

The most obvious next step in continuing this research would be the detail design and implementation of TAG in hardware. However before such a step, further study of alternatives not investigated in this project should be undertaken.

The lightpen functions of the present design have not been fully tested. Programs using the lightpen to establish visibility, for tracking star use, and for the use of a menu have been written, but they have not been run because of limitations in both time and monetary budgets.

B. Improvements to Simulation Routines

Two major problems exist with the present routines. They are expensive to use, and the language used to describe the subject matter is difficult to write. For the limited application intended in this project, these were tolerable, and the costs of implementing changes did not appear justified. But for further use, improvements would be desirable.

The cost problem has many facets. First, the main routine, specifying the demonstration being simulated, is quite large, and must be compiled for each different run. Since this utilizes the compile-time processing of PL/1, the compilation is a significant expense. Second, the memory required is large. Routines were overlaid to conserve space, but even then, 160 K bytes were required. A major portion which over-
lays cannot reduce is that required by the main routine. Third, the cost of the link-edit step is not small. Since the main routine must be re-compiled for each different application, the link-edit step must also be performed. Finally, the major cost is the execution of the simulation.

Several possible improvements could be made in the assembly routines. First, a change to permit the description of the subject to be read as data rather than as program would allow a load module to be used, eliminating the costly re-compilation and re-link-editing for each application. Either this or a pre-compilation by other than the PL/1 compile-time facility would give greater flexibility to the source language design and also permit more efficient coding to be generated.

A two-step procedure, in which a computer simulation is performed as one step generating a file of communications which is read by the terminal simulation in a second step, would reduce the memory requirements. But this would preclude any dialog between the two.

Several improvements could be made to improve the efficiency of the simulating routines. One of the less efficient items in PL/1 is the procedure prologue. An improvement could be made in the calls on PLOT by PICT by using one call with both origin and terminus coordinates, rather than one for each. This would correspond less closely to the hard-
ware, but there appears to be no reason to require that all portions be simulated structurally at the same time. The use of OPT=2 for the compilation of the heavily-used routines in the terminal simulation should improve execution speed. Since these routines need not be re-compiled, the extra cost of compilation is a one-time cost. Eliminating diagnostic outputs, or at least making them optional, should improve performance. The coding of some portions in assembly language could offer potential improvements also.

C. Design Modification

The goal of this project was to demonstrate the feasibility of a design, and not to develop the best one. With the design complete and simulated, several possible improvements are now apparent.

It was noted in the simulation that in several situations the communications required was one byte over that possible for a double-frame. This situation arises from a ten-word correction. This will occur in the present design for most zooms. A zoom changes only the third column of the camera matrix, and usually only the first four elements of this column. But these are not consecutive elements of memory, and so ten elements must be transmitted to accomplish this change. A focal-length specification not concealed in the camera matrix would improve zoom efficiency, possibly even achieving this operation without computer assistance.
A second aspect of this problem is the communications overhead. For single-framing, only twelve bytes may be communicated per frame, and three are required each way for overhead, leaving only six bytes (three words) transmitting changes to the terminal. Three possible improvements are suggested. First, the computer could send a message while the terminal is drawing a frame, with the message being held in a buffer at the terminal until an ACCEPT MESSAGE instruction, was executed by the terminal. Since the terminal would not be halted awaiting a message, no message need be returned to the computer. Second, a shorter load form, which would assume the same location and length as the previous load, would reduce the heading information to one byte for successive corrections to the same locations. Third, a one byte return message could be used for anticipated stops, since there is no need to send the location of the halt back to the computer.

A potential problem exists in the use of the procedure head in the presence of errors in coding. If control is transferred sequentially into the procedure head, the return address will specify the instruction following that from which the last transfer of control occurred. If this were a procedure return, the previously returning procedure would be re-activated with the return address not modified, and the result would be a looping program. The instruction counter
and timer provide some protection against such a malfunction, but less than could be desired.

In the preparation of the un-tested programs using the tracking star, it became painfully apparent that the only arithmetic capability available at the terminal for modifying the star coordinates was that for incrementing the frame counter. Though this will permit tracking, it is difficult to code and would be inefficient to use. The terminal should be capable of tracking without computer assistance, and so some more convenient arithmetic capability should be provided. Providing a three-element offset register with incrementing capabilities for this purpose would be desirable.

D. Other Design Alternatives

A few design alternatives are suggested that differ sufficiently from that presented in this report that they are not considered as modifications.

The hardware required for the matrix multiplication for camera and substructure operations must be fast. Most probably this speed would be achieved by the use of parallel processing on different elements of the vectors by different multipliers. Though the same matrix multiplier could be used for both substructure and camera manipulations, this reduces the time available for each to half. Instead, the two matrices could be pre-multiplied, and only a single matrix multiplication by this product matrix would be required during re-
This may require the storage of a number of product matrices equal to the product of the numbers of substructure and camera matrices. Or the product matrices could be calculated from the two factors each time a matrix change is to be made. The matrix multiply hardware could be used for this purpose, the only change being that the output would be stored in the product matrix rather than being directed to the display processor.

Before the terminal is implemented in its present design, the alternative of designing a terminal around a minicomputer should be investigated. Even though the computer would provide more capabilities than needed, the economies of mass production may make this choice most economical.

E. The Visibility Problem

No attempt has been made to solve the visibility problem, except for the crude operator-implemented visibility through the lightpen. The design has the information on the 'Z' coordinate available, which would be useful in visibility determination.

F. An Efficient Drawing System

A drawing system, without the restrictions of following the hardware structure, could be made for efficient production of three-dimensional drawings on the host computer. The communications link would be unnecessary, and the assembly
routines could assemble program directly into the refresh memory used by the refresh routines. If multiple drawings of the same subject were not required, this memory could be eliminated and the "assembly" routines could call directly on the "display" routines. If an intermediate storage were to be used, it should probably be a list structure.

A major part of the design of a useful drawing system is the design of an input language. Most computer input devices are character-oriented. It is difficult to describe a three-dimensional spacial system with a string of characters.
VI. SUMMARY

A. Evaluation of the Tests

The TAG graphic terminal with voice-grade communications has been demonstrated capable of performing moderate animations with double-framing. Very limited animation is possible with single-framing.

A substantial increase in capabilities occurs with communications slightly over the 2.4 kilobaud limit. Several possible improvements are suggested to reduce the communications volume for these situations to within the stated limit.

B. Cost Considerations

No attempt to estimate the cost of the system is made. However some very general observations may be made. The cost may be divided into three major portions: the display hardware, the processor, and the memory.

The display hardware would not differ significantly from that of present systems.

The processor would be more extensive, with the matrix multiplication capability the significant addition. This addition would very possibly be as expensive as the present processor, but would probably not exceed double this cost. Thus the processor could be expected to cost between two and three times that of a conventional design.
The dominant item in memory would be the draw instructions. A conventional system would require two 10 bit coordinates, one bit for draw, and one bit to indicate other instructions, or a total of 22 bits. TAG requires 64 bits (four words) per draw instruction. The cost here, then, would be approximately tripled. The total memory requirements, even though triple that for the same display with two-dimensional design, are not large. The simulated system had only a 512 word (1 K byte) memory, and the demonstrated applications have never used over 392 words of this.

Thus it is estimated that the cost of implementing such a system would be between double and triple that of implementing a conventional system. This would be economically desirable if the alternative were a local computer.
VII. LITERATURE CITED


VIII. ACKNOWLEDGEMENTS

The author wishes to express his sincere appreciation to Dr. Arthur V. Pohm for guidance of his graduate program and thesis research, for patience and realization of the time required to complete an extensive research project while on full-time employment, and for confidence in its eventual completion. The freedom and opportunity for independent decision given the author has contributed greatly to the educational value of the design experience.

The assistance of the staff of the Iowa State University Computation Center has been indispensable in the debugging of the programs. Special thanks are due Tony Bible, Gib Scranton, Karl Kelley, Mary Beth Lieuwerknecht, and Rich Giddings.
IX. APPENDIX A - THE ASSEMBLER

The general structure of the assembly routines has been presented in Assembly Routines, p. 70. To fully understand the source listings for the demonstrations, details of the available macro instructions are presented here. An index of macro instructions is provided on p. 128.

In the ensuing discussion, macro instructions are underscored. Keywords appear in upper case and user-supplied names or parameters are in lower case. Such items are underscored where they appear in the explanatory discussion also.

A. Control Instructions

1. Assembly control

As has been stated, assembly of terminal refresh instructions into the computer's copy of the refresh memory is a two-pass process. The control instruction #ASSM_{loc}; indicates the start of a group of assembly instructions to be placed in refresh memory starting at loc. The end of the group is marked by the instruction #ENDA;.

Labels on the refresh program are indicated as label_, and must be placed within an assembly group. Labels on the refresh program are integer variables and must be either explicitly or implicitly declared as fixed binary (15, 0). In the demonstration programs, all instruction labels start with I and all matrix labels start with M. The space between
label and \# is necessary, as the compile-time processing recognizes the single character \# for substitution.

Labels of the familiar form \texttt{place:} may be used, and are labels on the assembly program. The user is cautioned that the instructions \texttt{#ASSM - #ENDA} create a DO-loop into which control must not be transferred.

On the first pass, labels are set to their proper values; on the second pass, the program is assembled. Thus during assembly, any label appearing within the current assembly group will be known, either before or after its appearance as a label. In addition, any label appearing in a previous assembly group will be known. A label, being an integer variable, may be assigned a value, though this requires that the user know details of memory allocation.

A listing of the program as assembled may be obtained by setting \texttt{#LISTA = `'I'B}. This listing option may be more easily controlled by the use of the instructions \texttt{#LISTON;} and \texttt{#LISTOFF;}. An instruction \texttt{#LIST (loca, locb);} calls for a listing of the current computer copy of the refresh memory from \texttt{loca} to \texttt{locb}. \texttt{#LISTAL;} will list all of memory, and \texttt{#LISTAS;} will list that portion assembled in the most recent \texttt{#ASSM - #ENDA} group.

The instruction \texttt{#LOC (loc);} may be used within an assembly group to change the location at which instructions are being assembled to \texttt{loc}. It is useful when making numerous
small corrections to memory, as it saves the re-assembly of intervening instructions and still keeps the assembly in one group so that labels from a later part will be known in an earlier one.

The instruction #CONT_(loc): will cause assembly to resume at loc. It is used after assembly has been terminated with an #ENDA, and the group defined by it is a separate group. It differs from an #ASSM_(loc): instruction only in that the record of the starting address used by #LISTAS and similar instructions is not altered. Thus a #LISTAS instruction following an #ASSM group and several #CONT groups would list memory from the start of the #ASSM group to the #ENDA of the last #CONT group. The #CONT instruction should be used instead of the #LOC instruction if intervening calculations are to be performed on which two passes are undesirable.

2. Transmission control

The instruction #START_(loc): will cause a START command to be sent to the terminal to start refresh at loc. The instructions #RESUME: and #RESTART: will produce RESUME and RESTART commands, causing refresh to commence at the point at which previously stopped or at location 0, respectively. The instruction #HALT: generates the HALT command.
The instruction \#LOAD\,(loca, locb); causes the transmission of load messages to bring the terminal's copy of the refresh memory into agreement with the computer's copy, from loca to locb.

The instruction \#CORR\,(loca, locb); will also update the terminal's memory, and will then resume display. If possible the SHORT LOAD AND RESUME command will be used.

The instructions \#LOADAL; and \#CORRAL; will update all of memory. \#LOADAS; and \#CORRAS; will update that portion just assembled.

The instruction \#REQ\,(loca, locb); will cause the transmission of a DATA REQUEST command. As a result of the REQUESTED DATA response to this command, the computer's copy of memory will be brought into agreement with the terminal's, from loca to locb.

B. Assembly Instructions

The assembly instructions are responsible for generating coding for the TAG terminal and for storing it in the computer's copy of the refresh memory.

1. Words

The word assembly instructions permit assembly of any desired code into memory. They are most useful for numeric quantities.
The instruction \#H(\text{hexstr})\; allows specification of a word in hexadecimal. The parameter \text{hexstr} is a string of four characters. The instruction \#N(\text{ivalue})\; allows specification of a word as an integer quantity. The parameter \text{ivalue} may be any known integer variable, constant, or expression, including a label. \#F(\text{value})\; specifies a word by a floating-point value which is converted to fixed binary (15, 10) for storage.

The instruction \#INIT; is used to specify a word as having an initial value which may be modified by the terminal. This instruction does not store a word, but precedes the the \#H, \#N, or \#F instruction specifying the value of the word to be so flagged.

The instruction \#SPACE(n); specifies the flagging of \text{n} words as DON'T CARE. Since the word values will be ignored, they are not specified.

2. Matrices

A matrix in memory normally consists of a head followed by fifteen words of data. Because of the amount of information contained therein, it is usually most convenient to build up a matrix in several steps and then to store it. Intermediate results may be a 3 x 3 rotation matrix or a 5 x 3 rotation and translation matrix.

a. Bit matrices Three instructions generate and store a bit matrix. They differ in the manner in which the
eight bit strings are specified. All require \textit{iadr}, the interrupt address for terminal interrupts, and \textit{ifr}, the initial value of the frame counter upon loading the generated matrix into the active bit matrix of the terminal.

\texttt{#BITS (bs1, bs2, bs3, bs4, bs5, bs6, bs7, iadr, ifr);} specifies the seven bit strings \texttt{bs1, ..., bs7} as bit (8). They represent respectively the draw, change, set, clear, terminal interrupt, lightpen computer interrupt, and instruction interrupt strings. The spare bit string which is not used is not specified, but will be set to zero. Words six through thirteen of the matrix will be flagged as DON'T CARE.

\texttt{#BITS (bitarray, iadr, ifr);} specifies these with \texttt{bitarray}, a seven-element bit (8) array.

\texttt{#BITH (b1a, b1b, b2a, b2b, b3a, b3b, b4a, b4b, b5a, b5b, b6a, b6b, b7a, b7b, iadr, ifr);} allows the specification of each bit string by two integers in the range 0 - 15, each specifying four bits or one nybble. If the variables A - F are declared integer and given the values 10 - 15, specification in hexadecimal is facilitated.

\subsection*{Substructure matrices}

A 3 x 3 rotation matrix for rotation about a coordinate axis is generated by

\texttt{#ROT (nax, ang, \_3\_);} The axis is identified by \texttt{nax}, 1 for X, 2 for Y, or 3 for Z. The angle, in degrees, through which the subject will be rotated, is specified by \texttt{ang}. For a positive \texttt{ang}, rotation will be clockwise about the positive axis.
specified by \texttt{max}. The generated matrix is returned to the 3 x 3 floating-point array \$R$. 

Rotation matrices about axes not parallel to a coordinate axis may be generated by a matrix multiplication of two (or more) rotation matrices. \texttt{#MMPY(\textit{a}, \textit{b}, \textit{c})} will return in \$C$ the matrix product \textit{A} * \textit{B}. The dimensions of \textit{A}, \textit{B}, and \$C$ must be compatible.

A 5 x 3 matrix incorporating rotation, scaling, translation, and incremental translation is generated by \texttt{#TRANSL(\textit{f}, \textit{pp}, \textit{pi}, \textit{gp}, \textit{qi}, \textit{sf}, \textit{sm})}. The parameter \textit{f} is a 3 x 3 rotation matrix specifying the desired rotation. The scale factor \textit{sf} may specify enlargement or reduction of the subject. The three-element floating-point vectors \textit{pp} and \textit{gp} specify the coordinates of points to control translation. A point of the substructure, in substructure coordinates, is specified by \textit{pp}; and \textit{gp} specifies a goal point in system coordinates to which the point \textit{pp} is to be translated. To achieve a translation varying with the frame counter, \textit{pi} and \textit{qi}, also three-element vectors, specify increments to \textit{pp} and \textit{gp}. They correspond to a change of 1024 in the frame counter. (This will be 1024 frames of film only if the frame counter is incremented by one each time the film is advanced.) The generated matrix is returned in the 5 x 3 floating-point array \$H$. 
The instruction `#MATRIX(m);` will store the 5 x 3 matrix m. Though m is floating-point, the stored matrix will be scaled as fixed binary (15, 10). A matrix head will be supplied, making a total of sixteen words stored.

c. **Camera matrices** To specify a camera matrix, a translation-rotation matrix to move the subject to camera coordinates is first required. This could be generated by the substructure matrix techniques, but this specification is not convenient.

The instruction `#CAMERA(cp, ci, brg, sla, tilt, $m);` will generate such a matrix, returning it in $m$. The camera coordinates and their increments are specified by the three-element vectors cp and ci. The bearing of the line of sight is brg, with 0° representing due north (a view from the south) and 90° due east (from the west). The slope angle of the line of sight is sla. An elevation view is represented by 0°, and a positive angle indicates looking upwards. Rotation of the camera about its optical axis is controlled by tilt, which will normally be 0°.

Once a matrix m to establish the camera coordinate system is generated, camera matrices may be stored for several different systems of projection. `#PERSP(m, fl);` will store a perspective camera matrix with relative focal length fl. A relative focal length of 1.0 corresponds to a 53° pyramid of vision. `#ORTHO(m, sf);` will store an orthographic or axono-
metric camera matrix, scaling by the scale factor \( sf \). The scale factor for the front \((X', Y')\) plane is \( sf \). The receding \((Z'')\) axis is drawn at an angle \( \text{ang} \) with a relative scale factor \( \text{rsf} \). An \( \text{ang} \) of 90° indicates to the right, and 90° indicates up. A \( \text{rsf} \) of 1.0 specifies a cavalier projection, and 0.5 specifies a cabinet projection. Any intermediate value will produce a general oblique projection.

To facilitate the generation of successive orthographic projections, the instruction \#PROJ\(\text{(ang, hp, m, $m)}\) is provided. The existing view is specified by the matrix \( m \), and the new matrix is returned in \$m. The orientation of the new plane of projection is specified by \( \text{ang} \), and the position by \( \text{hp} \). If the existing view were a horizontal view, the angle \( \text{ang} \) would be the bearing of the line of sight for the new view. The parameter \( \text{hp} \) is a vector, and specifies the position of the plane of projection (hinge line) in one of three manners. If \( \text{hp} \) is a three-element vector, it is the coordinates of a point on the hinge line. If it has two elements, it specifies only the X and Y coordinates, the Z coordinate being assumed to be zero. If \( \text{hp} \) has one element, this is used as the distance of the hinge line from the center of the display.

d. Window matrices The first window matrix specifies the center coordinates \( x_c \) and \( y_c \), the number of edges \( n \),
and the four edge equations corresponding to the display edges. It is generated and stored by \#CTR\((x_c, y_c, n)\).

The second window matrix specifies up to five additional edges. Its specification is a multi-instruction process. The instruction \#IND\(\) initializes the generation. The instruction \#IND\(i, \text{ang}, \text{dist}\) will generate an equation for a boundary at angle \text{ang} and at distance \text{dist} from the center. The bottom is represented by 0°, and the left by 90°. This equation is used for the ith edge in the generated matrix. After all edges have been specified, the instruction \#INDS\(\) will store the matrix.

3. **Control**

The instructions \#GOTO\(\)(loc)\) and \#CALL\(\)(loc)\) generate and store branch instructions to transfer control to loc.

A procedure is started by the instruction \#PROC\(\), which stores a procedure head. It must be ended by the instruction \#END\(\), which will store a branch to the second word of the head, to effect return should control sequence reach the end of the procedure. Procedure return is normally specified by the \#RETURN\(\) instruction, which also stores a branch to the second word of the procedure head.

4. **Frame and conditional**

Instructions to set, increment, or increment and test the frame counter are stored by \#SET\(\)(iset)\), \#INCR\(\)(incr)\)
and #IF (incr, itest);. Following the #IF instruction, the end of the low-option instruction group is indicated by #ELSE;.

Camera control instructions are stored by #CAMON;, #CAMOFF;, and #FRAME;. A timer check instruction is stored by #TIMER;. These may be combined in one instruction by #TF (nf); where the integer nf specifies the second variant nibble of the instruction. Camera and timer controls may be combined with frame operations with #SETF (iset, nf);, #INCRF (incr, nf);, and #IFF (incr, itest, nf);.

5. Draw

The instruction #DRAW (x, y, z); will generate and store a draw instruction with variant bits '00000001'. Parameters x, y, and z are floating-point, and are the coordinates of the point.

The instruction #TRAV (x, y, z); will generate and store a traverse instruction, which is a draw instruction with all variant bits '0'.

The instructions #DRAWV (y); and #TRAVV (y); allow the specification of the point coordinates by a three-element floating-point vector y.

The instructions #DRAWF (x, y, z, nv1, nv2); and #DRAWVF (y, nv1, nv2); allow specification of the variant nibbles by the integer parameters nv1 and nv2.
6. Matrix manipulation

The instruction \#MATF (mn, loc); or \#MATS (mn, loc); will generate and store instructions to fetch or store respectively the full matrix mn from/into memory starting at loc. \#MATFN (mn, nr, loc); or \#MATSNS (mn, nr, loc); will similarly fetch or store the first nr words of the matrix.

The instruction \#MATC (mn, nr, nf); will generate a matrix correct instruction to replace nf elements of matrix mn starting with the nrth element with the nf following words of memory. Though these nf words are a part of the matrix correct instruction, they are not generated by \#MATC. The \#N and \#F instructions may be used to generate them.

7. Other

The instruction \#INTRPT (nv1, nv2); will store an interrupt instruction with variant nybbles specified by nv1 and nv2.

The instruction \#STOP; will store a stop instruction.
C. Source Code for Examples

Figure 37 shows the source code used for the demonstrations of Chapter IV.

T_ALL: PROCEDURE OPTIONS (MAIN);

ON ERROR SNAP SYSTFM;
ON CONDITION (ERROR) SNAP;

DECLARE START ENTRY (FIXED BINARY (15), BIT (1));
TIMERST ENTRY (FIXED BINARY (15), FIXED BINARY (15))
FINISH ENTRY,
UMAIN ENTRY (FIXED BINARY (15), FIXED BINARY (15),
    FIXED BINARY (15), FIXED BINARY (15),
    FIXED BINARY (15)),
;
OPEN FILE (SYSPRINT) PAGE SIZE(32000);
CALL TIMERST (400, 550);
CALL START 10, '1/B';
CALL UMAIN (32, 1024, 32, 375, 32);
CALL FINISH;
END T_ALL;

UMAIN: PROCEDURE (NA, NB, NC, ND, NE);

The coding shown in Figure 38 is inserted at this point to provide compile-time processing of the subsequent macro instructions.

DECLARE RI (3, 3),
R (3, 3),
A (5, 3),
B (7) BIT (8),
P (3),
PI (3),
PI (1), ;

#SCALE = 4.;

Figure 37. Source code for examples
Figure 37. (continued)
Figure 37. (continued)
Figure 37. (continued)
\#ASSM (MBITS):
MBITS # B = '00000000'B;
   B (1) = '00000001'B;
#BITS (B, IEND, 10);

#LOC (MCTR):
   #CTR (0.5, 0.5, 4);

#LOC (ICYC):

ICYC # #MATF (2, MH);
   #MATF (3, MCTR);
   #CALL (IDRAW);
   #MATF (2, MF);
   #CALL (IDRAW);
   #MATF (2, MP);
   #CALL (IDRAW);
   #INCR (1);
   #TIMER;
   #FRAME;
   #MATF (2, MOBL);
   #CALL (IDRAW);
   #INCR (1);
   #TIMER;
   #FRAME;

IEND # #INTRPT (0, 0):
   #STOP;
   #GOTO (ICYC);

MH # P (1) = -5.:
   P (2) = 1.1
   P (3) = -4.1;
   #TRANSL (RI, 0., 0., P, 0., 1., A);
   #ORTHO (A, 0.5);

MF # P1 = 0.1:
   #PROJ (0, P1, A, A);
   #ORTHO (A, 0.5);

MOBL # #OBL (30., 0.5, A, 1.0);

MP # P1 = 3.0:
   #PROJ (270, P1, A, A);
   #ORTHO (A, 0.5);

#ENDA;
#LISTAS;
#LOADAS;
#RESTRT;

PUT PAGE;

Figure 37. (continued)
Figure 37. (continued)
MSUM# P (1) = 1.;
P (2) = 1.;
P (3) = 0.1;
P (1) = 0.1;
P (2) = 0.1;
P (3) = 8.1;
#TRANSL (R1, 0, 0, P, Pt, 0.5, A1);
#MATRIX (A1);

#LOC (MCAM);

MCAM # P (1) = 9.1;
P (2) = -12.1;
P (3) = 6.1;
#CAMERA (P, C, -30, -15, 0, A1);
#PERSP (A1, 1.1);

#LOC (ICYC);

ICYC # #MATF (2, MCAM);
#MATF (3, MCTR);
#CALL (IDRAW);
#INCR (NC);
#TIMER;
#FRMF;
IEND # #STOP;
#GOTO (ICYC);

#ENDA;

#LISTAS;

#LOADAS;

#START (1ST);

P (1) = 0.1;
P (2) = 3.1;
P (3) = 2.1;
A1 = ND / 100.1;
DO I = 0 TO NE;

PUT PAGE;

#ASSM (PLID);

#ROT (1, -I * A1, R1);
#TRANSL (R, P, 0, P, 0, 1., A1);
#MATRIX (A1);

#ENDA;

#LISTAS;

#CORRAS;

END;

RETURN;

END UMAIN;

Figure 37. (continued)
D. Compile-time Processing

As previously stated, the user's program is pre-processed by the compile-time capabilities of PL/1. Figure 38 shows the coding which is added to the start of the user's program to achieve this. In addition to the compile-time instructions, declarations of entry attributes are included.

```
DECLARE BLOCKS (2) FIXED BINARY (15, 0) EXTERNAL,
  8LOC8 POINTER EXTERNAL,
  8L FIXED BINARY (15, 0) BASED (8LOC8),

  8P FIXED BINARY (15, 0) EXTERNAL,
  8ELSE FIXED BINARY (15, 0) EXTERNAL,

  8H ENTRY (CHARACTER (41)),
  8N ENTRY (FIXED BINARY (15, 0)),
  8T ENTRY (FLOAT),
  8H ENTRY,

  8Matrix ENTRY ((5, 3) FLOAT),
  8PERSP ENTRY ((5, 3) FLOAT, FLOAT),
  8ORTH ENTRY ((5, 3) FLOAT, FLOAT),
  8BL ENTRY (FLOAT, FLOAT, (5, 3) FLOAT, FLOAT),

  8CTR ENTRY (FLOAT, FLOAT, FIXED BINARY (15, 0)),

  8WINDI ENTRY,
  8WIND ENTRY (FIXED BINARY (15, 0), FLOAT, FLOAT),
  8WINDS ENTRY,

  8BITS ENTRY ((7) BIT (8), FIXED BINARY (15),
               FIXED BINARY (15)),
  8BIT ENTRY (BIT (8), BIT (8), BIT (8), BIT (8),
             BIT (8), BIT (8), FIXED BINARY (15), FIXED BINARY (15)),
  8BITH ENTRY (FIXED BINARY (15), FIXED BINARY (15),
              FIXED BINARY (15), FIXED BINARY (15),
              FIXED BINARY (15), FIXED BINARY (15),
              FIXED BINARY (15), FIXED BINARY (15),
              FIXED BINARY (15), FIXED BINARY (15),
              FIXED BINARY (15)),

  8PROC ENTRY,
  8RETURN ENTRY,
  8END ENTRY,
  8GOTO ENTRY (FIXED BINARY (15)),
  8CALL ENTRY (FIXED BINARY (15)),

  8TF ENTRY (FIXED BINARY (15)),
  8SETF ENTRY (FIXED BINARY (15), FIXED BINARY (15)),
  8SET ENTRY (FIXED BINARY (15)),
  8INCR ENTRY (FIXED BINARY (15), FIXED BINARY (15)),
  8INCR ENTRY (FIXED BINARY (15)),
  8OFF ENTRY (FIXED BINARY (15), FIXED BINARY (15),
              FIXED BINARY (15)),
  8IF ENTRY (FIXED BINARY (15), FIXED BINARY (15)),

Figure 38. Compile-time instructions
DRAW ENTRY (FLOAT, FLOAT, FLOAT),
TRAY ENTRY (FLOAT, FLOAT, FLOAT),
DRAWF ENTRY (FLOAT, FLOAT, FLOAT,
FIXED BINARY (15), FIXED BINARY (15)),
DRAWV ENTRY ((3) FLOAT),
TRAVV ENTRY ((3) FLOAT),
DRAWVF ENTRY ((3) FLOAT,
FIXED BINARY (15), FIXED BINARY (15)),
MATF ENTRY (FIXED BINARY (15), FIXED BINARY (15)),
MATFN ENTRY (FIXED BINARY (15), FIXED BINARY (15),
FIXED BINARY (15)),
MAT ENTRY (FIXED BINARY (15), FIXED BINARY (15),
FIXED BINARY (15)),
MATS ENTRY (FIXED BINARY (15), FIXED BINARY (15),
FIXED BINARY (15)),
MATC ENTRY (FIXED BINARY (15), FIXED BINARY (15),
FIXED BINARY (15)),
INTRPT ENTRY (FIXED BINARY (15), FIXED BINARY (15)),
STOP ENTRY,

/*
STORE: */
SPACE ENTRY (FIXED BINARY (15)),

PROJ ENTRY (FLOAT, (*) FLOAT, (5,3) FLOAT, (5,3) FLOAT),

CAMERA ENTRY ((3) FLOAT, (3) FLOAT, FLOAT, FLOAT, FLOAT,
(5, 3) FLOAT),

TRANSL ENTRY ((3, 3) FLOAT, (3) FLOAT, (3) FLOAT,
(3) FLOAT, (3) FLOAT, FLOAT, (5, 3) FLOAT),

ROT ENTRY (FIXED DECIMAL (1), FLOAT, (3, 3) FLOAT),
MMPY ENTRY,
LIST ENTRY (FIXED BINARY (15), FIXED BINARY (15)),
LOAD ENTRY (FIXED BINARY (15), FIXED BINARY (15)),
CORR ENTRY (FIXED BINARY (15), FIXED BINARY (15)),

/*
SEND: */
START ENTRY (FIXED BINARY (15)),
HALT ENTRY,
RESUME ENTRY,
RESTART ENTRY,
REQ ENTRY (FIXED BINARY (15), FIXED BINARY (15)),

NEXT FIXED BINARY (15, 0) EXTERNAL,
RETURN FIXED BINARY (15, 0) EXTERNAL,
LISTA RIT (1) EXTERNAL INITIAL (*1'B),
SCALE FLOAT EXTERNAL INITIAL (1.0),

LOC  = ADDR (LOC  (2));

Figure 38. (continued)
% DECLARE OaiF, aELSE, aaiF2, a2, a3) CHARACTER;
% DECLARE OaiF, aELSE, aaiF2, a2, a3;
% a2 = ' ';
% a3 = ' ';
% aaiF = 'IF aP=1 THEN aL=al+':
% aaiF2 = 'IF aP=2 THEN CALL ':

% DECLARE (#ASSM, #CONT) ENTRY RETURNS (CHARACTER);
% DECLARE (#ENDA, #LOC, #, a1) CHARACTER;
% DECLARE I FIXED;
% DECLARE I, a1:
% 1 = 1000:

% #ASSM: PROCEDURE RETURNS (CHARACTER):
I = I + 1:
ai = 'a' || SUBSTR (I, 6):
#ENDA = 'END ' || a1:
RETURN ('' IF #LISTA THEN PUT SKIP (3); a1 || a3 ||
ai || ' DO aP = 1 TO 2:** a2 || a3 || ' aLOCs =*');
% END;

% #CCNT: PROCEDURE RETURNS (CHARACTER):
I = I + 1:
ai = 'a' || SUBSTR (I, 6):
#ENDA = 'END ' || a1:
RETURN ('' IF #LISTA THEN PUT SKIP (2); a1 || a3 ||
ai || ' DO aP = 1 TO 2*');
% END;

% #LOC = '': IF #LISTA & aP = 2 THEN PUT SKIP (2); a1 || a3;
% # = '': aL; IF #LISTA & aP = 2 THEN PUT SKIP; a1 || a3;

% DECLARE (#LISTON, #LISTOFF) CHARACTER;
% #LISTON = '#LISTA = **I**B;';
% #LISTOFF = '#LISTA = **O**B;';

% DECLARE #SPACE ENTRY (CHARACTER) RETURNS (CHARACTER);
% #SPACE: PROCEDURE (X) RETURNS (CHARACTER):
DECLARE X CHARACTER:
RETURN (aaiF || X || aaiF2 || 'aSPACE (** X || *):*');
% END;

% DECLARE (#H, #N, #F, #INIT) CHARACTER:
% #H = aaiF || ' 1' || aELSE || 'aH ';
% #N = aaiF || ' 1' || aELSE || 'aN ';
% #F = aaiF || ' 1' || aELSE || 'aF ';
% #INIT = ' ' || aaiF2 || 'aINIT; ' || a2 || a2;

% DECLARE (#MATRIX, #PERSP, #ORTHO, #OBL) CHARACTER:
% #PERSP = aaiF || '16' || aELSE || 'aPERSP ';
% #ORTHO = aaiF || '16' || aELSE || 'aORTHO ';
% #OBL = aaiF || '16' || aELSE || 'aOBL ';
% #MATRIX = aaiF || '16' || aELSE || 'aMATRIX ';

Figure 38. (continued)
% DECLARE (#BITS,  #BIT,  #BITH,  #CTR,  #WIND1,  #WIND,  #WINDS) CHARACTER;
%  #BITS  = @IF 1 16 ELSE 1 *#BITS *;
%  #BIT  = @IF 1 16 ELSE 1 *#BIT *;
%  #BITH  = @IF 1 16 ELSE 1 *#BIT *;
%  #CTR  = @IF 1 16 ELSE 1 *#CTR *;
%  #WIND1 = @IF 2 1 *#WIND1 *;
%  #WIND  = @IF 2 1 *#WIND *;
%  #WINDS = @IF 1 16 ELSE 1 *#WINDS *;

% DECLARE (#CAMERA,  #ROT,  #MMPY,  #TRANSL,  #PROJ) CHARACTER;
%  #CAMERA = @IF 2 1 *#CAMERA *;
%  #ROT = @IF 2 1 *#ROT *;
%  #MMPY = @IF 2 1 *#MMPY *;
%  #TRANSL = @IF 2 1 *#TRANSL *;
%  #PROJ = @IF 2 1 *#PROJ *;

% DECLARE (#PROC,  #END,  #RETURN,  #GOTO,  #CALL) CHARACTER;
%  #PROC = @IF 3 1 *#PROC *;
%  #END = @IF 1 2 *#END *;
%  #RETURN = @IF 1 2 *#RETURN *;
%  #GOTO = @IF 1 2 *#GOTO *;
%  #CALL = @IF 2 1 *#CALL *;

% DECLARE (#TF,  #TIMER,  #FRAME,  #CAMON,  #CAMOFF) CHARACTER;
%  #TF = @IF 1 1 *#TF *;
%  #TIMER = @IF 1 1 *#TIMER *;
%  #FRAME = @IF 1 1 *#FRAME *;
%  #CAMON = @IF 1 1 *#CAMON *;
%  #CAMOFF = @IF 1 1 *#CAMOFF *;

% DECLARE (#SET,  #SETF,  #INCR,  #INCRF,  #ELSE,  #I1) CHARACTER;
%  #SET = @IF 2 1 *#SET *;
%  #SETF = @IF 2 1 *#SETF *;
%  #INCR = @IF 2 1 *#INCR *;
%  #INCRF = @IF 2 1 *#INCRF *;

% WIF:  PROCEDURE RETURNS (CHARACTER);
%  I = I + 1;
%  I1 = 'I' I SUBSTR (I, 6);
%  #ELSE = I1 I = 'AL'; 'A3' A3;
%  RETURN ('#ELSE = ' I I I I ' ' ' A2 A3 A3 ' A1 I I I I ' A1 I I I I ' A1 ');
% END;

% #IFF:  PROCEDURE RETURNS (CHARACTER);
%  I = I + 1;
%  I1 = 'I' I SUBSTR (I, 6);
%  #ELSE = I1 I = 'AL'; 'A3' A3;
%  RETURN ('#ELSE = ' I I I ' ' A2 A3 A3 ' A1 I I I ' I I I I ' A1 ');
% END;

Figure 38.  (continued)
% DECLARE (#CPAW, "CRAWF, #TRAV, "ORAWV, "DRAWVF, #TRAVV) CHAR;
% "DRAW = @IF 1* 4* @ELSE 1*"DRAW*;  
% "DRAWF = @IF 1* 4* @ELSE 1*"DRAW*;  
% "TRAV = @IF 1* 4* @ELSE 1*"TRAV*;  
% "DRAWV = @IF 1* 4* @ELSE 1*"DRAWV*;  
% "TRAVV = @IF 1* 4* @ELSE 1*"TRAVV*;  
%
% DECLARE (#MATF, "MATFN, "MATSN, "MATC) CHARACTER;
% "MATF = @IF 1* 2* @ELSE 1*"MATF*;  
% "MATFN = @IF 1* 2* @ELSE 1*"MATFN*;  
% "MATS = @IF 1* 2* @ELSE 1*"MATS*;  
% "MATSN = @IF 1* 2* @ELSE 1*"MATSN*;  
% "MATC = @IF 1* 1* @ELSE 1*"MATC*;  
%
% DECLARE (#INTRPT, "STOP" CHARACTER;
% "INTRPT = @IF 1* 1* @ELSE 1*"INTRPT*;  
% "STOP = @IF 1* 1* @ELSE 1*"STOP*;  
%
% DECLARE (#LIST, "LISTAS, "LISTAL) CHARACTER;
% "LIST = "CALL "LIST*;  
% "LISTAS= "CALL "LIST (1, @LOCS (1), @LOCS (2) -1)*;  
% "LISTAL = "CALL "LIST (0, 511)*;  
%
% DECLARE (#LOAD, "LOADAS, "LOADAL) CHARACTER;
% "LOAD = "CALL "LOAD*;  
% "LOADAS = "CALL "LOAD (1, @LOCS (1), @LOCS (2) -1)*;  
% "LOADAL = "CALL "LOAD (0, 511)*;  
%
% DECLARE (#CORR, "CORRAS, "CORRAL) CHARACTER;
% "CORR = "CALL "CORR*;  
% "CORRAS = "CALL "CORR (1, @LOCS (1), @LOCS (2) -1)*;  
% "CORRAL = "CALL "CORR (0, 511)*;  
%
% DECLARE (#START, "RESUME, "RESTRT, "HALT, "REQ) CHARACTER;
% "START = "CALL "START*;  
% "RESUME = "CALL "RESUME*;  
% "RESTRT = "CALL "RESTRT*;  
% "HALT = "CALL "HALT*;  
% "REQ = "CALL "REQ*;  

Figure 38. (continued)
Figure 39 shows an example of the PL/1 source code generated by the compile-time processing applied to a portion of the macro instructions of Figure 37.

```
DECLARE RI (3, 3),
    R (3, 3),
    A (5, 3),
    B (7) BIT (8),
    P (3),
    PI (3),
    PI (1),
;

SCALE = 4.;
PUT PAGE;
IF #LISTA THEN PUT SKIP (3);
@001: DC AP = 1 TO 2;
@002: = (0);
    IF AP=1 THEN AL=AL+ 2;
    ELSE CALL AGOTO (1ST);
MBITS = AL; IF #LISTA & AP = 2 THEN PUT SKIP;
   B = *00000000*R;
   B (1) = *00000001*B;
   IF AP=1 THEN AL=AL+16;
   ELSE CALL MBITS (B, IEND, 1);
MBOX = AL; IF #LISTA & AP = 2 THEN PUT SKIP;
   IF AP=2 THEN CALL @ROTI (1, 0, RI);
   IF AP=2 THEN CALL @TRANSL (RI, 0, 0, 0, 1.0 , A);
   IF AP=1 THEN AL=AL+16;
   ELSE CALL MBOX (A);
MLID = AL; IF #LISTA & AP = 2 THEN PUT SKIP;
   IF AP=2 THEN CALL @ROTI (1, -30, R);
   P (1) = 0.;
   P (2) = 3.;
   P (3) = 2.;
   IF AP=2 THEN CALL @TRANSL (R, P, 0, 0, 1.0 , A);
   IF AP=1 THEN AL=AL+16;
   ELSE CALL MLID (A);
MSUBI = AL; IF #LISTA & AP = 2 THEN PUT SKIP;
   P (1) = 1.;
   P (2) = 1.;
   P (3) = 0.;
   IF AP=2 THEN CALL @TRANSL (RI, 0, 0, P, 0, 0.5, A);
   IF AP=1 THEN AL=AL+16;
   ELSE CALL MSUBI (A);
```

Figure 39. Generated PL/1 source code
```
MSUB2 = @L: IF #LISTA & @P = 2 THEN PUT SKIP;
P (1) = 6.;
P (2) = 0.;
P (3) = 0.;
IF @P=2 THEN CALL @ROTM (3, 30, R);
IF @P=2 THEN CALL @TRANSL (R, 0, 0, P, 0, 0.25, A);
IF @P=1 THEN @L=0L+16;
ELSE CALL @MATRX (A);

MCTR = @L: IF #LISTA & @P = 2 THEN PUT SKIP;
IF @P=1 THEN @L=0L+16;
ELSE CALL @CTR (0.5, 0.5, 4);

MCAM = @L: IF #LISTA & @P = 2 THEN PUT SKIP;
P (1) = 9.;
P (2) = -12.;
P (3) = 6.;
IF @P=2 THEN CALL @CAMRA (P, 0, -30, -15, 0, A);
IF @P=1 THEN @L=0L+16;
ELSE CALL @PERSP (A, 1.);

IDRAW = @L: IF #LISTA & @P = 2 THEN PUT SKIP;
IF @P=1 THEN @L=0L+ 3;
ELSE CALL @PROC ;
IF @P=1 THEN @L=0L+ 2;
ELSE CALL @CALL (IBOX);
IF @P=1 THEN @L=0L+ 2;
ELSE CALL @MATRX (1, MSUB1);
IF @P=1 THEN @L=0L+ 2;
ELSE CALL @CALL (ISUBS);
IF @P=1 THEN @L=0L+ 2;
ELSE CALL @RETURN ;
IF @P=1 THEN @L=0L+ 2;
ELSE CALL @END ;

IBOX = @L: IF #LISTA & @P = 2 THEN PUT SKIP;
IF @P=1 THEN @L=0L+ 3;
ELSE CALL @PROC ;
IF @P=1 THEN @L=0L+ 2;
ELSE CALL @MATRX (1, MBOX);
IF @P=1 THEN @L=0L+ 4;
ELSE CALL @TRAX (5., 0., 0.);
IF @P=1 THEN @L=0L+ 4;
ELSE CALL @DRAW (0., 0., 0.);
IF @P=1 THEN @L=0L+ 4;
ELSE CALL @DRAW (0., 0., 2.);
IF @P=1 THEN @L=0L+ 4;
ELSE CALL @DRAW (5., 0., 2.);

Figure 39. (continued)
```
IF \( \text{ap}=1 \) THEN \( aL = aL + 4 \); 
ELSE CALL \( @DRAW \ (5.0, 3.0, 2.1) \);
IF \( \text{ap}=1 \) THEN \( aL = aL + 4 \); 
ELSE CALL \( @DRAW \ (0.0, 3.0, 2.1) \);
IF \( \text{ap}=1 \) THEN \( aL = aL + 2 \); 
ELSE CALL \( @MATF \ (1.0, \text{MLID}) \);
IF \( \text{ap}=1 \) THEN \( aL = aL + 4 \); 
ELSE CALL \( @DRAW \ (0.0, 0.0, 2.1) \);
IF \( \text{ap}=1 \) THEN \( aL = aL + 4 \); 
ELSE CALL \( @DRAW \ (5.0, 0.0, 2.1) \);
IF \( \text{ap}=1 \) THEN \( aL = aL + 4 \); 
ELSE CALL \( @DRAW \ (5.0, 3.0, 2.1) \);
IF \( \text{ap}=1 \) THEN \( aL = aL + 2 \); 
ELSE CALL \( @MATF \ (1.0, \text{MBOX}) \);
IF \( \text{ap}=1 \) THEN \( aL = aL + 4 \); 
ELSE CALL \( @DRAW \ (5.0, 3.0, 0.1) \);
IF \( \text{ap}=1 \) THEN \( aL = aL + 4 \); 
ELSE CALL \( @DRAW \ (5.0, 0.0, 2.1) \);
IF \( \text{ap}=1 \) THEN \( aL = aL + 4 \); 
ELSE CALL \( @DRAW \ (5.0, 3.0, 0.1) \);
IF \( \text{ap}=1 \) THEN \( aL = aL + 4 \); 
ELSE CALL \( @DRAW \ (0.0, 3.0, 0.1) \);
IF \( \text{ap}=1 \) THEN \( aL = aL + 4 \); 
ELSE CALL \( @DRAW \ (0.0, 0.0, 2.1) \);
IF \( \text{ap}=1 \) THEN \( aL = aL + 4 \); 
ELSE CALL \( @DRAW \ (0.0, 3.0, 0.1) \);
IF \( \text{ap}=1 \) THEN \( aL = aL + 2 \); 
ELSE CALL \( @proc \);
IF \( \text{ap}=1 \) THEN \( aL = aL + 3 \); 
ELSE CALL \( @PROC \);
IF \( \text{ap}=1 \) THEN \( aL = aL + 4 \); 
ELSE CALL \( @TRAV \ (0.0, 0.0, 0.1) \);
IF \( \text{ap}=1 \) THEN \( aL = aL + 4 \); 
ELSE CALL \( @DRAW \ (0.0, 0.0, 1.1) \);
IF \( \text{ap}=1 \) THEN \( aL = aL + 4 \); 
ELSE CALL \( @DRAW \ (1.0, 0.0, 1.1) \);
IF \( \text{ap}=1 \) THEN \( aL = aL + 4 \); 
ELSE CALL \( @DRAW \ (1.0, 0.0, 0.0) \);
IF \( \text{ap}=1 \) THEN \( aL = aL + 4 \); 
ELSE CALL \( @DRAW \ (0.0, 0.0, 0.0) \);
IF \( \text{ap}=1 \) THEN \( aL = aL + 4 \); 
ELSE CALL \( @DRAW \ (0.0, 1.0, 0.0) \);

\[ \text{ISUBS} = aL; \text{IF} \ \text{MLISTA} & \text{ap} = 2 \ \text{THEN} \ \text{PUT} \ \text{SKIP}; \]
\[ \text{IF} \ \text{ap} = 1 \ \text{THEN} \ aL = aL + 3; \ \text{ELSE} \ \text{CALL} \ \text{PROC}; \]
\[ \text{IF} \ \text{ap} = 1 \ \text{THEN} \ aL = aL + 4; \ \text{ELSE} \ \text{CALL} \ \text{TRAV} \ (0.0, 0.0, 0.1); \]
\[ \text{IF} \ \text{ap} = 1 \ \text{THEN} \ aL = aL + 4; \ \text{ELSE} \ \text{CALL} \ \text{DRAW} \ (0.0, 0.0, 1.1); \]
\[ \text{IF} \ \text{ap} = 1 \ \text{THEN} \ aL = aL + 4; \ \text{ELSE} \ \text{CALL} \ \text{DRAW} \ (1.0, 0.0, 1.1); \]
\[ \text{IF} \ \text{ap} = 1 \ \text{THEN} \ aL = aL + 4; \ \text{ELSE} \ \text{CALL} \ \text{DRAW} \ (1.0, 0.0, 0.0); \]
\[ \text{IF} \ \text{ap} = 1 \ \text{THEN} \ aL = aL + 4; \ \text{ELSE} \ \text{CALL} \ \text{DRAW} \ (0.0, 0.0, 0.0); \]
\[ \text{IF} \ \text{ap} = 1 \ \text{THEN} \ aL = aL + 4; \ \text{ELSE} \ \text{CALL} \ \text{DRAW} \ (0.0, 1.0, 0.0); \]

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{image}
\caption{Figure 39. (continued)}
\end{figure}
IF $\alpha P=1$ THEN $\alpha L=\alpha L+4$;
    ELSE CALL $\alpha ORAW (0., 1., 1.);$
ELSE CALL $\alpha ORAW (0., 0., 1.);$
IF $\alpha P=1$ THEN $\alpha L=\alpha L+4$;
    ELSE CALL $\alpha ORAW (0.5, 0.5, 1.5);$
IF $\alpha P=1$ THEN $\alpha L=\alpha L+4$;
    ELSE CALL $\alpha ORAW (1., 0., 1.);$
IF $\alpha P=1$ THEN $\alpha L=\alpha L+4$;
    ELSE CALL $\alpha ORAW (1., 1., 1.);$
IF $\alpha P=1$ THEN $\alpha L=\alpha L+4$;
    ELSE CALL $\alpha ORAW (0., 1., 1.);$
IF $\alpha P=1$ THEN $\alpha L=\alpha L+4$;
    ELSE CALL $\alpha ORAW (0.5, 0.5, 1.5);$
IF $\alpha P=1$ THEN $\alpha L=\alpha L+4$;
    ELSE CALL $\alpha ORAW (1., 1., 1.);$
IF $\alpha P=1$ THEN $\alpha L=\alpha L+4$;
    ELSE CALL $\alpha ORAW (1., 0., 0.);$
IF $\alpha P=1$ THEN $\alpha L=\alpha L+4$;
    ELSE CALL $\alpha ORAW (1., 0., 0.);$
IF $\alpha P=1$ THEN $\alpha L=\alpha L+4$;
    ELSE CALL $\alpha ORAW (1., 0., 0.);$
IF $\alpha P=1$ THEN $\alpha L=\alpha L+4$;
    ELSE CALL $\alpha ORAW (0., 1., 0.);$
IF $\alpha P=1$ THEN $\alpha L=\alpha L+4$;
    ELSE CALL $\alpha ORAW (1., 0., 1.);$
IF $\alpha P=1$ THEN $\alpha L=\alpha L+4$;
    ELSE CALL $\alpha ORAW (0., 1., 0.);$
IF $\alpha P=1$ THEN $\alpha L=\alpha L+4$;
    ELSE CALL $\alpha ORAW (1., 0., 1.);$
IF $\alpha P=1$ THEN $\alpha L=\alpha L+4$;
    ELSE CALL $\alpha ORAW (0., 1., 0.);$
IF $\alpha P=1$ THEN $\alpha L=\alpha L+4$;
    ELSE CALL $\alpha ORAW (0., 1., 0.);$
IF $\alpha P=1$ THEN $\alpha L=\alpha L+2$;
    ELSE CALL $\alpha TRAV ;$
IF $\alpha P=1$ THEN $\alpha L=\alpha L+2$;
    ELSE CALL $\alpha ORAW (0., 1., 0.);$
IF $\alpha P=1$ THEN $\alpha L=\alpha L+2$;
    ELSE CALL $\alpha ORAW (0., 1., 0.);$
START = \alpha L; IF \#LISTA & \alpha P = 2 THEN PUT SKIP;
    IF $\alpha P=1$ THEN \alpha L=\alpha L+ 2;
        ELSE CALL $\alpha MATF (0, MBITS);$
    IF $\alpha P=1$ THEN \alpha L=\alpha L+ 1;
        ELSE CALL $\alpha TRF (6);$
START = \alpha L; IF \#LISTA & \alpha P = 2 THEN PUT SKIP;
    IF $\alpha P=1$ THEN \alpha L=\alpha L+ 2;
        ELSE CALL $\alpha MATF (2, MCA);$
    IF $\alpha P=1$ THEN \alpha L=\alpha L+ 2;
        ELSE CALL $\alpha MATF (3, MCTR);$
    IF $\alpha P=1$ THEN \alpha L=\alpha L+ 2;
        ELSE CALL $\alpha CALL (IDRAW);$
    IF $\alpha P=1$ THEN \alpha L=\alpha L+ 2;
        ELSE CALL $\alpha INC (1);$
    IF $\alpha P=1$ THEN \alpha L=\alpha L+ 1;
        ELSE CALL $\alpha TRF (1);$
    IF $\alpha P=1$ THEN \alpha L=\alpha L+ 1;
        ELSE CALL $\alpha TRF (2);$
START = \alpha L; IF \#LISTA & \alpha P = 2 THEN PUT SKIP;
    IF $\alpha P=1$ THEN \alpha L=\alpha L+ 1;
        ELSE CALL $\alpha STOP ;$
    IF $\alpha P=1$ THEN \alpha L=\alpha L+ 2;
        ELSE CALL $\alpha TRF (2);$

Figure 39. (continued)
ELSE CALL AGOTO (ICYC);

END 3001:
CALL @LIST (@LOCS (1), @LOCS (2) -1) ;
CALL @LOAD (@LOCS (1), @LOCS (2) -1) ;
CALL @START (1ST);

PUT PAGE:

IF »LISTA THEN PUT SKIP (3):

3002: DO &P = 1 TO 2;
    @LOCS = (MCAM);
    IF &P=1 THEN &L=AL+16;
    ELSE CALL &PERSP (A, 4.0):

END 3002:
CALL @LIST (@LOCS (1), @LOCS (2) -1) ;
CALL @CORR (@LOCS (1), @LOCS (2) -1) ;

Figure 39. (continued)
### E. Index of Macro Instructions

<table>
<thead>
<tr>
<th>MACRO INSTRUCTION</th>
<th>COMPILE-TIME PROCESSING Page</th>
<th>APPLICATION DISCUSSION Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>120</td>
<td>99</td>
</tr>
<tr>
<td>#ASSM</td>
<td>120</td>
<td>99</td>
</tr>
<tr>
<td>#BIT</td>
<td>121</td>
<td>104</td>
</tr>
<tr>
<td>#BITH</td>
<td>121</td>
<td>104</td>
</tr>
<tr>
<td>#BITS</td>
<td>121</td>
<td>104</td>
</tr>
<tr>
<td>#CALL</td>
<td>121</td>
<td>108</td>
</tr>
<tr>
<td>#CAMERA</td>
<td>121</td>
<td>106</td>
</tr>
<tr>
<td>#CAMOFF</td>
<td>121</td>
<td>109</td>
</tr>
<tr>
<td>#CAMON</td>
<td>121</td>
<td>109</td>
</tr>
<tr>
<td>#CONT</td>
<td>120</td>
<td>101</td>
</tr>
<tr>
<td>#CORR</td>
<td>122</td>
<td>102</td>
</tr>
<tr>
<td>#CORRAL</td>
<td>122</td>
<td>102</td>
</tr>
<tr>
<td>#CORRAS</td>
<td>122</td>
<td>102</td>
</tr>
<tr>
<td>#CTR</td>
<td>121</td>
<td>108</td>
</tr>
<tr>
<td>#DRAW</td>
<td>122</td>
<td>109</td>
</tr>
<tr>
<td>#DRAWF</td>
<td>122</td>
<td>109</td>
</tr>
<tr>
<td>#DRAWV</td>
<td>122</td>
<td>109</td>
</tr>
<tr>
<td>#DRAWVF</td>
<td>122</td>
<td>109</td>
</tr>
<tr>
<td>#ELSE</td>
<td>121</td>
<td>109</td>
</tr>
<tr>
<td>#END</td>
<td>121</td>
<td>108</td>
</tr>
<tr>
<td>#ENDA</td>
<td>120</td>
<td>99</td>
</tr>
<tr>
<td>#F</td>
<td>120</td>
<td>103</td>
</tr>
<tr>
<td>MACRO INSTRUCTION</td>
<td>COMPILE-TIME PROCESSING Page</td>
<td>APPLICATION DISCUSSION Page</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>#FRAME</td>
<td>121</td>
<td>109</td>
</tr>
<tr>
<td>#GOTO</td>
<td>121</td>
<td>108</td>
</tr>
<tr>
<td>#H</td>
<td>120</td>
<td>103</td>
</tr>
<tr>
<td>#HALT</td>
<td>122</td>
<td>101</td>
</tr>
<tr>
<td>#IF</td>
<td>121</td>
<td>109</td>
</tr>
<tr>
<td>#IFF</td>
<td>121</td>
<td>109</td>
</tr>
<tr>
<td>#INCR</td>
<td>121</td>
<td>108</td>
</tr>
<tr>
<td>#INCRF</td>
<td>121</td>
<td>109</td>
</tr>
<tr>
<td>#INIT</td>
<td>120</td>
<td>103</td>
</tr>
<tr>
<td>#INTRPT</td>
<td>122</td>
<td>110</td>
</tr>
<tr>
<td>#LIST</td>
<td>122</td>
<td>100</td>
</tr>
<tr>
<td>#LISTAL</td>
<td>122</td>
<td>100</td>
</tr>
<tr>
<td>#LISTAS</td>
<td>122</td>
<td>100</td>
</tr>
<tr>
<td>#LISTOFF</td>
<td>120</td>
<td>100</td>
</tr>
<tr>
<td>#LISTON</td>
<td>120</td>
<td>100</td>
</tr>
<tr>
<td>#LOAD</td>
<td>122</td>
<td>102</td>
</tr>
<tr>
<td>#LOADAL</td>
<td>122</td>
<td>102</td>
</tr>
<tr>
<td>#LOADAS</td>
<td>122</td>
<td>102</td>
</tr>
<tr>
<td>#LOC</td>
<td>120</td>
<td>100</td>
</tr>
<tr>
<td>#MATC</td>
<td>122</td>
<td>110</td>
</tr>
<tr>
<td>#MATHF</td>
<td>122</td>
<td>110</td>
</tr>
<tr>
<td>#MATHFN</td>
<td>122</td>
<td>110</td>
</tr>
<tr>
<td>#MATRIX</td>
<td>120</td>
<td>106</td>
</tr>
<tr>
<td>MACRO INSTRUCTION</td>
<td>Compile-Time Processing Page</td>
<td>Application Discussion Page</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>#MATS</td>
<td>122</td>
<td>110</td>
</tr>
<tr>
<td>#MATSN</td>
<td>122</td>
<td>110</td>
</tr>
<tr>
<td>#MMPY</td>
<td>121</td>
<td>105</td>
</tr>
<tr>
<td>#N</td>
<td>120</td>
<td>103</td>
</tr>
<tr>
<td>#OBL</td>
<td>120</td>
<td>107</td>
</tr>
<tr>
<td>#ORTHO</td>
<td>120</td>
<td>106</td>
</tr>
<tr>
<td>#PERSP</td>
<td>120</td>
<td>106</td>
</tr>
<tr>
<td>#PROC</td>
<td>121</td>
<td>108</td>
</tr>
<tr>
<td>#PROJ</td>
<td>121</td>
<td>107</td>
</tr>
<tr>
<td>#REQ</td>
<td>122</td>
<td>102</td>
</tr>
<tr>
<td>#RESTART</td>
<td>122</td>
<td>101</td>
</tr>
<tr>
<td>#RESUME</td>
<td>122</td>
<td>101</td>
</tr>
<tr>
<td>#RETURN</td>
<td>121</td>
<td>108</td>
</tr>
<tr>
<td>#ROT</td>
<td>121</td>
<td>104</td>
</tr>
<tr>
<td>#SET</td>
<td>121</td>
<td>108</td>
</tr>
<tr>
<td>#SETF</td>
<td>121</td>
<td>109</td>
</tr>
<tr>
<td>#SPACE</td>
<td>120</td>
<td>103</td>
</tr>
<tr>
<td>#START</td>
<td>122</td>
<td>101</td>
</tr>
<tr>
<td>#STOP</td>
<td>122</td>
<td>110</td>
</tr>
<tr>
<td>#TF</td>
<td>121</td>
<td>109</td>
</tr>
<tr>
<td>#TIMER</td>
<td>121</td>
<td>109</td>
</tr>
<tr>
<td>#TRANSL</td>
<td>121</td>
<td>105</td>
</tr>
<tr>
<td>#TRAV</td>
<td>122</td>
<td>109</td>
</tr>
<tr>
<td>MACRO INSTRUCTION</td>
<td>COMPILE-TIME PROCESSING Page</td>
<td>APPLICATION DISCUSSION Page</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>#TRAVV</td>
<td>122</td>
<td>109</td>
</tr>
<tr>
<td>#WIND</td>
<td>121</td>
<td>108</td>
</tr>
<tr>
<td>#WINDI</td>
<td>121</td>
<td>108</td>
</tr>
<tr>
<td>#WINDS</td>
<td>121</td>
<td>108</td>
</tr>
</tbody>
</table>
X. APPENDIX B - LISTING OF SIMULATION PROGRAMS

The PL/1 source code for the programs developed and employed in the simulation of the TAG system is given. These routines are called by the programs presented in Appendix A, which would differ for each demonstration.

An index of entry names is provided on page 169.

A. The Hexidecimal Conversion Routine

The routine @HEX converts a value to or from a string of hexidecimal characters, to permit input or output in this form.
@HEX: PROCEDURE (NI) RETURNS (CHAR (4));

DECLARE NI FIXED BINARY (15, 0),
    NB BIT (8),
    HI CHARACTER (4),
    CH CHARACTER (4),
    CS (4) CHARACTER (1) DEFINED CH POSITION (1),
    C2 CHARACTER (2) DEFINED CH POSITION (1),
    NS BIT (16),
    NA (4) BIT (4) DEFINED NS POSITION (1),
    NC BIT (8) DEFINED NS POSITION (1),
    CI FIXED BINARY (15, 0),
    C CHARACTER (4),
    H6XCH (0: 15) CHARACTER (1) STATIC INITIALIZED ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
    'A', 'B', 'C', 'D', 'E', 'F'),
    N, I;

    NS = UNSPEC (NI);
    DC I = 1 TO 4;
        UNSPEC (CI) = (12) '0' || NA (I);
        CS (I) = H6XCH (CI);
    END;
    RETURN (CH);

@HEX2: ENTRY (NB) RETURNS (CHARACTER (2));
    NC = NB;
    DO I = 1 TO 2;
        UNSPEC (CI) = (12) '0' || NA (I);
        CS (I) = H6XCH (CI);
    END;
    RETURN (C2);

I@HEX: ENTRY (HI) RETURNS (FIXED BINARY (15));
    CH = HI;
    DC I = 1 TO 4;
        C = CS (I);
        IF C < 'A' THEN GO TO ERR;
        UNSPEC (N) = (8) '0' || UNSPEC (C);
        IF C < '0'
            THEN IF C < 'G'
                THEN N = N + 9; /* A - F */
                ELSE GO TO ERR;
            ELSE; /* 0 - 9 */
                NA (I) = SUBSTR (UNSPEC (N), (13));
            END;
        UNSPEC (N) = NS;
        RETURN (N);

ERR: PUT LIST (' INVALID CHARACTER ', C, ' IN I@HEX CALL')
    SKIP (2);
    SIGNAL CONDITION (ERROR);
    RETURN (0);

END @HEX;

Figure 40. The hexadecimal conversion routine
B. The Matrix Generation Routines

The matrix generation routines are called by the user's program to generate rotation and translation matrices for use as camera and substructure matrices in the program. These routines each return a $3 \times 3$ or a $5 \times 3$ matrix in the last argument of the call list.

```
ACAMERA: PROCEDURE (CP, CI, BRG, SLA, TILT, SCM);

DECLARE CP (3), /* CAMERA POSITION */
        CI (3), /* CAMERA INCREMENT */
        BRG, /* BEARING OF AXIS */
        SLA, /* AXIS SLOPE ANGLE */
        TILT, /* CAMERA TILT */
        SCM (5, 3), /* MATRIX RETURNED */

#SCALE FLOAT EXTERNAL INITIAL (1.0),
B (5, 3),
BP POINTER,
B3 (3, 3) BASED (BP),
B4P POINTER,
B4 (2, 3) BASED (B4P),
A (3, 3),
C2 (2, 3),
CC (2, 3),
MMMPY ENTRY,
MRRT ENTRY,

BP = ADDR (B);
B4P = ADDR (B (4, 1));
CC (1, *) = CP / #SCALE;
CC (2, *) = CI / #SCALE;
call MRRT (3, BRG, A);
call MRRT (1, - SLA - 90.0, B);
if TILT = 0 then go to TR;
call MRRT (3, TILT, B);
call MMMPY (A, (B3), A);
TR: call MMMPY (CC, A, C2);
B4 = - C2;
B3 = A;
SCM = B;
return;
end ACAMERA;
```

Figure 41. The matrix generation routines
@TRANSL: PROCEDURE (R, PP, PI, GP, GI, S, $V);

DECLARE R (3, 3), /* ROTATION MATRIX */
PP (3), /* PIVOT POINT */
PI (3), /* PIVOT INCREMENT */
GP (3), /* GOAL POINT */
GI (3), /* GOAL INCREMENT */
S, /* SCALE FACTOR */
$V (5, 3), /* MATRIX RETURNED */

SCALE FLOAT EXTERNAL INITIAL (1.0),
A (5, 3),
AP POINTER,
A3 (3, 3) BASED (AP),
A4P POINTER,
A4 (2, 3) BASED (A4P),

P (2, 3),
G (2, 3),
RS (3, 3),
A2 (2, 3),
\$MMPY ENTRY,

AP = ADDR (A);
A4P = ADDR (A (4, 1));

P (1, *) = PP / SCALE;
P (2, *) = PI / SCALE;
G (1, *) = GP / SCALE;
G (2, *) = GI / SCALE;
RS = R * S;
CALL \$MMPY (P, RS, A2);
A4 = G - A2;
A3 = RS;
$V = A;
RETURN;

END @TRANSL;

Figure 41. (continued)
DECLARE ANG, /* ANGLE OF NEW VIEW */
   PT (*), /* POINT ON HINGE L */
   PREV (5, 3), /* MATRIX FOR PREV V */
   $NEW (5, 3), /* MATRIX RETURNED */
#SCALE FLOAT EXTERNAL INITIAL (1.0),
P (3),
PP POINTER,
P2 (2) BASED (PP),
A (3, 3),
B (3, 3),
W (5, 3),
D FIXED BINARY (15, 0),
OSW (3) LABEL INITIAL (D1, D2, D3),
@TRANSL ENTRY ((3, 3) FLOAT, (3) FLOAT, (3) FLOAT,
   (3) FLOAT, (3) FLOAT, FLOAT, (5, 3) FLOAT),
@ROT ENTRY (FIXED DECIMAL (1), FLOAT, (3, 3) FLOAT),
@MMPY ENTRY,
;
PP = ADDR (P);
D = DIM (PT, 1);
GO TO OSW (D); GO TO ERR;
D1: P (1) = -SINO (ANG) * PT (1) / #SCALE;
P (2) = -COSO (ANG) * PT (1) / #SCALE;
P (3) = 0.; GO TO PR;
D2: P2 = PT / #SCALE;
P (3) = 0.; GO TO PR;
D3: P = PT / #SCALE;

PR: CALL @ROT (3, ANG, A);
   CALL @ROT (1, -90.0, B);
   CALL @MMPY (A, B, B);
   CALL @ROT (3, -ANG, A);
   CALL @MMPY (B, A, A);
   CALL @TRANSL (A, P, 0., P, 0., 1., W);
   CALL @MMPY (PREV, A, $NEW);
   $NEW (4, *) = $NEW (4, *) + W (4, *);
RETURN;

ERR: PUT LIST ('SPROJ DIMENSION ERROR: PT ('', ''), SKIP;
   SIGNAL CONDITION (ERROR);
   $NEW = 0.;
RETURN;

END @PROJ;

Figure 41. (continued)
aROT: PROCEDURE (AX, AN, $MAT);

DECLARE AX FIXED DECIMAL (11), /* AXIS: 1=X, 2=Y, 3=Z */
AN FLOAT, /* ANGLE IN DEGREES */
$MAT (3,3) FLOAT, /* MATRIX RETURNED */

SM (3,3) FIXED BINARY (15,0) STATIC
INITIAL (0, 1, -1, -1, 0, 1, 1, -1, 0);
IM (3,3) FIXED BINARY (15,0) STATIC
INITIAL (1, 0, 0, 0, 1, 0, 0, 0, 1);

$MAT = SM * SIND (AN) * IM * COSD (AN);
$MAT (AX, *) = 0;
$MAT (*, AX) = 0;
$MAT (AX, AX) = 1;
RETURN;
END aROT;

aMMPY: PROCEDURE (A, R, $C);

DECLARE A (*, *), /* FIRST FACTOR */
B (*, *), /* SECOND FACTOR */
$C (*, *), /* PRODUCT RETURNED */
T (L, N) CONTROLLED,
I, J, L, M, N, LL, MM, NN:

L = DIM (A, 1);
M = DIM (A, 2);
MM = DIM (B, 1);
N = DIM (B, 2);
LL = DIM ($C, 1);
NN = DIM ($C, 2);
IF (LL += L | MM += M | NN += N) THEN GO TO ERR;
ALLOCATE T;

CO I = 1 TO L;
DO J = 1 TO N;
   T (I, J) = SUM (AI, *) * Bj (*, J));
END;
END;
$C = T;
RETURN;

ERR: PUT EDIT (' MATMPY DIMENSION ERROR', '(', L, ',', M, '*, M, *
* *) * ('*, MM, '*, N, ') = ('*, LL, '*, NN, ')')
(SKIP (2), A (23), SKIP, X (5), A (1), F (2), A (2), F (2),
A (5), F (2), A (2), F (2), A (5), F (2), A (2), F (2),
A (1), SKIP);
SIGNAL CONDITION (ERROR);
$C = 0;
RETURN;
END aMMPY;

Figure 41. (continued)
C. The Assembly Routines

The assembly routines are called by the user's program to generate code for the terminal. Each routine, after generating the desired code, calls &STORE to plant it in &MEM, the computer's copy of the display memory.

\[ \text{DECLARE CS CHARACTER (4),} \]
\[ \text{NR FIXED BINARY (15, 0),} \]
\[ \text{X FLOAT,} \]
\[ \text{SCALE FLOAT EXTERNAL INITIAL (1.0),} \]
\[ \text{aVAL (0: 15) FIXED BINARY (15, 0) EXTERNAL,} \]
\[ \text{aVALP PCINTER,} \]
\[ \text{aVALO FIXED BINARY (15, 10) BASED (aVALP),} \]
\[ \text{aVALF (0: 15) BIT (2) EXTERNAL INITIAL ('11'B),} \]
\[ \text{laHEX ENTRY (CHARACTER (4)) RETURNS (FIXED BINARY (15)),} \]
\[ \text{aSTORE ENTRY (FIXED BINARY (15)),} \]
\[ \text{VALF BIT (2) STATIC INITIAL ('11'B),} \]
\[ \text{SVAL (0) = laHEX (CS);} \]
\[ \text{GO TO S;} \]
\[ \text{aN: ENTRY (NR);} \]
\[ \text{aVAL (0) = NR;} \]
\[ \text{GO TO S;} \]
\[ \text{aF: ENTRY (X);} \]
\[ \text{aVALP = ADDR (aVAL (0));} \]
\[ \text{aVALO = X / SCALE;} \]
\[ \text{S: aVALF (0) = VALF;} \]
\[ \text{VALF = '11'B;} \]
\[ \text{CALL aSTORE (0);} \]
\[ \text{RETURN;} \]
\[ \text{aINIT: ENTRY;} \]
\[ \text{VALF = '01'B;} \]
\[ \text{RETURN;} \]

END aH;

Figure 42. The assembly routines
MATRIX: PROCEDURE (CM):

DECLARE CM (5, 3),
   FL, SF, ANG, RSF,
   #SCALE FLOAT EXTERNAL INITIAL (1.0),
   #VAL (0: 15) FIXED BINARY (15, 0) EXTERNAL,
   #VALP POINTER,
   #VALM (5, 3) FIXED BINARY (15, 10) BASED (#VALP),
   IM (3, 3) FLOAT STATIC
      INITIAL (1., 0., 0., 0., 1., 0., 0., 0., 0., 1.),
   C3 (3, 3),
   C (5, 3),
   #MMPY ENTRY,
   #STORE ENTRY (FIXED BINARY (15)),
   S;
   C = CM;

CMS: #VAL (0) = -1;
   #VALP = ADDR (#VAL (1));
   #VALM = C;
   CALL #STORE (15);
   RETURN;

#PERSP: ENTRY (CM, FL):
   C = CM;
   IF FL = 0 THEN GO TO PERSERR;
   C (*, 3) = C (*, 3) / FL;
   GO TO CMS;

PERSERR: PUT LIST (' FOCAL LENGTH = 0.0 NOT VALID. 1.0 USED')
   SKIP;
   SIGNAL CONDITION (ERROR);
   GO TO CMS;

#ORTHO: ENTRY (CM, SF):
   C = CM;

CMC: C (*, 3) = 0.1
   S = SF * #SCALE;
   IF S < 0.3125 THEN GO TO ORTHERR;
   C (4, 3) = -10. / S;
   GO TO CMS;

ORTHERR: PUT LIST (' SLANT ANGLE NOT VALID. 15.0 DEGREES USED')
   SKIP;

#OBL: ENTRY (ANG, RSF, CM, SF):
   C3 = IM;
   C3 (3, 1) = RSF * -COSD (ANG);
   C3 (3, 2) = RSF * -SIND (ANG);
   CALL #MMPY (IM, C3, C);
   GO TO CMS;

Figure 42. (continued)
/* SMATRIX (CONTINUE) */

ORTHERR: PUT LIST (* SCALE FACTOR = *, SF, * REPLACED BY MIN*)
Ckip;
C (4, 3) = -32.
SIGNAL CONDITION (ERROR);
GO TO CMS;

END SMATRIX;

@RITS: PROCEDURE (BITS, LOC, ILOC);
DECLARE LOC, ILOC,
(RIT1, RIT2, RIT3, RIT4, RIT5, RIT6, RIT7) RIT (A),
RITS (7) RIT (8),
(R1, R2, R3, R4, P5, R6, R7, R8, R9, R10, R11, R12, R13,
R14) FIXED RINARY (15),
@VAL (0: 15) FIXED RINARY (15, 0) EXTERNAL,
@VALF (0: 15) RIT (2) EXTERNAL INITIAL ((16) (1)'11'B),
@STORE ENTRY (FIXED RINARY (15)),

UNSPEC (@VAL (1)) = RITS (1) || RITS (2);
UNSPEC (@VAL (2)) = RITS (3) || RITS (4);
UNSPEC (@VAL (3)) = RITS (5) || RITS (6);
UNSPEC (@VAL (4)) = RITS (7) || '00000000'B;
GO TO B;

@BITH: ENTRY (B1, B2, B3, B4, B5, R6, R7, R8, R9, R10, R11, R12,
R13, R14, LOC, ILOC);
@VAL (1) = 4096 * B1 + 256 * B2 + 16 * R3 + B4;
@VAL (2) = 4096 * B5 + 256 * B6 + 16 * R7 + B8;
@VAL (3) = 4096 * R9 + 256 * R10 + 16 * R11 + B12;
@VAL (4) = 4096 * R13 + 256 * R14;
GO TO B;

@BIT: ENTRY (RIT1, RIT2, RIT3, RIT4, RIT5, RIT6, RIT7, LOC, ILOC);
UNSPEC (@VAL (1)) = RIT1 || RIT2;
UNSPEC (@VAL (2)) = RIT3 || RIT4;
UNSPEC (@VAL (3)) = RIT5 || RIT6;
UNSPEC (@VAL (4)) = RIT7 || '00000000'B;

B: @VAL (0) = 15;
@VAL (5) = LOC;
CC [i = 6 TO 13];
@VALF (1) = '00'B;
END;
@VAL (14) = 1024;
@VAL (15) = LOC;
CALL @STORE (15);
RETURN;

END @BITH;

Figure 42. (continued)
end aCTR;

awiND: PROCEDURE;

DECLARE NR, ANG, DIST,

@VAL (0: 15) FIXED BINARY (15, 0) EXTERNAL,
@VALP POINTER,
@VALM (5, 3) FIXED BINARY (15, 10) BASED (@VALP),

WIND (5, 3) FIXED BINARY (15, 10) STATIC INITIAL
(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0),
@STORE ENTRY (FIXED BINARY (15)),

WIND = 0.0;
WIND (*, 3) = -1.0;
RETURN;

awiND: ENTRY (NR, ANG, DIST);
WIND (NR, 1) = SIN (ANG);
WIND (NR, 2) = COS (ANG);
WIND (NR, 3) = - DIST;
RETURN;

awiNDs: ENTRY;
@VALP = ADDR (@VAL (1));
@VALM = WIND;
WIND (0) = 15;
CALL @STORE (15);
RETURN;

end awiNDi;

Figure 42. (continued)
DECLARE LOC,

@LOC @POINTER @EXTERNAL,
@LOC @FIXED @BINARY (15, 0) @BASED (@LOC@P),

@VAL (0: 15) @FIXED @BINARY (15, 0) @EXTERNAL,
@VALF (0: 15) @BIT (2) @EXTERNAL @INITIAL ((16)(1)'11'B),

@PROCL (100) @FIXED @BINARY (15, 0) @STATIC,
@PROCT @FIXED @BINARY (15, 0) @STATIC @INITIAL (0),
@STORE @ENTRY @FIXED @BINARY (15)),

L;

@PROC = @PROC@CT + 1;
@PROC@L (@PROC) = @LOC + 1;
@VAL (0) = 8194;
@VAL (1) = 4057;
@VALF (2) = '00'B;
CALL @STOREF (2);
RETURN;

@RETURN: ENTRY;
IF @PROC = 0 THEN GO TO RETERR;
L = @PROC@L (@PROC);
GO TO G;

RETER@: PUT LIST (' #RETURN NOT IN #PROC IGNORED') SKIP (2);
SIGNAL CONDITION (ERROR);
L = @LOC + 2;
GO TO G;

@END: ENTRY;
IF @PROC = 0 THEN GC TO ENDERR;
L = @PRC@L (@PROC);
@PROC = @PROC - 1;
GO TO G;

ENDERR: PUT LIST (' EXTRA #END IGNORED') SKIP (2);
SIGNAL CONDITION (ERROR);
L = @LOC + 2;
GO TO G;

@GOTC:
@CALL: ENTRY (LOC);
L = LOC;

G: @VAL (0) = 4057;
@VAL (1) = L;
CALL @STORE (1);
RETURN;

END @PROC;

Figure 42. (continued)
@TF: PROCEDURE (NF);

   DECLARE NF, ISFT, INCR, ITEST,
   @VAL (0: 15) FIXED BINARY (15, 0) EXTERNAL,
   @ELSE FIXED BINARY (15, 0) EXTERNAL,
   @LOC PNTER EXTERNAL,
   @LOC FIXED BINARY (15, 0) BASED (@LOC PNTER),
   @STORF ENTRY (FIXED BINARY (15)),
   SIZE FIXED BINARY (15, 0),
   N;

   @VAL (0) = 12288 + 16 * NF;
   CALL @STORF (0);
   RETURN;

@SFTF: ENTRY (ISFT, NF);
   N = NF;
   G0 TO SFT;

@SFT: ENTRY (ISFT);
   N = 0;

   SET: @VAL (0) = 12545 + 16 * N;
   @VAL (1) = ISFT;
   CALL @STORF (1);
   RETURN;

@INCRF: ENTRY (INCR, NF);
   N = NF;
   GC TO INC;

@INCR: ENTRY (INCR);
   N = 0;

   INC: @VAL (0) = 12801 + 16 * N;
   @VAL (1) = INCR;
   CALL @STORE (1);
   RETURN;

Figure 42. (continued)
/* 3TF (CONTINUED) */

aiFF: ENTRY (INCR, ITEST, NF);
    N = NF;
    GO TO IF;

aiF: ENTRY (INCR, ITEST):
    N = 0;
    IF: SIZE = ELSE - ALOC - 1;
        IF SIZE < 3 THEN GO TO IFN;
        IF SIZE > 15 THEN GO TO IFL;
    IFS: AVAL (0) = 13056 + 16 * N + SIZE;
        AVAL (1) = INCR;
        AVAL (2) = ITEST;
        CALL ASTORE (2);
        RETURN;
    IFN: IF SIZE < 2 THEN PUT LIST (' NO ELSE FOLLOWING IF') SKIP (2);
        ELSE PUT LIST (' NO LOW OPTION INSTRUCTION BEFORE ELSE')
                            SKIP (2);
        IFF: SIGNAL CONDITION (ERROR);
               SIZE = 2;
               GO TO IFS;
    IFL: PUT LIST (' LOW OPTION INSTRUCTIONS ARE ' + SIZE,
                    ' WORDS LONG. MAXIMUM LENGTH IS 13 WORDS.') SKIP (2);
               GO TO IFE;

END 3TF;

Figure 42. (continued)
Figure 42. (continued)
SMATF: PROCEDURE (MN, LOC);
DECLARE MN, LOC, NR, NF,
  SVAL (0: 15) FIXED BINARY (15, 0) EXTERNAL,
  @STORE ENTRY (FIXED BINARY (15)),
  SVAL (0) = 24817 + 256 * MN;
M: SVAL (1) = LOC;
  CALL @STORE (1);
  RETURN;

SMATS: ENTRY (MN, NR, LOC);
  SVAL (0) = 24577 + 256 * MN + 16 * NR;
  GO TO M;

SMATSC: ENTRY (MN, NR, NF);
  SVAL (0) = 28672 + 256 * MN + 16 * NR + NF;
  CALL @STORE (1);
  RETURN;

END SMATF;

Figure 42. (continued)
\texttt{aINTRPT: PROCEDURE (I, J)};

\texttt{DECLARE I, J,}
\texttt{\quad aVAL (0: 15) FIXED BINARY (15, 0) EXTERNAL,}
\texttt{\quad aSTORE ENTRY (FIXED BINARY (15))},
\texttt{\quad aVAL (0) = -32768 * 256 + 1 + 16 * J;}
\texttt{\quad CALL aSTORE (0);}
\texttt{\quad RETURN;}

\texttt{aSTOP: ENTRY:}
\texttt{\quad aVAL (0) = -28672;}
\texttt{\quad CALL aSTORE (0);}
\texttt{\quad RETURN;}

\texttt{END aINTRPT;}

\texttt{aSTORE: PROCEDURE (N);}

\texttt{DECLARE N,}
\texttt{\quad aVAL (0: 15) FIXED BINARY (15, 0) EXTERNAL,}
\texttt{\quad aVALF (0: 15) BIT (2) EXTERNAL INITIAL ((16) (1)'11'B),}
\texttt{\quad aVALC (0: 15, 2) BIT (1) DEFINED aVALF POSITION (1),}
\texttt{\quad aLOCP POINTER EXTERNAL,}
\texttt{\quad aLOC FIXED BINARY (15, 0) BASED (aLOCP),}
\texttt{\quad aMEM (0: 511) FIXED BINARY (15, 0) EXTERNAL,}
\texttt{\quad aMEMF (0: 511) BIT (1) EXTERNAL INITIAL ((512) (1)'0'B),}
\texttt{\quad aMEMC (0: 511) BIT (1) EXTERNAL INITIAL ((512) (1)'0'B),}
\texttt{\quad #LISTA BIT (1) EXTERNAL INITIAL ('1'B),}
\texttt{\quad FLAG (0: 3) CHARACTER (6) STATIC INITIAL}
\texttt{\quad (' XXXX ', '{ }, '___'_'),}
\texttt{\quad aHEX RETURNS (CHARACTER (4)),}
\texttt{\quad I, J;}
\texttt{\quad IF #LISTA THEN DO:}
\texttt{\quad PUT EDIT (aHEX (aLOC), (aHEX (aVAL (I)) DO I = 0 TO N))}
\texttt{\quad (aLOC, X(2), A(4), X(4), A(4), 5(X(2), 3(X(2), A(4))));}
/* aSTCRE (CONTINUED) */

DO I = 0 TO N;
  IF aVALC (I, 1) THEN
    IF aVAL (I) = aMEMC (aLOC + I) THEN
      IF aMEMF (aLOC + I) = aMEMC (aLOC + I) THEN
        ELSE GO TO F;
      ELSE aMEMC (SLOC + 11)
    ELSE IF aVAL (I) = 3MEM (SLOC + I) THEN
      ELSE aVALC (I, 2)
    THEN aMEMC + I) = aVALC (I); /* TERM AGREES */
    ELSE /* TERM MAY MODIFY */
    IF aVALC (I, 2)
      THEN aMEMC + I) = aVALC (I); /* SEND ALWAYS */
    ELSE /* DON'T CARE */
      aMEMF (aLOC + I) = aVALC (I, 1);  
aMEMF (aLOC + I) = aVALC (I, 2);
E: END;

aLOC = aLOC + N + 1;
aVALF = 'll'R;
RETURN;

aSPACE: ENTRY (N);

J = aLOC + N - 1;
IF #LISTA THEN PUT ECIT (aHEX (aLOC), N, ' XXXX THRU LOC ', aHEX (J)) (SKIP (2), X (2), A (4), F (6), A (17), A (4));
 CC I = aLOC TO J;
  aMEMF (I) = 'O'B;
  aMEMC (I) = 'O'B;
END;
VLOC = J + 1;
RETURN;

FNO aSTCRE;

Figure 42. (continued)
D. The Listing and Transmission Routines

These routines are used to list the contents of SMEM and to transmit messages to the terminal.

```
DLIST: PROCEDURE (LSTART, LSTOP);

DECLARE LSTART FIXED BINARY (15, 0),
       LSTOP FIXED BINARY (15, 0),
       aMEM (0:511) FIXED BINARY (15, 0) EXTERNAL,
       aMEMC (0:511) BIT (1) EXTERNAL,
       aMEMF (0:511) BIT (1) EXTERNAL,
       FLAG (0:3) CHARACTER (6) STATIC INITIAL
           (' XXXX ', '(', ' ', ' ', ' '),
       S FIXED BINARY (15, 0),
       aHEX RETUNS (CHARACTER (4)),
       i, k;

PUT EDIT ('LIST FROM ', aHEX (LSTART), ' THRU ', aHEX (LSTOP))
       (SKIP (4), X (1), A (10), A (4), A (6), A (4));

UNSPEC (S) = UNSPEC (LSTART) & '11111111111111111111111';

PUT EDIT ((aHEX (k)) DO K = 0 TO 15))
       (SKIP (2), X (13), 4 (X (2), 4 (X (2), A (4))), SKIP (2));

DO I = 5 TO LSTOP BY 16;
    PUT EDIT (I, aHEX (I),
       (aHEX (aMEM (I+K))) DO K = 0 TO 15))
       (SKIP, F (5), X (2), A (4), X (2),
       4 (X (2), 4 (X (2), A (4))));
    PUT EDIT ((FLAG (aMEMC (I + K)) || aMEMF (I + K))
       DO K = 0 TO 15)) (SKIP (0), X (16), 4 (4 A (6), X (2)));

END;

RETURN;

END DLIST;
```

Figure 43. The listing and transmission routines
DECLARE LOC FIXED BINARY (15, 0),
FIN FIXED BINARY (15, 0),

MEMF (0: 511) BIT (1) EXTERNAL INITIAL ((512) (1) '0'B),
L FIXED BINARY (15, 0),
F FIXED BINARY (15, 0),
OKCT FIXED BINARY (15, 0),
FIRST BIT (1),
LOAD BIT (1),
FN FIXED BINARY (15, 0),
LEN FIXED BINARY (15, 0),
FS FIXED BINARY (15, 0),
FL FIXED BINARY (15, 0),

SEND ENTRY (FIXED BINARY (15), FIXED BINARY (15)),
SEND ENTRY (FIXED BINARY (15), FIXED BINARY (15)),
SEND ENTRY (FIXED BINARY (15), FIXED BINARY (15)),
RESUME ENTRY,
J, JP, I;
LOAD = '1'B;
GO TO C;

CORR: ENTRY (LOC, FIN);
LOAD = '0'B;
FIRST = '1'B;
C: L = LOC;
F = FIN;

Figure 43. (continued)
/* @LCAD (CONTINUED) */

SCAN: DO I = L TO F;
    IF @MEMF (I) THEN GO TO START;
END;
    IF LOAD THEN RETURN;
    ELSE IF FIRST THEN CALL @RESUME;
        ELSE CALL @SEND (FS, FL);
    RETURN;

START: L = I;
    OKCT = 0;
    DO I = L TO F;
        IF @MEMF (I)
            THEN OKCT = 0;
        ELSE DO;
            OKCT = OKCT + 1;
            IF OKCT >= 4 THEN GO TO SEND;
        END;
    END;

SEND: FN = I - OKCT;
    LEN = FN - L + 1;
    IF LFN < 16
        THEN IF FN < 4096
            THEN GO TO SHORT;
        IF LEN > 128 THEN GO TO LONG;
        CALL @SEND (L, FN);
        L = I + 1;
        GO TO SCAN;

SHORT: IF LOAD THEN GO TO SHORTS;
    IF FIRST THEN GO TO STORE;
SHORTS: CALL @SEND (L, FN):
        L = I + 1;
        GO TO SCAN;
STORE: FS = L;
    FL = FN;
    L = I + 1;
    FIRST = 'O'B;
    GO TO SCAN;
LONG: JP = L;
    DO J = L + 128 TO FN BY 128;
        CALL @SEND (JP, J - 1);
    JP = J;
    END;
    L = JP;
    GO TO SEND;

END @LOAD;

Figure 43. (continued)
DECLARE IST FIXED BINARY (15, 0),
    IFIN FIXED BINARY (15, 0),
    LOC FIXED BINARY (15, 0),
    MESSAGE CHARACTER (516) VARYING EXTERNAL,
    $\text{MEM} (0: 511) \text{ FIXED BINARY (15, 0) EXTERNAL},$
    $\text{HEMF (0: 511) BIT (1) \text{ EXTERNAL INITIAL ((512) (1) '0'B)},}$
    $\text{MEMC (0: 511) BIT (1) \text{ EXTERNAL INITIAL ((512) (1) '0'B)},}$
    RETURN FIXED BINARY (15, 0) EXTERNAL,
    NEXT FIXED BINARY (15, 0) EXTERNAL,
    MSG (516) BIT (8),
    MSGP POINTER,
    MSGC CHARACTER (516) BASED (MSGP),
    WD BIT (16),
    LB BIT (8) DEFINED WD POSITION (1),
    RB BIT (8) DEFINED WD POSITION (9),
    LINK ENTRY,
    I, J, L, LR;

L = IFIN - IST + 1;
MSG (1) = '11111001'B;
WD = UNSPEC (IST);
MSG (2) = LB;
MSG (3) = RB;
WD = UNSPEC (L);
MSG (4) = RB;
DO I = 1 TO L;
    J = IST + I - 1;
    WD = UNSPEC (MEM (J));
    $\text{HEMF (J)} = \text{HEMF (J)} \cup \text{MEMC (J)}$;
    MSG (2 * I + 3) = LB;
    MSG (2 * I + 4) = RB;
END;
L = 2 * L + 4;
GO TO SEND;

&SENOS: ENTRY (IST, IFIN);
L = IFIN - IST + 1;
MSG (1) = '11111010'B;
WD = SUBSTR (UNSPEC (IST), 5) || SUBSTR (UNSPEC (L), 13);
MSG (2) = LB;
MSG (3) = RB;
DO I = 1 TO L;
    J = IST + I - 1;
    WD = UNSPEC (MEM (J));
    $\text{HEMF (J)} = \text{HEMF (J)} \cup \text{MEMC (J)}$;
    MSG (2 * I + 2) = LB;
    MSG (2 * I + 3) = RB;
END;
L = 2 * L + 3;

Figure 43. (continued)
/* aSEND (CONTINUED) */
SEND: MSGP = ADDR (MSG);
MESSAGE = SUBSTR (MSGC, 1, L);
CALL LINK;
LR = LENGTH (MESSAGE);
IF LR = 0 THEN RETURN;
GC TO ERR:

aSTART: ENTRY (LOC);
MSG (1) = "11111110"B;
WD = UNSPEC (LOC);
MSG (2) = LB;
MSG (3) = RB;
L = 3;
GO TO SENDR:

aHALT: ENTRY;
MSG (1) = "11111111"B;
L = 1;
MSGP = ADDR (MSG);
MESSAGE = SUBSTR (MSGC, 1, L);
CALL LINK;
LR = LENGTH (MESSAGE);
IF LR = 3 THEN GO TO ERR;
MSGC = MESSAGE;
IF MSG (1) = "11110000"B THEN GO TO ERR;
RETURN:

aRESUME: ENTRY;
MSG (1) = "11111100"B;
L = 1;
GO TO SENDR:

aRESTR: ENTRY;
MSG (1) = "11111101"B;
L = 1;
GO TO SENDR:

aSENR: ENTRY (1ST, IFIN):
L = IFIN - IST + 1;
MSG (1) = "11111011"B;
WD = SUBSTR (UNSPEC (IST), 5) || SUBSTR (UNSPEC (L), 13);
MSG (2) = LB;
MSG (3) = RB;
DO I = 1 TO L;
    J = IST + I - 1;
    WD = UNSPEC (AMEM (J));
    AMEMF (J) = AMEMF (J) & ~ AMEMC (J);
    MSG (2 * I + 2) = LB;
    MSG (2 * I + 3) = RB;
END;
L = 2 * L + 3;

Figure 43. (continued)
/* aSEND (CONTINUED) */

SENDR: MSGP = ADDR (MSG);
MESSAGE = SUBSTR (MSGC, 1, L);
CALL LINK;

LR = LENGTH (MESSAGE);
IF LR ≠ 3 THEN GO TO ERR;
MSGC = MESSAGE;
UNSPF (I) = '00000000000'B || SUBSTR (MSG (1), 5);
IF I > 4 THEN GO TO ERR;
RETURN = I;
LB = MSG (2);
RB = MSG (3);
UNSPEC (#NEXT) = WD;
RETURN;

SREQ: ENTRY (1ST, IFIN);
L = IFIN - 1ST + 1;
MSG (1) = '11111000'B;
WD = UNSPEC (1ST);
MSG (2) = LB;
MSG (3) = RB;
WD = UNSPEC (L);
MSG (4) = RB;
MSGP = ADDR (MSG);
MESSAGE = SUBSTR (MSGC, 1, 4);
CALL LINK;

LR = LENGTH (MESSAGE);
IF LR ≠ 2 * L + 1 THEN GO TO ERR;
MSGC = MESSAGE;
IF MSG (1) = '11110111'B THEN GO TO ERR;
DO I = 1 TO L;
LB = MSG (2 * I);
RB = MSG (2 * I + 1);
UNSPEC (#MEM (1ST + I - 1)) = WD;
END;
RETURN;

ERR: PUT LIST (* RETURN MESSAGE FROM TERMINAL INVALID*) SKIP (2);
SIGNAL CONDITION (ERROR);
RETURN = 0;
RETURN;

END aSEND;

Figure 43. (continued)
LINK: PROCEDURE;

DECLARE MESSAGE CHARACTER (516) VARYING EXTERNAL,
MSG P POINTER,
MSG (516) RIT (8) BASED (MSGP),

L FIXED BINARY (15, 0),
TERM ENTRY,
@HEX2 RETURNS (CHARACTER (2)),
I;

MSGP = ADDR (MESSAGE);
L = LENGTH (MESSAGE);
IF L > 0 THEN PUT EDIT (' CCMP -> TERM ', L, ', BYTES ',
(2HEX2 MSG (1)) DO I = 1 TO L) (SKIP (2), A (14),
F (5), A (8), 20 (32 (X (1), A (21), SKIP, X (27)));

CALL TF3N;

L = LENGTH (MESSAGE);
IF L = 0 THEN RETURN;
PUT EDIT (' CCMP <- TERM ', L, ', BYTES ',
(2HEX2 MSG (1)) DO I = 1 TO L) (SKIP (2), A (14),
F (5), A (8), 20 (32 (X (1), A (21), SKIP, X (27))):
RETURN;

END LINK:

Figure 43. (continued)
E. The Terminal Routines

These routines simulate the terminal interface, refresh sequencer, and pictorial processor.

```
TERM: PROCEDURE;

DECLARE MESSAGE CHARACTER (516) VARYING EXTERNAL,
    MSGP POINTER,
    MSGB (5) BIT (8) BASED (MSGP),
    MSGH (6) BIT (4) BASFD (MSGP),
    MSGC CHARACTER (516) BASED (MSGP),
    DATAP POINTER,
    DATA (256) FIXED BINARY (15, 0) UNALIGNED BASED (DATAP),
    MFM (0: 511) FIXED BINARY (15, 0) EXTERNAL,
    CSW (8: 15) LABEL INITIAL
        (CSWB, CSW9, CSWA, CSWB, CSWC, CSWD, CSWE, CSWF),
    STOP FIXED BINARY (15, 0) STATIC,
    (COM, CODE, LOC, LEN, L, ZCT) FIXED BINARY (15, 0),
    @EX RETURNS (CHARACTER (4)),
    TIMER ENTRY,
    RUN ENTRY (FIXED BINARY (15), FIXED BINARY (15), FIXED BINARY (15)),

L = LENGTH (MESSAGE);
    MSGP = ADDR (MESSAGE);
    IF L = 0 THEN GO TO ERR;
    IF MSGH (1) ~ = '1111'b THEN GO TO TTY;
    UNSPEC (COM) = '000000000000'b || MSGH (2);
    GO TO CSW (COM);

ERR: PUT EDIT ('*INVALID MESSAGE RECEIVED AT TERMINAL*')
    (SKIP (2), X (39), A (36));
    MESSAGE = '
    MSGB (1) = '11110101'b;
    RETURN;

TTY: PUT EDIT ('*TELETYPE MESSAGE:*', MESSAGE)
    (SKIP (2), X (39), A (17), 8 (SKIP, X(43), 80 (A(1))));
    MESSAGE = '*'
    RETURN;
```

Figure 44. The terminal routines
/* TERM (CONTINUED) */

CSW8: UNSPEC (LOC) = MSGB (2) || MSGB (3); /* REQUEST DATA */
UNSPEC (LEN) = '00000000'B || MSGB (4);
DATAP = ADDR (MSGB (2));
PUT EDIT ('DATA REQUESTED: LOC ', aHEX(LOC), aHEX(LEN),
' WORDS') (SKIP(2), X(39), A(21), A(4), X(2), A(4), A(6));
MESSAGE = SUBSTR (MSGC, 1, 2 * LEN + 1);
MSGB (1) = '11111111'B;
DO I = 1 TO LEN;
   DATA (I) = FEM (LOG + 1 - 1);
END;
RETURN;

CSW9: UNSPEC (LOC) = MSGB (2) || MSGB (3); /* LONG LOAD */
UNSPEC (LEN) = '000000000000'B || MSGB (4);
DATAP = ADDR (MSGB (5));
PUT EDIT ('DATA LOAD (L) TO LOC ', aHEX(LOC), aHEX(LEN),
' WORDS') (SKIP(2), X(39), A(21), A(4), X(2), A(4), A(6));
ZCT = 0;
DO I = 1 TO LEN;
   IF DATA (I) = 0
      THEN DO;
         ZCT = ZCT + 1;
         IF ZCT > 15 THEN GO TO ERRZ;
      END;
   ELSE ZCT = 0;
   MEM (LOC + I - 1) = DATA (I);
END;
MESSAGE = '•
RETURN;

CSWA: UNSPEC (LOC) = '0000'R || MSGB (2) || MSGH (5); /* SH LD */
UNSPEC (LEN) = '00000000000000'B || MSGH (6);
DATAP = ADDR (MSGB (4));
PUT EDIT ('DATA LOAD (S) TO LOC ', aHEX(LOC), aHEX(LEN),
' WORDS') (SKIP(2), X(39), A(21), A(4), X(2), A(4), A(6));
DO I = 1 TO LEN;
   MEM (LOC + I - 1) = DATA (I);
END;
MESSAGE = '•
RETURN;

CSWB: UNSPEC (LOC) = '0000'B || MSGB (2) || MSGH (5); /* L & R */
UNSPEC (LEN) = '000000000000'B || MSGH (6);
DATAP = ADDR (MSGB (4));
PUT EDIT ('DATA LOAD (S) TO LOC ', aHEX(LOC), aHEX(LEN),
' WORDS') (SKIP(2), X(39), A(21), A(4), X(2), A(4), A(6));
DO I = 1 TO LEN;
   MEM (LOC + I - 1) = DATA (I);
END;

Figure 44. (continued)
/* TERM (CONTINUED) */

CSWC: LOC = STOP; /* RESUME */
    PUT EDIT ('DISPLAY RESUME LOC INSTR TIME') (SKIP (2), X (39), A (49), A (9));
    GC TO DISPLAY;

CSWD: LOC = 0; /* RESTART */
    PUT EDIT ('DISPLAY RESTART LOC INSTR') (SKIP (2), X (39), A (30));
    CALL TIMER;
    GO TO DISPLAY;

CSWE: UNSPEC (LOC) = MSGB (2) | MSGB (3); /* START */
    PUT EDIT ('DISPLAY START LOC INSTR') (SKIP (2), X (39), A (30));
    CALL TIMER;
    GO TO DISPLAY;

CSWF: PUT EDIT ('HALT RECEIVED') /* HALT */
    (SKIP (2), X (39), A (13));
    MESSAGE = ' '
    DATAP = ADDR (MSGB (2));
    DATA (1) = STOP;
    MSGB (1) = '11110000'B;
    RETURN;

DISPLAY: CALL RUN (LOC, STOP, CODE);
    PUT EDIT ('DISPLAY STOPPED TIME') (SKIP (2), X (39), A (49), A (9));
    MESSAGE = ' '
    DATAP = ADDR (MSGB (2));
    DATA (1) = STOP;
    IF CODE = 1 THEN MSGB (1) = '11110001'B; /* ICT INTRPT */
    IF CODE = 4 THEN MSGB (1) = '11110010'B; /* L PEN INTRPT */
    IF CODE = 8 THEN MSGB (1) = '11110111'B; /* INTRPT */
    IF CODE > 8 THEN MSGB (1) = '11110100'B; /* STOP */
    RETURN;

ERRZ: PUT EDIT ('EXCESSIVE GAP (ZEROS) IN DATA')
    (SKIP (2), X (39), A (29));
    MESSAGE = ' '
    MSGB (1) = '11110110'B;
    RETURN;

END TERM;

Figure 44. (continued)
RUN: PROCEDURE (START, $STOP, #CODE);

DECLARE START FIXED BINARY (15, 0),
$STOP FIXED BINARY (15, 0),
#CODE FIXED BINARY (15, 0),
MEM (0: 511) FIXED BINARY (15, 0) EXTERNAL,
MAT (0: 7, 15) FIXED BINARY (15, 0) EXTERNAL,
MATP POINTER,
MAT $H I BIT (1) BASED (MATP),
MATVP POINTER,
V3 (3) FIXED BINARY (15, 0) BASED (MATVP),
FRP POINTER,
FR FIXED BINARY (15, 0) BASED (FRP),
ICT FIXED BINARY (15, 0) EXTERNAL,
ICTLIM FIXED BINARY (15, 0) EXTERNAL,
LPSTR BIT (1) EXTERNAL,
INSTR BIT (16),
IV (4) BIT (4) DEFINED INSTR POSITION (11),
VB (8) BIT (1) DEFINED INSTR POSITION (5),
IV (4) FIXED BINARY (15, 0) INITIAL (0, 0, 0, 0),
VP POINTER,
V (2) FIXED BINARY (15, 0) BASED (VP),
NP FIXED BINARY (15, 0),
I FIXED BINARY (15, 0),
REVT FIXED BINARY (15, 0),
UPSW (0: 8) LABEL
INITIAL (NP0, CP1, CP2, CP3, CP4, CP5, CP6, CP7, CP8),
FRAME ENTRY (FIXED BINARY (15, 0)),
CAMIN ENTRY (FIXED BINARY (15, 0)),
CAMOFF ENTRY (FIXED BINARY (15, 0)),
PCT ENTRY (BIT (1)),
TIMR ENTRY,
AH-EX RETLNS (CHAR (4)),
A BIT (1),
LOC, I, J, K, M, N;

MATP = ADDR (MAT);
MATVP = ADDR (MAT (0, 11));
FRP = ADDR (MAT (0, 15));
VP = ADDR (IV (2));

LCC = START;
GO TO FETCH;

Figure 44. (continued)
/ * RUN (CONTINUED) */

NEXT: LOC = LOC + L + 1;
FETCH: INSTR = UNSPEC (MEM (LOC));
CTL = CTL + 1;
IF CT > ICTIM THEN GC TO ICINT;

UNSPEC (IVA) = (12) *OP # IV:
CP = IVA (11);
L = IVA (4);
PUT EDIT ('HEX (LOC), 'HEX (MEM (L))) DJ I = LOC TO LOC + L)
(SKIP, X(56), A(4), X(4), A(4), X(4),
3 (A(4), X(2)), 4 (SKIP, X(72), 3 (A(4), X(2)));
IF OP > 8 THEN GO TO INTERRUPT;
GO TO OPSW (OP);

OP0: GO TO NEXT;

OP1: RFVT = LOC + L + 1;
LOC = MFM (LCC + 1);
GO TO FETCH;

OP2: MFM (LOC + 2) = RFVT;
/* PROCEDURE HEAD */
GO TO NEXT;

OP3: IF VR (3) THEN GO TO OP3A;
/* FRAME TESTS */
IF VR (4) THEN FR = MFM (LOC + 1); /* SET FRAME COUNT */
GO TO OP3B;

OP3A: FR = FR + MFM (LOC + 1);
IF ~ VR (4) THEN GO TO OP3B;
IF FR > MEM (LOC + 2) /* TEST FRAME COUNT */
THEN GO TO OP3B; /* HIGH */
LCC = LOC + 3; /* LOW */
PUT EDIT ('FRAME =', FR, 'HEX (FR), 'LOW')
(SKIP, X (81), A(7), F(5), X(12), A(4), X(2), A(3));
GO TO FETCH;

OP3B:
PRINT EDIT ('FRAME =FR, 'HEX (FR))
(SKIP, X (81), A(7), F(5), X(12), A(4));
IF VB (6) /* CAMERA CONTROL */
THEN IF VR (7)
THEN CALL CAMON ((FR));
ELSE CALL CAMOFF ((FR));
ELSE IF VR (7)
THEN CALL FRAME ((FR));
ELSE;
IF VB (8) THEN CALL TIMER;
GO TO NEXT;

Figure 44. (continued)
/* RUN (CONTINUED) */

OP4: DO J = 1 TO 3;
    V3 (J) = MEM (LOC + J);
END;
LPSTR = "0B";
R = ANY (VR & RITS (1, *));
CALL PICT (H);
IF NOT LPSTR THEN GO TO NEXT;

IF ANY (VR & RITS (2, *)) THEN ON;
    VR = RITS (3, *) | VR & ~RITS (4, *);
    UNSPEC (MEM (LOC)) = (NSTR;
    FND;
IF ANY (VR & RITS (5, *)) THEN ON;
    REV = LOC + 4;
    LOC = MAT (3, 5);
    GC TO FETCH;
END;
IF ANY (VR & RITS (6, *)) THEN GO TO INTERRUPT;
GC TO NEXT;

OP5: M = MEM (LOC + 1);
    /* STORE MATRIX */
N = V (1);  
DO K = 1 TO V (2);
    MEM (M + K) = MAT (N, K);
END;
PUT EDIT ("MATS", N, \HEX (MAT (N, 1)) DO I = 1 TO 15)
    (SKIP, X(1)), A(1), F(1), X(4),
    S (3 (A(4), X(2)), SKIP, X(93));
GC TO NEXT;

OP6: M = MEM (LOC + 1);
    /* FETCH MATRIX */
N = V (1);  
DO K = 1 TO V (2);
    MAT (N, K) = MEM (M + K);
END;
PUT EDIT ("MATF", N, \HEX (MAT (N, 1)) DO I = 1 TO 15)
    (SKIP, X(1)), A(1), F(1), X(4),
    S (3 (A(4), X(2)), SKIP, X(93));
    GC TO NEXT;

OP7: N = V (1);
    /* CHANGE MATRIX */
    M = V (2) - 1;
DO J = 1 TO L;
    MAT (N, M + J) = MEM (LOC + J);
END;
PUT EDIT ("MATC", N, \HEX (MAT (N, 1)) DO I = 1 TO 15)
    (SKIP, X(1)), A(1), F(1), X(4),
    S (3 (A(4), X(2)), SKIP, X(93));
    GC TO NEXT;

OP8: IF ANY (VR & RITS (7, *)) THEN GO TO INTERRUPT;
    /* MAIN COMP INTRPT */
ELSF GO TO NEXT;

Figure 44. (continued)
/* UN (CONTINUED) */

/* STOP IN ERROR */

/* INTBRPLLPI: SCODF = IIP; */

/* STOP = LUC + L + 1; */

/* ICINT: SCOKE = 1; */

/* STOP = LUC; */

/* RETURN; */

/* END RUN; */

PICT: PROCEDURE (B);

DECLARE B RIT (1),

MAT (0:7, 15) FIXED BINARY (15, 0) EXTERNAL,
MATP POINTER,
MATS (0:7, 3, 3) FIXED BINARY (15, 10) BASED (MATP),
VCP POINTER,
VC (2) FIXED BINARY (15, 10) BASED (VCP),
MATVP POINTER,
VT (5) FIXED BINARY (15, 10) BASED (MATVP),
TV (3) FIXED BINARY (15, 10) BASED (MATVP),
WP POINTER,
WIND (9, 3) FIXED BINARY (15, 10) BASED (WP),
NWP POINTER,
NW FIXED BINARY (15, 0) BASED (NWP),

PV (3) FIXED BINARY (15, 10),
PVP POINTER,
P2 (2) FIXED BINARY (15, 10) BASED (PVP),

PI (2) FIXED BINARY (15, 0),
PIP POINTER,
PF (2) FIXED BINARY (15, 10) BASED (PIP),

OV (3) FIXED BINARY (15, 10) STATIC,
OW (9) FIXED BINARY (15, 10) STATIC,
OR (9) RIT (1) STATIC,
V (3) FIXED BINARY (15, 10),
TW (9) FIXED BINARY (15, 10),
TR (9) RIT (1),
TS (5) FIXED BINARY (15, 10),
PLOT ENTRY ((2) FIXED BINARY (15)),
PLOTS ENTRY ((2) FIXED BINARY (15)),
I, AO, A, AT;

MATP = ADDR (MAT); 
MATVP = ADDR (MAT (0, 11));
VCP = ADDR (MAT (3, 1));
NWP = ADDR (MAT (3, 3));
WP = ADDR (MAT (3, 4));
PVP = ADDR (PVP); 
PIP = ADDR (PI);

Figure 44. (continued)
Figure 44. (continued)
F. The Display Routines

The display routines are responsible for the simulation of the display processor, camera, and timer in the terminal.

PLOT: PROCEDURE (V):

DECLARE V (2) FIXED BINARY (15, 0),
NF FIXED BINARY (15, 0),
PLTR BIT (1),
LPGN BIT (1) EXTERNAL,
LPC (2) FIXED BINARY (15, 0) EXTERNAL,
APPER FIXED BINARY (15, 0) EXTERNAL,
LPSTR BIT (1) EXTERNAL,
INCRPL BIT (1) STATIC,
FR FIXED BINARY (15, 0) STATIC,
X (100) FLOAT STATIC,
Y (100) FLOAT STATIC,
VO (2) FIXED BINARY (15, 0) STATIC INITIAL (-1, -1),
VN (2) FIXED BINARY (15, 0) STATIC INITIAL (-1, -1),
XR (4) FLOAT STATIC INITIAL (0, 1024, 1024, 0),
YR (4) FLOAT STATIC INITIAL (0, 0, 1024, 1024),
GL CHARACTER (20) STATIC,
CL CHARACTER (20) STATIC,
ML CHARACTER (20) STATIC INITIAL (''),
N FIXED BINARY (15, 0) STATIC INITIAL (0),
CAMERA BIT (1) STATIC,
FF BIT (1) STATIC INITIAL (''),
GRAPH ENTRY (FIXED BINARY, FIXED BINARY, FIXED BINARY,
FLOAT, FLOAT, FLOAT, FLOAT, FLOAT, FLOAT,
CHAR (20), CHAR (20), CHAR (20), CHAR (20)),
GRAPHS ENTRY (FIXED BINARY, FIXED BINARY,
CHAR (20)),
ORIGIN ENTRY (FLOAT, FLOAT, FIXED BINARY),
OPER ENTRY (FIXED BINARY (15, 0)),
LPX (1) FLOAT,
LPY (1) FLOAT,
U (2) FIXED BINARY (15, 0),
AD (2) FIXED BINARY (15, 0),
AX FIXED BINARY (15, 0),
RAT (2),

PUT EDIT ('PLOT', V) (SKIP, X(114), A(4), X(3), F(5), F(5)):
C = V - VO;
IF ALL (0 = 0) THEN RETURN; /* ZERO LENGTH */
N = N + 1;
IF ~1PCN THEN GO TO ADD:

Figure 45. The display routines
/* PLCT (CONTINUEC) */

AD = ARS (D);  /* LIGHTPEN CHECK */
IF ANY (AD <= APPER)  /* TEST FOR AXES */
THEN GO TO AXIS;
Y THEN GO TO AXIS;
RAT = (LPC - VO) / D;
IF (ARS ((RAT (1) - RAT (2))
  * MIN (AD (1), AC (2))) <= APPER)
THEN GO TO RTEST;
ELSE GO TO ADD;

AXIS: IF AD (1) < AD (2)  /* FIND SHORT DIREC */
THEN AX = 1;
ELSE AX = 7;  /* VERTICAL LINE */
IF (LPC (AX) - VO (AX)) > APPFR
THEN GO TO ADD;
AX = 3 - AX;
RAT = (LPC (AX) - VO (AX)) / D (AX);

RTEST: IF -(ANY (RAT<0)  /* CHECK BOUNDS */
  I
  ANY (RAT >1)
  )
THEN ON;
LPSTR = 'I'B;
PUT EDIT ('LIGHTPEN STRIKF') (SKIP, X(8L), A(15));
END;

ADC: VO = V;
X (N) = V (1);
Y (N) = V (2);
IF N = 99 THEN GO TO OUT;
RETURN;

PLOTS: ENTRY (V);
PUT EDIT ('PLCT's', V) (SKIP, X(114), A(5), X(2), F(5), F(5));
IF ALL (V = VC) THEN RETURN;
N = 1; /* START MEW LIST */
GO TO ADD;

START: ENTRY (NF, PLTR);
INCRPL = PLTR;
CAMERA = 'O'B;
CL = 'CAMERA OFF';
CALL ORIGIN (0.0, 0.0, 4);
GO TO NFR;

Figure 45. (continued)
FRAME: ENTRY (NF):
FR: IF N > 1 THEN GO TO OUTF;
    IF FF THEN GO TO LPF;
    IF CAMERA THEN GO TO BLANK;
    FR = NF;
    GL = FR;
    RETURN;
BLANK:
    PUT EDIT ('BLANK FR ', FR) (SKIP, X(81), A(9), F(5));
    CALL GRAPH (1, XR, YR, 3, 7, 3.0, 11.0, 100.0, 0.0, 0.0, 100.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, BL, BL, GL, 'BLANK FRAME'); /* BLANK FRAME */
    FR = NF;
    GL = FR;
    IF INCRPL THEN CALL ORIGIN (5.0, 0.0, 4);
    RETURN;
OUTF: IF FF THEN CALL GRAPH (4, XR, YR, 3, 7, 13.0, 11.0, 100.0, 0.0, 0.0, 100.0, 0.0, 0.0, 0.0, 0.0, RL, BL, GL, CL);
    CALL GRAPH (1, XR, YR, 3, 7, 4, 11.0, 100.0, 0.0, 0.0, 100.0, 0.0, 0.0, 0.0, 0.0, RL, BL, GL, 'NEW FRAME'); /* FOR PRINTER PLTR */
    CALL ORIGIN (15.0, 0.0, 4);
    IF INCRPL THEN GO TO NFR;
    CALL ORIGIN (0.0, 0.0, 0);
    CALL GRAPH (1, XR, YR, 3, 7, 1.0, 11.0, 100.0, 0.0, 0.0, 100.0, 0.0, 0.0, 0.0, 0.0, RL, BL, GL, 'BL'); /* PAGE ADVANCE */
    CALL ORIGIN (6.0, 0.0, 4);
NFR: N = 0;
    PUT EDIT ('NEW FRAME', NF, CL) (SKIP(3), X(81), A(9), F(5), X(4), A(20));
    CALL ORIGIN (NF);
    IF INCRPL THEN GO TO NFR;
    RETURN;

Figure 45. (continued)
/* PLCT (CONTINUED) */

FINISH: ENTRY;
    PUT EDIT ('FINISH FR*', FR, CL)
        (SKIP, X(8I), A(9), F(5), X(4), A(20));
        IF N > 1 THEN GO TO CUTE; /* COMPLETE OLD FRAME*/
        IF ~FF THEN GO TO LPE;
        CALL ORIGIN (0.0, 0.0, 0);
        RETURN;

OUTE: IF FF
      THEN CALL GRAPH (4, XR, YR, 3, 7, 13.0, 11.0, 100.0, 0.0, 100.0, 0.0,
                      BL, BL, GL, CL);
      CALL GRAPHS (N, X, Y, 0, 4, 0);
LPE: IF LPGN THEN DO;
      LPX = LPC (1);
      LPY = LPC (2);
      CALL GRAPHS (1, LPX, LPY, 1, 107, LPL);
      END;
      CALL ORIGIN (15.0, 0.0, 4);
      CALL ORIGIN (0.0, 0.0, 0);
      RETURN;

CAMON: ENTRY (NF);
    IF CAMERA THEN GO TO FRA;
    CAMERA = 'I'R;
    IF N > 1 THEN GO TO OUTON;
    CL = 'CAMERA CN';
    IF ~FF THEN GO TO LPIF;
    FR = NF;
    GL = FR;
    RETURN;

OUTON: IF FF
      THEN CALL GRAPH (4, XR, YR, 3, 7, 13.0, 11.0, 100.0, 0.0, 100.0, 0.0,
                      BL, BL, GL, CL);
      CALL GRAPHS (N, X, Y, 0, 4, 0);
      CL = 'CAMERA CN';
      GO TO LPF;

CAMOFF: ENTRY (NF);
    IF ~CAMERA THEN GO TO FRA;
    CAMERA = 'O'R;
    IF N > 1 THEN GO TO OUTOFF;
    CL = 'CAMERA OFF';
    IF ~FF THEN GO TO LPF;
    GO TO BLANK;

OUTOFF: IF FF
       THEN CALL GRAPH (4, XR, YR, 3, 7, 13.0, 11.0, 100.0, 0.0, 100.0, 0.0,
                      BL, BL, GL, CL);
       CALL GRAPHS (N, X, Y, 0, 4, 0);
       CL = 'CAMERA OFF';
       GO TO LPF;

END PLOT;

Figure 45. (continued)
TIMFR: PROCEDURE;

DECLARE ICTL FIXED BINARY (15, 0),
TIML FIXED BINARY (15, 0),
ICT FIXED BINARY (15, 0) EXTERNAL,
ICTLIM FIXED BINARY (15, 0) EXTERNAL,

TIMSTR PICTURE '9999999999',
HRSTR PICTURE '99' DEFINED TIMSTR POSITION (1),
MINSTR PICTURE '99' DEFINED TIMSTR POSITION (3),
SECSTR PICTURE '99' DEFINED TIMSTR POSITION (5),
TSTOP PICTURE '9999999999' STATIC,
HR FIXED DECIMAL (2),
MIN FIXED DECIMAL (2),
SEC FIXED DECIMAL (3),

TIMSTR = TIME;
PUT EDIT ('TIME = ', TIMSTR, 'ICT = ', ICT)
(SKIP (2), X (81), A (7), A (91), X (2), A (6), F (4));
IF TIMSTR > TSTOP THEN GO TO STOP;
ICT = 0;
RETURN;

STOP: PUT EDIT ('TIMER STOP') (SKIP, X (81), A (101));
SIGNAL CONDITION (ERROR):
CALL FINISH;
STOP;

TIMERST: ENTRY (ICTL, TIML);
ICTLIM = ICTL;
ICT = 0;
TIMSTR = TIMP;
PUT EDIT ('TIME = ', TIMSTR, 'ICILIM = ', ICTLIM)
(SKIP (2), X (81), A (7), A (91), X (2), A (6), F (4));
SEC = SEGSTR + TIML;
MIN = SEC / 60;
SECSTR = SEC - 60 * MIN;
MIN = MIN + MINSTR;
HR = MIN / 60;
MINSTR = MIN - 60 * HR;
HRSTR = HR + HRSTR;
PUT EDIT ('TSTOP= ', TIMSTR) (SKIP, X (81), A (7), A (91));
TSTOP = TIMSTR;
RETURN;

END TIMFR;

Figure 45. (continued)
G. The Operator Routine

The operator routine simulates the operator at the display console, positioning the lightpen according to directions specified on cards.

```
OPER: PROCEDURE (FR):

DECLARE FR FIXED BINARY (15, 0),
   LPON BIT (1) EXTERNAL,
   LPC (2) FIXED BINARY (15, 0) EXTERNAL,
   APPER FIXED BINARY (15, 0) EXTERNAL,
   NFR FIXED BINARY (15, 0) STATIC INITIAL (-32768),
;

IF FR < NFR THEN RETURN;
GET DATA (LPON, LPC, APPER) COPY;
GET LIST (NFR) SKIP;
RETURN;
END OPER;
```

Figure 46. The operator routine

H. Index of Entry Names

The following listing, which includes the dummy parameter list for every entry, is useful for locating the routine for which the name is an entry. Parameters beginning with the character $ are used to return an array or value. No other arguments are altered in any call.

<table>
<thead>
<tr>
<th>Entry Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>@BIT</td>
<td>140</td>
</tr>
<tr>
<td>@BITH</td>
<td>140</td>
</tr>
<tr>
<td>@BITS</td>
<td>140</td>
</tr>
</tbody>
</table>

---

1The parameter list for this procedure exceeds the available space for listing in this index. See the referenced page for details.
@CALL: ENTRY (LOC);  
@CAMERA: PROCEDURE (CP, CI, BRG, SLA, TILT, $CM);  
@CORR: ENTRY (LOC, FIN);  
@CTR: PROCEDURE (XC, YC, NR);  
@DRAW: PROCEDURE (X, Y, Z);  
@DRAWF: ENTRY (X, Y, Z, I, J);  
@DRAWF: ENTRY (PT);  
@DRAWVF: ENTRY (PT, I, J);  
@END: ENTRY;  
@F: ENTRY (X);  
@GOTO: ENTRY (LOC);  
@H: PROCEDURE (CS);  
@HALT: ENTRY;  
@HEX: PROCEDURE (NI) RETURNS (CHAR (4));  
@HEX2: ENTRY (NB) RETURNS (CHARACTER (2));  
@IF: ENTRY (INCR, ITEST);  
@IFF: ENTRY (INCR, ITEST, NF);  
@INCR: ENTRY (INCR);  
@INCRF: ENTRY (INCR, NF);  
@INIT: ENTRY;  
@INTRPT: PROCEDURE (I, J);  
@LIST: PROCEDURE (LSTART, LSTOP);  
@LOAD: PROCEDURE (LOC, FIN);  
@MATCH: ENTRY (MN, NR, NF);
@MATF: PROCEDURE (MN, LOC);
@MATFN: ENTRY (MN, NR, LOC);
@MATRIX: PROCEDURE (CM);
@MATS: ENTRY (MN, LOC);
@MATSN: ENTRY (MN, NR, LOC);
@MMPY: PROCEDURE (A, B, $C);
@N: ENTRY (NR);
@OBL: ENTRY (ANG, RSF, CM, SF);
@ORTHO: ENTRY (CM, SF);
@PERSP: ENTRY (CM, FL);
@PROC: PROCEDURE;
@PROJ: PROCEDURE (ANG, PT, PREV, $NEW);
@RESTRT: ENTRY;
@RESUME: ENTRY;
@RETURN: ENTRY;
@REQ: ENTRY (IST, IFIN);
@ROT: PROCEDURE (AX, AN, $MAT);
@SEND: PROCEDURE (IST, IFIN);
@SENCS: ENTRY (IST, IFIN);
@SENCSR: ENTRY (IST, IFIN);
@SET: ENTRY (ISET);
@SETF: ENTRY (ISET, NF);
@SPACE: ENTRY (N);
@START: ENTRY (LOC);
@STOP: ENTRY;
@STORE: PROCEDURE (N);
@TF: PROCEDURE (NF);
@TRANS: PROCEDURE (B, PP, PI, GP, GI, S, $V);
@TRAV: ENTRY (X, Y, Z);
@TRAVV: ENTRY (PT);
@WIND: ENTRY (NR, ANG, DIST);
@WINDI: PROCEDURE;
@WINDS: ENTRY;
CAMOFF: ENTRY (NF);
CAMON: ENTRY (NF);
FINISH: ENTRY;
FRAME: ENTRY (NF);
I@HEX: ENTRY (HI) RETURNS (FIXED BINARY (15));
LINK: PROCEDURE;
OPER: PROCEDURE (FR);
PICT: PROCEDURE (B);
PLOT: PROCEDURE (V);
PLOTS: ENTRY (V);
RUN: PROCEDURE (START, $STOP, $CODE);
START: ENTRY (NF, PLTR);
TERM: PROCEDURE;
TIMER: PROCEDURE;
TIMERST: ENTRY (ICTL, TIML);