1989

Three-dimensional data input using Sutherland's two-dimensional tablet strategies plus enhanced user feedback

Myra Bennett Pelz

Rochester Institute of Technology
School of Computer Science and Technology

Three-Dimensional Data Input Using Sutherland's Two-Dimensional Tablet Strategies Plus Enhanced User Feedback

by

Myra Bennett Pelz

A thesis submitted in partial fulfillment of the requirements for the degree of

Masters of Science

Approved by:

Dr. John Schott

Dr. Andrew Kitchen

Dr. Peter Anderson

August 30, 1989

Title of Thesis: Three-Dimensional Data Input Using Sutherland's Two-Dimensional Tablet Strategies Plus Enhanced User Feedback

I _____ hereby (grant/deny) permission to the Wallace Memorial Library, of RIT, to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

or

I Myra Bennett Pelz prefer to be contacted each time a request for reproduction is made. I can be reached at the following address:

Hugh Carey Bldg
Room 2324 X6097

Date: 9/15/89

Table of Contents

1. Problem Statement

Although the representation of three-dimensional computer images has become increasingly more sophisticated and complex, the problem of making three-dimensional data available to the computer continues to plague graphics systems designers. In 1974, Ivan Sutherland developed a system that defines a three-dimensional point by digitizing the same point simultaneously in two different two-dimensional views of an object. This project will implement Sutherland's system, with refinements, particularly in the area of improved user feedback.

2. Previous Work

All computer graphics systems that present three-dimensional images must begin with a model, or description, of a three-dimensional object or scene. One of the more difficult problems presented to graphics systems designers is how to make these often complex models available to the computer, so that they can then be manipulated, modified and analyzed by the graphics software. The most common graphical representation used for modeling solid three-dimensional objects is the polyhedron. This is because polyhedra, comprised of polygon faces which are in turn comprised of only points and lines, can with adequate complexity approximate any solid object.

Many systems have been developed for constructing complex models using polyheda. Some have started with physical three-dimensional models and have then used a variety of methods - special lighting, drawn lines, light grids and laser beam patterns, as well as sound and microwaves, projected on to and reflected off of the object's surface. Appropriate detectors or specially placed cameras are then used to capture the images. For example, by illuminating a scene from four different directions, Shirai and Tseyi [1971] extracted four line drawings of an object using extensive image processing and artificial intelligence techniques, and then used two-dimensional logical operations to create a single three-dimensional image from the drawings. Crow [1978] projected reference grids onto an

actor's face, and took photographs simultaneously from three positions using carefully registered cameras. Measurements were then taken from these cameras, the coordinates transformed to three-dimensions, and then recast as polygons. More recently, Altschuler [1982] used a series of laser beam dot patterns which were rapidly projected onto a surface and viewed from several perspectives.

If a physical model is available, a three-dimensional sonic digitizer may be used, as well as digitizers that employ a transducer placed in contact with the model that can trace the model's actual contours.

Other systems, rather than working with a physical model, expect the user to create his three-dimensional model directly. Some of these systems, such as Braid's [1975], are built from more primitive objects that are made available to the user, such as cubes, wedges, and cylinders. Many other systems require the manipulation of polygon vertices, edges and faces. Unfortunately, systems that require models to be built from primitives can be time consuming and laborious when a great deal of complexity is required.

Still other approaches to providing the computer with three-dimensional data use two-dimensional representations of objects, where the objects themselves cannot be manipulated by projected effects or precise viewing. Negroponte [1973] used a system which allows orthographic projections to be drawn free-hand, which the computer "cleans up", using artificial

intelligence methods, to construct three-dimensional representations. Sutherland [1974] used a simpler system of digitizing multiple two-dimensional views of objects, i.e., mechanical drawings or photographs. The two-dimensional coordinates obtained from the multiple views were then transformed into three-dimensions.

More recently researchers have been concerned with the modeling of solid objects from mechanical drawings. Wesley and Markovsky [1981] developed a formal mathematical procedure for generating solid objects consisting of planar faces. Sakurai and Gossard [1983] expanded their work for non-planar objects and Yoshiura [1984] added natural language processing to handle missing information and ambiguities. However, all of these attempts require perfect and consistent orthographic projections.

In the end, the input system chosen is largely dependent on the application; that is, the type and complexity of the model required, and the availability of a physical model that can be manipulated or of mechanical drawings of that object. The Center for Imaging Science at RIT wanted a general purpose system that could generate three-dimensional models using photographs, rather than mechanical drawings, since the latter will not be available to them. They also hoped to model objects of any size, particularly ones too large to be brought into a laboratory to be projected with markings or precisely viewed. To implement such a system, Sutherland's 1974 paper was chosen as a model since it seemed to satisfy these fundamental requirements. It also

requires a minimum amount of hardware, all of which is currently available in the Center.

3. Sutherland's "Three-Dimensional Data Input by Tablet"

The position of a three-dimensional point in Sutherland's system is found by digitizing the point's projection in each of two views of the object simultaneously. Sutherland's system uses a large area digitizing tablet (3ft X 4ft), allowing for the digitizing of several views of an object at one time. Both orthographic projections (mechanical drawings) and perspective views (photographs) of an object can be accommodated.

A matrix, representing the three-dimensional to two-dimensional transformation for each view, is first developed. This is then used to "undo" the viewing tranformation of each two-dimensional data point and convert it to three-dimensions.

The user begins the process by specifying the tablet area occupied by each of the two-dimensional views he is using. He does this by pointing to the lower left and upper right corner of each view area, thereby defining a rectangular space aligned with the tablet axes for each view. This allows the system to automatically determine which view is being used, and allows the user to change from one pair of views to another at will.

The user must then specify what Sutherland refers to as "exemplary" points, which are two-dimensional data points in each view for which the corresponding three-dimensional world coordinates are known. They are called exemplary because they are examples of what the viewing transformation did to certain known points. These are used to specify the viewing transfor-

mation from three to two dimensions.

Using both the known coordinates of several points in three dimensions and their corresponding projections in two-dimensions, a matrix is formed for each view and is then inverted. Using this new matrix to transform all subsequent points accounts for not only the viewing direction of each view, but also for any tilt of the view on the tablet and skew of the tablet axes. It also easily accommodates different scale factors for different views. Since the exemplary points lie at different distances from the origin, the matrix will contain automatic scale factor compensation. Sutherland also points out that using exemplary points to specify the viewing transformation is often much simpler than specifying the viewing parameters directly, and it relieves the user of the need to know the mechanics of transformations.

Exemplary specification may be used for both orghographic and perspective views. However, orthographic views require that only the origin of the three-dimensional coordinate system in each view be specified, along with a point on each of the three-dimensional axes. Perspective projections require that coordinates of at least six points in three dimensions be identified. In either case, the exemplary points chosen must not be coplanar in three dimensions.

Once the transformation matrix has been defined, the user is ready to digitize the points in his two-dimensional views to form a three-dimensional image. Two corresponding points in two

views of the object are digitized simultaneously, using two pens. Sutherland argues that the two pen system makes it easier for the user to think about digitizing the same point in different views, and that it allows the user to keep better track of where he is in the digitizing process. To facilitate this process, Sutherland mounts the pen for one of the views in a stand, while digitizing corresponding points in the second view. This leaves the user free to concentrate on digitizing one view at a time. This system works well for orthographic projections, but not for pairs of photographs in which directions of the axes are not clear.

Each point in the pair of views, along with their respective transformation matrices, defines a ray in three-dimensional space. Together the two define a three-dimensional point. However, because there are two coordinates in each view, for a total of four coordinates rather than three, one coordinate position will be redundant. In orthographic projections, the user can specify that one coordinate in each of the views is to be ignored. In perspective views, however, the four planes may not intersect at a single point, so Sutherland suggests that the redundant degree of freedom be removed by some best fit method.

The three-dimensional points that have now been identified are the vertices that define a collection of polygon faces. These form the "skin" of the three-dimensional model.

Sutherland's system allows the user to define polygons simultaneously with the definition of individal points. He

does this by having the user define a "row" of polygons by digitizing a set of points that lie on each side of the row. The system automatically generates the quadrilaterals between each set of points. The user also must collect his points into either a Visible (V) set or an Invisible (I) set of vertices, depending on whether or not the polygon face is visible from outside the object. All points are entered in a clockwise direction. The points in the I set, i.e., those that are seen through the object, but not from outside it, are automatically reversed and output in a counterclockwise direction. This is typical of how graphics systems distinguish between the "outside" and the "inside" of polygon faces.

4. Theoretical and Conceptual Development

The mathematics that Sutherland uses to determine the three-dimensional coordinates of a point from two views are examined below. As the author points out, by handling the general perspective case, the simpler case of orthographic projections is automatically covered. The strategies that follow also take care of misalignment between views and tablet axes, since misalignment shows up as a simple rotation which is automatically included in the general perspective. Non-perpendicularity of tablet axes is likewise accounted for.

Sutherland determines the viewing transformation for each view, given each view's exemplary specification, by representing the transformation as a matrix. Points are represented in homogeneous coordinates; that is, the three-dimensional coordinate X, Y, Z is represented as the row vector [X, Y, Z, 1], where 1 is the homogeneous term, or as a multiple of that vector, as in [wX, wY, wZ, w], where w is arbitrarily chosen. A point in homogeneous terms may therefore be represented in lower case notation as [x, y, z, w] where X = x/w, Y = y/w, and Z = z/w.

A general perspective projection can be represented in homogeneous coordinates as a 4 X 4 transformation matrix, T, whereby:

$$[x,y,z,w] \quad X \quad \begin{bmatrix} T_{11} & T_{12} & T_{13} & T_{14} \\ T_{21} & T_{22} & T_{23} & T_{24} \\ T_{31} & T_{32} & T_{33} & T_{34} \\ T_{41} & T_{42} & T_{43} & T_{44} \end{bmatrix} = [x',y',z',w'] \qquad (1)$$

and [x, y, z, w] is a point in "object space", [x', y', z', w'] is a point in "image space", and X' = x'/w', Y' = y'/w' and Z' = z'/w'.

Sutherland notes that a perspective projection is actually a projection from one three-dimensional space (here termed "object space") to another three-dimensional space (or "image space"), which computes the perspective depth as well as the positions of objects on the screen. The matrix itself may be arbitrarily scaled, since division will remove the effects of such scaling.

To go from three to two dimensions, we must drop the depth coordinate Z' and the projection becomes:

$$[X, Y, Z, 1] \quad X \quad \begin{bmatrix} T_{11} & T_{12} & T_{13} \\ T_{21} & T_{22} & T_{23} \\ T_{31} & T_{32} & T_{33} \\ T_{41} & T_{42} & T_{43} \end{bmatrix} = [x', y', w'] = w'[U, V, 1] \qquad (2)$$

where [x', y', w'] are homogeneous coordinates in two-dimensional space, and U = x'/w' and V = y'/w'. The tablet measurements we will get from each of our two-dimensional views are actually expressed, not as x', y' and w' but as the ratios U and V as seen above. We have no way of knowing the value of w', and different X, Y and Z inputs may give us different w' values. However, since exemplary points will be used to scale each two-dimensional projection, the value of w' is not needed.

The three equations represented by the matrix expression above are:

$$T_{11}X + T_{21}Y + T_{31}Z + T_{41} = w'U$$
$$T_{12}X + T_{22}Y + T_{32}Z + T_{42} = w'V$$
$$T_{13}x + T_{23}Y + T_{33}Z + T_{43} = w' \quad (3)$$

By substituting the value of w' found in the third equation and grouping terms we get:

$$(T_{11} - T_{13}U)X + (T_{21} - T_{23}U)Y + (T_{31} - T_{33}U)Z + (T_{41} - T_{43}U) = 0$$
$$(T_{12} - T_{13}V)X + (T_{22} - T_{23}V)Y + (T_{32} - T_{33}V)Z + (T_{42} - T_{43}V) = 0 \quad (4)$$

If we know the location of several points in three dimensions (X, Y, Z) and their measured points in the view (U, V), then there are two equations per point in the twelve unknowns T_{ij} which would require six data points for their solution.

We are now ready to find the value of T_{ij}. If we set w to 1 we can rewrite the equations in (3) as:

$$T_{11}X + T_{21}Y + T_{31}Z + T_{41} - U = 0$$
$$T_{12}X + T_{22}Y + T_{32}Z + T_{42} - V = 0$$
$$T_{13}X + T_{23}Y + T_{33}Z + T_{43} - 1 = 0 \quad (5)$$

We can also include U and V in the last equation of (5) by multiplying by U and V, respectively.

$$T_{13}UX + T_{23}UY + T_{33}UZ + T_{43}U - U = 0$$
$$T_{13}VX + T_{23}VY + T_{33}VZ + T_{43}V - V = 0 \quad (6)$$

Since 0 - 0 = 0, we can combine equations (5) and (6) as follows:

$$T_{11}X + T_{21}Y + T_{31}Z + T_{41} - U - (T_{13}UX + T_{23}UY + T_{33}UZ + T_{43}U - U) = 0$$
$$T_{12}X + T_{22}Y + T_{32}Z + T_{42} - V - (T_{13}VX + T_{23}VY + T_{33}VZ + T_{43}V - V) = 0 \quad (7)$$

and then rewrite them as:

$$T_{11}X + T_{21}Y + T_{31}Z + T_{41} - T_{13}UX - T_{23}UY - T_{33}UZ - T_{43}U = 0$$

$$T_{12}X + T_{22}Y + T_{32}Z + T_{42} - T_{13}VX - T_{23}VY - T_{33}VZ - T_{43}V = 0 \quad (8)$$

Equation (8) can then be expressed in matrix form, using the subscripts a-f for each exemplary point as:

$$\begin{bmatrix}
X_a & Y_a & Z_a & 1 & 0 & 0 & 0 & 0 & -U_aX_a & -U_aY_a & -U_aZ_a & -U_a \\
X_b & Y_b & Z_b & 1 & 0 & 0 & 0 & 0 & -U_bX_b & -U_bY_b & -U_bZ_b & -U_b \\
X_c & Y_c & Z_c & 1 & 0 & 0 & 0 & 0 & -U_cX_c & -U_cY_c & -U_cZ_c & -U_c \\
X_d & Y_d & Z_d & 1 & 0 & 0 & 0 & 0 & -U_dX_d & U_dY_d & -U_dZ_d & -U_d \\
X_e & Y_e & Z_e & 1 & 0 & 0 & 0 & 0 & -U_eX_e & -U_eY_e & -U_eZ_e & -U_e \\
X_f & Y_f & Z_f & 1 & 0 & 0 & 0 & 0 & -U_fX_f & -U_fY_f & -U_fZ_f & -U_f \\
0 & 0 & 0 & 0 & X_a & Y_a & Z_a & 1 & -V_aX_a & -V_aY_a & -V_aZ_a & -V_a \\
0 & 0 & 0 & 0 & X_b & Y_b & Z_b & 1 & -V_bX_b & -V_bY_b & -V_bZ_b & -V_b \\
0 & 0 & 0 & 0 & X_c & Y_c & Z_c & 1 & -V_cX_c & -V_cY_c & -V_cZ_c & -V_c \\
0 & 0 & 0 & 0 & X_d & Y_d & Z_d & 1 & -V_dX_d & -V_dY_d & -V_dZ_d & -V_d \\
0 & 0 & 0 & 0 & X_e & Y_e & Z_e & 1 & -V_eX_e & -V_eY_e & -V_eZ_e & -V_e \\
0 & 0 & 0 & 0 & X_f & Y_f & Z_f & 1 & -V_fX_f & -V_fY_f & -V_fZ_f & -V_f
\end{bmatrix}
\times
\begin{bmatrix}
T_{11} \\ T_{21} \\ T_{31} \\ T_{41} \\ T_{12} \\ T_{22} \\ T_{32} \\ T_{42} \\ T_{13} \\ T_{23} \\ T_{33} \\ T_{43}
\end{bmatrix}
= [0] \quad (9)$$

Because the equations of (9) are homogeneous, the solutions for T_{ij} will contain an arbitrary scale factor if any such solutions exist. To solve for T, we may choose T_{43} to be unity, add the last column of the matrix to both sides of the equation, and solve for the remaining eleven T's. To do so, Sutherland

suggests using a mean-squares fit method, where, given:

$$AX = B \tag{10}$$

where matrix A has too many rows for the degrees of freedom in X, a least mean-squared error fit can be found by solving

$$A^TAX = A^TB \tag{11}$$

where A^T is the transpose of A and A^TA is square and the right dimension to suit X. If no solution to X is computable, conditions imposed on equation (11) are redundant and no solution represents the least error condition.

To solve for X in equation (11) we multiply both sides by the inverse of A^TA:

$$(A^TA)^{-1}(A^TA)X = (A^TA)^{-1}(A^TB) \tag{12}$$

and can then solve for X where:

$$X = (A^TA)^{-1}(A^TB) \tag{13}$$

Finally, once we have calculated the tranformation $T_{i,j}$ using the formula above, and we know the position of the point U, V we can take the original equations of (4) and, substituting the terms in parentheses with computable constants, we can rewrite them as:

$$\begin{aligned} a_1X + b_1Y + c_1Z + d_1 &= 0 \\ a_2X + b_2Y + c_2Z + d_2 &= 0 \end{aligned} \tag{14}$$

where each equation represents a plane in three space on which the point must lie, and each pair defines a ray extending from the camera through the point. Given transformations for two views and two-dimensional coordinates U and V for each view, we have four equations in the three unknowns X, Y and Z. Sutherland

then suggests using a least squares fit to solve for the values of X, Y and Z.

We can accomplish this by expressing the four equations in matrix form:

$$\begin{bmatrix} a_{u1} & b_{u1} & c_{u1} \\ a_{v1} & b_{v1} & c_{v1} \\ a_{u2} & b_{u2} & c_{u2} \\ a_{v2} & b_{v2} & c_{v2} \end{bmatrix} \quad X \quad [X, Y, Z] \quad = \quad \begin{bmatrix} d_{u1} \\ d_{v1} \\ d_{u2} \\ d_{v2} \end{bmatrix} \qquad (15)$$

where the subscripts u1 and v1 represent the constants in the two equations in the first view, and u2 and v2 represent the constants in the second view. We can then find a least-squares fit using the same formula as outlined in equations (10) - (13).

5. System Implementation

Introduction

A system, called IVAN, was developed similar to Sutherland's, but using photographs for input instead of orthographic drawings. Certain refinements were also added, particularly in the areas of user-interface and feedback.

IVAN allows two-dimensional points from up to four views of an object to be entered using a digitizing tablet and hand-held stylus. From these points, the system produces a file of points and vertices that represent the object in three-dimensions. The file can in turn be used by a general-purpose graphics package for displaying and manipulating the three-dimensional model that has been created.

The user is carefully guided through the entire process of creating his three-dimensional model - each step must be completed before going on to the next. The user may also end a session at any time, and work-in-progress may be stored in files to be used at later sessions, so that earlier steps need not be repeated.

System Overview

IVAN allows up to four different photographs or orthographic drawings of the three-dimensional object that is to be modeled. These views are first "grabbed" by the video digitizing component of the system, which creates a 512 X 512 X 8 bit representation of each view. These grayscale images are later displayed on a video monitor to provide user feedback.

The four photographs or drawings are then placed on an x-y digitizing tablet. For convenience, one area of the tablet is also set aside for a menu, allowing most commands to be entered without putting down the tablet stylus. All menu options that are currently available are displayed before each option is selected.

Once the views and menu are attached to the tablet, the system requests the tablet positions of each. At least six "exemplary" points must then be defined for each view. These are two-dimensional points that correspond to known three-dimensional world coordinates. They are called "exemplary" because they are examples of what the viewing transformation from three dimensions to two dimensions has done to certain known points in each view. The system uses these exemplary points to construct a viewing transformation, represented as a matrix, for each view.

Points may then be selected from the two-dimensional views, using the digitizing stylus. Each point on the object must be selected from at least two, but as many as four, different views.

Three-dimensional points are then calculated from the position of the point in each pair of views using the transformation matrices. The three-dimensional points form the vertices of three-sided facets which create the "skin" of the object model.

Points can be formed into facets directly, or they can be grouped by the user into rows, leaving it to the program to convert each pair of rows into the triangular facets.

While selecting the two-dimensional points, one of the grayscale images digitized earlier may also be displayed. Any two-dimensional point that is selected from the view currently displayed on the video monitor is recorded and superimposed on the video image. The user can use this feedback to keep track of his progress. Once facets are formed, the points displayed on the screen are replaced by labels that can be used to directly choose three-dimensional points for subsequent facets, thereby bypassing the need to convert the points again from the two-dimensional views.

The final output of the system is a list of three-dimensional points and a set of triangular facets created from those points. These can then be written to a file and in turn used as input to a graphics system that can, using hidden line removal, shading, etc., create a three-dimensional image.

Hardware Requirements

IVAN uses a Scientific Accessories Inc. GP-8 2-d Sonic Digitizer for digitizing the object views. The GP-8 measures the transit time of a sound impulse, generated by a stylus, to calculate the distance from the "X" and "Y" microphones. The microphones are mounted on an L-frame and positioned on a drafting table, and both the microphones and stylus are attached to a control unit. The digitizer has an active area of 60" by 72" and resolution of 0.01 inches. The control unit communicates with its host computer, an IBM AT, via an RS232 serial cable.

The IBM AT has 640K of memory and a Sony 12" video monitor. It is equipped with a keyboard and mouse for input and an Imaging Technologies, Inc. imaging board for grabbing and displaying the grayscale images. A Monochrome vidicon camera with standard video output, equipped with a 25mm lens and mounted on a vertical copy stand, is used to 'grab' the images, and a second Sony 12" video monitor is used for displaying them. An illustration of the setup can be seen in **Figure 5.1.**

Software Requirements

IVAN's files are written in Microsoft C, and must be compiled using the large memory model and the /AL /Ze options. The C language was chosen over FORTRAN primarily because of its enhanced data structures. Microsoft C is the only C compiler that can currently call the FORTRAN Werner Frei IMAGETOOL routines, which are required to control the imaging board for

grabbing, storing and displaying images, and for writing points to the video screen. The Microsoft Mouse Library provides routines for using the mouse to locate the image boundaries.

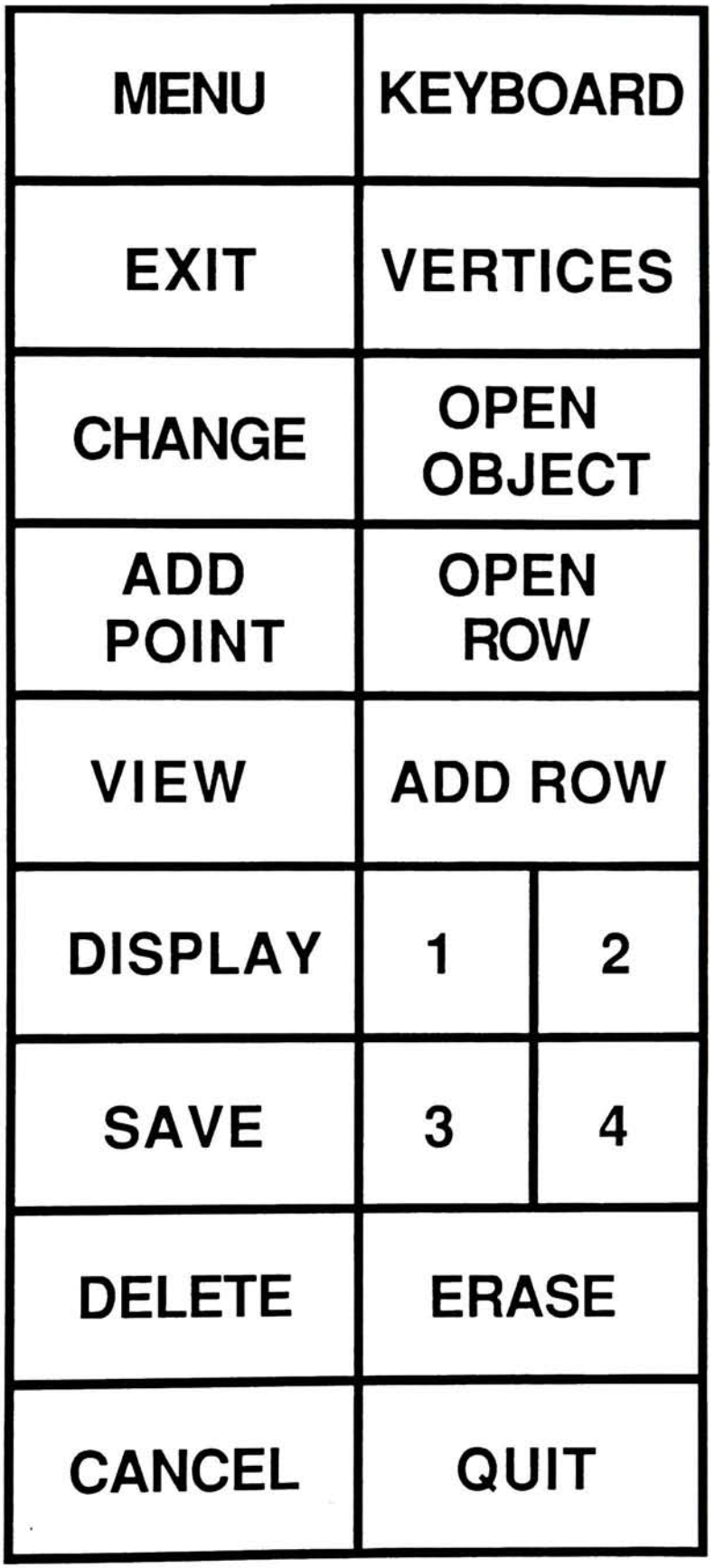
Figure 5.1

System Input

Ivan accepts input from both the keyboard and the digitizing tablet. Digitizer input is necessary for selecting two-dimensional points from the various object views, and for selecting menu options. Input from the digitizer is requested with the prompt "!>", and is entered by clicking the stylus on either the desired two-dimensional point or on the appropriate area of the menu. A copy of the menu is found in **Figure 5.2.**

Keyboard input is requested with the prompt "->". It is implied that any keyboard input, except for <RETURN> itself, will be followed by a <RETURN>.

Incorrect input, from either the tablet or the keyboard, results in an appropriate message and a "beep", sparing the user the need to constantly keep an eye on the screen to see if his work is progressing correctly.



Tablet Menu (3/1/89)

Figure 5.2

Initialization

To start using the system, **ivan** is typed at the keyboard. The system begins by initializing the data structures that are global to all files. A global array of records contains information about each view - its boundary values, exemplary points, transformation matrix, and image boundaries. The array is accessed by the view number. A global integer array, done, holds flags that indicate which steps the user has completed so that the system can allow him to move ahead to the next step. A second integer array, flags, holds the flags that indicate which menu options are currently available, and is accessed by the number given to each menu option, assigned in the **#define** file. A more detailed description of IVAN's files and data structures can be found in the Technical Documentation in Appendix B.

The IMAGETOOL routines **INITIM, CLKSEL, LUTSEL, WRLUT** and **SETSCR** are then called to select, initialize and enable the imaging board and the video screen.

Data Files

IVAN next requests input from files created by the system at a previous session. IVAN creates several different types of files. The parameter data file stores values for the global array of view records, as well as other information which indicates which steps were completed at a previous session so that these need not be repeated for the current session. A sample parameter data file is found in **Figure 5.3**, with a description of the format.

The system output file stores the three-dimensional points and facet vertices that represent the object being modeled. These are grouped into distinct object sets. The first data line in the output file contains the number of object sets that have been created. Then, for each set, the file contains a count of the points and facets for the set, a list of all the three-dimensional points that have been created, and a list of all the facets that have been formed. A sample output file is found in **Figure 5.4**.

The display file stores the point labels that have been written onto the grayscale images. For each view, the file holds the number of points for the view, and then a series of lines that contain the x and y coordinates for the pixel location of the label, and an index into the object list that represents the three-dimensional point for that location. This file is used to re-display all points created at a previous session, and the labels can be chosen to create new facets from these points

during the current session.

Figure 5.5 shows a sample display file, with 6 points for view 1, 4 points for view 2, 2 points for view 3, and no points displayed for view 4. Note that for view 4 a zero is stored to indicate that no points were displayed for that view.

IVAN also creates files for testing the accuracy of exemplary points, which will be discussed in greater detail in the Testing Exemplary Point Accuracy section.

Generally, when the system is instructed to save the current data, the appropriate file name or names is requested, and the appropriate data written to each. Values not yet created for the parameter file are written as 0's. Data file names are restricted to eight characters in length.

```
 3.52   -11.39
13.46    -3.37
14.28   -11.38
24.23    -3.36   view boundaries
 3.69   -20.25
13.55   -12.10
14.48   -20.15
24.36   -12.01
 5.08     0.79    6.05    0.00   -3.47
 2.03     2.75    0.00    0.00    6.97
 5.01     6.79    6.05    8.25    1.69
 5.04     5.45    6.05    6.25   -1.28
 2.81     5.71    1.88    6.13    5.88
 8.11     2.87   12.03    0.00    6.97
 7.13     5.96    9.91    6.30    5.78
 0.00     0.00    0.00    0.00    0.00   view 1
 0.00     0.00    0.00    0.00    0.00   exemplary points-
 0.00     0.00    0.00    0.00    0.00
 0.00     0.00    0.00    0.00    0.00
 0.00     0.00    0.00    0.00    0.00
 0.00     0.00    0.00    0.00    0.00
 0.00     0.00    0.00    0.00    0.00
 0.00     0.00    0.00    0.00    0.00
 1.57     3.52    0.00    0.00    6.97
 5.52     6.40    6.05    8.25    1.69
 5.57     7.15    7.53    8.25    4.38
 4.19     6.88    4.47    8.25    4.38
 6.21     4.68    6.05    6.25   -1.28   view 2
 2.57     5.98    1.88    6.13    5.88   exemplary points-
 6.49     1.21    6.05    0.00   -3.47
 6.49     4.66   12.03    0.00    6.97
 0.00     0.00    0.00    0.00    0.00
 0.00     0.00    0.00    0.00    0.00
 0.00     0.00    0.00    0.00    0.00
 0.00     0.00    0.00    0.00    0.00
 0.00     0.00    0.00    0.00    0.00
 0.00     0.00    0.00    0.00    0.00
 0.00     0.00    0.00    0.00    0.00
 2.29     1.71    6.05    0.00   -3.47
 4.28     6.79    6.05    8.25    1.69   view 3
 5.88     6.97    7.53    8.25    4.38   exemplary points-
 4.67     7.22    4.47    8.25    4.38
 3.12     5.32    6.05    6.25   -1.28
 4.21     6.62    1.88    6.13    5.88
 8.37     2.77   12.03    0.00    6.97
 7.43     5.91    9.91    6.30    5.78
 0.00     0.00    0.00    0.00    0.00
 0.00     0.00    0.00    0.00    0.00
 0.00     0.00    0.00    0.00    0.00
 0.00     0.00    0.00    0.00    0.00
 0.00     0.00    0.00    0.00    0.00
 0.00     0.00    0.00    0.00    0.00
 0.00     0.00    0.00    0.00    0.00
 4.95     1.47    6.05    0.00   -3.47
 2.00     5.64    0.00    0.00    6.97
 4.94     1.18    6.05    8.25    1.69
 5.76     2.42    7.53    8.25    4.38       view 4
 4.08     2.35    4.47    8.25    4.38       exemplary points-
 2.62     3.70    1.88    6.13    5.88
 7.81     5.64   12.03    0.00    6.97
 7.15     3.79    9.91    6.30    5.78
 0.00     0.00    0.00    0.00    0.00
 0.00     0.00    0.00    0.00    0.00
 0.00     0.00    0.00    0.00    0.00
 0.00     0.00    0.00    0.00    0.00
 0.00     0.00    0.00    0.00    0.00
 0.00     0.00    0.00    0.00    0.00
 0.00     0.00    0.00    0.00    0.00

 0.60
-0.02
 0.17
 1.28
 0.00
 0.58   transformation
 0.26   matrix - view 1
 1.59
 0.00
 0.00
 0.03
 0.51
-0.02
-0.14
 2.76
 0.14   transformation
 0.39   matrix - view2
 0.34
 1.53
 0.00
-0.01
 0.01
 0.17
-0.05
 0.38   transformation
 1.95   matrix - view 3
-0.27
 0.30
 0.24
 3.77
-0.02
-0.01
 0.02
 0.38
-0.12
-0.20   transformation
 2.82   matrix - view 4
 0.03
-0.24
 0.23
 2.33
 0.00
-0.02
-0.04

# exemp pts     7      8      8      8
digitized?     1      1      1      1
          437     38
image bounds     441     31
          445     29
          445     34

view boundaries done?   1
current command code    0
```

Figure 5.3

```
  1
 16
 22
 4.6   3.8   2.2
 5.2   3.3   3.0
 5.3   3.1   3.0
 5.4   2.4   2.2
 7.3   4.3   1.5
 7.5   3.9   2.0
 6.8   3.5   2.2
 6.7   3.1   1.7
 4.4   5.6   2.0
 6.2   6.3   1.5
 4.3   5.9  -0.1
 6.1   6.3  -0.9
 6.6   1.6  -0.2
 7.6   1.6  -0.6
 6.9   1.3   2.1
 7.6   1.6   1.7
 5.3   4.8   2.4
 5.5   4.4   2.4
 5.2   4.5   2.4
 5.3   4.2   2.4
 5.5   4.1   2.4
 5.8   4.1   2.5
0  4  1
1  4  5
1  5  2
2  5  6
2  6  3
3  6  7
8  10  9
9  10  11
10  12  11
11  12  13
12  14  13
13  14  15
18  17  16
19  17  18
20  17  19
21  17  20
```

Figure 5.4

```
6
189          204          0
249          230          1
187          246          2
253          266          3
242          185          4
300          218          5
4
228          238          0
287          234          1
228          284          2
295          264          3
2
328          208          4
269          237          5
0
```

Figure 5.5

If the user begins a session with IVAN by reading in a parameter data file that has been created during a previous session, it is assumed that the view boundaries and exemplary points contained in the file still apply; that is, the views have not been moved. If the views have been re-positioned, all subsequent points selected will not be accurate.

If a complete parameter data file has been used to begin the current session, a previously created output file may also be read. Any new points and vertices are then appended to those read in from the file, and subsequent saves will store both previous and new data.

If an output file has been read in, the user is given the option to amend the file by deleting facets that may have been incorrectly created. This is done by supplying the system with the three vertices that comprise each facet to be deleted. The vertices are represented by each point's location in the object list, just as they are in the output file.

Grabbing the Views

The user is now instructed, step by step, in "grabbing" the object views to produce grayscale images. The video camera, mounted to a stand, should be turned on and the lights adjusted for proper illumination of the views.

The system first requests that a name be given to the object that is to be created. This is limited to eight characters. The system uses the object name for reading and writing the appropriate image for the current object and the correct view. If no name is entered, no display feedback is available.

The user is asked to arbitrarily assign each view a number from 1 to 4. The number serves to identify the views and must be used consistently throughout, since it also identifies the view's grayscale image, boundary definitions, exemplary points, etc.

The user is then prompted for a view number, representing one of the four views that the user would like to grab. The user may grab as many of the four views as he likes, or none at all. If the views have already been grabbed and stored at a previous session, or if none are desired, <RETURN> will move the user on to the next step in the process. If a view number is given, the IMAGETOOL routine DIGITZ is called to begin producing continuous video output. The view corresponding to the number entered is then placed under the camera, and adjusted so that all four edges of the view are visible on the video monitor.

<RETURN> causes two perpendicular lines to appear on the

monitor, written by the IMAGETOOL routine DRVEC. The lines cross at the lower left hand corner of the screen and serve as a guide for proper alignment of the photograph. The photograph should be placed carefully, so that the lower left corner is aligned with both the horizontal and vertical lines, since proper alignment is critical for accurate display. Taping the four corners of the photographs to a heavier piece of cardboard makes it easier to manipulate them during this process. <RETURN> grabs the image on the screen with another call to DIGITZ in single frame grab mode.

In order to calculate the tablet-to-screen coordinates so that a point selected on the digitizer can be displayed on the monitor, the system needs the boundary values of the image. The lower left hand corner is known, since the user has aligned the photograph with the perpendicular lines that appear on the screen. Using calls to the cursor control routines CMOUSEL, CURTYP, CURON, and CURSET in the Microsoft Mouse Library, the mouse is then moved until the cursor is on the upper right corner of the view and clicked to locate and store the upper right corner boundary. The image boundaries are then stored in the parameter file along with the view boundaries that have been stored for each corresponding view.

The system then stores the grayscale image in a file with a call to the IMAGETOOL routine WDISK. The name of the file is automatically created by the system from the object name given earlier and the view number. For example, view 1 of an image with the name CART is stored in a file named CART_1, view 2 of

the same image in CART_2, etc.

The system then comes back to request another view number, and the above procedure is repeated for each of the views the user would like to grab for future display. When all views have been grabbed, a <RETURN> lists which of the views have been grabbed.

The system then offers an opportunity to save the image boundaries. If the name of a file is entered at the prompt, the boundaries that were located for each view are saved in the parameter data file named.

Although helpful for user feedback, the images that have been grabbed are not adequate for use in selecting the two-dimensional points needed in creating objects. First of all, the range of visible gray tones on the video image is severely reduced, making the locating of points in either very light or very dark areas impossible. Discrimination of these points is much better on the original photograph. Secondly, the spatial resolution of the video image is less than half that of a 9" X 12" photograph (0.010 inches/pixel vs 0.023 inches/pixel). Finally, since each point must be selected in at least two views, and possibly more, multiple views must be visible at the same time. This would require up to four separate monitors, since it would be prohibitively slow and confusing to move from one view image to another on a single monitor. Given these factors, the original photographs are used for creating three-dimensional points, placed on a two-dimensional digitizing tablet.

Preparing the Tablet

Once the grabbing of the video images in complete, the four image views are affixed to the tablet. The views should be placed toward the left side of the tablet, parallel to the tablet axes, and leaving enough digitizing area to the right for the menu. Next, the menu should be attached to the right of the views, but within the range of the digitizer microphones, and keeping in mind that the photographs and menu cannot overlap. **Figure 5.6** is an illustration of what the arrangement should look like.

Figure 5.6

The system first initializes communication with the digitizer by calling the external Microsoft C routine **_bios_serialcom**. The user is then asked to select the lower left and then the upper right menu boundaries with the digitizing stylus to define the position of the menu. This allows the system to calculate the position of the various menu options. Before proceeding, the system checks to be sure that the menu is properly aligned. If not, the system prints an error message, asks that the menu be adjusted, and then prompts again for the position of the menu's corners. This process is repeated until the menu is properly aligned.

Communication with the digitizer is also accomplished with the **_bios_serialcom** routine. The digitizer sends bytes in the format ±XXXX±YYYY<LF>. **_bios_serialcom** receives data 16 bits at a time. The low order byte contains the data byte received and the high order byte contains error status bits. Once the data is free of error, IVAN strips off the low order bits, collects the x and y data in a character array and converts them to floating point values.

Once the menu boundaries have been accepted, most of the input to the system is from the menu using the digitizing stylus. At each request for input, the integer array **flags** is reset and only those options currently available are set. The available options are then displayed, and only those are accepted as valid. All others result in the message 'Invalid Selection.'

In general, the **SAVE**, **DISPLAY**, **MENU** and **QUIT** options are

always valid. **MENU** lists all current options and what they currently do. **SAVE** permits the user to save the current data in a file. **DISPLAY** displays view boundaries, exemplary points, object points and facets, etc., depending on what step the user is currently engaged in. **QUIT** ends the entire session, giving the user a final opportunity to save all data before terminating. The **EXIT** command is also used throughout the program, to indicate that the user has completed the current option and would like to go on.

Defining View Boundaries

Before the points in the two dimensional views can be selected, boundary values that define a rectangular view area for each view must be defined. The system first prints the current boundaries; either those boundary definitions read from the file of a previous session, or 0's if no boundaries have been defined.

If view boundaries have been read in, the system goes on to the definition of exemplary points. There is no opportunity to change the boundaries once they have been defined, since all subsequent points are dependent on their values.

If no boundaries have been read, the system begins the process of defining view boundaries by prompting for one of the views numbered 1 - 4, corresponding to the numbers 1 to 4 on the tablet menu. After selecting a number with the tablet stylus, the system requests that the user select first the lower left corner and then the upper right corner of each view. The system will repeat the request for view numbers and the corresponding view boundaries until the menu option **EXIT** is selected. Once **EXIT** is chosen, the final boundaries are displayed. The user is then allowed one final chance to change the boundary values. **CHANGE** allows the user to go back and re-define the boundaries, **EXIT** indicates acceptance of the values. Since, as stated earlier, all subsequent data points are dependent on the values of the view boundaries, it is important that these boundaries be correct.

If the boundary definitions are accepted, the system

requests the name of a file for saving the boundaries for use at a later session. If the values are not saved, the boundaries must be redefined at the next session, and all points must be selected again.

Several other menu options are also available during the process of defining view boundaries. They are **DISPLAY**, which displays the current view boundaries for the view given; **SAVE**, which saves the boundaries in a file specified by the user; **MENU**, which displays a list of the menu options available; and **QUIT**, which ends the session.

Defining Exemplary Points

The next step is the definition of exemplary points. Exemplary points are three-dimensional points represented in world coordinates, and their corresponding points on the two-dimensional views. At least six exemplary points are needed for each view, with a maximum of 15 points for each allowed.

The three dimensional component of each exemplary point must be measured or otherwise taken from the object, or from the scene in which the object appears. This entails choosing an arbitrary origin and axes in three-dimensional space, and then determining the three-dimensional coordinates of points on either the object or features in the scene.

Although at least six exemplary points are needed for each view, a point in real-world coordinate space can be shared by all the views in which it appears. Choosing 3-d coordinates that appear in all four views require that a minimum of six real-world points be created. Regardless of the number of points chosen, they must not all be coplanar in all three dimensions.

The process of defining exemplary points involves several levels of menu commands. The top level requests that the user choose to either add or delete points for any view. An EXIT at this level ends exemplary point development, and the system attempts to calculate transformation matrices from the points that have been stored.

Whether or not previously defined exemplary points have been read in, the system begins by allowing the user to add or

delete the current exemplary points for any of the views.

Choosing the **ADD POINT** option on the menu allows the user to continue adding exemplary points until **EXIT** is chosen. The system begins the process of collecting exemplary points by asking for a point in world coordinates, taken from the object or from the scene in which it appears. Each x, y and z value is entered separately, one per line. A two-dimensional point that is the projection of this three-dimensional point on any of the views is then requested. The view itself need not be indicated since the system determines the view using the boundary definitions. Once selected, the coordinates for the 2-d point are converted to be relative to the appropriate view's boundaries and the new point is then stored. After the point is chosen, a list is displayed of all the exemplary points thus far chosen for that view. At least six points are displayed; if fewer than six points have been chosen, the rest are shown as 0's. This is a reminder that six points are required, and makes it easy to see how many more are necessary.

After the first two-dimensional projection of the current point is selected, the prompt is repeated until the user has selected the same point in all the views in which it appears. When all the two-dimensional projections for the current three-dimensional point have been exhausted, **EXIT** is selected. This results in the return of the prompt for a new three-dimensional point, and the process begins again. When the user is finished entering points, **<RETURN>** is entered at the prompt.

DELETE also remains in "delete" mode until **EXIT** is chosen. First the view from which the point is to be deleted must be chosen from the tablet options 1 to 4. Choosing a view causes a list of the exemplary points for that view to be displayed. The position in the list of the point to be deleted (where 1 is the first, or top, position) is then requested. Once the location of the point has been entered, the point is deleted and the list of points is displayed again, reset so that all points below the deleted point in the list have been moved up. Entering **<RETURN>** brings back the delete prompt without deleting a point, and **EXIT** returns to the add or delete option. One more **EXIT** on returning to the add/delete prompt indicates to the system that all exemplary points have been defined. The system then calculates the transformation matrix for each view. If too few exemplary points for a view have been chosen, the system responds with a message, and allows the user to add more points.

In addition to the menu commands above, **DISPLAY** displays the exemplary points for the view given; **SAVE** saves the data in the file named; **MENU** displays a list of the menu options available; and **QUIT** ends the session.

Errors are handled with the following messages:

Invalid selection - choosing an invalid menu selection, or digitizing a two-dimensional point that does not lie within one of the view areas.

Maximum number of exemplary points reached - attempting to create more than 15 exemplary points.

Invalid deletion - attempting to delete an exemplary point that has not been stored.

Insufficient number of exemplary points for view # - creating fewer than six exemplary points for a view.

Point already stored - attempting to store an exemplary point that has already been stored.

No point stored at that location - attempting to delete a point at a location in which no point is stored.

Creating Transformation Matrices

Transformation matrices are created from the exemplary points that have been defined. The mathematics described in the Theoretical and Conceptual Development section are used to perform the calculations. Procedures are first called that create the matrix of coefficients (A) and the column vector for the right hand side of the equation (B). A series of routines then solve the equation $A^TAX = A^TB$ for finding the least squares fit for X. If no inverse of A^TA is found, an error message is printed, and the user may modify the points he has chosen. Routines for allocating pointer storage for matrices and vectors and for finding the inverse of a matrix using the LU decomposition method are from William Press's Numerical Methods in C [1988].

Testing Exemplary Point Accuracy

Two different options are available for testing the accuracy of exemplary points. The first option allows the user to create three-dimensional points using the transformation matrices created for each view, the same way it would be done for creating object points in three-dimensions. However, by asking the system to create points for which the three-dimensional coordinates are already known, the measured and created points can be compared to see how accurately points are being formed. The system assists by allowing the user to label each point that is created, and by automatically calculating the difference between the measured and created points. The exemplary points that have been used to create the transformation matrix for each view can be tested in this way as well.

The system begins testing by asking the user if he would like to test the exemplary points that have been defined. Entering 'y' from the keyboard at the prompt allows access to the first testing option. To use this first option, the measured three-dimensional coordinates for a point are requested from the keyboard. The system uses these values to determine the difference between the measured and created points. The system next requests a character, also entered from the keyboard, that will serve as a label for the point that is being created. The user can then begin selecting two-dimensional points that correspond to the three-dimensional point that has been chosen. The points can be selected from any two or more views, and as

many times as is desired. An explanation of how three-dimensional points are created from view pairs can be found in the section Creating Three-dimensional Points.

When the current point has been tested sufficiently, EXIT allows the user to choose another measured three-dimensional point, and its corresponding label, and then create points for this new measured point. Entering <RETURN> indicates that the user has completed this testing option.

The second testing option allows the user to remove one exemplary point at a time for a particular view, and to then create points as in the first option, but with a transformation matrix that has been formed without the removed point for that view. Choosing this testing option requires that at least seven exemplary points be available for a view to be tested, since six are required to create the matrix.

This option begins by requesting a view number from the menu. The list of exemplary points for the given view is displayed, and the user is requested to choose a point to be removed by indicating its location in the list. A transformation matrix is then formed from the remaining points. <RETURN> indicates that there are no more points to be removed for this view.

The system now follows the same procedure as in testing option one; i.e., requesting the measured coordinates for a three-dimensional point, and then a label to identify a point that is being created. The user can again create three-dimensional points from the view pairs, using transformation matrices

with different points removed, and can compare how accurately different points are created by each matrix.

With this second testing option each view must be tested separately. When one view is completed, entering <RETURN> at the prompt for a location in the exemplary point list results in a prompt for another view. Choosing **EXIT** at this point ends the testing session. If the tested points are to be saved, the SAVE option should be chosen before going on to the next view.

Data on each tested point can be either **DISPLAY**ed or **SAVE**d at any time. The values saved and displayed include the location of the point that has been removed from the view (0 for test option 1), the label that has been assigned to the point that is being tested, the views that were used to create the three-dimensional point, and the difference in x, y and z between the measured point and the created point. An example of the files generated by the first and second testing options appear in **Figure** 5.7 and 5.8, respectively.

Errors are handled with the following messages:

Invalid point to test - attempting to remove a point from the exemplary point list that does not exist.

Must have more than 6 points to test - Attempting to remove a point for a view that has fewer than seven exemplary points.

Removed	Tested	Views				Delta X	Delta Y	Delta Z
0	*	1	1	0	0	-0.01	0.27	0.04
0	x	0	1	1	0	0.01	0.15	0.03
0	+	1	0	0	1	-0.18	-0.10	0.00
0	]	0	0	1	1	-0.22	-0.04	0.12
0	a	1	1	0	0	-0.04	-0.12	0.14
0	a	1	0	1	0	0.08	0.18	0.10
0	a	1	0	0	1	-0.08	-0.26	0.10
0	a	0	1	1	0	-0.20	0.00	0.12
0	a	0	1	0	1	0.79	-0.93	0.25
0	a	0	0	1	1	-0.30	-0.13	0.08

Figure 5.7

View: 1

Removed	Tested	Views				Delta X	Delta Y	Delta Z
1	b	1	1	0	0	-0.07	0.13	0.15
2	b	1	1	0	0	-0.43	0.60	0.03
3	b	1	1	0	0	-0.10	0.22	0.16
4	b	1	1	0	0	0.02	0.26	0.00
5	b	1	1	0	0	0.07	0.10	0.15
6	b	1	1	0	0	0.01	0.22	0.06
7	b	1	1	0	0	0.05	0.15	0.10
1	b	1	0	1	0	0.78	1.46	0.36
2	b	1	0	1	0	-0.13	0.25	0.02
3	b	1	0	1	0	0.67	2.56	-0.32
4	b	1	0	1	0	0.96	1.91	0.25
5	b	1	0	1	0	0.74	1.46	0.24
6	b	1	0	1	0	0.68	1.50	0.21
7	b	1	0	1	0	1.22	2.17	0.29
1	b	1	0	0	1	-0.05	0.00	0.10
2	b	1	0	0	1	0.92	0.22	0.00
3	b	1	0	0	1	-0.33	0.16	0.29
4	b	1	0	0	1	-0.07	0.16	-0.01
5	b	1	0	0	1	-0.01	0.00	0.17
6	b	1	0	0	1	-0.05	0.08	0.05
7	b	1	0	0	1	-0.05	0.05	0.12

Figure 5.8

Creating Three-dimensional Points

The system is now ready to accept two-dimensional points, entered from the digitizing tablet, to create a three-dimensional model. To calculate the coordinates of a three-dimensional point, the same two-dimensional point must be selected from at least two different views, and as many as all four. The two-dimensional points are placed on a stack, along with the view from which each point was chosen. After selecting a single point in its various views, the **EXIT** option is chosen. The two-dimensional stack is then emptied, and a three-dimensional point is calculated from the points that had been stored. Again, the mathematics used to perform the calculations are found in the Theoretical and Conceptual Development section.

If a point is selected from only two views, the transformation matrices for those two views are used to calculate the three-dimensional point. For points chosen on more than two views, a point is created for each pair of views, and then an average for the various points is calculated. Once the calculations are complete, the next set of points can be selected.

The system supplies the following messages when handling errors:

Invalid selection - choosing an invalid menu selection, or digitizing a two-dimensional point that does not lie within one of the view areas.

Point from view already chosen - attempting to select two two-dimensional points on the same view.

Points chosen on fewer than two views. Recent selected point ignored. - trying to EXIT after choosing a point from only one view. The point that was selected is automatically erased.

Displaying User Feedback

The user can keep track of the points he has chosen by displaying them on the image screen. First, a view that has been previously digitized is chosen for display using the VIEW menu option. Once the view appears on the screen, all selected points that are visible on that view are drawn on the screen, superimposed on the view in the proper location. The tablet-to-screen conversion is done by multiplying the x and y coordinates selected on the tablet times the ratio of image-to-tablet coordinates for each of the dimensions, and then adding this offset to the minimum x and y values for each image.

The points are drawn by the IMAGETOOL software routines RDPXL and WRPXL. First, RDPXL reads the values of the pixels that are to be written over. The values are averaged, and the average is stored to be used later in re-writing the pixels for the ERASE command. The point is then written as a white dot surrounded by a black cross-hair so that it will be visible on a variety of fields.

Once facets have been created from the current set of points, the displayed points are replaced by integer labels. The integer values represent the location of each three-dimensional point in the object list that corresponds to the displayed two-dimensional point for that view.

This is implemented by using a global array that holds each three-dimensional point as it is pushed on the stack, and then the position of that point in the display list for each view in

which the point appears. Once the points in the stack are moved over to the object list, the position of each point in the object list is found, and stored back into the display list with the corresponding two-dimensional point for each view.

The integers are written to the screen using the IMAGETOOL routines BINCHAR, which converts an integer to a character string, and CHARST, which writes a character string to the screen. Before the text is written, the routine RDPXL samples a point on the background. If the background is dark, the text is written in white. If it is light, the integer appears in black. The integer values may be used to select three-dimensional points that have already been created, without converting them again from the two-dimensional views. This will be discussed in greater depth below.

The user may decide to display a different view at any time. The new view replaces the old view on the screen, and all the points that had been previously chosen for this current view are automatically superimposed on the new image, as are all subsequent points. This is accomplished by storing all display points on a linked list for each view, with an array that holds a header node for each view's list, accessed by the view number.

Creating Three-Dimensional Objects

IVAN represents each object as a list of unique points that have been created, and a list of three-sided "facets" formed by the points. Table 5.1 below shows an example of points and facets. Each line of output represents the three coordinates x, y and z for a point. The facet vertices are represented by the location of each point in the list of points. For example, in the table we have four points. Point 0 is on the first line, point 1 is on the second line, etc. The two facets are formed by point 0, 2, 1 and point 1, 2, 3.

```
8.2   -0.7   8.4
2.5    0.1   4.3
12.0   5.4   8.9
5.8    5.1   3.9

0 2 1
1 2 3
```

Table 5.1

Object points and facets are grouped into one or more distinct object sets, and are stored on a linked list of object nodes, referred to as the object list. Each object node represents one object set, and includes a MAXEDGES X 3 integer array for storing object points, and a pointer to a linked list of facet nodes for that set. Each node in the facet list includes a three-element integer array, which stores the locations of the points that form an object facet.

Points for different object sets are unrelated to each other, and each set is given a number to distinguish it from other objects sets. Before points can be created an object set must be "opened" by choosing the menu option **OPEN OBJECT**. However, if a set has been opened during a previous session and the output file from that session has been read in, points will automatically be appended to the set most recently opened.

Object points and facets can be formed in two ways. The **VERTICES** option allows the user to create three-sided facets directly. Choosing the **OPEN ROW SET/ADD ROW** option allows the user to organize points into rows. For this option, a row of points is selected. This first row is referred to as the first row in the current "row set". Once this first row has been chosen, any number of contiguous rows can be added, provided that the same number of points are selected for each row, by choosing **ADD ROW** for each additional row. For rows with an unequal number of points, a point may be selected more than once for the row with fewer points. There must be at least two rows in a row set and at least two points in a row.

Each time a three-dimensional point is created, it is placed on a stack, which stores the point's x, y and z coordinate values. If the **VERTICES** option is active, after each set of three points has been created they are automatically copied onto the object list if they have not been previously selected. The three points on the stack are then located in the object list, and their positions stored as facet vertices, in

counter-clockwise order. Each facet formed in this way is unrelated to others, and the points for facets with shared edges must be entered separately.

EXIT allows the user to exit from **VERTICES**. If **EXIT** is chosen without completing all three vertices of a facet, the system responds with an error message and the user may either **DELETE** points or complete the facet.

When using the **OPEN ROW SET** option, each time **ADD ROW** is chosen, the unique points in the two most recently selected rows are copied from the three-dimensional stack onto the object list. The stack points are then formed into facets, again by finding their position on the list. This requires that for three or more rows, the stack must be re-ordered so that the points that made up the second row for the previous set become the points for the first row in the new set.

Facets formed from rows of points are also created in a counter-clockwise direction. The algorithm begins with the first point in the first row, goes to the first point in the second row, then returns to the second point in the top row. The second facet is formed starting from this last point, returning to the first point on the second row, and then moving to the second point on the second row. This process is then repeated, beginning with the second point in the first row, and then on through all of the points that have been stored for the row set. **Figure 5.9** illustrates how facets are formed for a row set of three points.

EXIT will end the current row set, create the last set of facets, and remove the user from the **OPEN ROW/ADD ROW** option.

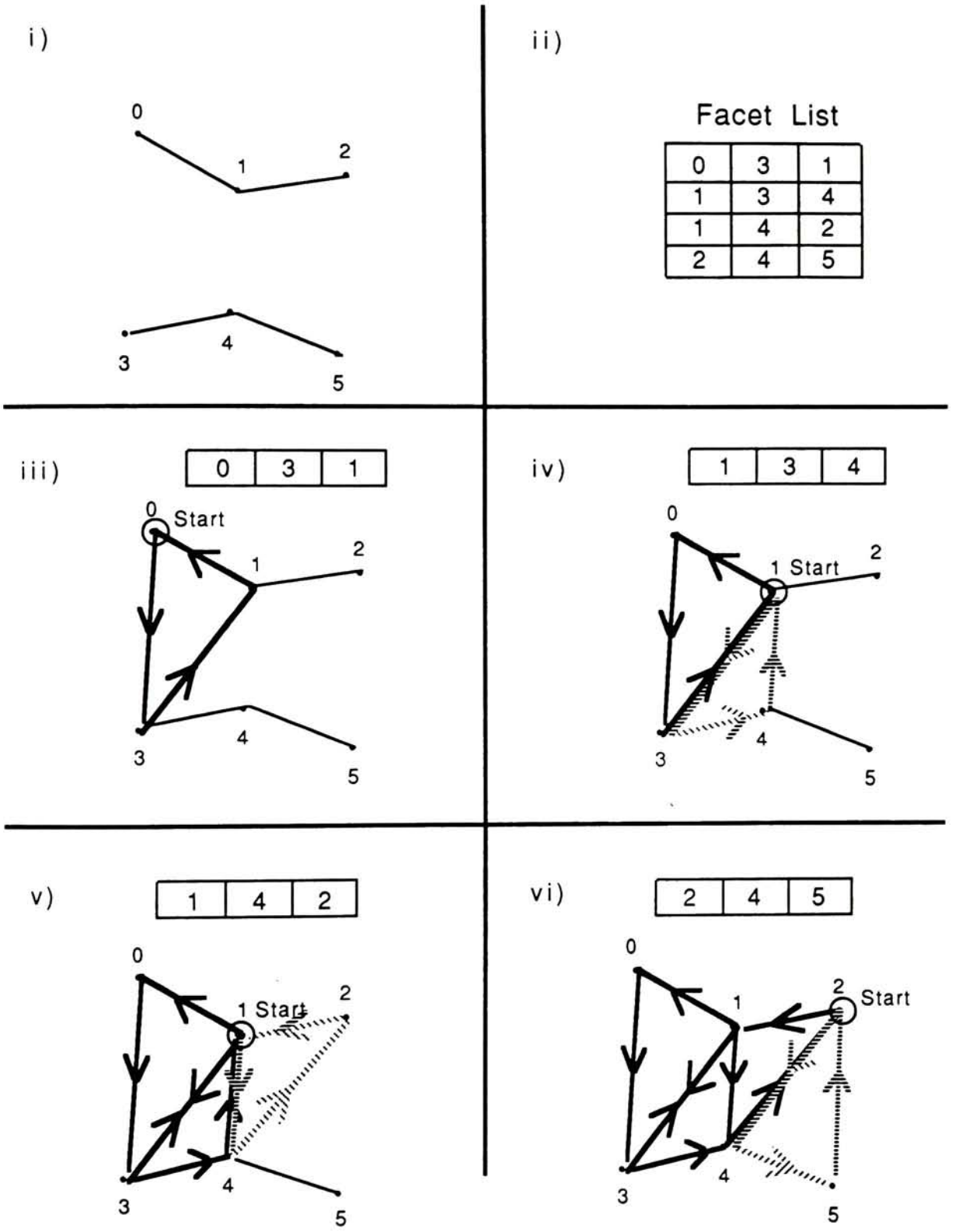

i)
0
1
2
3
4
5
ii)
Facet List
0 3 1
1 3 4
1 4 2
2 4 5
iii)
0 3 1
0 Start
1
2
3
4
5
iv)
1 3 4
0
1 Start
2
3
4
5
v)
1 4 2
0
1 Start
2
3
4
5
vi)
2 4 5
0
1
2 Start
3
4
5

Figure 5.9

The user can choose freely between the **OPEN ROW SET/ADD ROW** options and the **VERTICES** option, as long as the user **EXIT**'s from one before choosing the other. Failure to do so will result in an error message.

Points that have already been converted to three dimensions may be used in subsequent rows and vertices by using the point labels that appear on the display screen. This allows the user to bypass the more time-consuming process of re-generating the points from two dimensions.

To use an already created object point in the current row or vertex, the **KEYBOARD** option is chosen. This permits the user to enter the point labels directly from the keyboard. Points may be entered in this mode until a <RETURN> is entered. This method guarantees that shared points will be recognized and facets created accordingly.

During the process of creating three-dimensional points, the **DISPLAY** command displays the current contents of the two-dimensional and three-dimensional stacks, the current object list, the current command, and the number of points in each row or facet, depending on the current option. The **SAVE** command saves the list of points and facets in the file named, the **MENU** command displays a brief explanation of the available menu options, and **QUIT** ends the session.

The system also responds with the following error messages:

Invalid selection - choosing an invalid menu selection, or digitizing a two-dimensional point that does not lie within one of the view areas.

Row length not equal - attempting to define a row with a length that is not equal to the length of the current row set.

Fewer than two points for row - attempting to define a row set with fewer than two points for the row.

Unfinished facet - attempting to **EXIT** from the **VERTICES** command with only 1 or 2 points created.

Invalid point label - attempting to use a point label that does not appear on the image.

Correcting User Errors

The most recent two-dimensional point chosen in a view may be removed with the **ERASE** command. This will also erase the point from the view image currently displayed. Points are erased from the image by writing back over the point with the average pixel value that was stored for that point when the point was originally written. Since **ERASE** works by popping the two-dimensional stack, only points that have not yet been used to calculate a three-dimensional point can be erased.

The most recent three-dimensional point calculated may be deleted by using the **DELETE** option. Only three-dimensional points that have not yet been formed into facets can be deleted, since **DELETE** works by popping the three-dimensional stack. Deleted points are erased from the view image currently displayed, as well as from all other views that may have been selected to create the deleted point. If the point deleted is the third point selected while in **VERTICES** mode, the facet formed for those three points must also be deleted, and the three-dimensional stack restored to its previous state so that a new third point can be chosen.

The **CANCEL** command is also available. This command will cancel the **OPEN OBJECT**, **OPEN ROW SET**, **ADD ROW** or **VERTICES** command, if any of these was the most recent command given, and will restore all values to what they were before the canceled command was given. This implies that any facets formed with the **ADD ROW** or **EXIT** commands in **OPEN ROW/ADD ROW** mode must also

be deleted and the stack restored to its previous state.

The system also supplies the following error messages:

Invalid selection - choosing an invalid menu selection, or digitizing a two-dimensional point that does not lie within one of the view areas.

No 3-d points to delete - attempting to delete a 3-d point if none currently exists in the 3-d stack.

No 2-d points to erase - attempting to erase a 2-d point if none currently exists in the 2-d stack.

Final Output

The final result of the above procedures is the output file illustrated in **Figure 5.4**. Only object points that have already been created into facets are stored in the file; all two- and three-dimensional points left in the stacks are lost.

The output file is intended to serve as input to a general-purpose graphics package for further display and manipulation of the three-dimensional model that has been created.

6. Results

Choosing appropriate views

The system requires that the user provide up to four views of an object. Choosing those views is critical, both in terms of how much of the object can be represented in three-dimensions, and how accurately the three-dimensional points can be generated by the system.

The four views must include all relevant sides in at least two views. This implies that if the front, back, top and two sides are needed, each view must include information on at least three faces. **Figure 6.1** shows four example views. Note that they include the front in views 1 and 2, the garage side in views 1 and 4, the back in views 3 and 4 and the chimney side in views 2 and 3. Similarly, the front roof section is seen in views 1 and 2 and the back roof section in 3 and 4. The bottom cannot be seen; however, if the top were flat rather than angled the bottom could have been included.

There is, however, an additional note of caution: although a particular face may be visible, a part of that face may be hidden from view by another feature of the object. In the example views in **Figure 6.1**, a window on the garage side is partially hidden. For this feature to be seen in at least two views, and given the fact that all sides are needed, additional views would be necessary.

Note also that four views, while appropriate for photo-

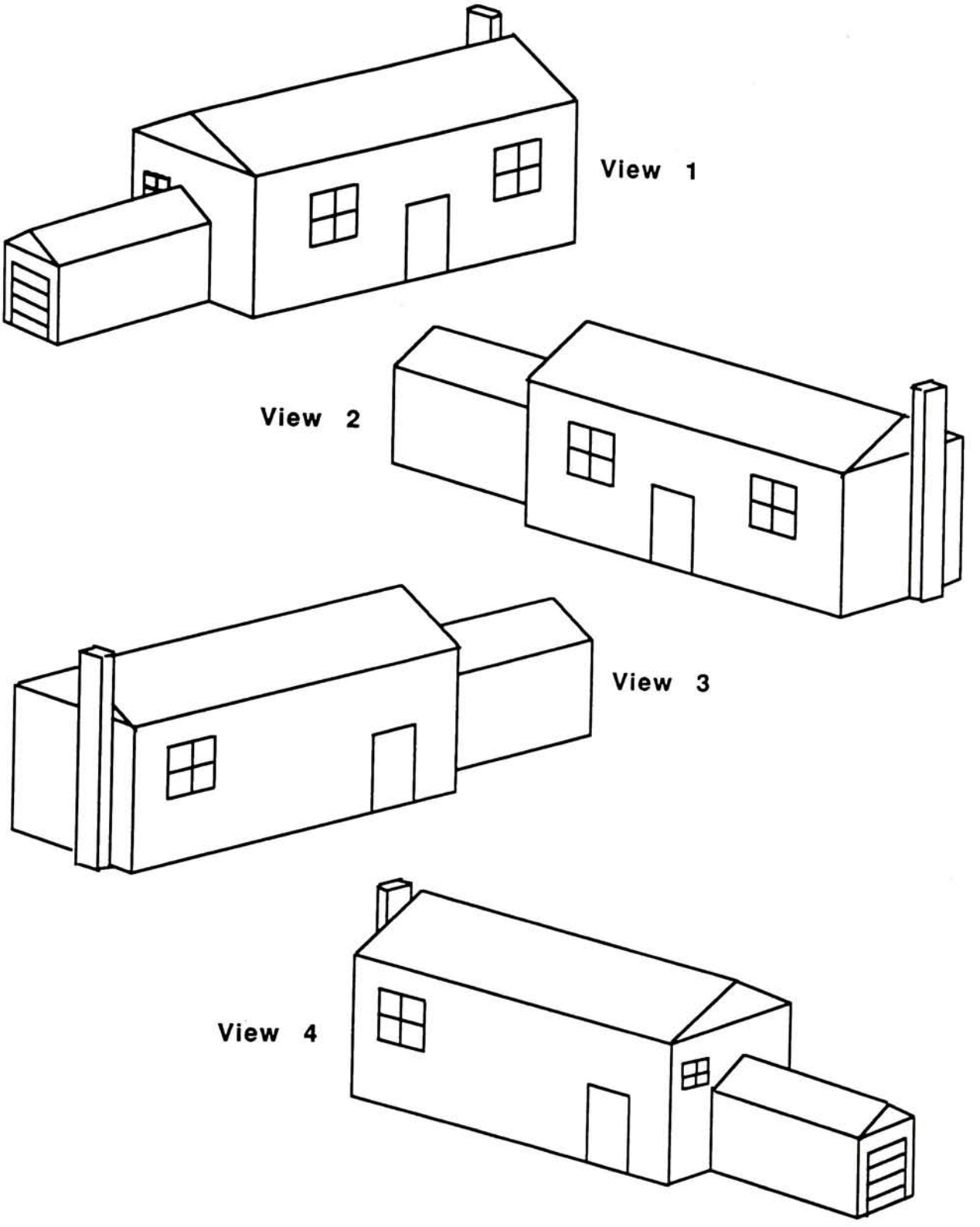


Figure 6.1

graphic input, would limit the use of the system for orthographic drawings. To capture the same number of sides of an object, five drawings would be required; i.e., one view each of the top, front, back and each side.

The Significance of View Pairs

Choosing views pairs when creating three-dimensional points is also critical in how accurately the system can calculate the points. When lines are drawn from two different viewpoints to an object point, the farther the angle formed by those lines is from perpendicular, the greater the error when those two views are used together. **Figure 6.2** illustrates this problem. The closer the two views are to linear (0 degrees or 180 degrees), the more significant any error becomes. Note that the distribution of error in x, y and z will vary with the angle as well.

Figures 6.3a and **6.3b** list the error found for some sample points, using the minimum number of exemplary points. The object used for these examples was a plastic model house, approximately 12" long by 5" wide by 6" high. **Figure 6.3a** illustrates how accurately one point on the house, labeled A, was calculated using two different pairs of views. Point A was calculated far more accurately using views 2 and 3, which are very close to perpendicular, than it was using views 2 and 4, which are almost linear.

The angles formed by two views can also vary from object point to object point. **Figure 6.3b** shows how two points were

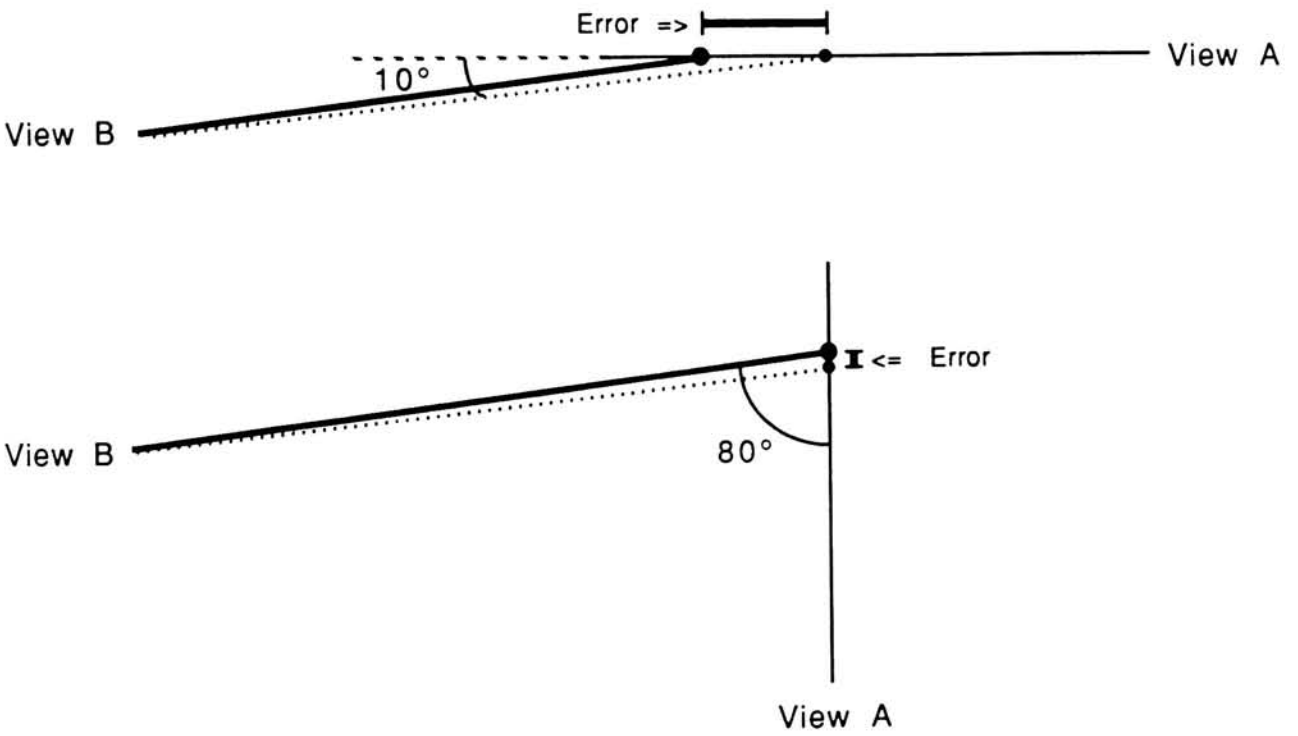
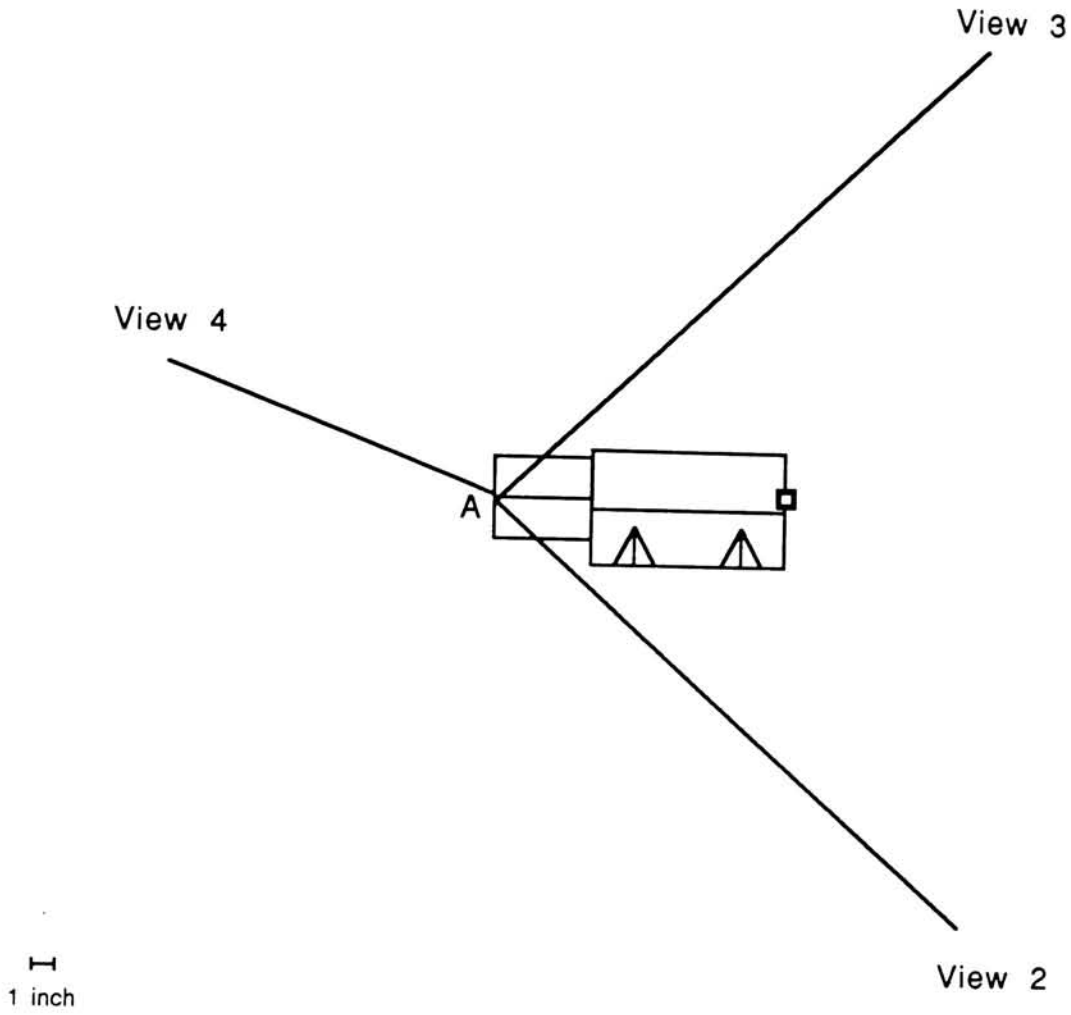


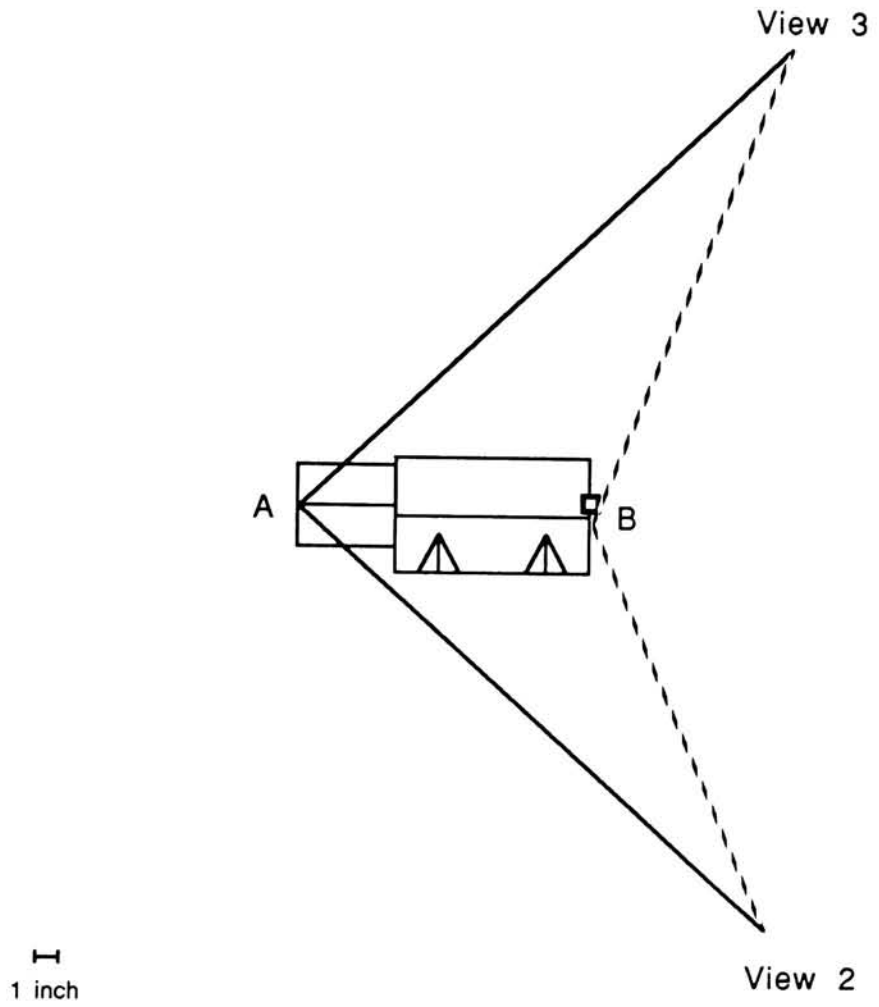
Figure 6.2



Point A

View Pair	Δx	Δy	Δz
Views 2 & 3	-0.08	0.04	-0.12
Views 2 & 4	2.79	-2.84	0.49

Figure 6.3a



Views 2&3

Point	Δ x	Δ y	Δ z
A	-0.08	0.04	-0.12
B	0.34	0.80	0.10

Figure 6.3b

calculated using the same pair of views. Using the view pair 2 and 3, point A was calculated more accurately than point B, since the viewpoints form an angle closer to perpendicular for point A.

This finding also argues that using more than two views when calculating three-dimensional points will not necessarily improve accuracy. The results of each pair of views is averaged when more than one pair is used. Therefore, a single pair of views taken at approximately 90 degrees can produce better results than would be produced by adding a third view that is close to linear with respect to either of the first two.

In his paper, Sutherland notes that, although he uses orthographic views in his examples and for most of his discussion, photographic views can be used as well. Clearly, orthographic views, which by definition are perpendicular to each other, eliminate this source of error. To reduce the error with photographs, the user should avoid choosing view pairs that form angles that are far from 90 degrees. To reduce the variation in angle within a single view, a seen in **Figure 6.3b**, photographs should be taken from a distance with a long focal length lens.

Defining Exemplary Points

The user is required to choose at least six and up to 15 exemplary points for each of the views he has chosen. The exemplary points are used to create the transformation matrix for each view. Matrices for pairs of views are then used to calcu-

late points in three dimensions. How accurately each pair of matrices generates a point is clearly dependent on the accuracy of the exemplary points themselves.

Each exemplary point must be measured carefully on the object, and then the same point is chosen on the view or views in which it appears. Several factors can influence the accuracy of these critical points.

How accurately the points are measured on the object is clearly a factor. Human error is easily introduced: for this project, exemplary points were measured and re-measured several times to verify their validity. Successive measurements yielded differences of up to a tenth of an inch on an object less than a foot square. Another factor is that the object used was a plastic model, whose sides may have been somewhat skewed, and with points that were often difficult to measure accurately.

How accurately the user chooses the position of the point on the photograph is also a factor. The digitizing stylus that was used for this project does not actually digitize the point that is selected. Instead, it digitizes a point below the true point, and then calculates an offset. Any rotation of the stylus will therefore contribute error to the measured points.

Several other factors can introduce error into the system as well. Small defocus errors can make it difficult to choose the exact point that was measured. Lens distortion, which can vary across the view, can also cause errors.

One additional factor related to human error is the possi-

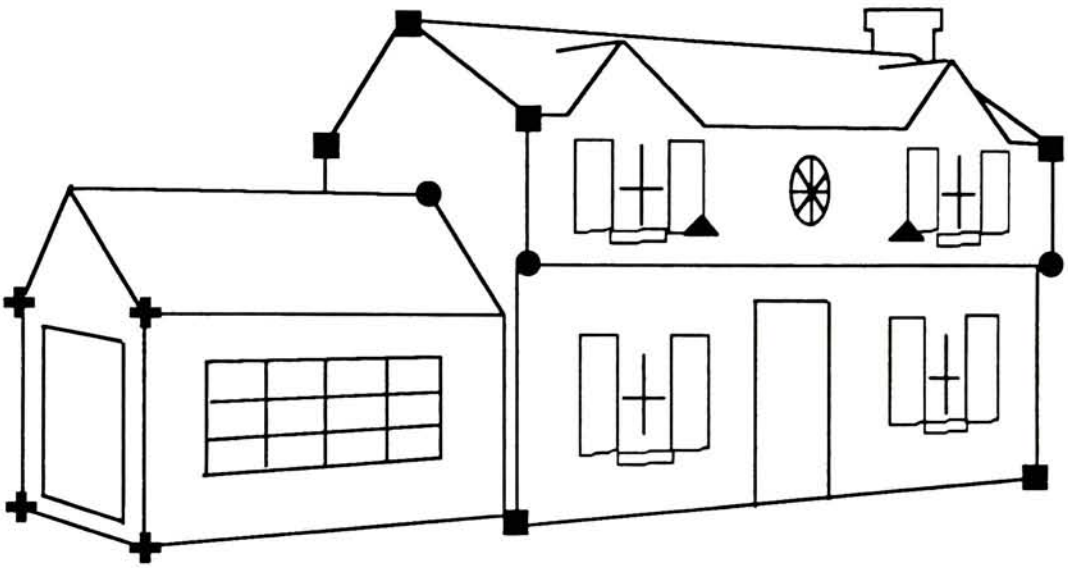
bility that the user may select the wrong point on the print for a measured three-dimensional point. It is very easy to become confused when looking at different views of the same object and the same points. As an aid in this regard, the user might consider actually labeling the exemplary points either on the prints or on the object itself.

How many exemplary points?

It might be inferred that adding more exemplary points will automatically increase accuracy. While this is generally true, it was found that the contribution of each added exemplary point was a function of its own accuracy, and of the information it contributed in a particular dimension.

To test the accuracy of the set of exemplary points chosen for each view, three-dimensional points were created by the system and then compared to their measured values. Points were created using six, nine, eleven and fifteen exemplary points for each view. An example of the exemplary points chosen is illustrated in **Figure 6.4**.

Testing began with six points on each view on the main house. Three additional points were then added. These three points contributed additional information in both the x and y dimension, but not much in z. In general, accuracy was improved in both the x and y dimensions, and z improved as well. This is illustrated in the plots of two sample view pairs shown in **Figure 6.5a and 6.5b**. The plots show the error found in each dimension



■ Exemplary points # 1-6

● Exemplary points # 7-9

▲ Exemplary points # 10-11

✚ Exemplary points # 12-15

Figure 6.4

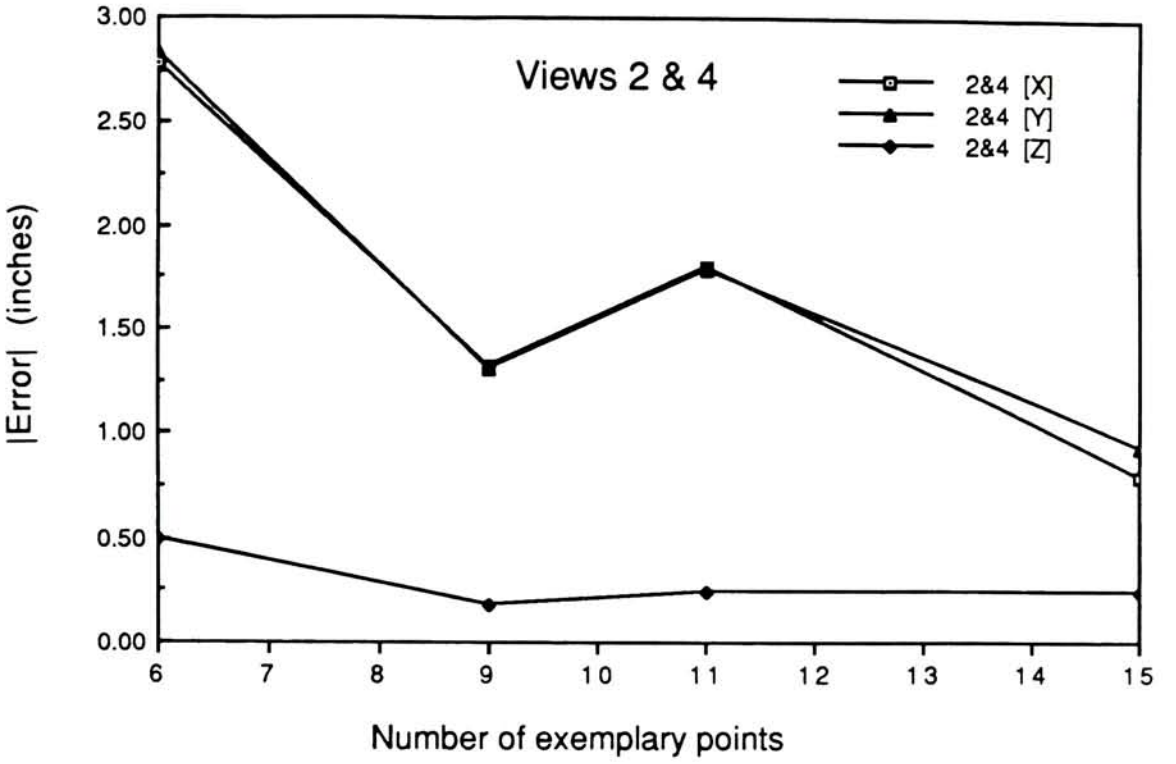


Figure 6.5a

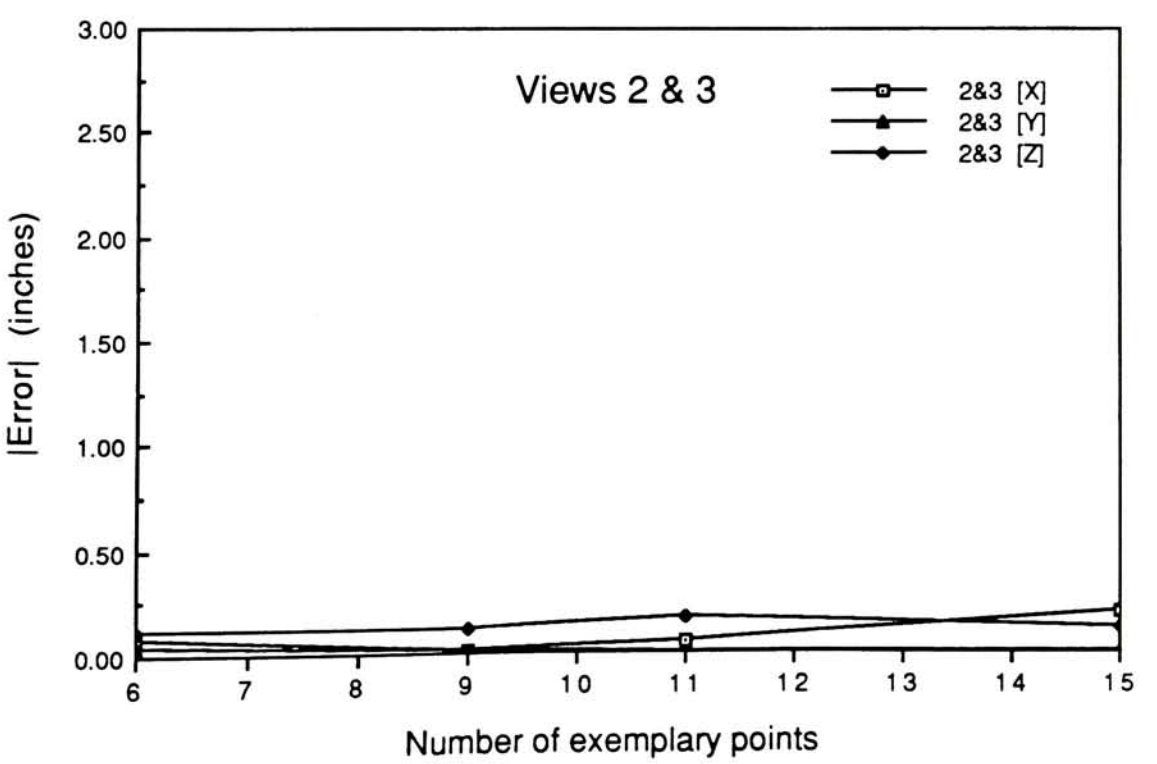


Figure 6.5b

for six, nine, eleven and fifteen exemplary points for each view pair. As noted earlier in **Figure 6.3a**, views 2 and 4 are very close to linear. Correspondingly, large errors can be found in **Figure 6.5a** in both the x and y dimensions for this pair of views. Since views 2 and 3 are closer to perpendicular (referring back to **Figure 6.3b**), the error seen in **Figure 6.5b** is much smaller for these two views.

The next two points that were added contributed little additional y information, and clearly some noise instead, since from 9 to 11 points the accuracy of the y dimension deteriorated. Points 10 and 11 also contributed very little in the x dimension, and some error was introduced as well. This is presumably

because the points are very close together, which increases the effect of measurement error.

Finally, adding the last four points did contribute to the overall accuracy.

Sutherland's only stipulation regarding exemplary points is that they must not be co-planar in all dimensions. However, it is recommended that the points chosen be as widely distributed as possible over all three dimensions.

Checking Exemplary Points

How, then, does the user check the accuracy of his exemplary points? The system provides two different built-in procedures for testing, as described in full in the User Guide found in Appendix C. Using these procedures to calculate three-dimensional points, then comparing the calculated values for the points to their measured values, demonstrates how exemplary points are contributing to the overall accuracy of the calculations.

Creating Object Points

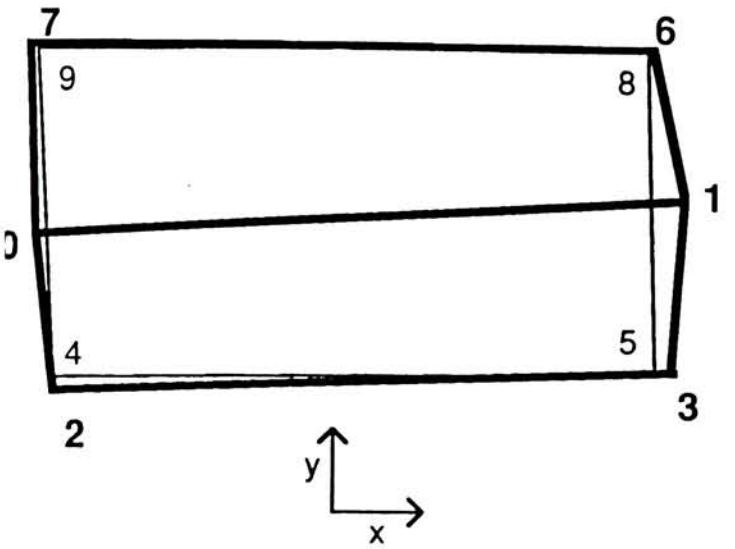
Table 6.1 contains a list of three-dimensional object points created by IVAN, and **Figure 6.6** shows two-dimensional projections of those points. When compared to the measured values for these points, the maximum error found in the best dimension, z, was 0.13 inches. The maximum error in x and y was found to be 0.40 and 0.41 inches, respectively. In both cases, this

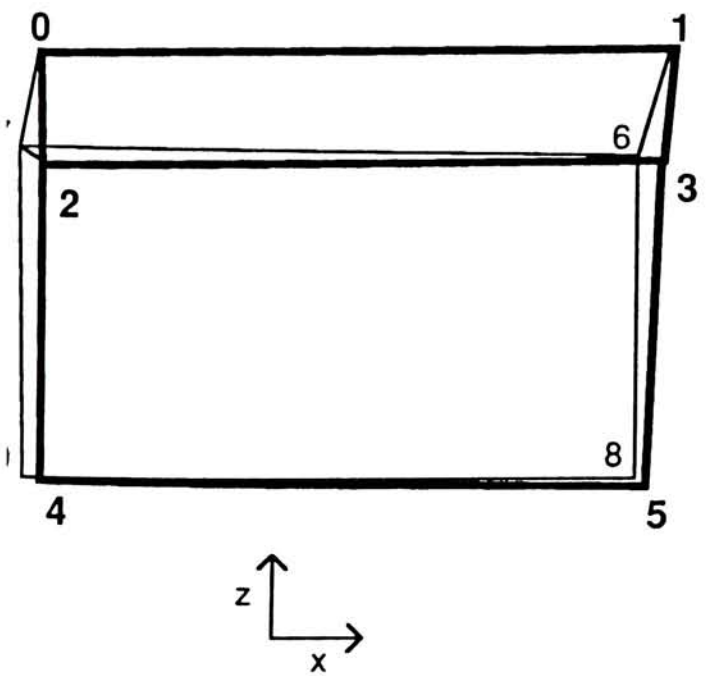
-0.24	1.86	5.37
7.90	2.30	5.47
0.00	-0.16	3.94
7.76	0.10	4.04
0.00	-0.02	0.02
7.56	0.05	-0.04
7.47	4.19	4.05
-0.31	4.22	4.06
7.45	4.14	0.03
-0.21	4.10	0.05

Table 6.1

0	2	1
1	2	3
2	4	3
3	4	5
1	6	0
0	6	7
6	8	7
7	8	9
3	5	6
6	5	8
7	9	2
2	9	4
1	3	6
0	7	2

Table 6.2





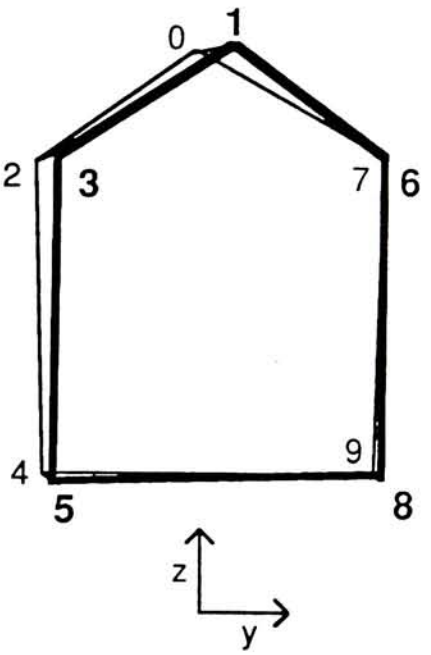


Figure 6.6

was for point 1, which was very difficult to digitize due to lack of detail on the photographs. If we discount this point, the maximum error is 0.22 in x and 0.26 in y, and can be attributed to the errors discussed above. For example, it was found that the actual measurements alone may carry an error of 0.1 inches.

The list of points maintained by the system contains only unique points. Which points are judged as the same and which are considered unique is determined by the precision value defined at the start of the program. Recognizing that the same point has been chosen on different occasions is particularly critical for the correct formation of facets.

The difference in the calculated values for a single point was found to be no greater than .03 inches if the same view pair was used and the points are selected correctly. As seen in **Figure 6.3a**, however, the difference can be quite large if two different view pairs are used. The choice of 0.1 inches as a precision value was chosen as a generous tolerance, but also implies that the user will always use the same view pairs for calculating a particular point.

Displaying Points on the View Image

Some error was also found in writing points to the displayed feedback image. Properly aligning the photographs both when grabbing the image and when placing the views on the tablet will influence how accurately the points appear on the screen. Lens distortion may also be a factor, since copy lenses were not used.

Creating Object facets

Table 6.2 lists the facets that have been created from the object points in **Table 6.1**. **Figure 6.7** is a sketch of the simplified object that was created, with several of the points labeled and an illustration of how the facets were formed. For example, two counter-clockwise facets have been created between the points 0 and 1 and the points 2 and 3. The arrows indicate that the vertices forming the first facet are 0 -> 2 -> 1, and for the second facet are 1 -> 2 -> 3.

In order to form facets correctly, the user must follow several rules when selecting his points. In **OPEN ROW/ADD ROW** mode, all points in a row set must be visible on at least one view. Choosing this view as a "reference" view, the user must enter all rows from top to bottom, and all points within a row from left to right as they appear on his reference view. Proceeding in this way will cause all facets that are visible on the reference view to appear counter-clockwise, and, conversely, all facets that are not visible from the reference view will appear clockwise.

Similarly, in **VERTICES** mode all points that comprise a facet must be visible in a reference view, and the points must be entered in a counter-clockwise direction.

Future Work

Allowing the user more than four views will provide more information about an object, and allow the user to choose the

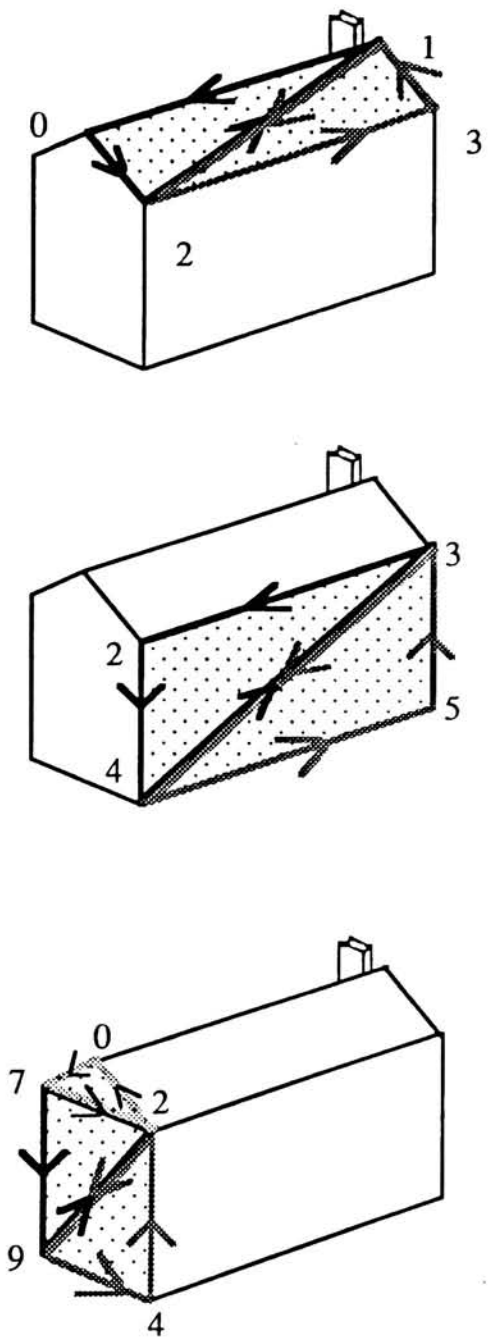


Figure 6.7

best view pair for creating any given point. It would also minimize the problem of parts of the object being hidden from view by other object features. Although this project was intended specifically for photographic input, additional views would also allow greater use of orthographic rather than photographic views.

In addition, some of the current system's prototype code may be optimized to speed computations. However, the time required to do calculations using the present method did not noticably slow the use of the system.

Some improvement in the accuracy of the calculated three-dimensional points is also a goal for the future. A more accurate digitizing stylus could reduce some error. More accurately measuring and selecting the exemplary points would contribute to improving accuracy as well. Constructing a special "exemplary device" to be placed in the scene with the object, such as a frame of metal rules, is one possibility. This could be more easily and accurately measured and selected than points on the object or scene.

7. Bibliography

Altschuler, Martin, et. al., "Laser Electro-Optic System for Rapid 3-D Topographic Mapping of Surfaces," **Optical Engineering**, 20:953, 1981.

Braid, I. C., "The Synthesis of Solids Bounded by Many Faces," **Communications of the ACM**, 18, 1985.

Crow, F. C., "Shaded Computer Graphics in the Entertainment Industry," **Computer**, 11:11, 1978.

Csuri, C. A., "3-D Computer Animation," in Chen, C. H., (ed.), **Pattern Recognition and Artificial Intelligence**, Academic Press, New York: 1976.

Draper, N. R. and Smith, H., **Applied Regression Analysis**, John Wiley and Sons, New York: 1981.

Foley, J. D. and A. Van Dam, **Fundamentals of Interactive Computer Graphics**, Addison-Wesley Publishing, Reading, Mass: 1983.

Harrington, Steven, **Computer Graphics: A Programming Approach**, McGrill-Hill, New York: 1983.

Negroponte, N., "Recent Advances in Sketch Recognition," **Proceedings of the National Computer Conference**, 1973.

Newman, W. M. and R. F. Sproull, **Principles of Interactive Computer Graphics**, McGraw Hill, New York, 1979.

Noakes, P. D., "A New Peripheral for 3-D Computer Input," **IEEE Micro**, 4:26, 1984.

Perucchio, R., et. al., "Interactive Computer Graphic Pre-processing for 3-D Finite Element Analysis," **International Journal of Numerical Methods Engineering**, 18:909, 1982.

Press, W., et. al., **Numerical Recipes in C**, Cambridge University Press, New York: 1988.

Sakurai, H. and Gossard, D. C., "Solid Model Input Through Orthograhic Views", **Computer Graphics**, 17:243-252, 1983.

Shirai, Y. and S. Tseyi, "Extraction of the Line Drawings of Three-Dimensional Objects by Sequential Illumination from Different Directions," **Proceedings ot the International Joint Conference on Artificial Intelligence**, 2nd, 1971.

Sutherland, I. E., "Three-Dimensional Data Input by Tablet," **Proceedings of the IEEE**, 62:453, 1974.

Wesley, M. A. and Markovsky, G., "Fleshing Out Projections," **IBM Journal of Research and Development**, 25:934-954, 1981.

Yoshiura, H., et. al., "Top-down Construction of 3-D Mechanical Object Shapes from Engineering Drawings", **Computer**, 17:32-40, 1984.

APPENDIX A

TECHNICAL DOCUMENTATION

1. Files

C files:

ivan.c
viewbds.c
menu.c
tab.c
exemp.c
trans.c
tabsel.c
calcpt.c
dlsq.c
util.c
dobj.c
dstack.c
wern.c
wrout.c

#define file:

defs.h - contains the #define statements global to all files.

extern file:

structs.h - contains variable definitions for data structures global to all files.

Compiling, Linking and Running:

Ivan is written in Microsoft C, which is the only C compiler that is compatible with the Werner Frie IMAGETOOL software. In order to access that software, the Microsoft C large memory model must be used, and compiling must be done with the /AL and /Ze options.

Both the Werner Frie IMAGETOOL and Microsoft Mouse libraries must be linked before running.

2. Data Structures Global to all Files

Name: viewtype views

Structure:

```
typedef struct
{  double bounds [2] [2];
   double points [MAXPTS] [5];
   double *T;
   int num_points;
   int digitized;
   int imbounds[2];
}
```

Purpose: views holds essential information for all four views of the image. It is defined as a 1 x VIEWROWS array, where VIEWROWS is one more than MAXVIEWS. This allows easy access of the array using the numbers 1 to MAXVIEWS as indices, with the 0th element in the array unused.

Fields:

bounds - a 2x2 array of double to hold the boundaries of view, where bounds[0][0] and bounds[0][1] are the x and y coordinates of the lower left boundary and bounds[1][0] and bounds[1][1] are the x and y coordinates of the upper right boundary.

points - a MAXPTS x VIEWROWS array of double that holds the exemplary points for the view, where each row is a unique exemplary point, and the columns for each point are defined as:

- 0 - 2d x coordinate
- 1 - 2d y coordinate
- 2 - 3d x coordinate
- 3 - 3d y coordinate
- 4 - 3d z coordinate

*T - a pointer to a double vector which holds the transformation matrix for each view.

num_points - an integer that stores the number of exemplary points created for that view.

digitized - an integer flag to indicate whether the view has been grabbed and can be displayed.

imbounds - a 1x2 integer array that holds the location of the views's upper right hand corner on the image screen, where imbounds[0] is the value in x and imbounds[1] is the value in y.

Name: uvstack uv

Structure:

```
typedef struct
{  double su;
   double sv;
   int cview;
}  uvstruct;

typedef struct
{  int uvtop;
   uvstruct uvitem[5];
}
```

Purpose: The stack that stores the 2-d points used to calculate a single 3-d point. Points are stored until the 3-d point is calculated.

Fields:

int uvtop - the top of the stack.

uvstruct uvitem[5] - an array of structs, each of which stores a 2-d point in double su and double sv and the view from the point was selected in int cview.

Name: xyzstack

Structure:

```
typedef struct
{  double sx;
   double sy;
   double sz;
} xyzstruct;

typedef struct
{  int xyztop;
   xyzstruct xyzitem[MAXEDGES];
}
```

Purpose: Stack that stores the 3-d points required to form facets. Points are held in the stack until the facets for those points are formed.

Fields:

int xyztop - the top of the 3-d stack.

xyzstruct xyzitem[MAXEDGES] - an array of structs, each of which stores a 3-d point in double sx, double sy and double sz.

Name: **linktype linkdisp[MAXOPTS]**

Structure:

```
typedef struct
{  double linkpt[3];
   int linkview[5];
 } linktype;
```

Purpose: Array provides a link between the location of each three-dimensional point on the object list and the location of the point on each of the displayed view images.

Fields:

double linkpt[]3 - stores a three-dimensional point.

int linkview[5] - an array that holds the location in the display list for the point on each of the four views, accessed by 1 - 4, with 0 ignored.

Name: **int flags[MAXFLAGS]**

Purpose: Indicates which of the menu options are available at any given time. Each of the elements of flag are accessed by a command number, as defined in **defs.h**. Elements containing a 0 are invalid commands, while elements containing a 1 are valid.

3. File Definitions

ivan.c

Contains **main**, which drives the entire system. Includes routines that initalize the global data structures and the digitizer parameters, and load and store the parameter file data.

Routines:

main
initialize
init_digit
init_views
read_trans_files
save_trans_files

Global variable declarations:

viewtype views[MAXROWS] - stores data for all four views.

int flags[MAXFLAGS] - holds flags that indicate which menu options are currently valid.

int currcomm - a flag that indicates an object currently exists, allowing new points and facets to be appended to the current object.

int done[MAXDONE] - a list of flags that indicate whether view boundaries have been stored (**done[0]**) and which of the views have a sufficient number of exemplary points (**done[1]** - **done[4]**, corresponding to the views 1 - 4.

int savetype - a flag that indicates which type of file is to be saved.

ivan.c

Name: **main**

Purpose: Drives the entire system by calling the high level routines listed below. When QUIT is chosen, main gives the user the opportunity to save the current session before ending the session.

Calling sequence: main()

Input/Output: None

Global variables accessed/modified: Accesses the flag stored in **int done[0]** to determine if view boundaries have not yet been defined.

Major routines called:

initialize
define_views
find_views
get_menu
convert_file

Utility routines called:

setflags
show_bds
interp_point

ivan.c

Name: **initialize**

Purpose: Initializes the imaging software, digitizer and the globally defined data structures. Directs the reading of parameter and output data files.

Calling sequence: initialize()

Input/Output: None

Global variables accessed/modified: None

Major routines called:

init_werner
get_image
init_digit
init_views
read_trans_files
init_obj_list
read_obj_file
init_disp

Utility routines called: None

Name: **init_digit**

Purpose: Calls routines that initialize the digitizer and menu paramters.

Calling sequence: init_digit()

Input/Output: None

Global variables accessed/modified: None

Major routines called:

init_parms
get_range
get_orig
set_orig
set_menu

Utility routines called: None

ivan.c

Name: init_views

Purpose: Initializes the globally defined variables to zero.

Calling sequence: init_views()

Input/Output: None

Global variables accessed/modified: viewtype views[VIEWROWS], int done[MAXDONE] and int currcomm are all set to 0.

Major routines called: None

Utility routines called:

*dvector

Name: read_trans_files

Purpose: Reads values from the parameter data file into the globally defined variables.

Calling sequence: read_trans_files(newfile)

Input/Output: Returns a flag indicating whether a file has been read.

Global variables accessed/modified: Values are read into viewtype views[MAXVIEWS], int done[MAXDONE] and int currcomm.

Major routines called: None

Utility routines called:

getline

ivan.c

Name: **save_trans_files**

Purpose: Writes values from the globally defined data structures to the parameter data file given.

Calling sequence: save_trans_files()

Input/Output: None

Global variables accessed/modified: The values in **viewtype views[MAXVIEWS], int done[MAXDONE]** and **int currcomm** are saved.

Major routines called: None

Utility routines called:

getline

viewbds.c

Accepts, stores and displays the boundary values for each view.

Routines:

define_views
show_bds

Global variable declarations:

done[MAXDONE] - a list of flags that indicate whether view boundaries have been stored (**done[0]**) and which of the views have a sufficient number of exemplary points (**done[1]** - **done[4]**, corresponding to the views 1 - 4.

viewbds.c

Name: define_views

Purpose: Prompts for, accepts and stores the boundary values for each view. The user continues choosing views and boundary values until EXIT is chosen.

Calling sequence: define_views(selection)

Input/Output: int selection - if equal to QUIT, the procedure terminates.

Global variables accessed/modified: The view boundary values are stored in viewtype views[].bounds[][]. When the values have been stored and accepted, the int done[0] flag is set.

Major routines called:

save_trans_files

Utility routines called:

show_bds
setflags
interp_point

Procedure name: show_bds

Purpose: Writes the data values of the view boundaries in views.bounds[][] to the screen.

Calling sequence: show_bds()

Input/Output: None

Global variables accessed/modified: viewytpe views[].bounds[][] is accessed.

Major routines called: None

Utility routines called: None

menu.c

Contains routines related to the digitizer menu for establishing digitizing precision, menu boundaries and the position of each menu option. Also contains a series of routines that interpret the digitized point and return the menu option chosen.

Routines:

get_range
get_orig
set_orig
set_menu
inc_menu
interp_point

getadd	**getcanc**
getchange	**getcoords**
getdel	**getdisplay**
geteras	**getexit**
getmenu	**getnewrow**
getopobj	**getoprow**
getquit	**getsave**
getvert	**getview**
viewone	**viewtwo**
viewthree	**viewfour**
showopts	

Global variable declarations:

double menu[MAXFLAGS] [4] - contains the position on the digitizer of each menu option.

int accuracy - stores precision required for two points to be considered the same point by the system.

int savetype - flag that indicates the type of file to be saved.

menu.c

Name: **get_range**

Purpose: Allows the user to modify the precision value.

Calling sequence: get_range()

Input/Output: None

Global variables accessed/modified: The precision value is stored in **double accuracy.**

Major routines called: None

Utility routines called:

getline
atof

Name: **get_orig**

Purpose: Accepts the boundaries of the menu.

Calling sequence: get_orig(lxmenu, lymenu)

Input/Output: **double lxmenu and double lymenu**

Global variables accessed/modified: None

Major routines called: None

Utility routines called:

read_tablet
fabs

menu.c

Name: **set_orig**

Purpose: Stores the tablet boundaries for the views.

Calling sequence: set_orig(lxmenu)

Input/Output: Accepts **double lxmenu**, the maximum x value for the menu boundaries.

Global variables accessed/modified: **menu[COORDS][0])**, **menu[COORDS][2]** and **menu[COORDS][3]** are set to values defined by the default tablet origin. Since the menu and views cannot overlap, and the menu is placed to the right of the views, **menu[COORDS][1]** is set to the value of **lxmenu** minus a precison value of .1.

Major routines called: None

Utility routines called: None

menu.c

Name: **set_menu**

Purpose: Stores the min and max x and y values for each menu option. Each option is calculated to be relative to the menu boundary value accepted in **lxmenu.**

The x values are based on the column position of the option. There are two columns:

x value for left-hand column -
menu[][0] - **lxmenu** plus a defined LINEWIDTH value.
menu[][1] - min x for left column plus a defined OPWIDTH.

x value for right-hand column -
menu[][0] - max x for left column plus a defined LINEWIDTH.
menu[][1] - min x for right column plus a defined OPWIDTH.

The y values for each option are calculated by the option's vertical position in the menu. For the options on the far bottom of the menu, the min y is equal to **lymenu** plus a defined LINEWIDTH and max y is equal to the min y plus a defined OPHEIGHT. The min y for all other options are calculated by adding a defined LINWIDTH to the previous y value (i.e., the one below it) and the max y is equal to that min y value plus a defined OPHEIGHT.

Calling sequence: set_menu(lxmenu,lymenu)

Input/Output: **lxmenu** and **lymenu** are the max x and max y values of the menu boundaries.

Global variables accessed/modified: Each menu option is stored in **menu[][].**

Major routines called:

inc_menu

Utility routines called: None

menu.c

Name: **inc_menu**

Purpose: Calculates a new min y and max y value for each set of menu options.

Calling sequence: inc_menu(miny, maxy)

Input/Output: Returns **double miny** and **double maxy**, which are the new min y and max y values for each set of menu options.

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

menu.c

Name: interp_point

Purpose: Interprets a point received from the tablet by determining which, if any, menu option has been chosen. If the point is not within any of the option boundaries, or if the selection chosen is currently invalid, an error is returned.

Calling sequence: interp_point(terpx, terpy, terror)

Input/Output: Accepts **double terpx** and **terpy**, a point that has been read in from the tablet. Returns in **int terror** a 0 if the selection is a valid menu option, a 1 if not. Function returns the actual menu selection.

Global variables accessed/modified: Accesses **int flags[]** to determine if the selection chosen is valid and **int savetype** to determine the type of files to be saved for the **SAVE** command.

Major routines called:

getadd	getcanc
getchange	getcoords
getdel	getdisplay
geteras	getexit
getmenu	getnewrow
getopobj	getoprow
getquit	getsave
getvert	getview
viewone	viewtwo
viewthree	viewfour

Utility routines called:

show_opts
read_tablet
save_trans_files
save_obj_files

menu.c

Name: **getverts**

Purpose: Returns the menu option VERTICES if it has been selected; 0 otherwise.

Calling sequence: getvert(x,y)

Input/Output: The tablet point entered is accepted in **double x** and **double y.**

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

Name: **getexit**

Purpose: Returns the menu option EXIT if it has been selected; 0 otherwise.

Calling sequence: getexit(x,y)

Input/Output: The tablet point entered is accepted in **double x** and **double y.**

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

Name: **getsave**

Purpose: Returns the menu option SAVE if it has been selected; 0 otherwise.

Calling sequence: getsave(x,y)

Input/Output: The tablet point entered is accepted in **double x** and **double y.**

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

menu.c

Name: **getchange**

Purpose: Returns the menu option CHANGE if it has been selected; 0 otherwise.

Calling sequence: getchange(x,y)

Input/Output: The tablet point entered is accepted in **double x** and **double y.**

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

Name: **getquit**

Purpose: Returns the menu option QUIT if it has been selected; 0 otherwise.

Calling sequence: getquit(x,y)

Input/Output: The tablet point entered is accepted in **double x** and **double y.**

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

Name: **getadd**

Purpose: Returns the menu option ADD if it has been selected; 0 otherwise.

Calling sequence: getadd(x,y)

Input/Output: The tablet point entered is accepted in **double x** and **double y.**

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

menu.c

Name: **getdisplay**

Purpose: Returns the menu option DISPLAY if it has been selected; 0 otherwise.

Calling sequence: getdisplay(x,y)

Input/Output: The tablet point entered is accepted in **double x** and **double y.**

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

Name: **getmenu**

Purpose: Returns the menu option MENU if it has been selected; 0 otherwise.

Calling sequence: getmenu(x,y)

Input/Output: The tablet point entered is accepted in **double x** and **double y.**

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

Name: **viewone**

Purpose: Returns the menu option VIEW1 if it has been selected; 0 otherwise.

Calling sequence: viewone(x,y)

Input/Output: The tablet point entered is accepted in **double x** and **double y.**

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

menu.c

Name: **viewtwo**

Purpose: Returns the menu option VIEW2 if it has been selected; 0 otherwise.

Calling sequence: viewtwo(x,y)

Input/Output: The tablet point entered is accepted in **double x and double y.**

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

Name: **viewthree**

Purpose: Returns the menu option VIEW3 if it has been selected; 0 otherwise.

Calling sequence: viewthree(x,y)

Input/Output: The tablet point entered is accepted in **double x and double y.**

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

Name: **viewfour**

Purpose: Returns the menu option VIEW4 if it has been selected; 0 otherwise.

Calling sequence: viewfour(x,y)

Input/Output: The tablet point entered is accepted in **double x and double y.**

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

menu.c

Name: **getcoords**

Purpose: Returns the menu option COORDS if it has been selected; 0 otherwise.

Calling sequence: getcoords(x,y)

Input/Output: The tablet point entered is accepted in double x and double y.

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

Name: **getopobj**

Purpose: Returns the menu option OPOBJ if it has been selected; 0 otherwise.

Calling sequence: getopobj(x,y)

Input/Output: The tablet point entered is accepted in double x and double y.

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

Name: **getoprow**

Purpose: Returns the menu option OPROW if it has been selected; 0 otherwise.

Calling sequence: getoprow(x,y)

Input/Output: The tablet point entered is accepted in double x and double y.

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

menu.c

Name: **getnewrow**

Purpose: Returns the menu option NEWROW if it has been selected; 0 otherwise.

Calling sequence: getnewrow(x,y)

Input/Output: The tablet point entered is accepted in **double x** and **double y.**

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

Name: **geteras**

Purpose: Returns the menu option ERASE if it has been selected; 0 otherwise.

Calling sequence: geteras(x,y)

Input/Output: The tablet point entered is accepted in **double x** and **double y.**

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

Name: **getdel**

Purpose: Returns the menu option DELETE if it has been selected; 0 otherwise.

Calling sequence: getdel(x,y)

Input/Output: The tablet point entered is accepted in **double x** and **double y.**

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

menu.c

Name: **getcanc**

Purpose: Returns the menu option CANCEL if it has been selected; 0 otherwise.

Calling sequence: getcanc(x,y)

Input/Output: The tablet point entered is accepted in **double x** and **double y.**

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

Name: **getview**

Purpose: Returns the menu option VIEW if it has been selected; 0 otherwise.

Calling sequence: getview(x,y)

Input/Output: The tablet point entered is accepted in **double x** and **double y.**

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

Name: **show_opts**

Purpose: **Prints all currently valid menu options.**

Calling sequence: show_opts()

Input/Output: None

Global variables accessed/modified: Accesses **int flags[]** to determine the valid menu options.

Major routines called: None

Utility routines called: None

tab.c

Contains procedures for interfacing with the digitizing tablet and for reading 3-dimensional points from the keyboard.

Routines:

init_parms
read_tablet
getnum
read_3d
convert
setflags
getline

Global variable declarations: None

tab.c

Name: **init_parms**

Purpose: Initializes communication with the digitizing tablet. The following parameters are chosen:

serial port:	0
# data bits:	7
# stop bits:	2
parity:	odd
baud rate:	4800

Calling sequence: init_parms()

Input/Output: None

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

External functions called:

_bios_serialcom

tab.c

Name: **read_tablet**

Purpose: Receives data from the digitizing tablet and converts it to 2-d coordinate values. Data is sent 16 bits at a time. The low order byte contains the data byte received and the high order byte contains error status bits.

Once the data is free of error, the high order bits are masked and the low order bits are stored in a character array. The format in which the bytes are received is ±XXXX±YYYY<LF>. The X and Y values are then stripped off one at a time and converted to floating point values.

Calling sequence: read_tablet(x,y)

Input/Output: 2-d coordinates input from the digitizing tablet are returned in **double x** and **y**.

Global variables accessed/modified: None

Major routines called:

getnum

Utility routines called: None

External routines called:

_bios_serialcom

tab.c

Name: **read_3d**

Purpose: Reads in a three-dimensional coordinate from the keyboard, one value on a line.

Calling sequence: read_3d(x, y, z, alldone)

Input/Output: Returns a 3-d coordinate read from the keyboard in **double x, y,** and **z.** Zero is returned in **int alldone** if <RETURN> is entered, signalling the end of input; otherwise 1 is returned.

Global variables accessed/modified: None

Major routines called: None

Utility routines called:

getline
atof
tab.c

Name: **getnum**

Purpose: Converts a string of characters to a signed real number and returns the number.

Calling sequence: getnum(buffer)

Input/Output: A string of characters is accepted in **char buffer[5].**

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

tab.c

Name: **convert**

Purpose: Determines which view the digitizer coordinates are on, and converts the absolute digitizer coordinates to coordinates for the appropriate view.

Calling sequence: convert(x,y)

Input/Output: **double x** and **y** are entered as absolute digitizer coordinates and are returned as offsets for the appropriate view.

Global variables accessed/modified: **viewtype views[].bounds[][]** is accessed.

Major routines called: None

Utility routines called: None

Name: **setflags**

Purpose: Resets all the available menu options to zero.

Calling sequence: setflags()

Input/Output: None

Global variables accessed/modified: The flags for all menu options stored in **int flags[MAXFLAGS]** are set to zero.

Major routines called: None

Utility routines called: None

tab.c

Name: **getline**

Purpose: Reads a line of characters from the keyboard, including the carriage return.

Calling sequence: getline(s, lim)

Input/Output: Accepts the length of the character string in **int lim** and returns the string read in **char s[MAXVAL]**.

Global variables accessed/modified: None

Major routines called: None

Utility routines called:

getchar

exemp.c

Allows the user to define exemplary points and checks to see if there are sufficient points to go on to create the transformation matrices. Calls the routines to calculate the transformation matrices.

Routines:

find_views	findpts
check_views	addpts
trans_pts	deletepts
test_ex1	dup_pts
test_ex2	store_pts
trymat	showpts
testcrds	show_view
calctest	checkpts
disptest	
savetest	
prfile	
stored_T	

Global variable declarations:

int done[MAXDONE] - holds flags that indicates which views have a sufficient number of exemplary points

int accuracy - the precision required for selecting points.

uvstack uv - stack that holds the 2-d points to be used for creating a 3-d point.

```
struct test_struct
   {
    char point_tested;
    int point_removed;
    int viewflags[5];
    double testrow[3];
   }
```

test_struct *test_ptr - holds one row of test data about a given point: a character label; which, if any, exemplary point was removed to create the current point; which views were chosen for the current test; and the resulting calculated point.

test_struct *test_data[MAXOPTS] - a list of all test data points.

int diverror - a flag indicating that an error was found in calculating the transformation matrix.

exemp.c

Name: **find_views**

Purpose: Controls the creation of exemplary points and blocks the next step of processing until a sufficient number of exemplary points have been chosen. Initiates testing of exemplary points.

Calling sequence: find_views(selection)

Input/Output: **int selection** returns the menu selection chosen, and is used to QUIT processing if the user wishes to end the session.

Global variables accessed/modified: **int done[]** is checked to determine if a sufficient number of exemplary points have been created for each view.

Major routines called:

findpts
checkpts
trans_pts
buildt
test_ex1
test_ex2

Utility routines called:

dmatrix
free_dmatrix

exemp.c

Name: **findpts**

Purpose: Controls the processing of menu options related to the creation of exemplary points, i.e., ADD, DISPLAY, DELETE, and MENU.

Calling sequence: findpts(selection,newpts)

Input/Output: **int newpts** returns a flag, indicating whether the ADD or DELETE option has been chosen. **int selection** returns the menu selection chosen, and is used to QUIT processing if the user wishes to end the session.

Global variables accessed/modified: None

Major routines called:

addpts
showpts
deletepts

Utility routines called:

setflags
interp_point

exemp.c

Name: **check_views**

Purpose: Checks each view for a sufficient number of exemplary points to allow further processing. If not enough points, the user may choose to ADD more. A transformation matrix is calculated for each view that has sufficient number of points unless there is already a matrix stored and no new points have been added.

Calling sequence: check_views(selection, newpts)

Input/Output: **int newpts** is accepted to determine if any new points have been created. **int selection** returns the menu selection chosen, and is used to QUIT processing if the user wishes to end the session.

Global variables accessed/modified: **int done[]** is used to determine if there is a sufficient number of exemplary points for each view.

Major routines called:

findpts
stored_T
buildt
checkpts

Utility routines called:

setflags

Name: **checkpts**

Purpose: Determines if a particular view has less than 6 exemplary points. Returns 0 is less than 6, 1 if 6 or more.

Calling sequence: checkpts(nview)

Input/Output: Accepts **int nview**, the view to be checked.

Global variables accessed/modified: **viewtype views[].numpoints** is incremented.

Major routines called: None

Utility routines called: None

exemp.c

Name: **trans_pts**

Purpose: Creates a matrix of exemplary points to be used for building the transformation matrix.

Calling sequence: trans_pts(pointmat,numpts,tview)

Input/Output: Accepts the number of exemplary points for the view in **int numpts** and the view number in **int tview**. Returns a matrix of exemplary points in **double **pointmat.**

Global variables accessed/modified: Accesses the exemplary points in **viewtype views[tview].points[][].**

Major routines called: None

Utility routines called: None

Name: **test_ex1**

Purpose: Drives the first testing option.

Calling sequence: test_ex1(selection)

Input/Output: **int selection** returns the menu selection chosen, and is used to QUIT processing if the user wishes to end the session.

Global variables accessed/modified: **test_struct *testptr** is created, and the location of the removed point set at zero.

Major routines called:

trymat

Utility routines called:

talloc

exemp.c

Name: **test_ex2**

Purpose: Drives the second testing option.

Calling sequence: test_ex2(selection)

Input/Output: **int selection** returns the menu selection chosen, and is used to QUIT processing if the user wishes to end the session.

Global variables accessed/modified: Accesses the exemplary points in **viewtype views[].points[][]** and the number of exemplary points in **views[].num_points**. The location of the point to be removed is stored in **test_struct *testptr**.

Major routines called:

buildt
trymat
trans_pts

Utility routines called:

interp_point
getline
dmatrix
free_dmatrix

exemp.c

Name: **trymat**

Purpose: Controls the various menu options for creating three-dimensional points for the testing of exemplary points.

Calling sequence: trymat(selection,tview,numtests)

Input/Output: **int selection** returns the menu selection chosen. For the second testing option, **int tview** provides the view that is being tested. The number of test rows is returned in **int numtests.**

Global variables accessed/modified: The label for the point being created is stored in **test_struct *testptr.**

Major routines called:

testcrds
calctest
disptest
savetest
trymat
trans_pts

Utility routines called:

interp_point
getline

Name: **testcrds**

Purpose: Controls the selection of 2d points to be used for creating three-dimensional points for testing.

Calling sequence: testcrds(u,v,&incords)

Input/Output: The 2d points selected are returned in **double u** and **v** and a flag that indicates that points are currently being selected is returned in **int incords.**

Global variables accessed/modified: The global 2d stack **uvstack uv** is accessed.

Major routines called: None

Utility routines called:

convert
pushuv

exemp.c

Name: **calctest**

Purpose: Controls the creation of three-dimensional points for testing.

Calling sequence: calctest(incords,numtests,x,y,z)

Input/Output: The values of the measured three-dimensional point is accepted in **double x, y** and z. The flag indicating that points are currently being selected is returned in **int incords** and the number of tests done is returned in **int numtests.**

Global variables accessed/modified: The views used and the calculated point for each test are stored in **test_struct *test-ptr. testptr** is then stored in **test_struct test_data[].**

Major routines called:

calc_3d

Utility routines called:

dmatrix
dvector
popuv
free_dmatrix
free_dvector
talloc

Name: **disptest**

Purpose: Displays the exemplary point test data.

Calling sequence: disptest(tview,numtests)

Input/Output: The number of tests stored is accepted in **int numtests** and the current view being tested in accepted in **int tview.**

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

exemp.c

Name: **savetest**

Purpose: Writes the exemplary point test data to a file.

Calling sequence: savetest(tview,numtests)

Input/Output: The number of tests stored is accepted in int **numtests** and the current view being tested in accepted in int **tview.**

Global variables accessed/modified: None

Major routines called: None

Utility routines called:

getline
prfile

Name: **prfile**

Purpose: Writes one test row to a file.

Calling sequence: prfile(i)

Input/Output: The current view being tested in accepted in **int i.**

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

exemp.c

Name: **addpts**

Purpose: Allows the user to create new exemplary points, by prompting first for a 3-d point and then its 2-d occurance in as many views as it appears.

Calling sequence: addpts(selection)

Input/Output: **int selection** returns the menu selection chosen, and is used to QUIT processing if the user wishes to end the session.

Global variables accessed/modified: **viewtype views[].numpoints** is incremented.

Major routines called:

read_3d
store_pts
show_view

Utility routines called:

setflags
interp_point
convert

Name: **store_pts**

Purpose: Stores the 3-d and 2-d components of an exemplary point for the appropriate view.

Calling sequence: store_pts(nview, num, u, v, x, y, z, error)

Input/Output: Accepts **int nview**, the view for which the point should be stored. Accepts **double u** and **v**, the 2-d coordinate values, and **double x, y,** and **z**, the 3-d coordinate values. **int error** is returned, and indicates whether or not the point has already been stored for that view.

Global variables accessed/modified: Points are stored in **viewtype views[].points[][].**

Major routines called: None

Utility routines called:

dup_pts

exemp.c

Name: **deletepts**

Purpose: Allows the user to delete previously stored exemplary points. The user is prompted for the location of the point to be deleted, entered at the keyboard. A point is deleted by shifting up one location all the points that occur in the list after the deleted point.

Calling sequence: deletepts(selection)

Input/Output: **int selection** returns the menu selection chosen, and is used to QUIT processing if the user wishes to end the session.

Global variables accessed/modified: Points are deleted from **viewtype views[].points[][]** and **views[].numpoints** is decremented.

Major routines called: None

Utility routines called:

inter_points
show_view
getline
atoi
dup_pts

Name: **dup_pts**

Purpose: Checks if an exemplary point has already been stored. Returns 0 if a duplicate is not found, 1 if the accepted point is a duplicate.

Calling sequence: dup_pts(nview, num, u, v, x, y, z)

Input/Output: Accepts **int nview**, which is the view for which the point is to be checked. Accepts **double u** and **v**, which are the values for the 2-d coordinate, and **double x, y,** and **z,** the values for the 3-d coordinate.

Global variables accessed/modified: Uses the global **double accuracy** for checking the duplication of the 2d component of the point.

Major routines called: None

Utility routines called: None

exemp.c

Name: **showpts**

Purpose: Directs the display of exemplary points.

Calling sequence: showpts(selection)

Input/Output: **int selection** returns the menu selection chosen, and is used to QUIT processing if the user wishes to end the session.

Global variables accessed/modified: None

Major routines called: None

Utility routines called:

interp_point
show_view

Name: **show_view**

Purpose: Displays the exemplary points.

Calling sequence: show_view(nview)

Input/Output: **int nview** is the view to be displayed.

Global variables accessed/modified: Accesses **viewtype views[].points[][] and views[].numpoints.**

Major routines called: None

Utility routines called: None

trans.c

Contains routines that create the transformation matrices for each view.

Routines:

buildt
a_tbuild
b_tbuild

Global variable declarations: None

trans.c

Name: **buildt**

Purpose: Directs the creation of a transformation matrix.

Calling sequence: buildt(tmat,numpts,nview)

Input/Output: **int nview** is the view for which the current matrix is being created. **double **tmat** is a pointer to a 2-d matrix which holds the exemplary points and **int numpts** is the number of exemplary points_for the current view.

Global variables accessed/modified: The transformation matrix created for the view is stored in **views[].T.**

Major routines called:

a_tbuild
b_tbuild
calc_squares

Utility routines called:

free_dmatrix
free_dvector

Name: **a_tbuild**

Purpose: Builds the matrix of coefficients (A) needed to solve the linear equations to find the transformation matrix, T.

Calling sequence: a_tbuild(a, tmat, numpts)

Input/Output: The number of exemplary points for the view is passed in **int numpts** and a pointer to a 2-d matrix holding the exemplary points is passed in **double **tmat.** The A matrix is returned in **double **a.**

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

trans.c

Name: **b_tbuild**

Purpose: Builds the right hand side of the set of linear equations in the form of a column vector, B.

Calling sequence: b_tbuild(b,tmat,numpts)

Input/Output: The number of exemplary points for the view is passed in **int numpts** and a pointer to a 2-d matrix holding the exemplary points is passed in **double **tmat.** The B matrix is returned in **double *b.**

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

tabsel.c

Contains the routines responsible for the processing of menu options related to creating three-dimensional objects from two-dimensional views.

Each of the menu option routines determine the next available options. Before any value is changed, it is saved in case the command is CANCELed.

Routines

get_menu
open_object
open_row
new_row
vertices
calc_point
exit_selec
display_data
view_image
coordinate
pick_menu
menu1
menu2

save_last
save_flags
save_curr_disp
rest_flags

erase
delete
cancel

Global variable declarations

uvstack uv - stack that holds two dimensional points from up to four different viewsm that are used to calculate three dimensional points.

xyzstack xyz - stack that holds the three-dimensional points that are used to create facets.

int diverror - an error flag set by a divide by zero when the three dimensional point is calculated.

char imfile[12] - the name of the view image file.

int dispview - the view whose image is currently displayed on the screen.

int nopts - number of new points added in each callto . Used for canceling commands.

int nofacs - number of new facets created in each call to. Used for canceling commands.

int sflags[MAXFLAGS] - saves the state of the option flags before the current command so they can be restored if the current command is canceled.

int curr_disp[5] - contains flags which indicate which views were used to create the current three-dimensional point.

int saved_disp[5] - saves the values of **curr_disp** before a new point is created so that they can be restored in case the current point is deleted. Used for erasing points from the image screen.

tabsel.c

Name: **get_menu**

Purpose: Initializes global variables and directs the processing of menu selections for the creation of the object.

Calling sequence: get_menu()

Input/Output: None

Global variables accessed/modified: **uvtype uv.uvtop, xyztype xyz.xyztop, linktype linkdisp[].linkview[], int linkpos, int no_pts** and **int no_facs** are intialized to 0.

Major routines called:

open_object
open_row
new_row
vertices
calc_point
exit_selec
display_data
view_image
coordinate
keyboard
erase
delete
cancel
pick_menu

Utility routines called:

interp_point

tabsel.c

Name: open_object

Purpose: Saves current values and creates a new object.

Calling sequence: open_object(incords,currcomm)

Input/Output: Returns **int incords**, a flag indicating whether the user is at the menu level or is in the process of choosing a set of 2-d points, and **int currcomm**, the current command code.

Global variables accessed/modified: **int bk_comm** is set equal to the current command code.

Major routines called:

getobj
addobj

Utility routines called:

save_flags
save_obj
save_fac
setflags

Name: open_row

Purpose: Saves current values and allows the user to begin processing a new row set.

Calling sequence: open_row(incords,currcomm)

Input/Output: Returns **int incords**, a flag indicating whether the user is at the menu level or is in the process of choosing a set of 2-d points, and **int currcomm**, the current command code.

Global variables accessed/modified: **int bk_comm** is set equal to the current command code.

Major routines called: None

Utility routines called:

save_flags
setflags

tabsel.c

Name: **new_row**

Purpose: Allows the user to begin an additional row in the current row set. If this is only the start of the second row in the current set, the first row is checked to be sure that there are at least two points, and the count of rows in the set is initialized. If at least two rows have already been selected, the last row is checked to be sure it was equal in length to previous rows in the set. If more than two rows have been completed, the 3-d stack is reset so that the the correct points are created into facets. Points and facets are then created and stored in the output list.

Calling sequence: new_row(incords,currcomm,numset,row2,row1)

Input/Output:

Returns:
int incords, a flag indicating whether the user is at the menu level or is in the process of choosing a set of 2-d points.
int currcomm, the current command code.
int numset, the number of rows in the set.

Accepts:
int row2 - the number of points in the most recent row.
int row1 - the number of points in the previous row.

Global variables accessed/modified: **int bk_comm** is set equal to the current command code and **int no_pts** and **int no_facs** sent as arguments to count the number of new points and facets.

Major routines called:

set_stack
row_file

Utility routines called:

save_fac
save_flags
setflags

tabsel.c

Name: **calc_point**

Purpose: Directs the creation of a three dimensional point from several two dimensional points if points from at least two views were selected. If the point is returned free of error, the next step depends on the current command code. If the code is OPEN ROW, row1 is incremented; if NEW ROW, row2 is incremented. If the current command is VERTICES and three vertices have been created, they are formed into a facet and the count of vertices is initialized to 0; otherwise, the number of vertices is incremented.

Calling sequence:

calc_point(incords,currcomm,row1, row2,vert_pts)

Input/Output:

Returns:
int incords, a flag indicating whether the user is at the menu level or is in the process of choosing a set of 2-d points.
int currcomm, the current command code.
int vert_pts, the number of vertices for the current facet.
int row2 - the number of points in the most recent row.
int row1 - the number of points in the previous row.

Global variables accessed/modified: **int diverror** contains a flag that indicates whether or not a divide by zero error occured. **bk_verts** is set equal to the number of vertices for the current facet. The current 3-d point is stored in **linktype linkdisp[]**, the array that links the 3-d points with the location of the point on each of the displayed views, is incremented, and **int linkpos**, the current position in the array, is incremented. **int currdisp[]** and **uvtype uv.uvtop** is initialized to zero.

Major routines called:

calc_3d
vert_file

Utility routines called:

save_fac
save_stack
save_curr_dips
setflags
popuv
pushxyz
free_dvector
free_dmatrix

tabsel.c

Name: **exit_selec**

Purpose: The procedure processes the EXIT command at the menu level. If the current command is NEW ROW and the final two rows are of equal length, facets are created. If the current command is VERTICES, the user if allowed to choose a new option only if the last facet has been finished.

Calling sequence:

exit_selec(currcomm,numset,row2,row1,vert_pts)

Input/Output:

Returns:
int currcomm, the current command code.
int numset, the number of rows in the row set.
int row2, the number of points in the most recent row.
int row1, the number of points in the previous row.

Accepts:
vert_pts, the number of vertices created for the current facet.

Global variables accessed/modified: **int bk_comm** is set equal to the current command code and **int no_pts** and **int no_facs** sent as arguments to count the number of new points and facets.

Major routines called:

row_file

Utility routines called:

save_fac
save_flags
setflags
save_stack
set_stack

tabsel.c

Name: **display_data**

Purpose: Displays all values to the screen: the 2-d and 3-d stacks, the list of 3-d points and facets, the list of points created for each grayscale image, and the currently set flags.

Calling sequence:

```
display_data(currcomm,row1,row2,vert_pts)
```

Input/Output: Accepts **currcomm**, **row1**, **row2**, and **vert_pts** for display to the screen.

Global variables accessed/modified: None

Major routines called:

show_uvstack
show_xyzstack
show_obj_file
show_werner
show_flags

Utility routines called:

getline

Name: **view_image**

Purpose: Stores the number of the view to be displayed on the image screen, and writes the image for that view to the screen.

Calling sequence: `view_image()`

Input/Output: None

Major routines called:

get_imfile
read_image
write_impts

Utility routines called:

save_flags
setflags
interp_point
rest_flags

tabsel.c

Name: **coordinate**

Purpose: Processes the selection of a point on one of the 2-d views. The view is calculated and the point is pushed onto the 2-d stack. The point is then added to the list of points for display on the image screen.

If the point is from a view already selected for the current object point, or if it is not within the boundaries of one of the views, the point is rejected. In this case, if this is the first point selected (i.e., the 2-d stack is now empty) the flags are reset and the user is returned to the menu level.

Calling sequence: coordinate(incords, u, v)

Input/Output: Returns **int incords** which is set to 1. Accepts **double u** and **v**, the coordinate values for the selected 2-d point.

Global variables accessed/modified: **uv.uvitem[].cview** and **uv.top** are accessed to determine if the current point is on a view already selected.

Major routines called:

make_display

Utility routines called:

save_flags
setflags
restflags

convert
pushuv
emptyuv

tabsel.c

Name: **keyboard**

Purpose: Allows the user to choose previously created points for the current row or vertex. Point labels are entered at the keyboard, the point for the label is found, and the point is pushed onto the 3-d stack.

Calling sequence: keyboard(currcomm,row1, row2,vert_pts)

Input/Output:

Accepts:
int currcomm, the current command code.
Returns:
int vert_pts, the number of vertices for the current facet.
int row2 - the number of points in the most recent row.
int row1 - the number of points in the previous row.

Global variables accessed/modified: **bk_verts** is set equal to the number of vertices for the current facet.

Major routines called:

find_obj_pt
vert_file

Utility routines called:

save_fac
save_stack
setflags
pushxyz
free_dvector
getline
atoi

tabsel.c

Name: **erase**

Purpose: **Erases a 2-d point by removing it from the 2-d stack. The point is also deleted from the list of image points and from the image screen if it is currently displayed.**

If this results in an empty stack, the user is returned to the menu level.

Calling sequence: erase(incords)

Input/Output: Returns **int incords,** which is reset to 0 if the erase results in an empty 2-d stack.

Global variables accessed/modified: None

Major routines called:

del_view

Utility routines called:

restflags

pushuv
emptyuv

tabsel.c

Name: **delete**

Purpose: Deletes a 3-d point by removing it from the 3-d stack. All 2-d points that were used to calculate the point are deleted from the list of image points. If any of these 2-d points are currently displayed on the image screen, that point is erased as well.

If the current command is OPEN ROW, row 1 is decremented. If the current command is NEW ROW, row2 is decremented. If the current command is VERTICES, and this point created a new facet, the facet is deleted and the previous values in the 3-d stack are restored. The number of vertices for the current facet is then decremented.

Calling sequence:

delete(incords,currcomm, row1, row2, vert_pts)

Input/Output: Returns a decremented **int row1, int row2,** or int **vert_pts,** depending on the current command. Accepts int **currcomm,** the current command, and **int incords,** which must be set to 0 to indicate that the user is at the menu level.

Global variables accessed/modified: **int bk_verts** is accessed to determine if facets were created with the last (deleted) command.

Major routines called:

del_all_views

Utility routines called:

rest_stack
rest_opt

popxyz
emptyxyz

tabsel.c

Name: **cancel**

Purpose: Cancels the most recent of several menu selections, and restores the status of all variables to their values prior to the processing of the command to be canceled. In addition to the resetting of the arguments, below, the 3-d stack is restored and any facets that may have been created by the canceled command are deleted.

Calling sequence:

```
cancel(currcomm,numset, row2,row1)
```

Input/Output: Accepts **currcomm** to indicate what command to cancel. Returns **row1**, **row2**, and **numset** which are reset to their original values.

Global variables accessed/modified: **int bk_comm**, **int bk_row1** and **int bk_row2** are used to restore saved values to the current command code and the two most recent rows of the current row set.

Major routines called: None

Utility routines called:

rest_flags
rest_obj
rest_stack
rest_opt

bk_stack

tabsel.c

Name: **pick_menu**

Purpose: Chooses the appropriate menu selections for display, depending on whether the user is at the menu level or selecting 2-d points.

Calling sequence: pick_menu1(incords)

Input/Output: Accepts **incords** which indicates whether the user is at the menu level or selecting points from the 2-d views.

Global variables accessed/modified: None

Major routines called:

menu1
menu2

Utility routines called: None

Name: **menu1**

Purpose: Prints the menu options currently available at the menu selection level.

Calling sequence: menu1()

Input/Output: None

Global variables accessed/modified: **int flags[]** is accessed to determine the available menu selections.

Major routines called: None

Utility routines called: None

tabsel.c

Name: **menu_2**

Purpose: Prints the menu options currently available while the user is selecting 2-d points.

Calling sequence: menu2()

Input/Output: None

Global variables accessed/modified: **int flags[]** is accessed to determine the available menu selections.

Major routines called: None

Utility routines called: None

tabsel.c

Name: **save_last**

Purpose: Saves several values in global variables in case these values must be later restored following a CANCEL command.

Calling sequence: save_last(row1, row2, currcomm, numset)

Input/Output: Saves **row1**, **row2**, **currcomm**, and **numset** in the respective global variables below.

Global variables accessed/modified: **int bk_row1**, **bk_row2**, **bk_comm** and **bk_numset** hold **row1**, **row2**, **currcomm**, and **numset**, respectively.

Major routines called: None

Utility routines called: None

Name: **save_flags**

Purpose: Saves the status of **flags[MAXFLAGS]** in **sflags[MAXFLAGS]** so the available menu options can be restored.

Calling sequence: save_flags()

Input/Output: None

Global variables accessed/modified: **int flags[MAXFLAGS]** is saved in **int sflags[MAXFLAGS]**.

Major routines called: None

Utility routines called: None

tabsel.c

Name: **rest_flags**

Purpose: Restores the status of **flags[MAXFLAGS]**, which was saved in **sflags[MAXFLAGS]**.

Calling sequence: rest_flags()

Input/Output: None

Global variables accessed/modified: **int flags[MAXFLAGS]** is restored, using the values saved in **int sflags[MAXFLAGS]**.

Major routines called: None

Utility routines called: None

tabsel.c

Name: **save_curr_disp**

Purpose: Saves the status of **curr_disp[5]**, which is a list of all views that were used in creating the most recent 3-d point, in **saved_disp[5]**. **saved_disp[]** is required in case the most recent point is deleted, and the point must be erased on all views.

Calling sequence: save_curr_disp()

Input/Output: None

Global variables accessed/modified: **int curr_disp[5]** is saved in **int saved_disp[5]**.

Major routines called: None

Utility routines called: None

calcpt.c

Contains the routines for calculating a 3-d point from its position on several views.

Routines:

calc_3d
makea_3
makeb_3

Global variable declarations:

uvstack uv - the stack of 2-d points.

int diverror - a error flag indicating a divide by zero error.

calcpt.c

Name: **calc_3d**

Purpose: Creates a 3-d point from every pair of views stored in the 2-d stack. The points are then averaged to provide one point for all possible 2-d points selected.

Calling sequence: calc_3d(a,b,X,vis)

Input/Output: The matrix of coefficients (A) and the matrix on the right hand side of the equations (B) are accepted in **double **a** and ***b**, respectively. Returns the 3-d points in the form of a matrix in **double *X** and the direction flag for the vertices in **int vis.**

Global variables accessed/modified: **int diverror** is tested after a new 3-d point is calculated to be sure that a valid matrix has been created. **uvstack uv.uvtop** is also accessed.

Major routines called:

makea_3
makeb_3
calc_squares

Utility routines called: None

Name: **makea_3**

Purpose: Builds the matrix of coefficients (A) needed to solve the linear equations to find the 3-d point, X.

Calling sequence: makea_3(a,view1, view2, u1,v1,u2,v2)

Input/Output: The two view numbers are passed in **int view1** and **int view2**, and the two points are passed in **double u1, v1** and **double u2, v2.** The matrix of coefficients is passed in **double **a.**

Global variables accessed/modified: **viewtype views[].*T** is accessed.

Major routines called: None

Utility routines called: None

calcpt.c

Name: **makeb_3**

Purpose: Builds the right hand side of the set of linear equations needed to find the 3-d point in the form of a column vector, B.

Calling sequence: makeb_3(b,view1, view2, u1,v1,u2,v2)

Input/Output: The two view numbers are passed in **int view1** and **int view2**, and the two points are passed in **double u1**, v1 and **double u2**, **v2**. The column vector is passed in **double *b**.

Global variables accessed/modified: **viewtype views[].*T** is accessed.

Major routines called: None

Utility routines called: None

dlsq.c

Contains the routines that finds the least squares fit for the matrix **X** given the matrices **A** and **B**.

The routines **lubksb** and **ludcmp** are modified versions of routines found in Press, William H. [1988].

Routines:

calc_squares
transpose
mat_1_mult
matmult
inverse
lubksb
ludcmp

Globally variable declarations:

int diverror - flag indicating that a singular matrix has been found.

dlsq.c

Name: **calc_squares**

Purpose: Calls a series of routines to solve the equation $A^TAX = A^TB$ for finding the least squares fit for X.

Calling sequence: calc_squares(a, arows, acols, b, x)

Input/Output: Accepts the A matrix in **double **a** and the B matrix in **double *b**, and accepts the number of rows and columns for A in **int arows** and **int acols**, respectively. The number of rows in A is equal to 2 * the number of exemplary points, and the number of columns is equal to 11. The solution matrix is returned in ***x**.

Global variables accessed/modified: **int diverror** is initialized, and then checked on its return from creating the solution matrix.

Major routines called:

transpose
mat_1_mult
matmult
inverse
lubksb
ludcmp

Utility routines called:

free_dmatrix
free_dvector

Name: **transpose**

Purpose: Transposes the matrix sent as argument.

Calling sequence: transpose(a, arows, acols, transa)

Input/Output: Accepts the matrix to be transposed in **double **a**, and the number of rows and columns for the matrix in **int arows** and **acols**, respectively. Returns the transpose of the matrix in **double **transa**.

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

dlsq.c

Name: **mat_1_mult**

Purpose: Multiplies a N X M matrix times a N X 1 matrix.

Calling sequence: mat_1_mult(a, arows, acols, b, brows, bcols,c)

Input/Output: Accepts the N X M matrix to be multiplied in double **a, and the number of rows and columns for the matrix in int arows and acols, respectively. Accepts the N X 1 matrix in double *b, and the number of rows and columns in int brows and bcols. Returns the product of A and B in double *c.

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

Name: **matmult**

Purpose: Multiplies a N X M matrix times a M X N matrix.

Calling sequence: matmult(a, arows, acols, b, brows, bcols,c)

Input/Output:Accepts the first matrix to be multiplied in double **a, and the number of rows and columns for the matrix in int arows and acols, respectively. Accepts the second matrix in double *b, and the number of rows and columns in int brows and bcols. Returns the product of A and B in double *c.

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

dlsq.c

Name: **inverse**

Purpose: Calculates the inverse a matrix using the LU decomposition method. For a discussion of this method, see Press, William., et. al.' [1988].

Calling sequence: inverse(a, y, n)

Input/Output:Accepts a square matrix to be inverted in **double **a**, and the number of rows and columns in **int n**. Returns the inverse of A in **double *y**.

Global variables accessed/modified: None

Major routines called:

ludcmp
lubksb

Utility routines called: None

Name: **ludcmp**

Purpose: Performs the LU decomposition on the square matrix accepted as argument. Written by William H. Press [1988].

Calling sequence: ludcmp(a, n, indx,d)

Input/Output:Accepts a square matrix to be decomposed in **double **a**. ****a** is replaced by an LU decomposition of a rowwise permutation of itself, and returned. The number of rows and columns in A is accepted in **int n**. **int *indx** records the row permutation effected by the partial pivoting; **float d** is +/-1 depending on whether the number of row interchanges were even or odd, respectively.

Global variables accessed/modified: **int diverror** is set if a singular matrix is encountered.

Major routines called: None

Utility routines called:

dvector
free_dvector

dlsq.c

Name: **lubksb**

Purpose: Solves the set of linear equations A x X = B. Written by William H. Press [1988].

Calling sequence: lubksb(a, n, indx,b)

Input/Output: Accepts the LU decomposition of a matrix in **double **a.** The number of rows and columns in A is accepted in **int n.** The permutation vector is accepted in **int *indx. double *b** is input as the right hand side vector, and returns the solution vector **x.**

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

util.c

Contains routines for allocating pointer storage. These are found in Press, William H, et. al. [1988].

Routines:

****dmatrix**
***dvector**
***ivector**

free_dvector
free_dmatrix

Global variable declarations: None

util.c

Name: **dmatrix**

Purpose: Allocates a double matrix. Written by William H. Press, et. al.[1988].

Calling sequence: double dmatrix(nrl,nrh, ncl, nch)

Input/Output: Matrix allocated is of size **[nrl..nrh][ncl..nch]** where nrl, nhr, ncl, nch are of type **int.**

Global variables accessed/modified: None

Major routines called: None

Utility routines called:

malloc

Name: **dvector**

Purpose: Allocates a double vector. Written by William H. Press, et. al. [1988].

Calling sequence: dvector(nrl,nrh, ncl, nch)

Input/Output: Vector allocated is of size **[nl..nh]** where nl and nh are of type **int.**

Global variables accessed/modified: None

Major routines called: None

Utility routines called:

malloc

util.c

Name: **ivector**

Purpose: Allocates an int vector. Written by William H. Press, et. al. [1988].

Calling sequence: ivector(nl,nh)

Input/Output: Vector allocated is of size **[nl..nh]** where nl and nh are of type **int**.

Global variables accessed/modified: None

Major routines called: None

Utility routines called:

malloc

dobj.c

Contains routines for creating the final list of 3-d points and their facets.

Routines:

init_obj_list
getobj
addobj
getfac

row_file	**vert_file**
row_facets	**vert_facets**
make_points	**odd**
stack_pos	**dup_verts**
store_facets	
store_obj_file	**find_obj_pt**
***alloc**	***falloc**
save_obj	**rest_obj**
save_fac	**rest_fac**
read_obj_file	**save_obj_file**
file_edit	**del_fac**
convert_file	

Global variable declarations:

```
struct facetnode
{
     int color;
     int facets[3];
     struct facetnode *nextfac;
}_*lastfac, *bk_lastfac;
```

Each facet is stored in a record that contains a color value (not used), an int array which holds three vertices, and a pointer to the next facet. ***lastfac** is a pointer to the last facet in the list; ***bk_lastfac** points to the one before the last and is used for the cancel command.

```
struct objnode
{
     int objnum;
     int numpts;
     int numfacs;
     double objpts[MAXEDGES][3];
     struct objnode *nextobj;
     struct facetnode *firstfac;
} *objlist, *lastobj, *bk_lastobj;
```

Each object is represented by a record. The first three integer values store the number of objects, the number of object points, and the number of facets. The list of object points is represented as an integer array of MAXOBJPTS possible points by the three coordinate values x, y and z. The record also contains a pointer to the next object, and a pointer to the first facet for that object. ***objlist** is a header node for the list of objects. ***lastobj** points to the current object and ***bk_lastobj** points to the object before the last one.

```
struct transarr
{
  double tfac[3][3];
  struct transarr *nextt;
} *tlist;
```

A data structure that holds the translated and scaled points that are a result of the conversion process for the Silicon graphics routines.

uvstack uv is the 2-dimensional stack and **xyzstack xyz** is the 3-dimensional stack.

dobj

Name: **init_obj_list**

Purpose: Initializes the head of the object list.

Calling sequence: init_obj_list()

Input/Output: None

Global variables accessed/modified: ***objlist** is created and initialized, and ***lastobj** is set equal to it.

Major routines called: None

Utility routines called:

***oalloc**

Name: **get_obj**

Purpose: Creates a new object node.

Calling sequence: get_obj()

Input/Output: None

Global variables accessed/modified: A new object node is created and hooked onto the object list, and ***lastobj** is set equal to it. A new facet node is also created, and both the ***nextfac** pointer of the new object and ***lastfac** are set equal to it.

Major routines called: None

Utility routines called:

***oalloc**
***falloc**

dobj

Name: **add_obj**

Purpose: Increments the number of objects in the object list header.

Calling sequence: add_obj()

Input/Output: None

Global variables accessed/modified: **objlist->objnum** is incremented.

Major routines called: None

Utility routines called: None

Name: **getfac**

Purpose: Creates a new facet node and hooks it on to the end of the facet list for the current object.

Calling sequence: getfac()

Input/Output: None

Global variables accessed/modified: ***lastfac** is set equal to the newly created facet node.

Major routines called: None

Utility routines called: None

dobj.c

Name: **row_file**

Purpose: Drives the development of the list of object points, vertices and facets for the row mode routines.

Calling sequence: row_file(no_pts, no_facs)

Input/Output: **int no_pts** is a count of the number of new points that are added to the object list. **int no_facs** is a count of the new facets that are created.

Global variables accessed/modified: None

Major routines called:

make_points
row_facets

Utility routines called: None

show_obj_file

Name: **vert_file**

Purpose: Drives the development of the list of object points, vertices and facets for vertices mode.

Calling sequence: vert_file(no_pts, no_facs)

Input/Output: **int no_pts** is a count of the number of new points that are added to the object list. **int no_facs** is a count of the new facets that are created.

Global variables accessed/modified: None

Major routines called:

make_points
vert_facets

Utility routines called: None

dobj.c

Name: **make_points**

Purpose: Stores unique points from the 3-d stack onto the object list, and updates the count of new object points and total count of object points.

Calling sequence: make_points(no_pts)

Input/Output: **int no_pts** is a count of the number of new points that are added to the object list.

Global variables accessed/modified: **lastobj->numpts** is incremented, and the 3-d stack **xyz** is accessed. The linking structure **linkdisp[].linkview[]** for displaying object point location on the screen is initialized.

Major routines called:

stack_pos
store_obj_file
store_disp_loc
dup_verts

Utility routines called: None

dobj.c

Name: **vert_facets**

Purpose: Creates a three sided facet from the three points in the 3-d stack. The three vertices of the facet are the locations of the three points in the object list. The order of the vertices is determined by whether all three points are visible in (has been selected in) the view 1 of the 2-d views. If all three points are visible, the vertices are stored counter-clockwise; otherwise, clockwise.

Calling sequence: vert_facets(no_facs)

Input/Output: **int no_fac** is a count of the number of new facets that are added to the object list.

Global variables accessed/modified: **lastobj->numfacs** is incremented.

Major routines called: None

Utility routines called:

stack_pos
store_facets
dup_verts

dobj.c

Name: **row_facets**

Purpose: Creates facets from two rows of points stored in the 3-d stack. The algorithm is as follows:

```
n - total number of points in the stack
p1 - location in 3d stack of 1st point in the facet
p2 - location in 3d stack of 2nd point in the facet
p3 - location in 3d stack of 3rd point in the facet
p1ctr - moves along the 1st row of points
p2ctr - moves along the 2nd row of points
oddctr and evenctr - alternate to form the 3rd point

init p1ctr to 1.5
init p2ctr to (n/2+1)     {1st point in the second row}
init oddctr to 2          {2nd point in the first row}
init evenctr to (n/2+2)   {2nd point in the second row}

for i = 1 to n-2          {always two fewer facets than points}
   p1 = trunc(p1ctr)
   p2 = trunc(p2ctr)
   if i is odd
     p3 = oddctr
     increment oddctr
   else p3 = evenctr
     increment evenctr
   endif

   add 0.5 to p1ctr       {causing the incrementing of p1 and p2
   add 0.5 to p2ctr          to alternate}

end loop
```

Example: Assume that the stack holds eight points, which is two rows of four points each. The facets will be formed from the eight points as follows:

p1	p2	p3
1	5	2
2	5	6
2	6	3
3	6	7
3	7	4
4	7	8

The order in which the facet vertices should be stored is determined by whether all points are visible in view 1, and then the vertices are stored in the object list.

Calling sequence: row_facets(no_facs)

Input/Output: **int no_fac** is a count of the number of new facets that are added to the object list.

Global variables accessed/modified: **lastobj->numfacs** is incremented.

Major routines called: None

Utility routines called:

odd
stack_pos
store_facets
dup_verts

Name: **store_facets**

Purpose: Stores a new facet in the object list. Points are stored counter-clockwise if all points are visible in view 1; otherwise, clockwise.

Calling sequence:
store_facets(ptpos1, ptpos2, ptpos3, direction)

Input/Output: Accepts **int ptpos1, ptpos2** and **ptpos3,** which are the locations of the facet vertices in the object list. Also accepts **int direction,** the order in which the vertices should be stored.

Global variables accessed/modified: The vertices are stored in **lastfac->facets[].**

Major routines called: None

Utility routines called: None

dobj.c

Name: **odd**

Purpose: Determines if the argument is odd or even. Function returns a 1 if the number is odd; 0 if it is even.

Calling sequence: odd(i)

Input/Output: **int i** is accepted and determined to be odd or even.

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

Name: **stack_pos**

Purpose: Returns coordinate values of a point in the 3-d stack at the location sent as argument.

Calling sequence: stack_pos(i, x, y, z)

Input/Output: Accepts **int i**, which is a position in the 3-d stack. Returns **double x, y,** and z, which are the coordinate values of the point found at that location.

Global variables accessed/modified: **xzz.xyzitem[i].sx, xzz.xyzitem[i].sy** and **xzz.xyzitem[i].sz** are copied into x, y, and z, respectively.

Major routines called: None

Utility routines called: None

dobj.c

Name: **store_obj_file**

Purpose: Stores the 3-d point passed as argument in the object point list.

Calling sequence: store_obj_file(num, x, y, z)

Input/Output: Accepts **int num**, which is a next position in the object point list. Also accepts **double x, y,** and **z,** the coordinate values to be stored at that location.

Global variables accessed/modified: x, y and z are stored at **lastobj->objpts[num][0], lastobj->objpts[num][1]** and **lastobj->objpts[num][2],** respectively

Major routines called: None

Utility routines called: None

Name: **find_obj_pt**

Purpose: Returns the point at a given location on the object list.

Calling sequence: find_obj_pt(X,pos)

Input/Output: Accepts the object list position in **int pos** and returns the point stored at that position in **double *X.**

Global variables accessed/modified: **lastobj->objpts[][]** is accessed.

Major routines called: None

Utility routines called: None

dobj.c

Name: **dup_verts**

Purpose: Returns the position of a 3-d point in the object list.

Calling sequence: dup_verts(x, y, z)

Input/Output: Accepts **double x, y,** and **z,** the coordinate values of the point whose location is sought.

Global variables accessed/modified: **lastobj->objpts[][]** is searched for a match with the argument. **double accuracy** is also accessed to determine the match.

Major routines called: None

Utility routines called: None

dobj.c

Name: **show_obj_file**

Purpose: Displays the entire object list.

Calling sequence: show_obj_file()

Input/Output: None

Global variables accessed/modified: All structures that form the object list pointed to by **objlist** are accessed.

Major routines called: None

Utility routines called: None

Name: ***oalloc**

Purpose: Returns a pointer to **struct objnode.**

Calling sequence: struct objnode *oalloc()

Input/Output: None

Global variables accessed/modified: None

Major routines called: None

Utility routines called:

malloc

Name: ***falloc**

Purpose: Returns a pointer to **struct facetnode.**

Calling sequence: struct facetnode *falloc()

Input/Output: None

Global variables accessed/modified: None

Major routines called: None

Utility routines called:

malloc

dobj.c

Name: **save_obj**

Purpose: Saves the pointer to the last object in **bk_lastobj.**

Calling sequence: save_obj()

Input/Output: None

Global variables accessed/modified: A pointer to the last object is saved in **struct objnode bk_lastobj.**

Major routines called: None

Utility routines called: None

Name: **save_fac**

Purpose: Saves the pointer to the last facet in **bk_lastfac.**

Calling sequence: save_fac()

Input/Output: None

Global variables accessed/modified: A pointer to the last facet is saved in **struct facetnode bk_lastfac.**

Major routines called: None

Utility routines called: None

Name: **rest_obj**

Purpose: The pointers saved in **bk_lastobj and bk_lastfac** are restored as pointers to the last object and the last facet.

Calling sequence: rest_obj()

Input/Output: None

Global variables accessed/modified: Pointers **struct objnode bk_lastobj and struct facetnode bk_lastfac** are accessed.

Major routines called: None

Utility routines called: None

dobj.c

Name: **rest_opt**

Purpose: Restores the object list to its state prior to the most recent command by subtracting the number of new points and facets formed and by restoring the pointer to the last facet.

Calling sequence: rest_opt()

Input/Output: None

Global variables accessed/modified: **struct objnode lastobj->numpts** and **lastobj->numfacs** are restored to their values prior to the last command, and **struct facetnode bk_lastfac** is used to restore the pointer to the last facet.

Major routines called: None

Utility routines called: None

dobj.c

Name: **read_obj_file**

Purpose: Reads data from a file into the object list.

Calling sequence: read_obj_file()

Input/Output: None

Global variables accessed/modified: **int currcomm** is initialized to 0 if the output file does not exist, indicating that the object list is empty and the user must first open an object before selecing points. If a file is read in, the last value of **currcomm** was saved, and will be restored when the file is read. **struct objnode lastobj** and **struct facetnode lastfac** are used to read in values for each object.

Major routines called:

file_edit
read_werner

Utility routines called:

getobj
getline
falloc

Name: **save_obj_file**

Purpose: Write data from the object list into a file

Calling sequence: save_obj_file()

Input/Output: None

Global variables accessed/modified: **struct objnode objlist** as a pointer to the object list.

Major routines called:

write_werner

Utility routines called:

getline

dobj.c

Name: **file_edit**

Purpose: Allows the user to delete a facet from an object file.

Calling sequence: file_edit()

Input/Output: None

Global variables accessed/modified: None

Major routines called:

del_fac

Utility routines called:

getline

Name: **del_fac**

Purpose: Finds the facet to delete and deletes it.

Calling sequence: del_fac(loc1, loc2, loc3)

Input/Output: **int loc1, loc2 and loc3** are the values entered by the user that serve as indices into the object point list and represent the three points of the facet to be deleted.

Global variables accessed/modified: **struct objnode objlist** is used to access the object list.

Major routines called: None

Utility routines called: None

dobj.c

Name: **convert_file**

Purpose: Converts the object file to a format usable by the Silicon Graphics routines under development.

Calling sequence: convert_file()

Input/Output: None

Global variables accessed/modified: **struct objnode *objlist** is used to access the object list and **struct transarr *tlist** is a pointer to the data structure that holds the converted point values.

Major routines called: None

Utility routines called: None

dstack.c

Contains all the stack routines for both the 2-d and 3-d stacks, including the save and restore commands, as well as the temporary stack routines for the CANCEL command.

Routines:

popuv	**popxyz**
pushuv	**pushxyz**
emptyuv	**emptyxyz**
show_uvstack	**show_xyzstack**
save_stack	**rest_stack**
set_stack	**bk_stack**
poptemp	**pushtemp**
emptytemp	**show_tempstack**

Global variable declarations:

```
extern uvstack uv - the 2-d stack
extern xyzstack xyz  - the 3-d stack

typedef struct
{
   int tstop;                    A stack structure for the temporary
   xyzstruct tsitem[50];         stacks for saving and restoring
}  tempstack;                    the 3-d stack.

tempstack ts1, ts2, ts2;         Three temporary stacks.
```

dstack.c

Name: **pushuv**

Purpose: Pushes data onto the 2-d stack

Calling sequence: pushuv(u,v,currview)

Input/Output: Accepts the 2-d coordinates values **double u** and v, and the view number **int currview.**

Global variables accessed/modified: Data is stored in **uvstack uv.**

Major routines called: None

Utility routines called: None

Name: **popuv**

Purpose: Pops data from the 2-d stack.

Calling sequence: popuv(tempu,tempv,tempview)

Input/Output: Copies the values on the top of the stack into **double tempu** and **tempv, and int currview.**

Global variables accessed/modified: Data is copied from **uvstack uv.**

Major routines called: None

Utility routines called: None

dstack.c

Name: **pushxyz**

Purpose: Pushes data onto the 3-d stack

Calling sequence: pushxyz(Z,direction)

Input/Output: Accepts the coordinate values in **double *X,** and the **int direction** (counterclockwise = 1, clockwise = 0) for a 3-d point.

Global variables accessed/modified: Data is stored in **xyzstack xyz.**

Major routines called: None

Utility routines called: None

Name: **popxyz**

Purpose: Pops data from the 3-d stack.

Calling sequence: popxyz(tempu,tempv,tempz,direction)

Input/Output: Copies the values on the top of the stack into **double tempu, tempv** and **tempz,** and **int direction.**

Global variables accessed/modified: Data is copied from **xyzstack xyz.**

Major routines called: None

Utility routines called: None

dstack.c

Name: **emptyuv**

Purpose: Returns 1 if the 2-d stack is empty; 0 otherwise.

Calling sequence: emptyuv()

Input/Output: None

Global variables accessed/modified: **uvstack uv** is accessed.

Major routines called: None

Utility routines called: None

Name: **emptyxyz**

Purpose: Returns 1 if the 3-d stack is empty; 0 otherwise.

Calling sequence: emptyxyz()

Input/Output: None

Global variables accessed/modified: **xyzstack xyz** is accessed.

Major routines called: None

Utility routines called: None

dstack.c

Name: **show_uvstack**

Purpose: Prints the contents of the 2-d stack.

Calling sequence: show_uvstack()

Input/Output: None

Global variables accessed/modified: **uvstack uv** is accessed.

Major routines called: None

Utility routines called: None

Name: **show_xyzstack**

Purpose: Prints the contents of the 3-d stack.

Calling sequence: show_xyzstack()

Input/Output: None

Global variables accessed/modified: **xyzstack xyz** is accessed.

Major routines called: None

Utility routines called: None

dstack.c

Name: **save_stack**

Purpose: Saves the current contents of the 3-d stack in a temporary stack.

Calling sequence: save_stack()

Input/Output: None

Global variables accessed/modified: **xyzstack xyz** is copied into the temporary stack **tempstack ts3.**

Major routines called: None

Utility routines called:

emptyxyz
popxyz
pushtemp

Name: **rest_stack**

Purpose: Restores the contents of the temporary stack to the 3-d stack.

Calling sequence: rest_stack()

Input/Output: None

Global variables accessed/modified: The temporary stack **tempstack ts3** is copied into **xyzstack xyz.**

Major routines called: None

Utility routines called:

devector
emptytemp
poptemp
pushxyz

dstack.c

Name: **set_stack**

Purpose: Each time three rows of points have been placed into the 3-d stack, the stack must be re-set so that the second row is placed in the stack in the position of the first row, and the third row is placed in the position of the second. This is because the routines for creating object points and facets expect only two rows in the stack, in the correct order.

The points in the third and second rows are popped off the 3-d stack and stored in the temporary stack **ts2**. The points in the first row are then popped off and stored in the temporary stack **ts1**, in case the current command is canceled and the stack must be restored. The points in **ts2** are then returned to the 3-d stack.

Calling sequence: set_stack(num)

Input/Output: **int num** is the total number of points in the two rows that should be used to create the new set of object points and facets.

Global variables accessed/modified: **tempstack ts1** and **ts2** are used to temporarily store the 3-d points.

Major routines called: None

Utility routines called:

dvector
emptyxyz
popxyz
pushxyz
emptytemp
poptem
pushtemp

dstack.c

Name: **bk_stack**

Purpose: Restores the 3-d stack to its original state before the set_stack command.

The points are popped off the 3-d stack and stored in the temporary stack **ts2**. The points in the temporary stack **ts1** (which holds the original first row) **are** popped, and stored back in their original position in the 3-d stack. The points in **ts2** are then returned to the 3-d stack.

Calling sequence: bk_stack(num)

Input/Output: **int num** is the total number of points in the two rows that should be used to create the new set of object points and facets.

Global variables accessed/modified: **tempstack ts1** and **ts2** are used to temporarily store the 3-d points.

Major routines called: None

Utility routines called:

dvector
emptyxyz
popxyz
pushxyz
emptytemp
poptem
pushtemp

dstack.c

Name: **pushtemp**

Purpose: Pushes data onto a temporary 3-d stack.

Calling sequence: pushtemp(ts, tempx, tempy,tempz,direction)

Input/Output: **tempstack *ts** is the temporary stack to be used for storing the 3-d point. The coordinate values **double *tempx, tempy** and **tempz** are accepted, as is **int direction**, which determines the order in which the facet vertices are to be stored.

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

Name: **poptemp**

Purpose: Pops data from a temporary 3-d stack.

Calling sequence: poptemp(ts, tempx, tempy,tempz,direction)

Input/Output: **tempstack *ts** is the temporary stack to be popped. The values on the top of the stack are copied into **double *tempx, tempy** and **tempz** and **int direction**.

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

dstack.c

Name: **emptytemp**

Purpose: Returns 1 if the temporary stack is empty; 0 otherwise.

Calling sequence: emptytemp(ts)

Input/Output: **tempstack *ts** is checked.

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

Name: **show_tempstack**

Purpose: Prints the contents of the temporary stack. Used for debugging purposes.

Calling sequence: show_tempstack(ts)

Input/Output: The data in **tempstack *ts** is displayed.

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

wern.c

Contains the first level of routines for the display of images and feedback points on the image screen. None of these routines actually call the Werner Frie IMAGETOOL software.

Routines:

init_disp	*nalloc
get_image	get_imview
get_imfile	write_impts
make_display	calc_ratio
store_disp_loc	
show_werner	
del_all_views	
delete_pt	
del_view	
read_werner	
write_werner	

Global variable definitions:

```
struct displist
{
  int dx;
  int dy;
  int count;
  int im_value;
  int objloc;
  struct displist *nextd;
}
```

Each feedback display point is stored in the record structure above. **int dx** and **int dy** are the positions of the point on the screen. **int count** is used only by the header of each view's list of points, and is a count of the number of points for that view. **int im_value** is an averaged grayscale value of the pixels replaced by the drawing of the point of the screen. This average value is used for erasing the drawn point. **int objloc** stores the location in the object list of the three-dimensional point corresponding to the 2-d point displayed on the image.

struct displist *lists[5] - an array of header nodes, each of which point to a list of display points for a particular view.

struct displist *p - the currently erased point.

extern int dispview - the view currently displayed on the image screen.

char imname[MAXIMAGE] - the name of the current object.

int linkpos - the current position in the array that links the location in the object list for a point with its position in the display list for each view.

wern.c

Name: **init_disp**

Purpose: Initializes the display point lists by creating for header nodes and placing each in any array.

Calling sequence: init_disp()

Input/Output: None

Global variables accessed/modified: The array **struct displist *list[5]** is created and initialized.

Major routines called: None

Utility routines called:

***nalloc**

Name: ***nalloc**

Purpose: Returns a pointer to **struct idsplist *nalloc.**

Calling sequence: struct displist *nalloc()

Input/Output: None

Global variables accessed/modified: **None**

Major routines called: None

Utility routines called:

***malloc**

wern.c

Name: **get_image**

Purpose: Gets the object name, and for each view for that object directs the digitizing and storing of an image for that view.

Calling sequence: get_image()

Input/Output: None

Global variables accessed/modified: The object name is stored int the global **char imname[MAXIMAGE]**, and a flag is set in **views[].digitized** for each view that has been digitized.

Major routines called:

get_imview
get_imfile
digit_image

Utility routines called:

get_line
clear_screen
save_trans_files

Name: **get_imview**

Purpose: Reads a view number from the keyboard and converts it to an integer between 1 and 4. Returns the view number.

Calling sequence: get_imview()

Input/Output: None

Global variables accessed/modified: None

Major routines called: None

Utility routines called:

get_line

wern.c

Name: **get_imfile**

Purpose: **Creates a file name from the object name and view number for each view that is digitized. The file name is a the object name, followed by an underscore, and the view number. For example, view 2 of an object called house would be house_2.**

Calling sequence: get_imfile(selection,imfile)

Input/Output: Accepts the view number in **int selection** and returns the file name in **char imfile[12].**

Global variables accessed/modified: Accesses the object name in **char imname[MAXIMAGE].**

Major routines called: None

Utility routines called: None

Name: **write_impts**

Purpose: Writes to the image screen all points on the display point list for the currently displayed view.

Calling sequence: write_impts()

Input/Output: None

Global variables accessed/modified: Accesses the currently displayed view in **int dispview** and uses **displist lists[]** to point to the head of the display list for the appropriate view.

Major routines called: None

Utility routines called:

write_point
write_text

wern.c

Name: **make_display**

Purpose: Creates a new feedback display point, calculates the tablet-to-screen coordinate values, stores the point on the display point list, and writes it to the screen if the point was selected from the currently displayed view.

The algorithm for tablet-to-screen conversion is as follows:

$$screen\_offset\_x = view\_x * \frac{image\_max\_x - image\_min\_x}{view\_max\_x - view\_min\_x}$$

$$screen\_offset\_y = view\_y * \frac{image\_max\_y - image\_min\_y}{view\_max\_y - view\_min\_y}$$

Calling sequence: make_display(x, y, view, curr_disp)

Input/Output: Accepts the tablet coordinate values **double x** and **double y** and the view from which the points were chosen in **int view**. Returns **int curr_disp[5]**, which holds flags indicating which views have been selected to create the current 3-d point. This is later used for the DELETE command.

Global variables accessed/modified: **lists[]** is used as a pointer to the list for the current view. **int dispview** is accessed to determine if the point's view is the one currently displayed.

Major routines called:

calc_ratio
write_point

Utility routines called:

***nalloc**

wern.c

Name: **calc_ratio**

Purpose: Calculates the ratio of image coordinates to tablet coordinates required to convert from tablet to screen coordinates.

Calling sequence: calc_ratio(i, ratiox, ratioy)

Input/Output: Accepts the view from which the point was selected **in int i** and returns the ratio of tablet-to-screen coordinates **in double ratiox** and **double ratioy.**

Global variables accessed/modified: **views[].imbounds[][]** is accessed for the view's min and max values.

Major routines called: None

Utility routines called: None

Name: **show_werner**

Purpose: Displays the feedback point display list values.

Calling sequence: show_werner()

Input/Output: None

Global variables accessed/modified: **displist lists[]** is used to point to the display list for each view.

Major routines called: None

Utility routines called: None

wern.c

Name: **store_disp_loc**

Purpose: Stores the point's location in the object list in the linking structure for later display.

Calling sequence: store_disp_loc(num, x, y, z)

Input/Output: Accepts a three-dimensional point in **double x, double y** and **double z** and the location in the list in **int num.**

Global variables accessed/modified: The position of the point is stored in **linkdisp[].linkview[]** for the appropriate view.

Major routines called:

write_text

Utility routines called: None

wern.c

Name: **del_all_views**

Purpose: Deletes points from the display point list for all 2-d points that were used to create a deleted 3-d point.

Calling sequence: del_all_views(saved_disp)

Input/Output: The views from which the deleted 2-d points were selected are stored as flags in **int saved_disp[5].**

Global variables accessed/modified: The currently displayed view is accessed in **int dispview.**

Major routines called:

delete_pt
erase_point

Utility routines called: None

Name: **del_view**

Purpose: Deletes a point from the appropriate display point list and erases the point from the image screen if the view on which the point was selected is currently displayed.

Calling sequence: del_view(view)

Input/Output: Accepts **int view,** which is the view from which the point that is to be deleted was selected.

Global variables accessed/modified: Accesses **int dispview,** which is the view currently displayed, to determine if the point must be erased from the display screen.

Major routines called: None

Utility routines called:

delete_pt
erase_point

wern.c

Name: **delete_pt**

Purpose: **Finds** the last point on the display list for the accepted view, and decrements the count of display points for that view.

Calling sequence: delete_pt(i)

Input/Output: Accepts the view from which the point to be deleted was selected in **int i.**

Global variables accessed/modified: The head of the display list for the appropriate view is accessed using **displist *list[].**

Major routines called: None

Utility routines called: None

wern.c

Name: **read_werner**

Purpose: Reads in a file of display points that correspond to the object that have just been read.

Calling sequence: read_werner()

Input/Output: None

Global variables accessed/modified: The head of the display list for the appropriate view is accessed using **displist *list[].**

Major routines called: None

Utility routines called: None

Name: **write_werner**

Purpose: Writes a file of display points.

Calling sequence: write_werner()

Input/Output: None

Global variables accessed/modified: The head of the display list for the appropriate view is accessed using **displist *list[].**

Major routines called: None

Utility routines called: None

wrout.c

Contains the routines that interact with the Werner Frie IMAGE-TOOL software and the Microsoft Mouse software routines. These are used to write images to the image screen and write points selected on the digitizer to the corresponding screen image.

Routines:

init_werner	clear_screen
digit_image	grab_image
image_bds	
write_vector	write_point
erase_point	write_text
write_image	read_image

Global variable delcarations: None

External routines:

Werner Frie IMAGETOOL:

BRDSEL	CHARZ
CLKSEL	CUROFF
CURON	CURSET
CURTYP	DIGITZ
DWVEC	INITIM
LUTSEL	RDISK
RDPXL	SETMSK
SETSCR	WRPXL
WRLUT	WDISK

Microsoft Mouse:

CMOUSEL

The user is directed to the appropriate manuals for an explanation of these routines.

wrout.c

Name: **init_werner**

Purpose: Initializes the imaging board by calling the several IMAGETOOL routines.

Calling sequence: init_werner()

Input/Output: None

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

External routines called:

INITIM - initializes the Image Memory Base Address Segment (0xd000) and the Control Register Base Address (0x300) which correspond to the jumper settings of the imaging board installed in the system. **BRDSEL** selects, initializes and enables imaging board #1 for access by the other IMAGETOOL routines.
CLKSEL - initializes the Video Clock Mode to PLL.
LUTSEL - selects each of the four, eight bit input lookup tables of the imaging board.
WRLUT - downloads values into each of the lookup table selected.

Name: **clear_screen**

Purpose: Clears the image screen.

Calling sequence: clear_screen()

Input/Output: None

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

External routines called:

SETSCR - clears the screen and sets the values of all the pixels to 0.

wrout.c

Name: **digit_image**

Purpose: Drives the grabbing and storing of images of the various views,

Calling sequence: digit_image(i,imfile)

Input/Output: Accepts the view number to be grabbed in **int** i and the name of the file for the current object and view in **char imfile[12].**

Global variables accessed/modified: The image boundary values are stored in **viewtype views[].imbounds[].**

Major routines called:

grab_image
image_bds
write_image

Utility routines called: None

External routines called: None

Name: **grab_image**

Purpose: Directs the user in grabbing each view.

Calling sequence: grab_image(imfile)

Input/Output: Accepts the name of the file for the current object and view in **char imfile[12].**

Global variables accessed/modified: None

Major routines called: None

Utility routines called:

clear_screen
write_vector

External routines called:

SETMSK - sets the bitplane protection mask for subsequent **DIGITZ** calls.
DIGITZ - controls the video digitizer of the imaging board, allowing continuous digitization or single frame grabbing.

wrout.c

Name: **image_bds**

Purpose: Directs the user in locating the image boundaries.

Calling sequence: image_bds(hpos,vpos)

Input/Output: Stores the x value of the image boundary in **int hpos** and the y values in **int vpos.**

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

External routines called:

CMOUSEL - provides interface for mouse input.
CURTYP - selects the shape of the cursor to be drawn.
CURON - turns the cursor on at the last cursor location.
CURSET - erases the cursor and re-draws it at the new location.

Name: **write_vector**

Purpose: Writes a black (127) vector to the image screen.

Calling sequence: write_vector(x,y,avg)

Input/Output: Accepts one endpoint of the vector to be drawn in **int x1** and **int y1** and the location of the other endpoint in **int x2** and **int y2.**

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

External routines called:

DRVEC - Writes an arbitrary vector to the screen.

wrout.c

Name: **write_point**

Purpose: Writes a point to the image screen. The point drawn is one white pixel surrounded by a black crosshair.

Calling sequence: write_point(x,y,avg)

Input/Output: Accepts the location of the point to be drawn in **int x** and **int y**. Returns an average of the pixel values found on the image where the point is to be written.

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

External routines called:

RDPXL - Reads a single pixel value.
WRPXL - Writes a single pixel value.

Name: **erase_point**

Purpose: Erases a point on the image screen. When the point was written, the values of the pixels under the point was read, averaged and stored. The point is erased by writing over the pixels with that stored value.

Calling sequence: erase_point(x,y,pvalue)

Input/Output: Accepts the location of the point to be drawn in **int x** and **int y** and the average of the pixel values that were overwritten in **int pvalue**.

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

External routines called:

WRPXL - Writes a single pixel value.

wrout.c

Name: write_text

Purpose: Writes a number to the image screen. If the background is dark, the number written is white; otherwise it is black.

Calling sequence: write_text(x,y,loc)

Input/Output: Accepts the location of the point to be drawn in int x and int y and the number to be written in loc.

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

External routines called:

RDPXL - Reads a single pixel value to determine background color.
BINCHR - converts an integer to a character string.
CHARST - writes a character string to the screen.

wrout.c

Name: write_image

Purpose: Writes the image to disk.

Calling sequence: write_image(imfile)

Input/Output: Accepts the file name for the image in char imfile[12].

Global variables accessed/modified: None

Major routines called: None

Utility routines called: None

External routines called:

CHARZ - Appends a zero byte to character string it receives in argument.
WDISK - Writes from the imaging board to disk.

Name: read_image

Purpose: Reads the image of a view from disk to the imaging memory.

Calling sequence: read_image(imfile)

Input/Output: Accepts the file name for the image in char imfile[12].

Global variables accessed/modified: None

Major routines called: None

Utility routines called:

clear_screen

External routines called:

RDISK - Reads from disk to the imaging board.

APPENDIX B

TESTING DOCUMENTATION

Feature Testing

I. Initializing System

A. Loading Files

Test	Command	Result
No parameter file	<RETURN>	'No file named.' Request for object name
Non-existent parameter file	file name	'File does not exit' Request for object name
File containing image boundaries	file name	Requests name of output file. Requests object name and allows additional digitizing. Lists all views previously digitized.
No output file	<RETURN>	'No file named.' Request for object name
Non-existent output file	filename	'File does not exist' Request for object name
Output file loaded	filename	Data loaded Request for file editing
Create new points and facets		Previous points recognized; new points added to list; new facets added to list
Choose no editing	n	Prompt for display file
Edit a file	y	Prompt for facet vertices
Vertices entered		Facet deleted in list; number of facets decremented
2nd facet vertices entered		Facet deleted; number decremented
Non-existent vertices entered		'Facet not found'

Test	Command	Result
End file editing	<RETURN>	Request for display file
No display file	<RETURN>	'No file named.' Request for object name
Non-existent display file	filename	'File does not exist' Request for object name
Display file loaded	filename	Point labels displayed on grayscale image
Create new points and facets		Previous labels recognized; new points added to list; new facets added to list

B. Saving Files

Test	Command	Result
Save image boundaries when created.	filename	File with image boundaries and digitized flags created; all other values 0
Save image boundaries after read from previously created file.	filename	File with image boundaries and digitized flags created
Save during exemplary point definition	SAVE	Parameter file requested and created
Save during creation of 3-d points	SAVE	Output and display files requested and created

C. Object name

Test	Command	Result
object name	object name	Request for digitizing views
no object name	<RETURN>	Digitizing not permitted

D. Digitizing Views

Test	Command	Result
Skip digitizing step	<RETURN>	Requests object name
Create image for object_views	[view] 1	Imagefile created for object_1 digitized flag set for view
	[view] 2	Imagefile created for object_2 digitized flag set for view
	[view] 3	Imagefile created for object_3 digitized flag set for view
	[view] 4	Imagefile created for object_4 digitized flag set for view
	[view] 5	Re-requests a view #
	[view] 0	Re-requests a view #
End of digitizing	<RETURN>	Lists all digitized views

E. Aligning menu

Test	Command	Result
Menu aligned	select corners	'Menu aligned' View boundaries displayed
Menu not aligned	select corners	'Menu not properly aligned' Repeats request for corners
Menu not aligned	select corners	'Menu not properly aligned' Repeats request for corners
Menu aligned	select corners	'Menu aligned' View boundaries displayed

II. View boundaries

A. No input file used

Test	Command	Result
No file name given	<RETURN>	View boundaries all 0's Request for view boundaries All valid commands displayed
Enter view bounds for view 1	[view] 1	Request for lower left then upper right corners for view 1 Request for new view number
Repeat above for all view numbers entered		Same as above for each view
End entering boundary values	EXIT	View boundaries displayed Chance to change or exit again
Save	SAVE	Request for file name; saved in file (views not accepted)
Accept boundaries	EXIT	Request for file name to save boundaries; saved in file (views accepted) Exemplary points requested
Change boundaries	CHANGE	Request for boundaries - all valid commands displayed
Enter new view boundaries	[view] 1	Boundary values updated Request for new view number repeated as above
End updating	EXIT	View boundaries displayed. Request for file name to save boundaries; saved in file Exemplary points requested.
Quit session	QUIT	Request to save; then quit.

III. Defining Exemplary Points

A. ADD Points

Test	Command	Result
Choose add point mode	ADD POINT	Prompt for entering 3-d point.
Add point on four views		Points for appropriate view displayed each time point on that view is chosen.
Add 2nd point on four views		Same as above.
End adding points	<RETURN>	
End add mode	EXIT	
Display points	DISPLAY	Correct points displayed.
Exit from display mode	EXIT	Return to prompt.
Add point on three views	ADD PT	Same as above.
Repeat for 3 more points so that total points are: view 1: 4 points view 2: 6 points view 3: 5 points view 4: 6 points		Same as above.
End adding points	<RETURN>	Return to prompt.
End add mode	EXIT	Return to prompt.
Display points	DISPLAY	Points correctly displayed.
End display mode	EXIT	Return to prompt.
Save points	SAVE	Request for file name; points saved in file camex1.
End exemplary point section	EXIT	'Insufficient exemplary points for view 1'

Test	Command	Result
Add points for view 1	EXIT	Transformation matrix built for view 1 Transformation matrix built for view 2 'Insufficient exemplary points for view 3'
Add points for view 3	EXIT	Transformation matrix built for view 3 Transformation matrix built for view 4 Begin creating 3-d object

B. Load exemplary points file

Test	Command	Result
Display points	DISPLAY	Points correctly displayed.
Delete mode - no deletions	DELETE, <RETURN>	No points deleted.
Add mode	ADD POINT	Prompt for 3-d point
Add points so that: view 1: 9 view 2: 9 view 3: 9 view 4: 9		
End adding points	<RETURN>	
Exit adding mode	EXIT	
Save current points	SAVE	Request file name; points saved in file

C. DELETE mode

Test	Command	Result
Delete points	DELETE	Request for view
	[view] 1	Request for location
	[loc] 7	Point deleted.
	[loc] 1	Point deleted.
	[loc] 4	Point deleted.
	[loc] 5	'No point stored'
	[loc] 0	Same as above.
End deleting points	<RETURN> EXIT	Return to prompt.
Add back points that were deleted	ADD PTS	
Display points	DISPLAY, all views	All points correctly added.
Add a point that is already there.		'Point already stored'; not accepted.
Delete 3 points from view 1.		Points deleted
End deleting points	EXIT	Return to prompt.
End exemplary points section	EXIT	'Insufficient exemplary points for view'
Add point mode	ADD PT	Prompt for 3-d point
Add more points to view 1 so that it has 6 points		
End Adding points	<RETURN> EXIT	Transformation matrix built for all views

IV. Testing Exemplary Points

Test	Command	Result
Choose no testing	n	Begin creating 3-d object
Choose testing	y	Request for 3-d point for testing
End Test option 1	RETURN	Testing option 2
Enter point		Request for character label
Enter character label		Menu options for choosing multiple 2-d points
End entering points	EXIT	Return to prompt for 3-d point
Choose points in multiple views		Test data created and stored
Display test	DISPLAY	Displays test results
Save test	SAVE	Test file created
End testing option	RETURN	Begin testing option 2
EXIT Test option 2	EXIT	Begin creating 3-d object
Enter view number		Prompt for location of point to remove
End choosing points for view	RETURN	Request for next view number
Enter a point to remove		new matrices made with remaining points; menu options for choosing multiple 2-d points
Choose points in multiple views		Test results created and stored
Display points for Test option 2	DISPLAY	Displays tests for view
Save test	SAVE	Tests for view saved
End choosing points for view	EXIT	Begin creating 3-d points

V. Creating Objects

A. OPEN OBJECT

Test	Command	Result
Open new object	OPEN OBJECT	New object created
Create facets for object; open new object	OPEN OBJECT	Second object created

B. OPEN ROW/ADD ROW

Test	Command	Result
Begin new row set	OPEN ROW	New row set begun
Select 2 points in first row	Point pairs, EXIT	Points created for first row Points appear on image
Begin second row	ADD ROW	Second row begun
Select 2 points in second row	Point pairs, EXIT	Points created for second row Points appear on image
End row set	EXIT	Object points and facets created; point labels displayed
Build two row set, add a third row	ADD ROW	Object points and facets created; point labels displayed; third row begun
Select points for third row	Point pairs, EXIT	Points created for third row Points appear on image
End three row set	EXIT	Object points and facets created; point labels displayed
Build 3 row set, add a fourth row	ADD ROW OW	Object points and facets created; point labels displayed; fourth row begun
Select points for fourth row	Point pairs, EXIT	Points created for fourth row Points appear on image
End row set	EXIT	Object points and facets created; point labels displayed
Repeat above for 3, 4 and 5 point rows		Object points and facets created; point labels displayed

C. VERTICES

Test	Command	Result
Choose vertices option	VERTICES	Option begun
Choose three points	Point pairs, EXIT	Points created; third point adds object points and facet to list
Choose three more points	Point pairs, EXIT	Points created, third point in set adds object points and facets to list

D. VIEW

Test	Command	Result
Choose a view for display	VIEW	Image displayed for view
Choose a new view after several points have been selected	VIEW	New view displayed; points and labels appear on new image
Choose a view for display after reading in display file		New view displayed; labels appear on image
DELETE a point, choose a new view for display	DELETE, VIEW	Point has been erased from new view

E. KEYBOARD

Test	Command	Result
Begin a new row	OPEN ROW	Menu options displayed
Choose keyboard	KEYBOARD	Prompt for point label
End option	<RETURN>	Return to menu options
Choose label		Point placed on stack; Label prompt returns
Digitize rest of row; add row; digitize 2nd row		Points and facets created correctly
Choose labels for 1st row; digitize 2nd row		Points and facets created correctly
Choose non-existent label		'Invalid point number'

VI. Correcting User Errors

A. ERASE

Test	Command	Result
OPEN ROW, one 2d point selected	ERASE	Point removed and erased from image
OPEN ROW, two 2d points selected	ERASE	Most recent point removed and erased from image
Erase first point	ERASE	First point removed and erased from image
OPEN ROW	ERASE	Not allowed
VERTICES, one 2d points selected	ERASE	Most recent point removed and erased from image
VERTICES, two 2d points selected	ERASE	Most recent point removed and erased from image
Same point erased twice	ERASE, ERASE	Point erased both times
All points in stack erased		All points erased

B. DELETE

Test	Command	Result
OPEN ROW, one 3d point created	DELETE	Point deleted and erased from image
OPEN ROW, two 3d points created	DELETE	Most recent point deleted and erased from image
Delete first point	DELETE	First point deleted and erased from image
First row created, ADD ROW, one point created in 2nd row	DELETE	First point in second row deleted and erased from image
ADD ROW, 2 points created in 2nd row	DELETEDELETE	Most recent point deleted and erased from image
OPEN ROW, 2 point row, ADD ROW, 2 point row	DELETE, EXIT	'Rows not of equal length'

Test	Command	Result
OPEN ROW, 2 point row, ADD ROW, 3 point row	DELETE, EXIT	Object points and facets created
VERTICES, one 3-d point created	DELETE	Point deleted and erased from image
VERTICES, three 3d points created	DELETE	3rd point deleted and erased; points and facets that were created are canceled.
VERTICES, three 3d points, DELETE, add 3rd 3d point		Object points and facets created.
All points in stack deleted		All points deleted.

C. CANCEL

Test	Command	Result
OPEN OBJECT	CANCELL	Object canceled
OPEN OBJECT, T, OPEN ROW, ADD ROW, OPEN OBJECT	CANCEL	Object canceled
OPEN OBJECT, VERTICES, OPEN OBJECT	CANCEL	Object canceled
OPEN OBJECT, VERTICES	CANCEL	VERTICES command canceled
OPEN OBJECT, OPEN ROW	CANCELL	OPEN ROW command canceled
OPEN ROW, ADD ROW	CANCEL	ADD ROW command canceled
OPEN ROW, ADD ROW, ADD ROW	CANCEL	ADD ROW command canceled, object points and facets that were created canceled
OPEN ROW, ADD ROW, ADD ROW, CANCEL EXIT		Object points and facets created.

Test	Command	Result
Select four points for a two point row set, ADD ROW, CANCEL, delete two points, ADD ROW		Object points and facets created.

APPENDIX B

TESTING DOCUMENTATION

User Testing

IVAN was tested by Tom Servoss, a graduate student in the Center for Imaging Science. Although unfamiliar with the program, Tom does have familiarity with computers, other software products, and programming. He is typical of the type of user for which IVAN was written.

Before we began, Tom created an eight inch square wire frame and mounted it to a wooden base. The object, a plactic model of a military jeep, was placed inside the frame. The hope was that exemplary points on the frame could be more easily and accurately measured than on the object itself. This would also allow other objects to be used, assuming that they fit inside the frame, without measuring additional exemplary points.

Tom had little problem understanding the procedures required for most aspects of the program. Choosing the views, digitizing the grayscale images, setting up the digitizer, and entering the menu boundaries proceeded quickly. Tom did need some clarification on how files are saved, and which data are stored in which files. He suggested that the prompts for the various files be made more clear, which they were.

Tom then proceeded easily through the definition of the view boundaries. Watching him work, it also occurred to me that a

"beep" when an error is encountered would relieve the user of having to continually check the screen to be sure that everything is progressing correctly. This feature has been added to the program.

Tom and I had some discussion about which exemplary points would be the best to use; i.e., that more than two co-linear points, and points close together, do not contribute much information. I have included the essence of that discussion in the User Manual. Tom then found the actual entering of exemplary points easy to do. He did, however, need some clarification on how exemplary points are deleted. Once he understood that indices into the list of point are used, he found it an efficient method.

Tom found the exemplary point testing useful for seeing how accurate his points were in re-creating those same points. However, he found that the testing was limited in helping him determine exactly which point or points was in error. Since the size of the errors in the first test indicated that he must have miscalculated at least one of his points, checking back over his measurements he did discover that he had given a point the wrong three-dimensional coordinate values. He made this correction.

Tom discovered a problem when attempting to save the exemplary point tests in a file. The program asked for the name of a parameter file and an object file first, and then a test file name. Entering <RETURN> for the first two file names caused an error. The program was amended, and was made to more

efficiently ask only for the test file name.

Tom then had no problem digitizing the object. He did ask advice on creating a circle, and this will also be included in the user documentation.

Once the object had been completed, Tom wanted to read the object back in and add additional facets to it. Although the object list could be read back in, procedures for reading in the list of displayed points for the grayscale images were not implemented. That problem has been corrected, and the user can now read in a previous session, display the points from that session, and use the displayed points to create new facets. I have also added the ability to delete facets from the object file.

Finally, Tom needed routines to convert the current output file format to one that would be readable by the Silicon Graphics routines under development. This was also done, and the object Tom created was displayed on the Silicon Graphics equipment.

Appended to this discussion is a copy of the exemplary point testing files and final output files created for Tom's object. The exemplary point file tests measured points labeled A-L, using exemplary points A-H.

Testing using all points on each view.

Point tested: -8.00 0.00 0.00

Removed	Tested	Views				Delta X	Delta Y	Delta Z
0	A	1	1	0	0	0.03	-0.22	-0.04
0	A	1	0	1	0	0.06	-0.17	0.08
0	A	0	1	1	0	-0.28	0.09	-0.02

Testing using all points on each view.

Point tested: -3.50 0.00 4.03

Removed	Tested	Views				Delta X	Delta Y	Delta Z
0	B	1	1	0	0	0.07	-0.47	0.27
0	B	1	0	1	0	0.31	-0.16	0.22
0	B	1	0	0	1	0.16	-0.07	0.18
0	B	0	1	1	0	0.09	-0.18	0.21
0	B	0	1	0	1	-0.18	-0.18	0.17
0	B	0	0	1	1	0.06	0.05	0.18

Testing using all points on each view.

Point tested: 0.00 4.44 4.03

Removed	Tested	Views				Delta X	Delta Y	Delta Z
0	C	0	0	1	1	0.02	-0.01	-0.02
0	D	1	1	0	0	-0.32	-0.23	0.03
0	D	1	0	0	1	-0.37	0.15	0.05
0	D	0	1	0	1	-0.79	0.16	0.08
0	E	1	1	0	0	-0.31	-0.15	0.18
0	E	1	0	1	0	-0.09	0.27	0.00
0	E	1	0	0	1	-0.54	-0.33	0.11
0	E	0	1	1	0	0.38	-0.34	0.32
0	E	0	1	0	1	-0.27	-0.05	0.34
0	E	0	0	1	1	0.00	0.18	0.29

Testing using all points on each view.

Point tested: -8.00 3.97 4.00

Removed	Tested	Views				Delta X	Delta Y	Delta Z
0	F	0	1	1	0	-0.06	-0.19	0.02
0	F	0	1	0	1	-0.28	0.24	0.08
0	F	0	0	1	1	-0.53	0.04	-0.19
0	G	1	1	0	0	-0.27	-0.22	0.15
0	G	1	0	1	0	-0.19	0.01	-0.17
0	G	1	0	0	1	-0.70	-0.21	0.05
0	G	0	1	1	0	0.70	-0.82	0.21
0	G	0	1	1	0	0.70	-0.83	0.22
0	G	0	1	0	1	-0.69	0.24	0.43
0	G	0	0	1	1	-0.56	0.26	-0.02
0	G	0	0	1	1	-0.56	0.24	-0.02
0	H	1	1	0	0	-0.37	0.53	0.34
0	H	1	0	1	0	-0.06	-0.13	0.03
0	H	1	0	0	1	-0.31	-0.07	0.29
0	H	0	1	1	0	0.30	-0.75	0.10
0	H	0	1	0	1	-0.75	-0.15	0.28
0	H	0	0	1	1	-0.44	0.05	0.03

Testing using all points on each view.

Point tested: -8.00 0.00 4.00

Removed	Tested	Views				Delta X	Delta Y	Delta Z
0	I	1	1	0	0	-0.46	-0.63	0.14
0	I	1	0	1	0	-0.31	-0.38	-0.16
0	I	1	0	0	1	-0.75	-0.03	0.15
0	I	1	0	0	1	-0.75	-0.09	0.15
0	I	0	1	1	0	0.33	-1.10	0.04
0	I	0	1	0	1	-1.38	0.21	0.41
0	I	0	0	1	1	-1.21	0.20	-0.17
0	J	1	1	0	0	-0.85	0.02	0.11
0	J	1	0	1	0	-0.62	0.51	-0.23
0	J	1	0	0	1	-1.24	-0.25	-0.08
0	J	0	1	1	0	0.37	-0.54	0.32
0	J	0	1	0	1	-0.80	0.20	0.37
0	J	0	0	1	1	-0.51	0.43	0.15
0	K	0	1	1	0	0.27	-0.03	0.26
0	K	0	1	0	1	-0.09	0.00	0.19
0	K	0	0	1	1	0.16	0.28	0.24
0	L	1	1	0	0	-0.01	-0.53	0.37
0	L	1	0	1	0	0.43	-0.04	0.17
0	L	0	1	1	0	0.14	-0.17	0.09

```
  1
 51
 78
-5.76   5.83   3.25
-5.64   3.60   3.03
-5.89   5.90   2.20
-5.69   3.29   2.35
-5.88   5.99   1.36
-5.92   3.34   1.25
-5.56   1.84   2.24
-5.66   1.90   1.05
-3.86   6.08   3.00
-3.76   3.75   2.83
-3.85   3.40   2.15
-3.75   1.96   2.15
-3.68   1.85   1.03
-3.90   6.07   2.03
-3.67   3.35   1.13
-3.95   6.00   1.20
-6.06   5.19   1.36
-6.03   4.75   1.20
-6.08   4.55   0.76
-6.08   4.63   0.34
-6.15   5.18   0.08
-6.15   5.68   0.25
-6.15   5.80   0.75
-6.12   5.70   1.05
-6.09   5.17   0.74
-5.95   2.69   1.32
-5.91   2.21   1.20
-5.89   2.03   0.68
-5.90   2.14   0.27
-6.01   2.68   0.02
-6.00   3.19   0.20
-6.01   3.34   0.63
-5.94   2.71   0.70
-3.48   2.61   1.26
-3.52   3.14   1.10
-3.55   3.29   0.64
-3.46   3.17   0.26
-3.47   2.58   0.01
-3.49   2.12   0.27
-3.49   2.07   0.67
-3.47   2.25   1.14
-3.54   2.68   0.64
-3.68   5.20   1.18
-3.76   5.66   0.98
-3.81   5.77   0.55
-3.78   5.65   0.18
-3.73   5.23  -0.05
-3.69   4.66   0.18
-3.70   4.50   0.56
-3.74   4.74   0.97
-3.74   5.13   0.57
 2 1
 2 3
 4 3
 4 5
 5 6
 5 7
 1 8
 1 9
```

1 3 9
9 3 10
3 6 10
10 6 11
6 7 11
11 7 12
9 10 8
8 10 13
10 14 13
13 14 15
11 12 10
10 12 14
8 15 0
0 15 4
16 24 17
17 24 24
17 24 18
18 24 24
18 24 19
19 24 24
19 24 20
20 24 24
20 24 21
21 24 24
21 24 22
22 24 24
22 24 23
23 24 24
25 32 26
26 32 32
26 32 27
27 32 32
27 32 28
28 32 32
28 32 29
29 32 32
29 32 30
30 32 32
30 32 31
31 32 32
31 32 5
5 32 32
33 41 34
34 41 41
34 41 35
35 41 41
35 41 36
36 41 41
36 41 37
37 41 41
37 41 38
38 41 41
38 41 39
39 41 41
39 41 40
40 41 41
42 50 43
43 50 50
43 50 44
44 50 50
44 50 45
45 50 50

45 50 46
46 50 50
46 50 47
47 50 50
47 50 48
48 50 50
48 50 49
49 50 50

APPENDIX C

USER'S GUIDE

Table of Contents

1. Introduction

IVAN is a software package that takes as input multiple two-dimensional views of an object and creates a three-dimensional representation of that object. The work is based largely on the efforts of Ivan Sutherland, who described a similar project in a paper written in 1974.*

Two-dimensional points from up to four views of an object are entered using a digitizing tablet and hand-held stylus. From these points, the system produces a file of points and vertices that represent the object in three-dimensions. The file can in turn be used by a general purpose graphics package for displaying and manipulating the three-dimensional model that IVAN has created.

The user is carefully guided through the entire process of creating his three-dimensional model - each step must be completed before going on to the next. The user may also end a session at any time, and work-in-progress may be stored in files to be used at later sessions, so that earlier steps need not be repeated.

* Sutherland, I. E., "Three-Dimensional Data Input by Tablet," Proceedings of the IEEE, 62:453, 1974.

2. System Overview

IVAN allows up to four different photographic or orthographic drawings of the three-dimensional object that is to be modeled. These views are first "grabbed" by the video digitizing component of the system, which creates a 512 X 512 X 8 bit representation of each view. These grayscale images are later displayed on a video monitor to provide user feedback.

The four photographs or drawings are then placed on an x-y digitizing tablet. One area of the tablet is also set aside for a menu, allowing most commands to be entered without putting down the tablet stylus. Some functions require keyboard input as well. All menu options available at any given time are displayed before each option is selected.

Once the views and menu are attached to the tablet, the system requests the tablet positions of each. At least six "exemplary" points must then be defined for each view. These are two-dimensional points that correspond to known three-dimensional world coordinates. They are called "exemplary" because they are examples of what the viewing transformation from three dimensions to two dimensions has done to certain known points in each view. The system uses these exemplary points to construct a viewing transformation, represented as a matrix, for each view.

Points are then selected from the two-dimensional views, using the digitizing stylus. Each point on the object must be selected from at least two, but as many as four, different views.

Three-dimensional points are then calculated from the position of the point in each pair of views using the transformation matrices. The three-dimensional points form the vertices of three-sided facets which create the "skin" of the object model.

Points can be formed into facets directly, or they can be grouped by the user into rows, leaving it to the system to convert each pair of rows into the triangular facets that comprise the three-dimensional model.

While selecting the two-dimensional points, one of the grayscale images digitized earlier may also be displayed. Any two-dimensional point that is selected from the view currently displayed on the video monitor will be recorded and superimposed on the video image. The user can use this feedback to keep track of his progress. Once facets are formed, the points displayed on the screen are replaced by labels that can be used to directly choose three-dimensional points for subsequent facets, thereby bypassing the need to convert the points again from the two-dimensional views.

The final output of the system is a list of three-dimensional points and a set of triangular facets created from those points. These can then be written to a file and in turn be used as input to a graphics system that can, using hidden line removal, shading, etc., create a three-dimensional image.

3. System Requirements

3.1 Hardware Setup

Figure 1 illustrates that system's hardware set-up:

1. The IBM AT Microcomputer

Includes a keyboard and mouse for input, and a monitor used to display commands, system prompts and coordinate values. Equipped with an Imaging Technology, Inc. imaging board.

2. Scientific Accessories Inc. GP-8 2-d Digitizer

Provides an active area of 60" by 72" and a resolution of 0.01. Communicates with the IBM AT via an RS232 serial interface cable.

3. Video Monitor

Sony 12" video monitor for displaying grayscale images.

4. Video camera and stand

Monochrome vidicon camera with standard video output, equipped with a 25mm lens and mounted on a vertical copy stand.

3.2 Supporting Software

Werner Frei IMAGETOOL

Provides routines for grabbing, storing and displaying images, and for writing points to the video screen.

Microsoft C

Runs IVAN and is necessary for interacting with the FORTRAN IMAGETOOL routines.

Microsoft Mouse Library

Provides routines for locating the image boundaries on the video screen using a mouse.

Figure 1

4. The Image Views

Care must be given to the selection of the views that will be used for creating the three-dimensional model. Since each point represented on the object must appear in at least two views, the burden is on the user to provide the system with views that adequately display the features of the object that the user requires.

The user must also arbitrarily assign each view a number from 1 to 4 for the purposes of identifying the view. Each view's number must be used consistently throughout, since it also identifies the view's grayscale image, boundary definitions, exemplary points, etc.

5. Files

IVAN saves sessions in several different files. One is referred to as the parameter data file, and stores boundary definitions, exemplary points, and transformation matrices for each view. This file also stores information that indicates to the system what steps have thus far been completed, allowing those steps to be skipped at all subsequent sessions for that object. **Figure 2** is an example of this file.

The system output file stores the three-dimensional points and vertices that represent the object being modeled. These are grouped into distinct object sets. The first data line in the output file contains the number of objects sets that have been created. Then, for each set, the file contains a count of points and facets, a list of all the three-dimensional points that have been created, and a list of all the facets that have been formed. **Figure 3** shows a sample output file.

The display file stores the pixel locations for points that have been written to the grayscale images of the various views, and the location in the object list of the three-dimensional point that corresponds to each location. This is used to re-display any points that have been created at a previous session. These labels can then be chosen to create new facets during the current session.

Figure 4 shows an example of a display file, with 6 points for views 1, 4 points for view 2, 2 points for view 3, and no points displayed for view 4. The list of points for each view

```
 3.52   -11.39
13.46    -3.37
14.28   -11.38
24.23    -3.36
 3.69   -20.25
13.55   -12.10
14.48   -20.15
24.36   -12.01
```

view boundaries

5.08	0.79	6.05	0.00	-3.47
2.03	2.75	0.00	0.00	6.97
5.01	6.79	6.05	8.25	1.69
5.04	5.45	6.05	6.25	-1.28
2.81	5.71	1.88	6.13	5.88
8.11	2.87	12.03	0.00	6.97
7.13	5.96	9.91	6.30	5.78
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
1.57	3.52	0.00	0.00	6.97
5.52	6.40	6.05	8.25	1.69
5.57	7.15	7.53	8.25	4.38
4.19	6.88	4.47	8.25	4.38
6.21	4.68	6.05	6.25	-1.28
2.57	5.98	1.88	6.13	5.88
6.49	1.21	6.05	0.00	-3.47
6.49	4.66	12.03	0.00	6.97
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
2.29	1.71	6.05	0.00	-3.47
4.28	6.79	6.05	8.25	1.69
5.88	6.97	7.53	8.25	4.38
4.67	7.22	4.47	8.25	4.38
3.12	5.32	6.05	6.25	-1.28
4.21	6.62	1.88	6.13	5.88
8.37	2.77	12.03	0.00	6.97
7.43	5.91	9.91	6.30	5.78
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
4.95	1.47	6.05	0.00	-3.47
2.00	5.64	0.00	0.00	6.97
4.94	1.18	6.05	8.25	1.69
5.76	2.42	7.53	8.25	4.38
4.08	2.35	4.47	8.25	4.38
2.62	3.70	1.88	6.13	5.88
7.81	5.64	12.03	0.00	6.97
7.15	3.79	9.91	6.30	5.78
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00

view 1 exemplary points-

view 2 exemplary points-

view 3 exemplary points-

view 4 exemplary points-

		0.60	
		-0.02	
		0.17	
		1.28	
		0.00	
		0.58	transformation matrix - view 1
		0.26	
		1.59	
		0.00	
		0.00	
		0.03	
		0.51	
		-0.02	
		-0.14	
		2.76	
		0.14	transformation matrix - view2
		0.39	
		0.34	
		1.53	
		0.00	
		-0.01	
		0.01	
		0.17	
		-0.05	
		0.38	transformation matrix - view 3
		1.95	
		-0.27	
		0.30	
		0.24	
		3.77	
		-0.02	
		-0.01	
		0.02	
		0.38	
		-0.12	
		-0.20	transformation matrix - view 4
		2.82	
		0.03	
		-0.24	
		0.23	
		2.33	
		0.00	
		-0.02	
		-0.04	
# exemp pts		7	8 8
digitized?		1	1 1
image bounds		437	38
		441	31
		445	29
		445	34
view boundaries done?		1	
current command code		0	

Figure 2

```
  1
 16
 22
 4.6   3.8   2.2
 5.2   3.3   3.0
 5.3   3.1   3.0
 5.4   2.4   2.2
 7.3   4.3   1.5
 7.5   3.9   2.0
 6.8   3.5   2.2
 6.7   3.1   1.7
 4.4   5.6   2.0
 6.2   6.3   1.5
 4.3   5.9  -0.1
 6.1   6.3  -0.9
 6.6   1.6  -0.2
 7.6   1.6  -0.6
 6.9   1.3   2.1
 7.6   1.6   1.7
 5.3   4.8   2.4
 5.5   4.4   2.4
 5.2   4.5   2.4
 5.3   4.2   2.4
 5.5   4.1   2.4
 5.8   4.1   2.5
0  4  1
1  4  5
1  5  2
2  5  6
2  6  3
3  6  7
8  10  9
9  10  11
10  12  11
11  12  13
12  14  13
13  14  15
18  17  16
19  17  18
20  17  19
21  17  20
```

Figure 3

6		
189	204	0
249	230	1
187	246	2
253	266	3
242	185	4
300	218	5
4		
228	238	0
287	234	1
228	284	2
295	264	3
2		
328	208	4
269	237	5
0		

Figure 4

is proceeded by the number of points. The first two numbers on each line are the x and y coordinates of the pixel location, and the final value is the index into the three-dimensional object list.

IVAN also creates files for testing the accuracy of exemplary points, which will be discussed in greater detail in the Testing Exemplary Point Accuracy section.

Generally, when the system is instructed to save the current data, the appropriate file name or names is requested, depending on where the user is in the creation of his object, and the appropriate data written to each. Values not yet created for the parameter file are written as 0's.

The user can begin a session with IVAN by reading in a parameter data file that has been created during a previous session. Using a previously created parameter file implies that the view boundaries and exemplary points contained in the file still apply; that is, the views have not been moved. If the views have been re-positioned, all subsequent points selected will not be accurate.

If a complete parameter data file has been used to begin the current session, a previously created output file may also be read. Any new points and facets are then appended to those read in from the file, and subsequent SAVES will store both previous and new data in the file name provided.

If an output file has been read in, the user is given the option to amend the file by deleting facets that may have

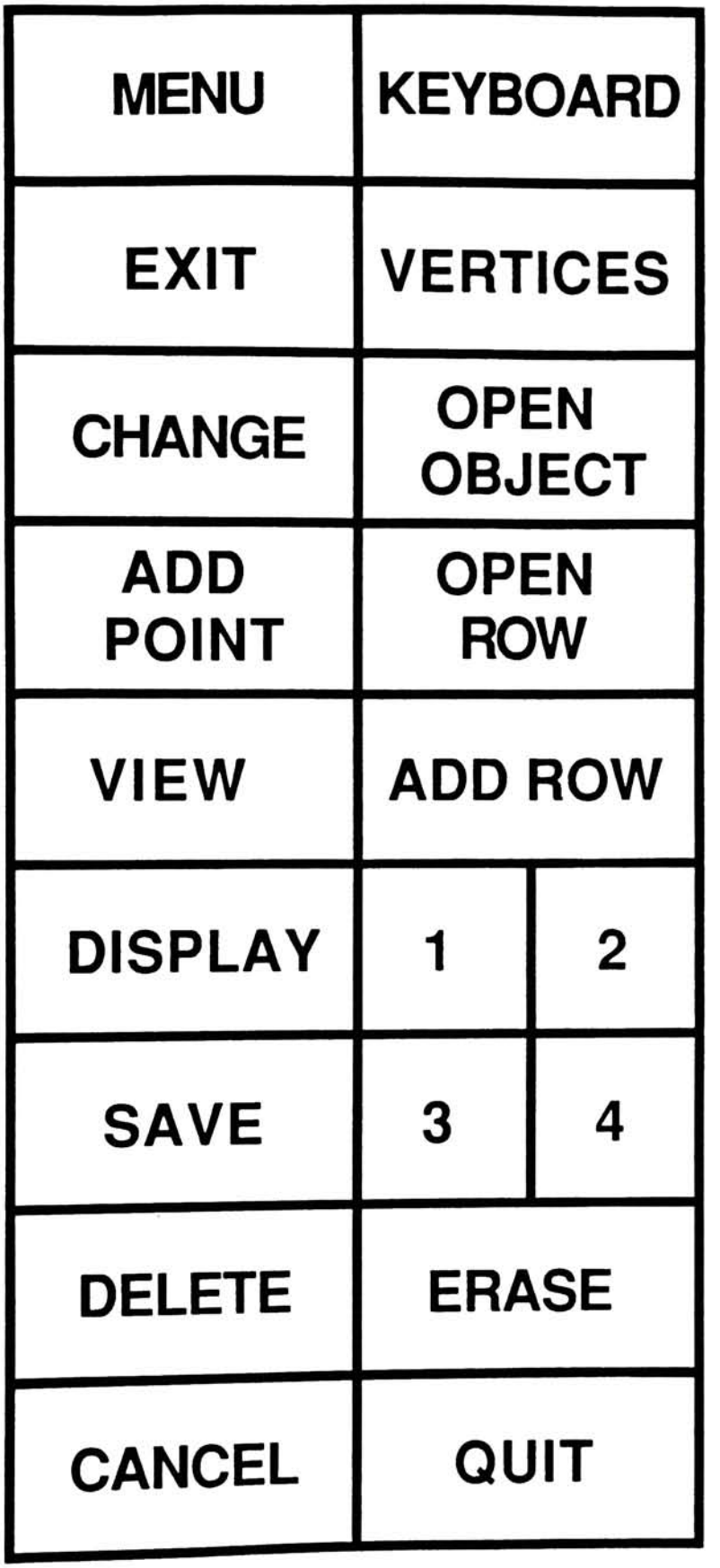
been incorrectly created. Reading in an output file will also result in a prompt for the name of the display file that corresponds to the current output file.

6. System Input

Ivan accepts input from both the keyboard and the digitizing tablet. Digitizer input is necessary for selecting both two-dimensional points from the various object views, and for selecting menu options. Input from the digitizer is requested with the prompt "!>", and is entered by clicking the stylus on either the desired two-dimensional point or on the appropriate area of the menu. A copy of the menu is found in **Figure 5**.

Keyboard input is requested with the prompt "->". It is implied that any keyboard input, except for <RETURN> itself, will be followed by a <RETURN>.

Incorrect input, from the either the tablet or the keyboard, results in an appropriate message and a "beep", sparing the user the need to constantly keep an eye on the screen to see if his work is progressing correctly.



Tablet Menu (3/1/89)

Figure 5

7. Getting Started

7.1 Starting Ivan

To start using the system, type **ivan** at the keyboard.

The system will respond with:

Welcome to IVAN.

7.2 Retrieving a Previous Session

Load parameter file? File name or <RETURN>.
->

The system will begin by requesting the name of a file containing parameter data saved from a previous session.

In response to the prompt, type the file name if one has been previously saved. The name should be no more than 8 characters in length. If no previous session has been saved in a file for the current image, hit **<RETURN>** after the prompt.

Read from what output file?
->

This is a request for the name of a previously created output file for the current object. If no such file exists, hit **<RETURN>**.

Edit output file? y/n
->

The system provides this opportunity for the user to delete any facets that have been created incorrectly. Entering "n" or <RETURN> will bypass this option; entering "y" will provide a prompt for the facet to be deleted.

Enter the indices for the 3 points that make up the facet to be deleted, one on a line. RETURN when done.
->

This prompt requests the 3 vertices that comprise the facet to be deleted. Note that the vertices are represented by the location of the points in the object list, just they are represented in the output file. Each vertex should be followed by a <RETURN>. An invalid vertex will result in the error message "Facet not found". When there are no facets to delete, <RETURN> ends the file editing session. Any subsequent saves will now write the amended file.

Read from what display file?
->

This requests the name of the display file that corresponds to the output file that has just been read in. <RETURN> indicates that no display file is to be entered.

7.3 Grabbing the Views

Object name?
->

Using the video camera mounted to a stand, the user is now ready to "grab" any of the object views, producing a grayscale image that will provide feedback later in the process. The camera should be turned on, and the lights adjusted for proper illumination of the views.

In response to the prompt, enter the name of the object to be modeled, up to eight characters in length. The system uses the object name to locate and read the appropriate image for the current object. If no name is entered, no display feedback will be available.

Digitizing which view? <RETURN> when done.
->

The system now requires that the user enter a number from the keyboard, 1 - 4, that represents one of the four views that the user would like to grab. The user may grab as many of the four views as he likes, or none at all.

If the views have already been grabbed and stored at a previous session, or if none are desired, <RETURN> will move the user ahead to the creation of exemplary points.

Camera in continuous digitizing mode. Set focus, F/stop and camera distance. Enter <RETURN> to continue.
->

The imaging component of the system will now begin producing continuous video output. The view corresponding to the number entered should then be placed under the camera, and adjusted so that all four edges of the view are visible on the video monitor.

<RETURN> will cause two perpendicular lines to appear on the video monitor. The lines cross at the lower left hand corner of the screen and serve as a guide for proper alignment of the photograph.

Place the photograph so that the lower left corner is aligned with the horizontal and vertical lines shown. Proper alignment is critical for accurate display. When aligned, hit <RETURN>.
->

Taping the four corners of the photographs to a heavier piece of cardboard will make it easier to manipulate them during this process. The user should keep in mind that correct alignment is very important to ensure that points are accurately written to the screen.

<RETURN> will grab the image on the screen.

Move the mouse until the cursor is on the upper right corner of the view. Click the mouse when properly positioned.
->

When the mouse is clicked, the location of the cursor will be stored as the upper-right boundary for the image. This allows tablet-to-screen coordinates to be calculated.

Writing file **filename** to disk. Please wait.

The system then stores the grayscale image in a file. The name of the file will be automatically created by the system from the object name given earlier and the view number.

For example, view 1 of an image with the name CART will be stored in a file named CART_1, view 2 of the same image in CART_2, etc.

The system then returns to request another view number, and the above procedure is repeated for each of the views the user would like to grab for future display. When all views have been grabbed, enter <RETURN> at the request for a view number. After the <RETURN>, the system will list which of the views have been grabbed.

Save the image boundaries? File name or <RETURN>.
->

Entering a file name will save the image boundaries that were located for each view. These are necessary for the proper display of points on the screen.

Saving these values now is advised if the current session will end here; otherwise, the views will have to be grabbed again so that the boundaries can be located. However, saving data later during this same session will result in these values being stored as well.

8. Preparing the Tablet

8.1 Placing the Views

Afix the four image views to the tablet. Caution: once placed, they must remain in that position. Failure to do so will result in all points having to be re-entered.__<RETURN> when done.
->

Unless the user has already done so, the four views of the object should now be attached to the tablet. The views should be placed on the left side of the tablet, leaving enough digitizing area to the right for the menu. The photographs should be placed parallel to the tablet axes.

8.2 Placing the Menu

Afix the menu to the tablet, to the right of the views, but within the range of the digitizer microphones. Align the menu carefully, or the system will request that it be re-positioned.
 <RETURN> when done.
->

The menu should be attached to the tablet, keeping in mind that the photographs and menu cannot overlap. **Figure 6** is an illustration of what the arrangement should look like.

Select the lower left corner of the menu.
!>

Selecting the lower left and then the upper right menu boundaries with the digitizing stylus will define the position of the menu, and allow the system to calculate the position of the various menu options. Note the prompt for tablet input.

Select the upper right corner of the menu.
!>

Before calculating the position of the menu options, the system must determine if the menu is properly aligned. If it is not, the system will print an error message, ask that the menu be adjusted, and then prompt again for the position of the menu's corners. This process is repeated until the menu is properly aligned.

Figure 6

9. Defining View Boundaries

Before the points in the two dimensional views can be selected, boundary values that define a rectangular view area for each view must be defined. The system will first print the current boundaries; either those boundary definitions read from the file of a previous session, or 0's if no boundaries have been defined.

If view boundaries have been read in, the system will go on to the definition of exemplary points. There is no opportunity to change the boundaries once they have been defined, since all subsequent points are dependent on their values.

9.1 Selecting view boundaries

```
Select boundaries for which view?  1, 2, 3, 4?  EXIT when
done.
!>
```

If no boundaries have been read, the system begins the process of defining view boundaries by prompting for one of the views numbered 1 - 4, corresponding to the numbers 1 to 4 on the tablet menu. After selecting a number with the tablet stylus, the system will request:

```
Select the lower left corner of view #.
!>
```

Once the user has clicked his digitizing stylus on the lower left corner of the view, the system requests:

```
Select the upper right corner of view #.
!>
```

The stylus should now be clicked on the upper right corner of the same view.

9.2 Exiting

The system will repeat the request for view numbers and the corresponding view boundaries until the menu option **EXIT** is selected.

Once **EXIT** is chosen, the final boundaries are displayed.

```
CHANGE view boundaries or EXIT?
!>
```

This last step of defining view boundaries allows one final chance to change the boundary values. **CHANGE** allows the user to go back and re-define the boundaries, **EXIT** indicates acceptance of the values. Since, as stated earlier, all subsequent data points are dependent on the values of the view boundaries, it is important that these boundaries be correct.

```
Save view boundaries?  File name or <RETURN>.
->
```

If the boundary definitions are accepted, the system requests the name of a file for saving the boundaries for use at a later session. If the values are not saved, the boundaries will have to be redefined at the next session, and all points will have to be selected again.

9.3 Additional Menu Options

DISPLAY - displays the current view boundaries for the view given.
SAVE - saves the data in the file named.
MENU - displays a list of the menu options available.
QUIT - ends the session.

9.4 Error Messages

Invalid selection - Checks for valid menu options, and that view boundary values do not lie on the tablet menu.

10. Defining Exemplary Points

The next step is the definition of exemplary points. Whether or not previously defined exemplary points have been read in, the system begins by allowing the user to enter or change the current exemplary points for any of the views.

Exemplary points are three-dimensional points represented in world coordinates, and their corresponding points on the two-dimensional views. At least six exemplary points are needed for each view, with a maximum of 15 points for each allowed.

The three dimensional component of each exemplary point must be measured or otherwise taken from the object, or from the scene in which the object appears. This entails choosing an arbitrary origin and axes in three-dimensional space, and then determining the three-dimensional coordinates of points on either the object or features in the scene.

Although at least six exemplary points are needed for each view, a point in real-world coordinate space can be shared by all the views in which it appears. Choosing 3-d coordinates that appear in all four views will require that a minimum of six real-world points be created. Regardless of the number chosen, they must not all be coplanar in three dimensions.

Which exemplary points are chosen will have a significant effect on how accurately three-dimensional points are created by the system. Choosing points that are as widely distributed as possible over all three dimensions will tend to increase accuracy.

10.1 Selecting Exemplary Points

ADD or DELETE exemplary points for views. Exit when done.
!>

ADD allows the user to add exemplary points for any view; DELETE allows deletion of points from any view. Since six points per view are required to calculate the transformation matrix, EXIT should not be chosen until all four views contain at least six points each.

10.2 Menu Options - Selecting Points

10.2.1 ADD POINT

Choosing the ADD POINT option on the menu allows the user to continue adding exemplary points until EXIT is chosen.

Enter x, y and z values in world coordinates from the keyboard, one value on a line. Enter <RETURN> when done adding points.
->

The system begins the process of collecting exemplary points by asking for a point in world coordinates, taken from the object or from the scene in which it appears. Note that the x, y and z value should be entered separately, one on a line.

Select a 2-d point from each view that corresponds to the 3-d point just entered. Choose EXIT when all views for that point have been chosen.
!>

A two-dimensional point that is the projection of this three-dimensional point on any view may be selected. The view itself need not be indicated since the system can determine the view using the boundary definitions.

After the point is chosen, a list is displayed of all the exemplary points thus far chosen for that view. At least six points are displayed; if fewer than six points have been chosen, the rest are shown as 0's. This is a reminder that six points are required, and makes it easy to see how many more are necessary.

Notice that the two-dimensional points are not absolute digitizer values, but have been converted to be relative measurements for each view.

After a point is selected, the prompt is repeated until the user has selected the same point in all the views in which it

appears. When all the 2-d points for the current three-dimensional point have been exhausted, **EXIT** will result in the return of the prompt for a three-dimensional point, and the process begins again. When the user is finished entering points, **<RETURN>** should be entered at the prompt.

10.2.1 DELETE

DELETE also remains in "delete" mode until **EXIT** is chosen.

DELETE points on which view? 1, 2, 3, 4? EXIT when done.
!>

First the view from which the point is to be deleted must be chosen from the tablet options 1 to 4. Choosing a view will cause a list of the exemplary points for that view to be displayed.

Enter location of 2-d point to DELETE from the keyboard.
<RETURN> when done deleting points for this view.
->

This is a request for the position in the list of the point to be deleted (where 1 is the first, or top, position). Once the location of the point has been entered, the point is deleted and the list of points is displayed again.

Entering **<RETURN>** will return to the delete prompt without deleting a point, and **EXIT** will return to the add or delete option.

10.3 Exiting

One more **EXIT** on returning to the ADD or DELETE prompt will indicate to the system that all exemplary points have been defined. The system will then calculate the transformation matrix for each view. If too few exemplary points for a view have been chosen, the system responds with a message, and allows the user to add more points.

10.4 Additional Menu Options

DISPLAY - displays the exemplary points for the view given.

SAVE - saves the data in the file named.

MENU - displays a list of the menu options available.

QUIT - quits the session.

10.5 Error Messages

Invalid selection - choosing an invalid menu selection, or digitizing a two-dimensional point that does not lie within one of the view areas.

Maximum number of exemplary points reached - attempting to create more than 15 exemplary points.

Invalid deletion - attempting to delete an exemplary point that has not been stored.

Insufficient number of exemplary points for view # - creating fewer than six exemplary points for a view.

Point already stored - attempting to store an exemplary point that has already been stored.

No point stored at that location - attempting to delete a point at a location in which there is no point stored.

10.6 Testing Exemplary Point Accuracy

Two different options are available for testing exemplary points to determine how accurately they are creating three-dimensional points. The first option allows the user to create three-dimensional points using the transformation matrices created for each view, the same way it would be done for creating object points in three-dimensions. However, by asking the system to create three-dimensional points for which the three-dimensional coordinates are already known, the measured and created points can be compared to see how accurately points are being

formed. The system assists by allowing the user to label each point that is created, and by automatically calculating the difference between the measured and created points. The exemplary points that have been used to create the transformation matrix for each view can be tested in this way as well.

Test exemplary points? y/n
->

The system first asks the user if exemplary point testing is desired. Entering a 'y' at the prompt from the keyboard allows access to the first testing option. <RETURN> will skip the testing options completely.

Enter X, Y and Z values for a point to be tested, one value on a line.
->

The measured three-dimensional coordinates for a point should be entered from the keyboard. The system will use these values to determine the difference between the measured and created points. Entering <RETURN> indicates that no more points are to be tested.

Enter a character representing a point for testing.
->

The character will serve as a label so that the user can identify the point that is being created.

The system will now allow the user to select two-dimensional points that correspond to the three-dimensional point that has been chosen for testing. The points can be selected from any two or more views, and as many times as is desired. For an explanation of how to create three-dimensional points from view pairs, see Section 11.1 Creating 3-d Points.

When the current point has been tested sufficiently, EXIT will allow the user to choose another measured three-dimensional point, and its corresponding label, and then create points for this new measured point. Entering <RETURN> will indicate that

the user has completed this testing option.

The second testing option allows the user to remove one exemplary point at a time for a particular view, and to then create points as in the first option, but with a transformation matrix that has been formed without the removed point for that view. Choosing this testing option requires that at least seven exemplary points be available for a view to be tested, since six are required to form the transformation matrix.

Enter a view number.
!>

The system begins by requesting a view number from the menu. Choosing **EXIT** will end the testing session.

Enter location of a point to be removed from list.
->

The list of exemplary points for the given view is displayed, and the user is requested to choose a point to be removed by indicating its location in the list. A transformation matrix is then formed from the remaining points. <RETURN> will indicate that there are no more points to be removed for this view.

The system now follows the same procedure as in testing option one; i.e., requesting the measured coordinates for a three-dimensional point, and then a label to identify a point that is being created. The user can again create three-dimensional points from the view pairs, using transformation matrices with different points removed, and can compare how accurately different points are created by each matrix.

Note that each view must be tested separately. When one view is completed, entering <RETURN> at the prompt for a location in the exemplary point list will result in a prompt for another

view. Since only one view can be saved at a time, the **SAVE** option should be chosen before going on to the next view.

Data on each tested point that is either **DISPLAY**ed or **SAVE**d includes the point that has been removed (0 for test option 1), the label for the point that is being tested, the views that were used to create the three-dimensional point, and the difference in x, y and z between the measured point and the created point. An example of the files created for test options 1 and 2 can be found in **Figure** 7 and 8, respectively.

10.6.1 Additional Menu Options

DISPLAY - displays data for all tested points for test option 1 and for the view given for test option 2.

SAVE - saves the testing data in the file named.

MENU - displays a list of the menu options available.

QUIT - ends the session.

10.6.2 Error Messages

Invalid point to test - attempting to remove a point from the exemplary point list that does not exist.

Must have more than 6 points to test - Attempting to remove a point for a view that has fewer than seven exemplary points.

Removed	Tested	Views				Delta X	Delta Y	Delta Z
0	*	1	1	0	0	-0.01	0.27	0.04
0	x	0	1	1	0	0.01	0.15	0.03
0	+	1	0	0	1	-0.18	-0.10	0.00
0	]	0	0	1	1	-0.22	-0.04	0.12
0	a	1	1	0	0	-0.04	-0.12	0.14
0	a	1	0	1	0	0.08	0.18	0.10
0	a	1	0	0	1	-0.08	-0.26	0.10
0	a	0	1	1	0	-0.20	0.00	0.12
0	a	0	1	0	1	0.79	-0.93	0.25
0	a	0	0	1	1	-0.30	-0.13	0.08

Figure 7

View: 1

Removed	Tested	Views				Delta X	Delta Y	Delta Z
1	b	1	1	0	0	-0.07	0.13	0.15
2	b	1	1	0	0	-0.43	0.60	0.03
3	b	1	1	0	0	-0.10	0.22	0.16
4	b	1	1	0	0	0.02	0.26	0.00
5	b	1	1	0	0	0.07	0.10	0.15
6	b	1	1	0	0	0.01	0.22	0.06
7	b	1	1	0	0	0.05	0.15	0.10
1	b	1	0	1	0	0.73	1.46	0.36
2	b	1	0	1	0	-0.13	0.25	0.02
3	b	1	0	1	0	0.67	2.56	-0.32
4	b	1	0	1	0	0.96	1.91	0.25
5	b	1	0	1	0	0.74	1.46	0.24
6	b	1	0	1	0	0.68	1.50	0.21
7	b	1	0	1	0	1.22	2.17	0.29
1	b	1	0	0	1	-0.05	0.00	0.10
2	b	1	0	0	1	0.92	0.42	0.00
3	b	1	0	0	1	-0.33	0.16	0.29
4	b	1	0	0	1	-0.07	0.16	-0.01
5	b	1	0	0	1	-0.01	0.00	0.17
6	b	1	0	0	1	-0.05	0.08	0.05
7	b	1	0	0	1	-0.05	0.05	0.12

Figure 8

11. Creating a Three-dimensional Object

The system is now ready to accept two-dimensional points, entered from the digitizing tablet, to create a three-dimensional model.

11.1 Creating Three-dimensional Points

To calculate the coordinates of a three-dimensional point, the same two-dimensional point must be selected from at least two different views, and as many as all four. After selecting a single point in its various views, the **EXIT** option is chosen. A three-dimensional point is then calculated from the two-dimensional points that have been selected.

If a point is selected from only two views, the transformation matrices for those two views are used to calculate the three-dimensional point. For points chosen on more than two views, a point is created for each pair of views, and then an average is calculated from the various points formed. Once the calculation is completed, the next set of points can be selected.

11.2 Displaying User Feedback

The user can keep track of the points he has chosen by displaying them on the image screen. First, a view that has been previously digitized is chosen for display using the **VIEW** menu option. Once the view appears on the screen, all selected points that are visible on that view will appear superimposed on the screen. The points appear as a white dot surrounded by a

black cross-hair.

Once facets have been created from the current set of points, each displayed point is replaced by an integer label. If the background is dark, the numbers appear white; if it is light, they are black. Each label represents the location of the three-dimensional point in the object list that corresponds to the displayed two-dimensional point for that view. The labels can be used to create new facets from three-dimensional points that have already been created. This will be discussed in greater depth below.

The user may decide to display a different view at any time. The new view he chooses will replace the old view on the screen, and all the points that he had previously chosen for this new view will be automatically superimposed on this newly displayed view as well.

11.3 Creating Three-dimensional facets

The system arranges the points it creates into three-sided "facets". The three vertices of each facet are formed from the three-dimensional points created by the process explained above.

The table below shows an example of points and facets. Each line of output represents the three coordinates x, y and z, of a point. The facet vertices are represented by the location of each point in the list of points. For example, in the table below we have four points. Point 0 is on the first line, point 1 is on the second line, etc. The two facets are formed by points 0, 2, 1 and points 1, 2, 3.

```
 8.2  -0.7  8.4
 2.5   0.1  4.3
12.0   5.4  8.9
 5.8   5.1  3.9

0 2 1
1 2 3
```

All facets are formed by the system in a counter-clockwise direction. In order to form facets correctly, the user must follow several rules in selecting his points. In **OPEN ROW SET/ ADD ROW** mode, all points in a row set must be visible on at least one view. Choosing this view as the "reference" view, the user must enter all rows from top to bottom, and all points from left to right on his reference view. Proceeding in this way will cause all facets that are visible in a given view to appear counter-clockwise, and, conversely, all facets that are invisible from a particular view will appear to be clockwise.

VERTICES mode works in a similar fashion. A reference view must be chosen for each three-sided facet, and the facet must be selected in a counter-clockwise direction.

11.4 Choosing Three-dimensional Points Directly

In addition to choosing points for both rows and vertices by using multiple two-dimensional views, the user may choose the points that have already been converted to three dimensions directly. Once facets are formed for a set of points, the number representing each point's location in the object list is written on the screen in the point's corresponding location on the displayed view. Entering the point labels directly from the keyboard allows the user to bypass the more time-consuming process of re-creating the points from two dimensions. This can be very convenient, particularly for features that share an edge with a facet that has been created previously.

To choose a labeled point for the current set of points, first the menu option **KEYBOARD** is chosen.

Choose a numbered point from the display screen.
->

> The user responds by typing in a labeled point. The systems retrieves the point from the list, and point is now treated just as it would have been if it had calculated from the two-dimensional views. The prompt returns after the point has been accepted until the user responds with <RETURN>. If the label entered does not appear on the screen, an error message results.

11.5 Creating Three-dimensional Objects

An object to be modeled is represented as a list of points and a list of facets. These are grouped together into distinct object sets. An object can be made up of one or more object sets; however, before points can be created, a set must be "opened" so that points can be stored in a list for that set. The points for different object sets are unrelated to each other, and each object set is given a number to distinguish it from other sets.

11.6 Menu Options - Creating Objects

Notice that all available options are displayed after each option is chosen. Choosing an option other than those listed will result in an INVALID SELECTION error.

11.6.1 OPEN OBJECT

```
New object opened.
!>
```

Before points can be selected, a new object set must be begun by choosing **OPEN OBJECT** on the tablet menu. However, if an object set has been opened during a previous session and the output file from that session has been read in, points will automatically be appended to the object set most recently opened.

11.6.2 VERTICES

Begin entering points to form 3-sided facets.
!>

Choosing **VERTICES** allows the user to form three-dimensional points into three-sided facets directly. All points that form a facet must be visible in the reference view, and points must be chosen in a counter-clockwise direction.

Every three points created will be automatically formed into a facet. Each facet formed in this way is unrelated to others, and the points for facets with shared edges must be entered separately.

EXIT allows the user to exit from **VERTICES**. If **EXIT** is chosen without completing all three vertices of a facet, the system responds with a error message and the user may either **DELETE** points or complete the facet.

11.6.3 OPEN ROW

Begin choosing points for the first row.
!>

Choosing **OPEN ROW SET** allows the user to select points to form a row. This first row is known as the first row in the current row "set". Once this first row has been chosen, any number of contiguous rows can be added, provided that all the rows have the same number of points, by choosing **ADD ROW** for each additional row. There must be at least two rows in a row set and at least two points in a row.

For rows with an unequal number of points, the same point or points can be selected more than once in the row with fewer points. This technique can be used to create both triangles and circles, the latter by choosing one row on the perimeter of the circle, and choosing the center of the circle as the second row.

All rows in a row set must be completely visible in one view. Rows selected in this reference view must be formed from top to bottom on the view, and points in the row selected from left to right.

11.6.4 ADD ROW

Begin selecting next row in current set.
!>

Each time **ADD ROW** is chosen, the points in the two most recently selected rows are connected automatically to form facets.

11.7 Exiting

EXIT will end the current row set, create the last set of facets, and remove the user from the **OPEN ROW/ADD ROW** option.

The user can choose freely between the **OPEN ROW SET/ADD ROW** options and the **VERTICES** option, as long as the user **EXIT**'s from one before choosing the other. Failure to do so will result in an error message.

11.8 Correcting User Errors

11.8.1 ERASE

Most recent 2-d point erased.
!>

The most recent two-dimensional point chosen in a view may be deleted with the **ERASE** command. This will also erase the point from the view currently displayed. Only two-dimensional points that have not yet been used to create a three-dimensional point can be erased.

11.8.2 DELETE

Most recent 3-d point deleted.
!>

The most recent three-dimensional point calculated may be deleted by using the **DELETE** option. Only three-dimensional points that have not yet been formed into facets can be deleted. A delete will erase the point on the currently displayed view image, as well as from all other views on from which the point has been selected.

11.8.3 CANCEL

```
Last command cancelled.
!>
```

The **CANCEL** command is also available. This command will cancel the **OPEN OBJECT, OPEN ROW SET, ADD ROW** or **VERTICES** command, if any of these was the most recent command given, and will restore all values to what they were before the canceled command was given.

11.9 Additional Menu Options

DISPLAY - displays the list of current two-dimensional points that have not yet been converted to three-dimensions, the list of three-dimensional points for the current row, and the points and facets already defined.

SAVE - saves the final list of points and facets in the file named.

MENU - displays a brief explanation of the available menu options.

QUIT - ends the session.

11.10 Error Messages

Invalid selection - choosing an invalid menu selection, or digitizing a two-dimensional point that does not lie within one of the view areas.

Points chosen on fewer than two views. Recent selected point ignored. - trying to EXIT after choosing a point from only one view. The point that was selected in automatically erased.

Point from view already chosen - attempting to select two two-dimensional points on the same view.

Row length not equal - attempting to define a row with a length that is not equal to the length of the current row set.

Fewer than two points for row - attempting to define a row set with fewer than two points for the row.

Unfinished facet - attempting to **EXIT** from the **VERTICES** command with only 1 or 2 points created.

No 3-d points to delete - attempting to delete a 3-d point if none currently exist in the 3-d stack.

No 2-d points to erase - attempting to erase a 2-d point if none currently exist in the 2-d stack.

Invalid point number - Entering a point label for which no point exists.

12. Final Output

The final result of the above procedures is the output file described in Section 5 and illustrated earlier in Figure 3. Note that only object points that have already been created into facets are stored in the file; all other two- and three-dimensional points are lost.

The final output file can be used as three-dimensional input to a graphics package for further display and manipulation of the three-dimensional object that has been created.