A MICROCOMPUTER GRAPHICS PACKAGE FOR USE WITH A HIGH-RESOLUTION RASTER-SCAN DOT-MATRIX PRINTER

by

EARL F. GLYNN II

B.S., Kansas State University, 1975

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1982

Approved by:

Major Professor

TABLE OF CONTENTS

Contents (continued)

1. Introduction

1.1 Purpose, Motivation and Objectives

A general purpose graphics software package was developed for use with a low-cost "personal" microcomputer system without the use of any special purpose hardware. This project was motivated by the acquisition of a graphics printer without adequate software to control its many features.

Software was developed to provide a mechanism to define a picture in terms of a logical screen -- which may or may not correspond to a video display screen -- which could be mapped to a graphics printer. Ideally, the logical screen would be mapped to either a video display terminal or a hard copy device.

The initial implementation supports only "black" and "white" pixels but the software was designed to support a more complex pixel definition. Two- and three-dimensional graphic primitives support user picture definition. Various vector/matrix operations support mathematical transformation of the picture. While the software package was developed for one particular graphics printer, other printers could be used by changing some internal constants and variable definitions and recompiling. Output to a video display terminal is also possible but is not included in this initial implementation.

In this report we assume the reader is either knowledgeable about computer graphics or has access to various textbooks on the subject. Detailed explanations of graphic operations are not included in this report. [NEWM79] and [FOLE82] are excellent references. The following are helpful and interesting: [ANGE81, GILO78, POSD77, ROGE76, SCOT82, WHIT82].

1.2 Background

In the last few years dot-matrix printers have been introduced for high-quality word processing and graphics applications. These printers have a variety of dot resolutions such as 74-by-72 (Centronics 739) 72-by-60/120 (HP 82905), and 84-by-84 (Integral Data Systems 460 or 560) dots per inch (DPI) for print areas as wide as 13.2 inches. In the last year new "quad density" printers have been announced. The NEC 8023 has 144-by-60 DPI resolution; the C. ITOH 8510 has 144-by-144 DPI; the Axiom IMP-4 has 19008 dots/square inch; and an upgrade to the Digital LA 120 offers 165 DPI resolution. These printers and others are currently selling from about $350 to $1500. In early 1982 Integral Data Systems announced a four-color ribbon with its high resolution "Prism" printer [UMLO82]. Mixing allows eight possible colors. The printer sells for about $2000. Higher resolutions and greater color capabilities will probably be developed in the near future.

These dot matrix printers have resolutions that favorably compare to the resolution available on video display terminals currently available. These printers are cheaper than the display terminals and provide hard copy output directly, however, these printers are much slower than display terminals.

1.3 Software Overview

The user defines a picture using world coordinates -- any convenient coordinate system. If desired these world coordinates can be manipulated mathematically using translations, rotations or scaling before they are converted to screen coordinates. The logical screen is defined to be a matrix of picture elements -- pixels. Each pixel is small enough (about 0.01 inch by 0.01 inch) that a circular dot represents what is actually a

rectangular area.

The screen coordinates of points and lines are defined by a set of pixels. Clipping may be necessary to eliminate points or lines which extend outside the logical screen pixel matrix.

The user defines the actual physical size of the logical screen and, therefore, implicitly defines the maximum size of the plot file. The maximum width of the picture is limited horizontally by the printer line length (in practice but not in theory) but vertically only by available diskette storage. Since the number of pixels in the matrix could reach two million or more, and each pixel requires at least one bit of storage, the entire pixel matrix requires perhaps 250 Kbytes of storage. This amount of memory is not generally available on most micros (but probably will be in a few years). To accommodate a potentially very large pixel matrix on existing micros, a demand paging system was developed so that as many pixels as possible can remain in memory for manipulation. Diskettes for micros typically hold 92-256 Kbytes but hard disks with capacities of 5 Mbytes or more are entering the microcomputer market. Efficient diskette storage and in-memory data structures allow some data compression and creation of some pictures which otherwise would exceed available diskette storage.

2. A Graphics Tutorial

Computer graphics is the creation, storage and manipulation of models of objects and their pictures via a digital computer. Computer graphics is used in such diverse areas as mathematics, medicine, architecture, engineering, chemistry, cartography, business, word processing, art, animation and entertainment. Computer graphics is

becoming the preferred interface between humans and computers instead of being considered a special form of communication requiring special I/O software and hardware. Data presented pictorially can be perceived and processed by humans more rapidly and efficiently than textual data. Graphics systems will be more widely available as microcomputer hardware costs continue to tumble. One is prompted to look for computer graphics in unexpected places within the home, office and laboratory in the next few years.

2.1 Interactive Versus Passive Computer Graphics

Interactive computer graphics allows a user to dynamically control a picture's context, format, size or color by means of interaction devices such as a keyboard, lever or joystick. Video displays such as CRTs and TV sets are used to show the picture dynamics. Points and lines must be continuously updated to add a dimension of time to the display. Applications of interactive graphics include flight training simulation, computer-aided design and video games.

Passive computer graphics is involved when using an impact printer or a drum or flatbed plotter. The user controls the picture creation but does not have real-time, dynamic options. Passive graphics is easier to implement than interactive since event handling is not necessary. Processor speed for passive graphics is not as critical since points and lines are not continuously updated. Passive graphics is all that is required in many applications, e.g., graphs, pie charts, histograms, flowcharts, architectural diagrams and circuit schematics.

The remainder of this tutorial emphasizes subject areas used in both interactive and passive graphics. Certain areas of interest only to interactive graphics (e.g., event handling) are not addressed since the

software system was developed to operate as a passive computer graphics package.

2.2 Vector Versus Raster Display

The term "vector" is not used strictly in the mathematical sense of an n-tuple of location coordinates. Graphics literature uses "vector" to describe a line segment or the process of drawing a segment.

Pictures can be created by a vector system with a random-scan -- segments are displayed in any order. A pen plotter in which a pen can be moved in any direction over a piece of paper is a random-scan vector device.

In a raster-scan system a drawing is divided into horizontal lines. Each raster scan traces out a small strip of a picture. U.S. TV sets, for example, have 525 lines and most CRT raster systems use between 256 and 1024 lines. The more lines, the higher the picture quality.

With a black-and-white CRT raster device, a raster scan is a left-to-right sweep of the electron beam which is modulated to create different shades of gray. There is a one-to-one mapping of a memory location to each small segment of the raster scan. Each of these small segments is called a "pixel" -- a picture element. A picture is therefore a matrix of pixels. See [NEWM79, Chapters 15-19] or [FOLE82, Chapters 10-12] for details of raster graphics.

Dot-matrix impact printers are also raster devices. Each pass of the print head traces out typically 7 to 9 rows of dots to form text characters. In the last few years microprocessor-controlled, dot-addressable, impact printers have been introduced as graphics devices.

The IDS 560 printer has a graphics resolution of 84 dots per inch both vertically and horizontally. Dots are formed by print head wires 14

mils in diameter; the printed dot is anywhere between 15 and 17 mils in diameter due to inherent variations in paper hardness, humidity and ribbon wear. Dots are printed on 1/84th-inch centers, about 12 mils apart. (A mil is 0.001 inch).

In graphics mode each pass of the IDS 560 prints 7 rows of dots. Each column of 7 dots within each row is mapped from a byte of memory. Given a byte with bits 7 to 0 (bit 7 the most significant and bit 0 the least significant), bit 0 is mapped to the top row of the raster scan while bit 6 is mapped to the bottom row of the scan. Bits 1 through 5 are mapped in between. Bit 7 (typically the parity bit) is ignored. Other details of the IDS 560 graphics mode can be found in [IDS81].

While the IDS 560 prints 7 rows of dots per raster scan, the microcomputer graphics package was developed to operate as a random-scan vector system.

2.3 Two-Dimensional Picture Definition

Output from graphics systems is in a two-dimensional form whether on a CRT screen, a drum or bed plotter, or an impact printer. This section will introduce concepts and algorithms used in two-dimensional graphics. Three-dimensional graphics involves projections into two dimensions and will be discussed in Section 2.5.

2.3.1 World Coordinates and Windows

The term "world coordinates" is used to describe the Cartesian coordinate system used by a user. The units of the world coordinates can be anything appropriate for a problem definition, e.g., inches, meters, gallons, liters, pounds, newtons, etc. A user should concentrate on the definition of the entities to be plotted and should not be overly concerned about conversions which are automatically performed by a

graphics package.

Many graphics systems require the user to specify coordinates in a device space -- the coordinates needed by the display hardware. The user's data is rarely within the same range as needed by the display hardware and must be mathematically manipulated to fall within the desired range.

The rectangular area bounding the extents of a user's world coordinates which defines the desired picture is called the "window". (Points outside the "window" cannot be "seen" by the graphics software, i.e., they are ignored.) Without an appropriate transformation, the default "positive" direction of the coordinate axes is left-to-right for the "x" axis and bottom-to-top for the "y" axis. A canonical space ranging from 0.0 to 1.0 in both the "x" and "y" Cartesian dimensions is an appropriate default window. However, a user can specify any window by setting the desired minimum and maximum values of the "x" and "y" dimensions.

2.3.2 Points and Lines

The entities described by a user in world coordinates consist of a set of points and lines. A point in world coordinates is mapped into a point in device space. If the dimensions of device space do not correspond to world coordinates, then a simple translation and/or scaling operation will allow world coordinates to be mapped to device space.

A line segment consists of two endpoints and all the collinear points in between. Transforming a line segment from world coordinates to device space is simple: The endpoints of the segment in world coordinates are each mapped to device space. The segment in device space consists of these transformed endpoints and all the collinear points in

between.

When using a random-scan vector device, the points or line segments in device space can be directly plotted. No calculations are necessary to define intermediate segment points. This is not true when using a raster-scan device with a picture consisting of a matrix of pixels. The transformed segment endpoints map directly to specific pixels but all the intermediate pixels between the endpoints must be individually selected.

2.3.3 Clipping

Unless a user is extremely careful lines may extend outside the defined window area. The portion of the line outside the window must be clipped. A common method used for line clipping is the Cohen-Sutherland algorithm. See [NEWM79, pp. 65-67] or [FOLE82, pp. 146-149] for a detailed description of this algorithm. (The Pascal programs in both texts are essentially the same. [FOLE82] "fixes" the single "goto" of [NEWM79].) A brief description follows.

This algorithm first considers the regions in which the line endpoints lie. These regions classically have been assigned binary codes as shown in both [NEWM79] and [FOLE82]. The binary codes seem to complicate the discussion. In this report we replace a set of region codes with a set of directions as if the window were a map surrounded by regions. For example, the window '0000' becomes [], '0010' becomes [east], '0110' becomes [south east], '0100' becomes [south], etc. Consider the rectangular window and the sets of regions shown in the diagram below:

[north west]	[north]	[north east]
[west]	[]	[east]
[south west]	[south]	[south east]

If the set of regions a point lies in is the empty set [], the point is contained within the window. If the set of regions is non-empty, the point lies outside the window. If the union of regions from both endpoints of a line is the empty set, the segment is entirely visible. If the intersection of regions from two points is not the empty set, the segment must lie entirely outside the window and is invisible. Thus, lines which are entirely visible or invisible are quickly processed.

If the line is partially visible, the point of intersection with one edge of the window is found and the segment that lies outside the window is discarded. The algorithm then repeats: The initial visibility test is then applied to the remaining segment and further subdivisions are made until only the visible part of the segment remains.

2.3.4 Viewports

A "window" is a logical (or virtual) screen and is mapped to a physical (or real) screen or a portion of a physical screen. This area of the physical screen onto which a window is mapped is termed a "viewport" or "view". Often the window fills the whole physical screen and the window and viewport have similar definitions. Sometimes many viewports fill a single physical screen.

Since the microcomputer system used in developing this software package did not support a graphics CRT but rather a graphics printer, a "viewport" is defined slightly differently for this package. A "viewport" is treated as a rectangular subset of a "window". The "view"

is used by the clipping modules instead of the "window". This allows a user to define one logical screen containing any number of other logical screens. The intent was to allow a user to restrict graphic operations to only a portion of the window. Many diagrams could be included in one final hard copy plot.

2.4 Matrix Transformations

2.4.1 Homogeneous Coordinates

The representation of an n-component position vector by an (n+1)-component vector is called homogeneous coordinate representation [ROGE76]. In homogeneous coordinate representation the transformation of n-dimensional vectors is performed in (n+1)-dimensional space and the transformed n-dimensional results are obtained by projection back into the particular n-dimensional space of interest. Thus, in two dimensions the position vector [x y] is represented by the three-component vector [hx hy h]. There is no unique homogeneous coordinate representation of a point in two-dimensional space. For ease of calculation and simplicity [x y 1] is used to represent a nontransformed point in two-dimensional homogenous coordinates.

The advantage of introducing homogeneous coordinates occurs in the general 3 x 3 transformation matrix

$$[x'\ y'\ h] = [x\ \ y\ \ 1] \begin{bmatrix} a & b & p \\ c & d & q \\ m & n & s \end{bmatrix}$$

where terms a, b, c and d produce scaling, shearing and rotation, m and n produce translation, and p and q produce a projection. The element s produces overall scaling.

2.4.2 Translation

Translation is the uniform motion of an object along a straight line. A translation could not occur with a transformation matrix without the use of homogeneous coordinates. Given the translation vector $\underline{T}$ with translation components $\underline{Tx}$ and $\underline{Ty}$, the matrix transformation for translation is

$$[x'\ y'\ 1] = [x\ \ y\ \ 1] \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Tx & Ty & 1 \end{vmatrix}$$

2.4.3 Scaling

A scaling operation can represent a change in units, or an enlargement or shrinking of the dimensions of an object. Negative scaling values can be used for mirror image "reflections". Given the scaling vector $\underline{S}$ with scaling components $\underline{Sx}$ and $\underline{Sy}$, the matrix transformation for scaling is

$$[x'\ y'\ 1] = [x\ \ y\ \ 1] \begin{vmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

2.4.4 Rotation

Rotation means that each point of an object moves in a circular path around the center of rotation. A rotation transformation matrix can be derived from simple geometry. The transformation matrix *to* rotate point (x,y) through a clockwise angle $\emptyset$ about the origin of the coordinate system is

$$[x'\ y'\ 1] = [x\ \ y\ \ 1] \begin{vmatrix} \cos\emptyset & -\sin\emptyset & 0 \\ \sin\emptyset & \cos\emptyset & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

This transformation matrix can only be used for a rotation about the

origin.

2.4.5 Change in Coordinate Systems

In the above transformations the coordinate system stays unaltered and the object is transformed with respect to the origin. An alternate but equivalent way of thinking of a transformation is a change in coordinate systems. This view is useful when multiple objects, each defined in its own local coordinate system, are combined into a single, global coordinate system.

A translation $\underline{T}$ of a point is a translation $\underline{-T}$ of the coordinate system. A rotation angle $\underline{\emptyset}$ of a point is a rotation angle $\underline{-\emptyset}$ of the coordinate system.

2.4.6 Concatenation of Simple Transformations

Transformations can be combined by matrix multiplication of simple transformation matrices. The order of transformations must be preserved since matrix multiplication is not commutative.

To demonstrate concatenation of transformations consider the rotation about an arbitrary point instead of the origin. Given point (x,y) to be rotated by an angle $\emptyset$ about the point (Rx,Ry). This can be accomplished by translating the origin to the point (Rx,Ry), performing the rotation, and then restoring the original origin:

$$[x'\ y'\ 1] = [x\ \ y\ \ 1] \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -Rx & -Ry & 1 \end{vmatrix} \begin{vmatrix} \cos\emptyset & -\sin\emptyset & 0 \\ \sin\emptyset & \cos\emptyset & 0 \\ 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Rx & Ry & 1 \end{vmatrix}$$

2.5 Extension to Three-Dimensional Graphics

Many two-dimensional operations can simply be generalized to produce three-dimensional operations. Some three-dimensional operations do not have two-dimensional analogs, however.

2.5.1 Homogeneous Coordinates

In three dimensions the position vector [x y z] is usually represented by the four-component homogeneous coordinates [x y z 1]. This row vector can be transformed by a general 4 x 4 matrix.

2.5.2 Transformations

Translation and scaling transformations can be easily generalized from two to three dimensions. The translation vector T consists of three components Tx, Ty and Tz. The 3D translation matrix is

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ Tx & Ty & Tz & 1 \end{bmatrix}$$

Given the scaling vector S with scaling components Sx, Sy and Sz, the 3D scaling matrix is

$$\begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotations in two-dimensions are assumed to be about the z-axis which is perpendicular to the x-y plane. With three dimensions, rotations can be about any one of the three axes in the plane formed by the other two axes. Positive rotation angles are measured in the clockwise sense when looking along an axis in the direction of the origin. Rotation about the z axis through an angle ϕ is achieved with the transformation:

$$\begin{bmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation about the y axis is given by:

$$\begin{vmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Rotation about the x axis is given by:

$$\begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

Permuting the axes in a cyclic fashion yields the expressions for rotation about the x and y axes from the z-axis rotation matrix.

2.5.3 View Transform

Objects are described in world coordinates. For 3D graphics these world coordinates usually must be converted to "eye coordinates" which have the location of the viewer's eye as the origin. A view transform matrix, composed of the concatenation of 5 simple transformation matrices, performs this change in coordinate systems. See procedure "view_transform_matrix" in Appendix Section A.3 "matrixops UNIT". Other details can be found in [NEWM79, Chapter 22] or [FOLE82, Chapter 8].

2.5.4 Projections

Perhaps the simplest projection from three- to two-dimensions is the "orthographic" projection. With this projection the z-coordinate is ignored and the (x,y) coordinates are used without modification. While this projection is simple it is not appropriate for many 3D applications.

A perspective view can be generated by projecting each point of an object onto the plane of the display screen. The view transform described in Section 2.5.3 is necessary to convert from world coordinates to eye coordinates.

The perspective transformation is different than previous transformations in that it requires dividing the the x and y values (in eye coordinates) by the z value (in eye coordinates). The earlier transformations only involved multiplications and additions. Generating a true perspective image requires dividing by the depth of each point.

2.5.5 Clipping

Clipping of three-dimensional segments can occur after the conversion from world coordinates to eye coordinates but it must be performed before the division by the depth of each point described in the previous section. The clipping operation simply cannot operate on projected line segments. Objects behind the eye can be projected onto the screen. The viewing "window" of 2D clipping is replaced by a "pyramid" in 3D clipping. The algorithm described in 2.3.3 can be used except line intersections with lines are replaced by line intersections with planes. Once 3D segments are clipped and projected they can be displayed.

3. Examples and Discussion

The microcomputer graphics package produced Exhibits 1 through 5. (The programs which produced these Exhibits are in Appendix B.) Most of these programs were originally developed as test cases while developing the various UNITs and PROCEDUREs.

3.1 Two-Dimensional Graphics

3.1.1 Cartography: Map of Kansas

Drawing straight lines is the simplest task performed using 2D graphics. Internally several special cases exist for drawing lines: single point, horizontal, vertical, slope >= 1, slope < 1. The points

for drawing the outline of the State of Kansas by county were obtained several years ago from a U.S. Department of Transportation tape. Since the map involves over 600 segments it was chosen for an initial thorough test of drawing simple lines of various slopes. Errors could be easily spotted. Exhibit 1.1 shows the whole state by county.

The plotting of straight lines was a major milestone in the program development -- further development obviously would have been senseless if simple line segments could not be handled. Memory was still available for user programs after they referenced the bulk of the library procedures. Processing time was slow, but tolerable. A large number of memory frames was available for the paging activity. (About 50 frames were initially available. Future additions have reduced that number to 10-20 with some programs now). All I/O and paging problems were ironed out in this first stage. Problems in mapping pixels to bits were also resolved.

Most of the vector and matrix mathematical operations were implemented in anticipation of 2D transformations. The 2D scaling, rotation and translation matrix operations were implemented and tested using the Kansas map data. Exhibit 1.1 shows scaling and translation while Exhibit 1.3 shows these operations as well as rotation about the "z" axis.

Initially, clipping was performed very crudely to avoid potential subscript range problems. Each pixel was checked to see if it was outside the "view" area. This crude clipping was initially implemented by a single IF statement but was very inefficient. 2D clipping was later "correctly" implemented using the Cohen-Sutherland algorithm from [NEWM79].

Exhibit 1.3 demonstrates intentional distortion to show what could happen with an inappropriate transformation matrix. For "fun" the black and white colors were inverted in this Exhibit.

3.1.2 Orthographic View of a Football Field

Exhibit 2 shows a 2D representation of the KSU football field -- an orthographic 3D view. This example was prepared to test 3D transformations and clipping.

3.2 Three-Dimensional Graphics

Once 2D graphic primitives were successfully implemented, work was begun on 3D graphics. When compiler symbol table space became more and more scarce it became apparent that many changes were necessary. Two- and three-dimensional primitives were combined into single PROCEDUREs. Combining the clipping operations was the most difficult change.

3.2.1 Perspective Views of a Football Field

Exhibits 3.1 through 3.4 show various perspective views of the KSU football field. Choice of a football field was made while attending a game last fall. A change in seats from the 50-yard line at one game to the endzone at another game gave me the idea. The field is useful for demonstrating perspective transformations since most people can visualize being in different seats and having the various views. These three pages of Exhibits take about 6.5 hours to plot from beginning to end. If processing time were faster, a movie composed of many still frames showing the aerial view flying around the stadium would have been considered.

3.2.2 Surface Graphics: z = f(x,y)

One of the original reasons for developing this graphics package was to plot perspective views of surfaces similar to the Surface II Graphics

System [SAMP75]. Exhibits 4.1 through 4.4 show various perspective views of the same surface but without hidden lines removed. One method of removing hidden lines for such surfaces of functional form is given in reference [KUBE68]. To date this hidden line removal algorithm has only been partially implemented.

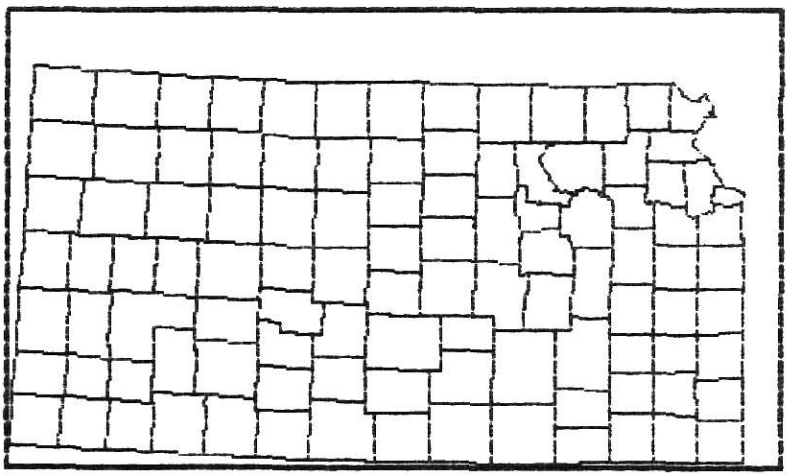
3.2.3 Surface Graphics: Pressure Map of Well Field

Exhibits 5.1 and 5.2 show a practical application for surface graphics. This surface described by a 16-by-16 grid is the solution of 256 simultaneous equations solved iteratively as the numerical solution of a partial differential equation.

Exhibit 1. Cartography Examples

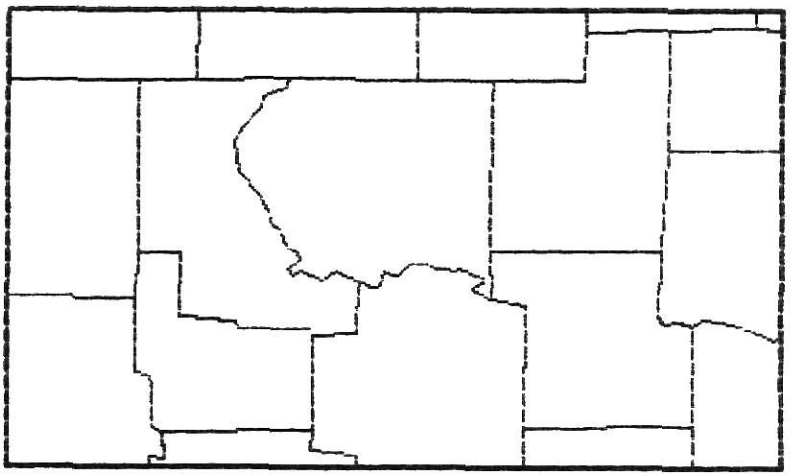
1.1 State of Kansas

Demonstrates plotting of unmodified line segments



1.1 Local Counties: Riley, Pottawatomie, et al

Demonstrates symmetrical scaling, translation and clipping of line segments



1.3 "Creative Cartography"

Demonstrates asymmetrical scaling, rotation, translation and clipping of line segments; color inversion

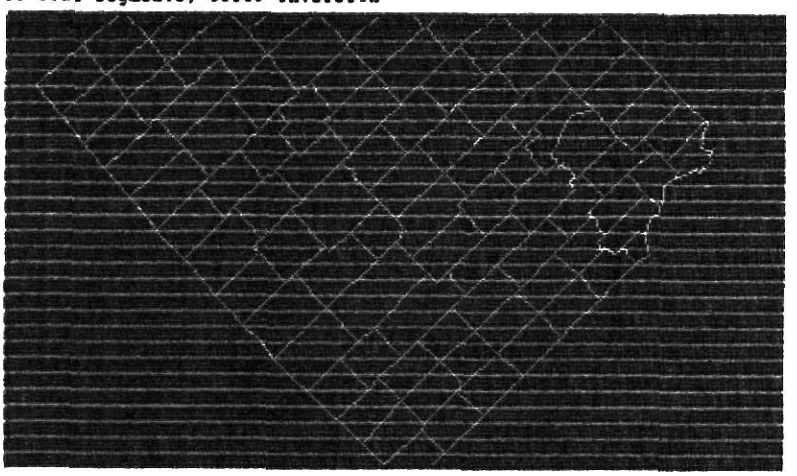


Exhibit 2. Orthographic View of KSU Football Field

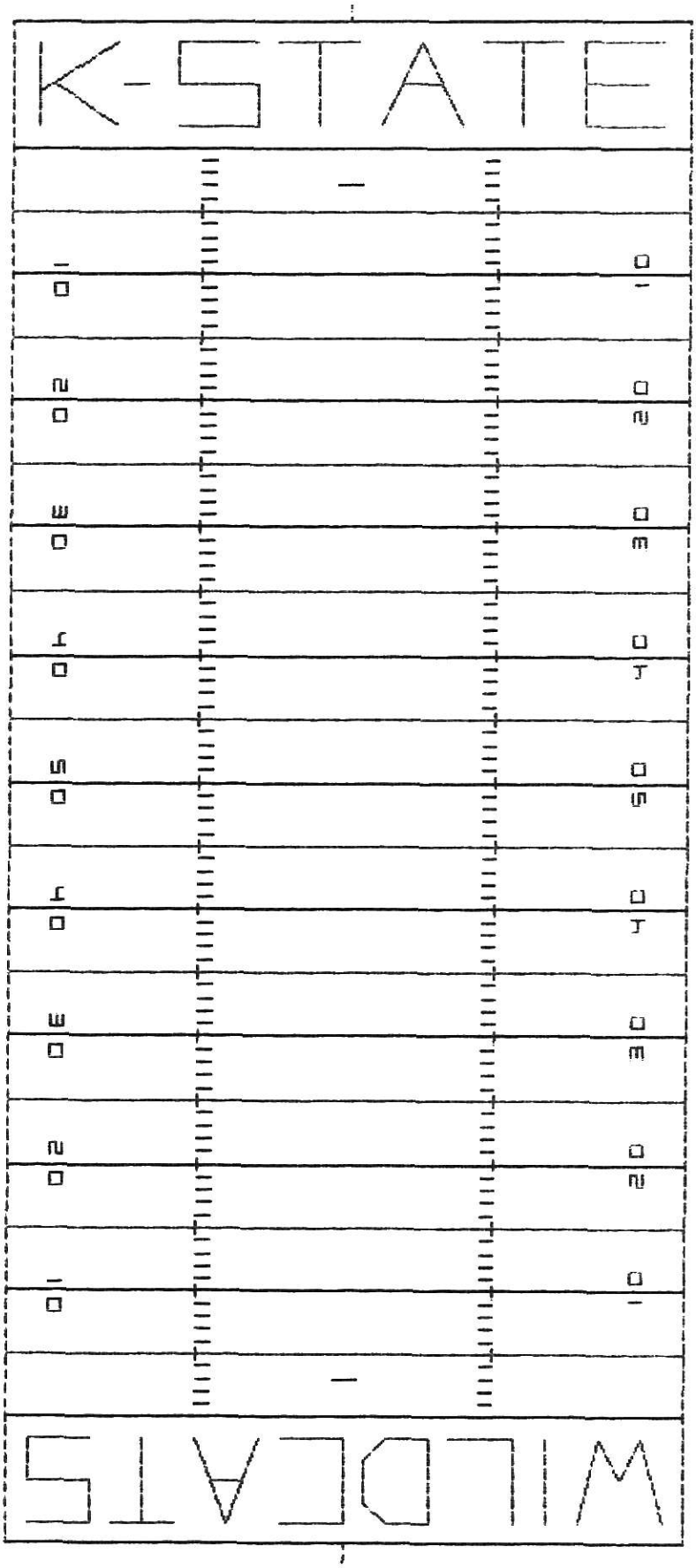
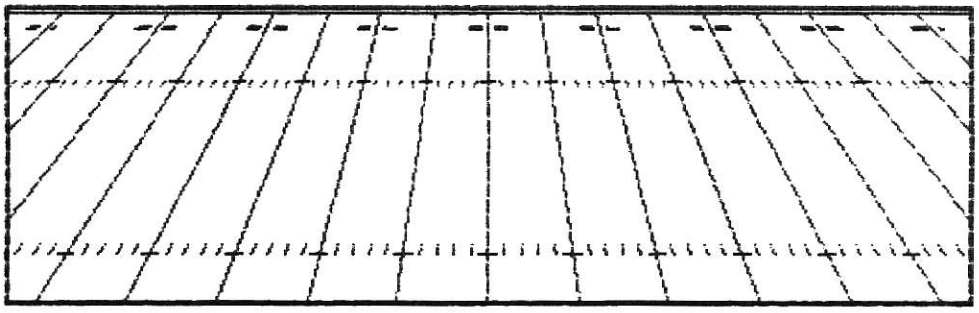


Exhibit 3. Perspective Views of KSU Football Field

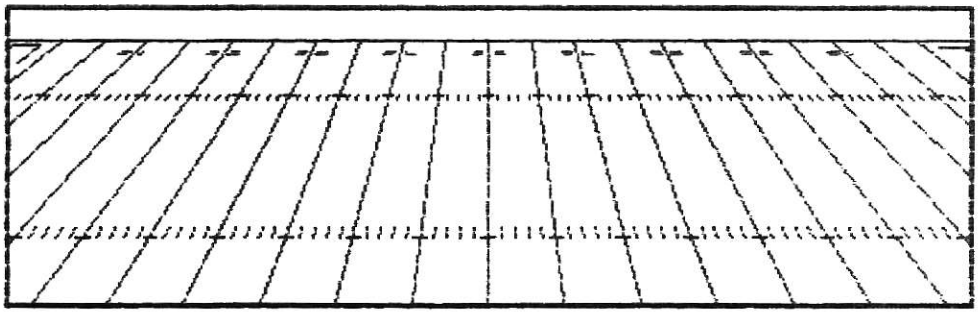
3.1 Side View (from the east side)

Azimuth = 90 degrees, Elevation = 30 degrees, Distance = 200 feet from center of field

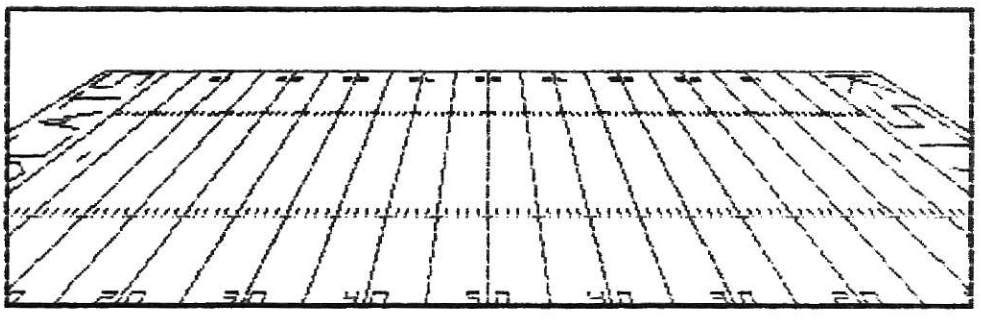
3.1a Proper perspective when viewed 5 inches from eye



3.1b Proper perspective when viewed 4 inches from eye



3.1c Proper perspective when viewed 3 inches from eye



3.1d Proper perspective when viewed 2 inches from eye

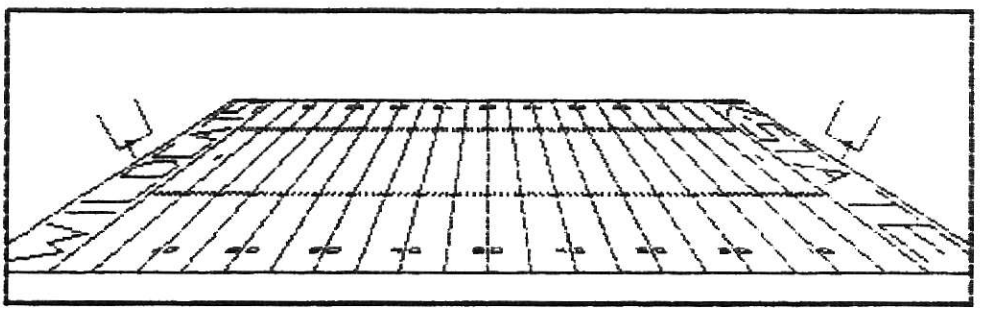
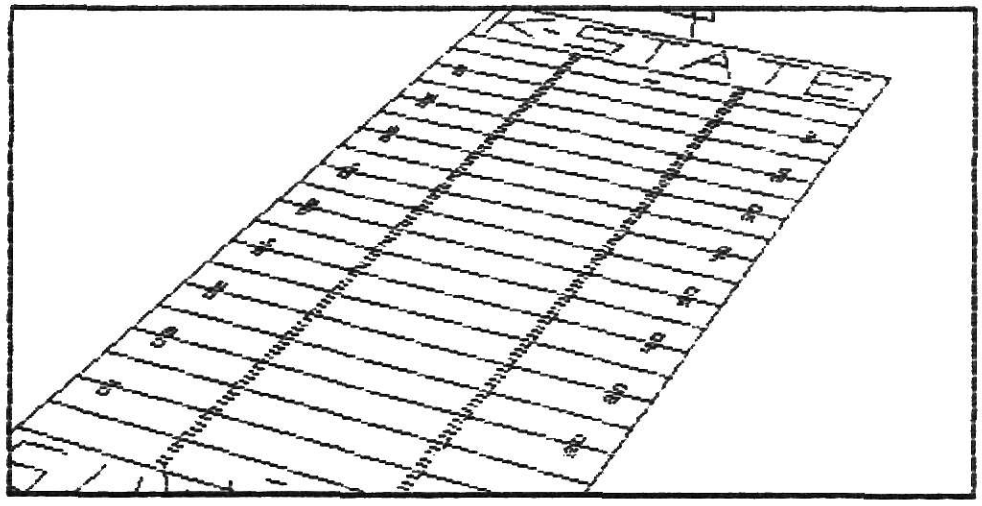
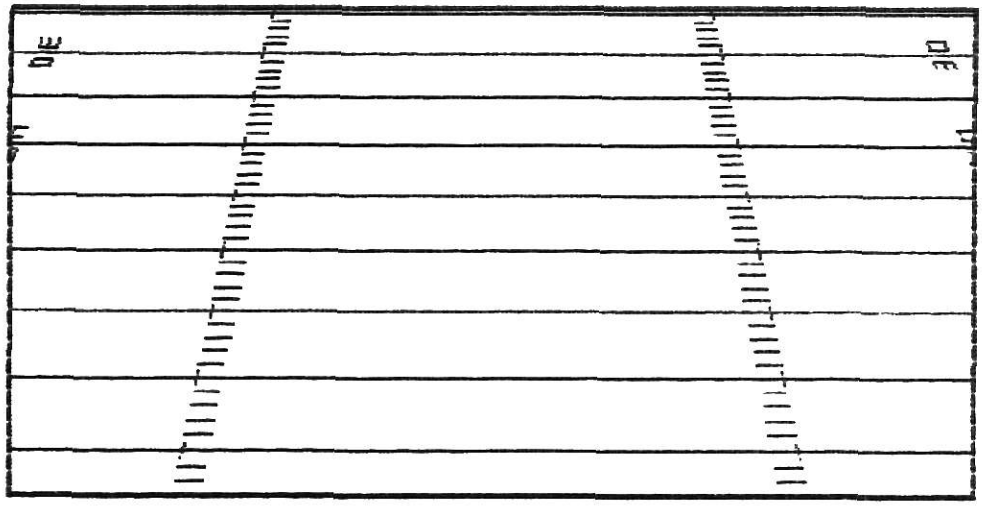


Exhibit 3. Perspective Views of KSU Football Field

3.2 Corner View (from the southeast corner)
Azimuth = 25 degrees, Elevation = 30 degrees, Distance = 750 feet from center of field



3.3 Endzone View (from the south endzone)
Azimuth = 0 degrees, Elevation = 30 degrees, Distance = 300 feet from center of field



3.4 Aerial View
Azimuth = 65 degrees, Elevation = 75 degrees, Distance = 1500 feet from center of field

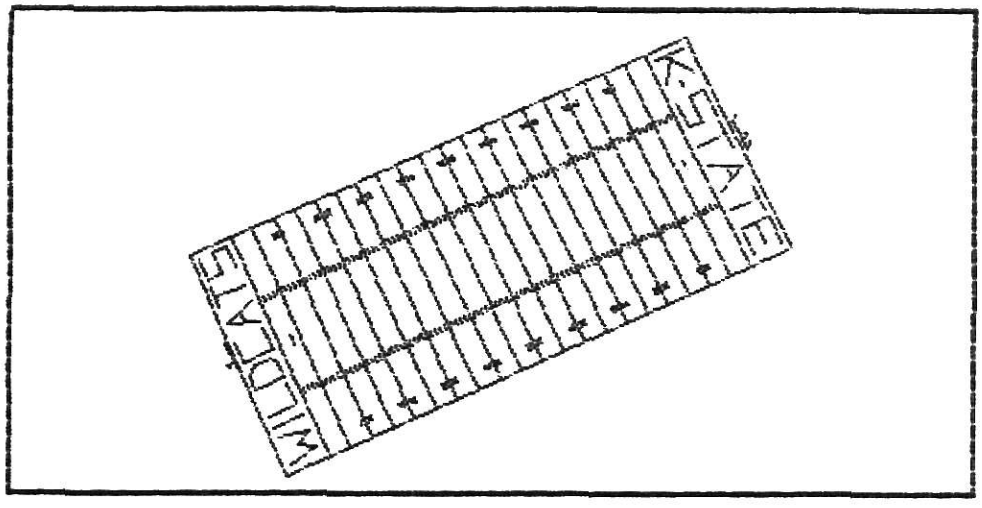
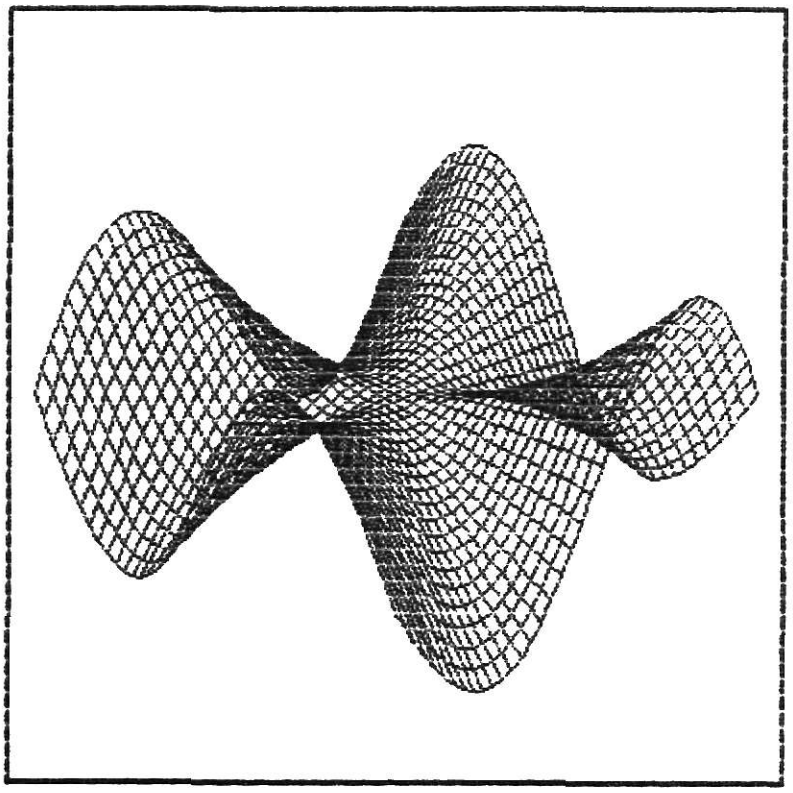


Exhibit 4. The Surface $Z(X,Y) = XY(X^2-Y^2)/(X^2+Y^2)$

"Hidden" lines are not removed. x=-2..+2; y=-2..+2.

4.1 Azimuth = 45 degrees, Elevation = 30 degrees, Distance = 15 units



4.2 Azimuth = 45 degrees, Elevation = 30 degrees, Distance = 5 units

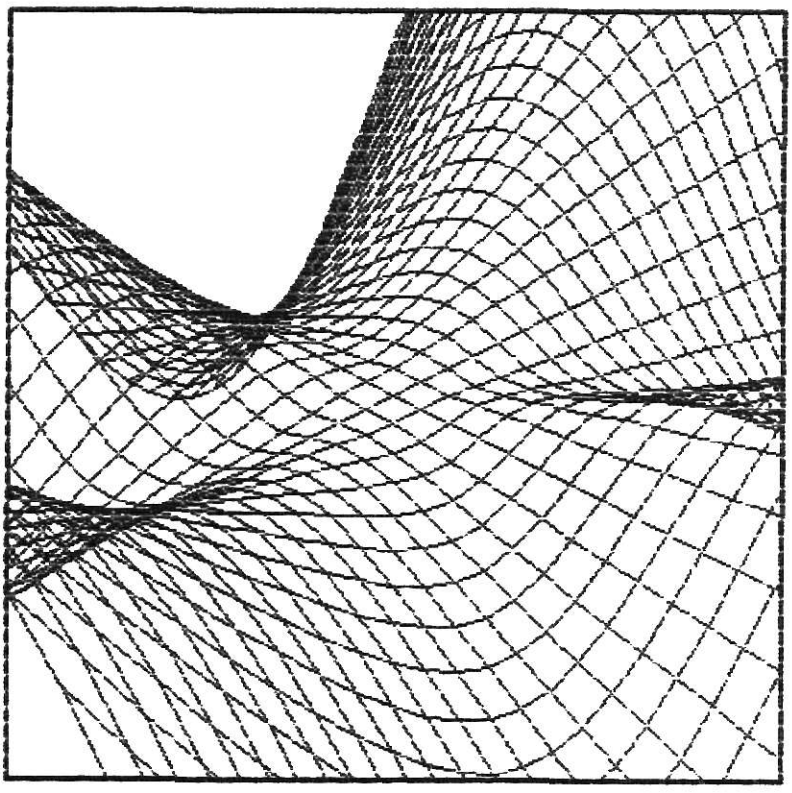
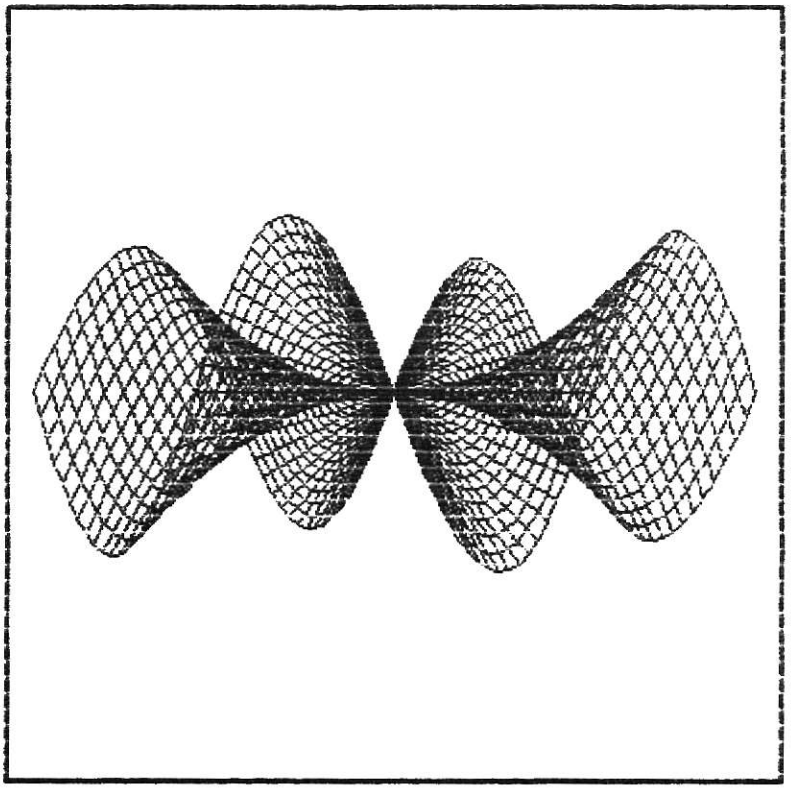


Exhibit 4. The Surface $Z(X,Y) = XY(X^2-Y^2)/(X^2+Y^2)$

"Hidden" lines are not removed. x=-2..+2; y=-2..+2.

4.3 Azimuth = 45 degrees, Elevation = 0 degrees, Distance = 15 units



4.4 Azimuth = 45 degrees, Elevation = 90 degrees, Distance = 15 units

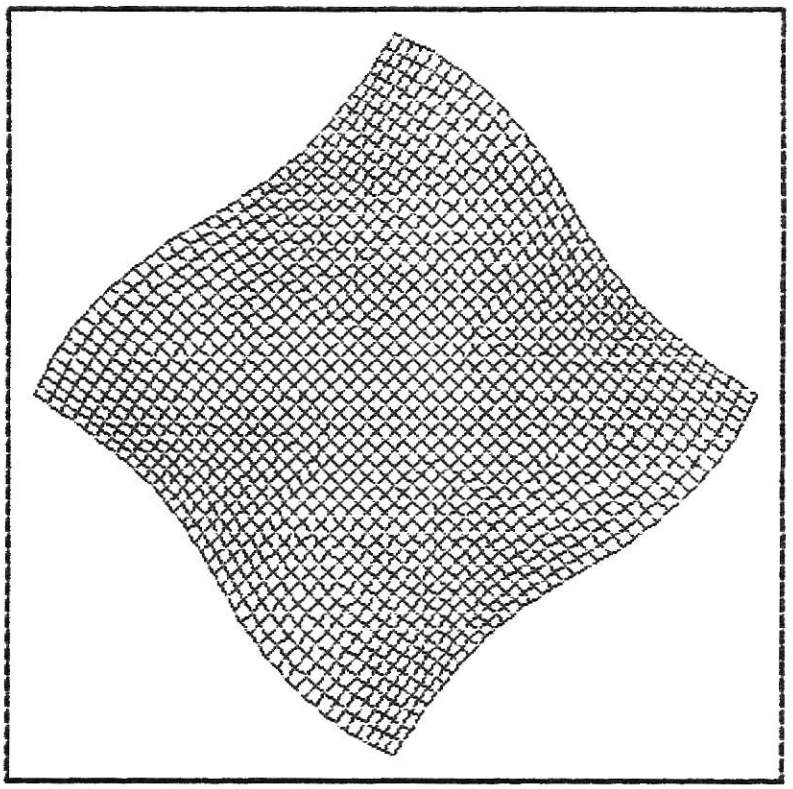
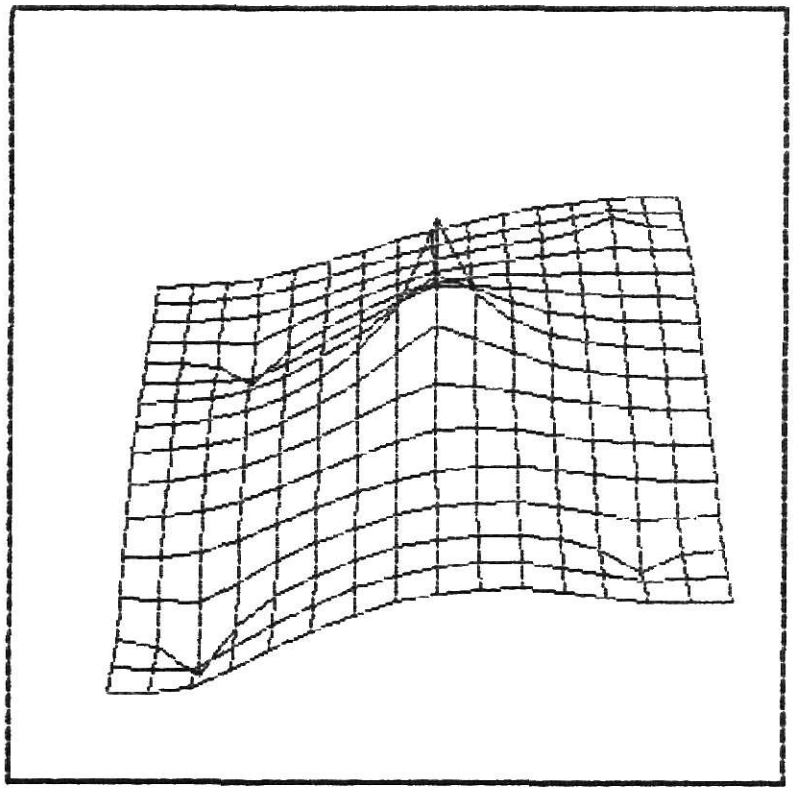
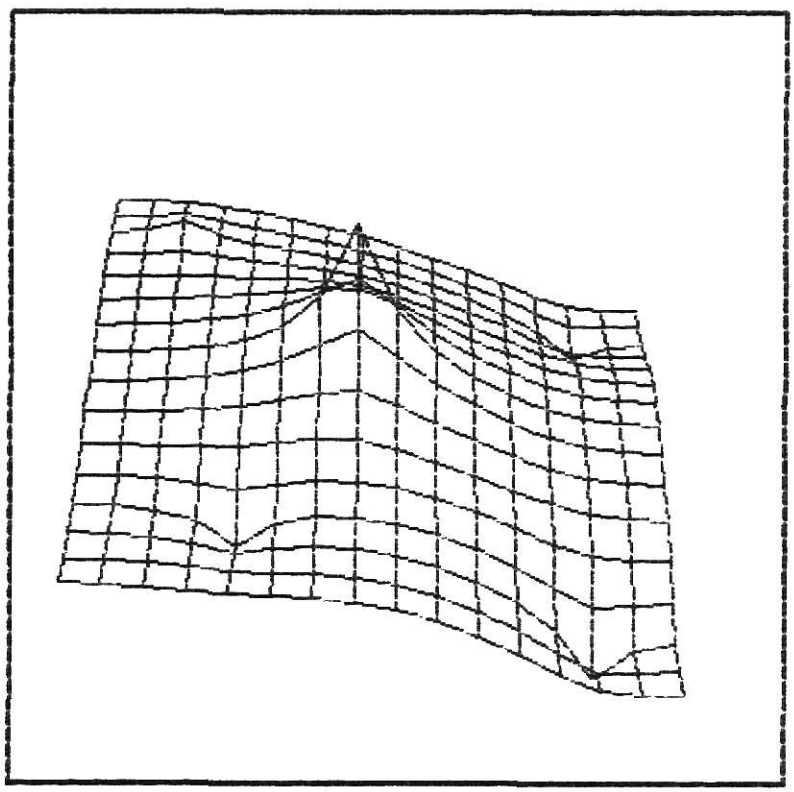


Exhibit 5. Pressure Map of Area with Injection/Production Wells
Partial Differential Equation: $k_x U_{xx} + k_y U_{yy} = c(x,y)$
$k_x=2$, $k_y=1$, $c(x,y)$= constant flow rate

5.1 Azimuth = 0 degrees, Elevation = 30 degrees, Distance = 60 units



5.2 Azimuth = 270 degrees, Elevation = 30 degrees, Distance = 60 units



4. Design Philosophy and Implementation Strategy

Software design on large mainframe computers often is not seriously restricted by memory, disk space or processor speed. Operating systems, programming languages and support libraries on large mainframes provide great flexibility not only in design but implementation. The microcomputer program designer is not always so fortunate. Memory, diskette space and processor speed can be major design factors -- but such factors will be less binding as microprocessor technology advances. Microcomputer operating systems and programming languages do not provide many features available on minis or mainframes. Support libraries and utility programs on micros are minimal or non-existent and the programmer must often improvise.

4.1 System Limitations

The software was designed to be as general as possible but generality was compromised to resolve certain problems created by hardware or software limitations.

4.1.1 Hardware Configuration

The graphics package is nearly independent of the type of processor but somewhat dependent on the particular graphics printer. The software was developed on a Heath/Zenith H/Z-89 Z-80 microcomputer system with a 2.048 MHz clock, 25-line by 80-column CRT, 64 Kbyte memory, three 102.4 Kbyte diskette drives, and the Integral Data Systems "Paper Tiger" 560 printer. The maximum printer width is 13.2 inches. High quality characters are formed with the 24-by-9 dot matrix. The printer's graphics mode provides 84 by 84 overlapping dots per inch. The printer speed is about 110 characters per second in mono-spaced mode, approximately half that speed in double width or graphics mode.

4.1.2 Operating System/Programming Language

The UCSD Pascal p-System (Version II.0) was used for software development [UCSD80]. This original implementation is strictly in Pascal even though certain operations could be enhanced dramatically by recoding them in Z-80 assembler language. UNITs were used to provide a library mechanism. The "include" file mechanism of the compiler was used since certain source programs were too large to edit as a single file. Compiler symbol table space was an overriding constraint during compilations of UNITs. A single UNIT was not possible and several UNITS were found to be necessary to compile all modules. Procedures were sometimes modularized more than desired due to compiler restrictions on their length. Single procedures were sometimes subdivided into several local internal procedures to appease the compiler yet restrict variable access. The use of global variables was kept to a minimum.

Features of UCSD Pascal that are non-standard Pascal (e.g., random I/O) were used only when necessary. System intrinsics were avoided but certain ones (e.g., MOVELEFT) still had to be used. Free union variant records -- variants without tag fields -- were used as necessary to facilitate overlaying different data types even though they are usually considered "dangerous" because of their flexibility.

4.2 Primitives

Standards in the area of computer graphics have been investigated for many years [GUED76], but as in many areas of computer science today, only "tendencies" exist. The primitives used in this project were modeled after those found in [NEWM79] and are similar to those in [FOLE82]. [NEWM79] uses the same names for two- and three-dimensional primitives of different dimensionality, e.g., "moveto (x,y)" and "moveto

(x,y,z)". Instead of separate two- and three-dimensional primitives, and because of limited symbol table space in compiling UCSD UNITs, an attempt was made to integrate two- and three-dimensional primitives. For example, instead of "moveto_2D (x,y)" and "moveto_3D (x,y,z)", a single "moveto (u)" was implemented where "u" is of the TYPE "vector" which holds either a two- or three-dimensional point. This integration does require the use of separate PROCEDUREs to define the vectors of different dimensionality: "define_2D_vector (x,y,u)" and "define_3D_vector (x,y,z,u)".

4.3 Logical Screen and Plot Files

Two alternatives were initially considered for the data structure containing the picture definition. One alternative was a somewhat complex data structure containing line segments which would be searched during each raster scan to determine if any part of the segment was contained in the scan. Several methods of this type are discussed in [FRAN79]. Unfortunately, the size of the data structure would be somewhat proportional to the complexity of the picture. A second approach is very simple in concept: a huge matrix of pixels would be maintained in memory or on disk in a "plot file". The size of the matrix would be proportional to the size of the physical screen but independent of the complexity of the picture. The decision to use this second method was made somewhat arbitrarily.

4.3.1 World and Screen Coordinate Systems

A logical screen is an abstraction of a real video display screen which may or may not correspond to an actual screen, or multiple logical screens might be mapped to a single physical screen. While a logical screen could take on any shape, only rectangular areas were used in this

project.

The coordinates of a logical screen (called a "window") are defined in terms of world coordinates instead of the absolute addressing often required by the hardware of most physical screens. The software automatically converts world units to the physical units describing the location of a given point on a logical screen.

4.3.2 Paging System

A rather large matrix of pixels cannot be held in only 64K of memory. A paging system was implemented to facilitate picture definition for a physical screen of any dimension which could be stored on existing or future disk space. This paging system is invisible to the user.

4.4 Feasibility Studies

All problems cannot be anticipated but certain potential problems were investigated which could have prevented successful completion of this project. Certain parts of the graphics package were developed in a "bottom-up" approach.

4.4.1 Hardware/Software Harmony

The UCSD p-system happened to be compatible with the hardware requirements of the IDS 560 printer. The UCSD Supplemental Users' Document for using the H89 did not indicate any printer restrictions, but the menu from the configuration program gave only three possible printers to choose from -- none of which were an IDS 560 or a "generic" printer. Problems were known to exist in graphics mode when the IDS 560 printer was used with the Heath HDOS operating system. HDOS uses a device driver which intercepts certain special codes and imposes limits on characters per line and lines per page which cannot be completely disabled (easily). UCSD could have had similar problems but did not.

One problem did develop which could have ruined most pictures on the IDS 560. The UCSD p-system, for some unknown reason, intercepts the control character hexadecimal '10' -- a fact that did not surprise Softech personnel when phoned about the problem. Without that character non-vertical and most non-horizontal lines would have breaks. Certain horizontal lines would even completely disappear. The IDS people anticipated such problems by recognizing only seven of the eight bits of a byte in graphics mode. The remaining bit could be '0' or '1'. By or-ing hex '80' with '10', the printer correctly treated '90' as a substitute for '10'.

4.4.2 Random I/O in UCSD Pascal

Standard Pascal does not support random files. UCSD supports random file access by means of the intrinsic SEEK and the GET and PUT procedures. This method of random file access only works for typed files, however. The system intrinsics UNITREAD/UNITWRITE and BLOCKREAD/BLOCKWRITE support random file access for untyped files but may be dangerous intrinsics to use (the manual says so). Whether or not the use of a typed file was necessary was unclear initially. Several alternatives were investigated. One big problem centered around whether or not the sequential parameter prefix file (see System Guide) had to be a separate file. Because of operating system fragmentation of disk space, it was desirable to incorporate the prefix with the rest of the random file containing blocks of pixels. This problem was resolved by introducing free union variant records and tolerating some extra move operations.

4.4.3 Mapping Pixels to Bits

The exact mapping of a pixel to one or more bits in UCSD PACKED

ARRAYs was not well understood. Each pixel could have required a two-byte word for storage and only very small pictures would be possible. This was not the case and it was possible to map 8 bits into each byte of physical storage. However, one small snafu still was present. A PACKED ARRAY of bits was found not to be stored contiguously in physical memory. The order is: bits 7-0 in byte 1, bits 15-8 in byte 2, and so on.

4.4.4 Processor Speed

Interpreting p-code on the UCSD system adds considerable overhead to run time compared with assembler code or even native code generated by a compiler. Optimization of execution time was not a primary goal of this project, but if execution times would have been much slower, implementation of this graphics package would have been too impractical for the microcomputer being used. The concern that some processes might take several hours was realized when mapping the plot file to the printer in graphics mode.

5. Future Extensions and Other Applications

Future work on this particular graphics package could address many current deficiencies that time did not permit to be considered in this first implementation. Some of these areas are discussed below.

5.1 Performance Enhancements

From a run time standpoint, this graphics package executes satisfactorily except for the mapping of the plot file to the graphics printer. That process literally takes hours -- eight hours or more for some large plots. The time involved in the mapping process must be decreased by as much as an order of magnitude for practical operation.

According to Softech and Heath sources interviewed by phone, UCSD

Pascal Version IV will be available in August or September 1982 for the H/Z-89. That version should provide a native code generator which should speed the mapping process considerably. Re-writing certain sections of code in Z-80 assembler language will also be considered. The mapping process could be sped up considerably if bytes are addressed instead of the individual bits representing each pixel.

5.2 Annotation Text and Symbols

[MACE82] demonstrates several character sets and special symbols whose coordinates are defined by [WOLC76]. Annotation text and symbols are very necessary for a complete graphics package.

5.3 Graphic Elements including CALCOMP-like Plotting Package

The only two graphic primitives in this graphics package are "moveto" and "lineto". These primitives could be combined to form graphic elements which define more complex operations. For compatibility with existing graphics packages on many mainframe computers, a CALCOMP-like library of graphic elements should be developed.

5.4 Hidden Line/Surface Removal

Several references have been investigated to review existing hidden line and surface removal algorithms: [GRIF78a, GRIF78b, KUBE68, SUTH74 and WILL72]. The [KUBE68] reference seemed particular easy for z = f(x,y) functional surfaces or for those which are already defined by a grid of points where interpolation is possible between grid points. No decision has been reached about which algorithm would be relatively quick, use little memory and use little compiler symbol table space.

5.5 Color Graphics

Microcomputer magazines reported last year that Integral Data Systems intended to announce a color printer this year. In January 1982

IDS announced their "Prism" printer which has a four-color ribbon. Mixing allows another four colors. This graphics package was designed to accommodate a multi-color pixel TYPE definition with this new product in mind. The driver which controls the transfer of the plot file to the graphics printer would have to be modified to accommodate the new color printer. Certain other changes in the definition of blocks and memory frames would also be necessary.

6. Conclusions

This microcomputer graphics package demonstrates that a general purpose package can not only be used on a "personal" computer but can be developed on one. This same software system on a future, more powerful microcomputer with nearly unlimited memory and disk space will give a user a very powerful graphics tool.

This project was undertaken in Fall 1981 as an academic endeavor but also as a recreational exercise on a personal computer. Many areas would not have been explored as thoroughly, and many tangents would not have been partially pursued, if the primary intent of this project was to develop and market a new software package. An outline of the chronology of this project is approximately as follows:

September 1981 -- 25 hours
- reviewed operation of IDS 560 printer
- reviewed operation of UCSD Pascal System
- ran benchmark tests comparing UCSD Pascal with Microsoft FORTRAN compiler and Microsoft BASIC interpreter
- started literature search

October 1981 -- 25 hours
- continued literature search
- investigated UCSD random I/O
- investigated interactive prompts for file names
- studied I/O errors related with the above

November 1981 -- 40 hours
- developed "hexdump" PROGRAM as debugging tool
- studied use of variant records to overlay different data types
- "parm" and "plot" files were first created
- framework of paging system was developed
- pixel setting for line segments developed

December 1981 -- 40 hours
- integrated "parm" file with "plot" file
- driver written to map pixel matrix to printer
- the initial structure of the "dotplotter" UNIT established
- initial 2D operations: simple line segments drawn

January 1982 -- 40 hours
- matrix operations/transformations implemented
- 2D tests and examples
- integration of 2D and 3D procedures
- 2D clipping

February 1982 -- 30 hours
- view transform matrix procedure developed
- 3D clipping
- 3D tests and examples
- literature investigation of hidden line/surface removal

March 1982 -- 30 hours
- hidden line/surface investigation continued
- outlines of Report, System and User Guide developed

April 1982 -- 40 hours
- last 3D example developed
- Report, System and User Guides written

REFERENCES

ANGE81 Angell, Ian O., A Practical Introduction to Computer Graphics, 1981, John Wiley & Sons

FOLE82 Foley, James D. and Andries Van Dam, Fundamentals of Interactive Computer Graphics, 1982, Addison-Wesley Publishing Company

FRAN79 Franklin, W. Randolph, "Evaluation of Algorithms to Display Vector Plots on Raster Devices", Computer Graphics and Image Processing, Vol. 11, 1979, pp. 377-397

GILO78 Giloi, Wolgang K., Interactive Computer Graphics, 1978, Prentice-Hall

GRIF78a Griffiths, J. B., "A surface display algorithm", Computer-Aided Design, Volume 10, Number 1, January 1978, pp. 65-73

GRIF78b Griffiths, J. B., "Bibliography of hidden-line and hidden-surface algorithms", Computer-Aided Design, Volume 10, Number 3, May 1978, pp. 203-206

GUED76 Guedj, Richard A. and Hugh A. Tucker (editors), IFIP Workshop on Methodology in Computer Graphics, Seillac, France, 1976, North-Holland Publishing Company

IDS81 IDS 560 "Paper Tiger" Impact Printer Owner's Manual, Integral Data Systems, Inc., 1981

KUBE68 Kubert, B., J. Szabo and S. Giulieri, "The Perspective Representation of Functions of Two Variables", Journal of the Association for Computing Machinery, Vol. 15, No. 2, April 1968, pp. 193-204

MACE82 Macero, Daniel, et al, "Graphics II by Selanar", BYTE, Vol. 7, No. 3, March 1982, pp. 172-196

NEWM79 Newman, William M. and Robert F. Sproull, Principles of Interactive Graphics (Second Edition), 1979, McGraw-Hill

POSD77 Posdamer, Jeffrey L., "A Vector Development of the Fundamentals of Computational Geometry", Computer Graphics and Image Processing, Vol. 6, 1977, pp. 382-393

ROGE76 Rogers, David F. and J. Alan Adams, Mathematical Elements for Computer Graphics, 1976, McGraw-Hill

SAMP75 Sampson, Robert J., Surface II Graphics System, 1975, Kansas Geological Survey, University of Kansas

SCOT82 Scott, Joan E, Introduction to Interactive Computer Graphics, 1982, John Wiley & Sons

SUTH74 Sutherland, Ivan E., Robert F. Sproull and Robert A. Schumacker, "A Characterization of Ten Hidden-Surface Algorithms", Computing Surveys, Vol 6. No. 1, March 1974, pp. 1-55

UCSD80 UCSD Pascal Users Manual, Softech Microsystems, Inc., 1980

UMLO82 Umlor, Ed, "Integeral Data Systems' Prism Printer", BYTE, Vol. 7, No. 3, March 1982. pp. 44-49

WHIT82 Whitted, Turner, "Some Recent Advances in Computer Graphics", Science, Vol. 215, No. 4534, 12 February 1982, pp. 767-774

WILL72 Williamson, Hugh, "Algorithm 420: Hidden-Line Plotting Program", Communications of the ACM, Vol. 15, No. 2, February 1972, pp. 100-103

WOLC76 Wolcott, N. M. and J. Hilsenrath, "A Contribution to Computer Typesetting Techniques: Tables of Coordinates from Hershey's Repertory of Occidental Type Fonts and Graphic Symbols", National Bureau of Standards Special Publication No. 424, U.S. Department of Commerce, U.S. Government Printing Office, 1976 (out of print)

Appendix A. Source Listings of Pascal UNITs

```
{$L- PRINTER:}
{$S+ Put compiler in swapping mode.}
{$R+ Turn range checking on.}
UNIT global;  {UCSD Pascal, Version II.}

  {$C  Copyright (C) 1982 by Earl F. Glynn, Manhattan, KS.}
  {Written in January 1982; last modified on 15 February 1982.}

  {The "global" UNIT is used to minimize sharing between the "dotplotter"
  and other UNITs to provide more symbol table space for compilation
  of the "dotplotter" UNIT.  The TYPE definitions are used extensively in
  other UNITs and user PROGRAMs.}


INTERFACE

CONST
  radians_per_degree = 1.745329E-2;  {3.14159.../180}

TYPE
  index     =  1..4;              {used for "matrix" and "vector" TYPEs}
  dimension =  (two_D,three_D); {two- or three-dimensional TYPE}
  matrix    =                   {transformation "matrix"}
    RECORD
      size  :  index;
      mtrx  :  ARRAY[index,index] OF REAL
    END;
  vector    =                   {"vector" TYPE used to define points}
    RECORD
      size  :  index;
      vctr  :  ARRAY[index] OF REAL
    END;

VAR
  prt       :  TEXT;    {"prt" is defined here so a UNIT which USES ...}
                 {another UNIT can compile and reference a common TEXT FILE.}

FUNCTION  defuzz(x:  REAL):  REAL;
PROCEDURE define_2D_vector (x,y:    REAL; VAR u:  vector);
PROCEDURE define_3D_vector (x,y,z:  REAL; VAR u:  vector);
PROCEDURE transform (u:  vector; a:  matrix; VAR v:  vector);


IMPLEMENTATION

CONST
  fuzz = 1.0E-6;

FUNCTION defuzz (PUBLIC(x:  REAL):  REAL);
  {"defuzz" is used for comparisons and to avoid propagation of "fuzzy"
  nearly-zero values.  REAL calculations often result in "fuzzy" values.}
BEGIN
  IF   ABS(x) < fuzz
  THEN defuzz := 0.0
  ELSE defuzz := x
END {defuzz};

PROCEDURE define_2D_vector (PUBLIC(x,y:  REAL; VAR u:  vector));
  {This procedure defines two-dimensional homogeneous coordinates (x,y,1)
  as a single "vector" data element "u".  The "size" of a two-dimensional
  homogenous vector is 3.}
BEGIN
  u.size := 3;
  u.vctr[1] := x;
  u.vctr[2] := y;
  u.vctr[3] := 1.0
END {define_2D_vector};

PROCEDURE define_3D_vector (PUBLIC(x,y,z:  REAL; VAR u:  vector));
  {This procedure defines three-dimensional homogeneous coordinates (x,y,z,1)
  as a single "vector" data element "u".  The "size" of a three-dimensional
  homogenous vector is 4.}
BEGIN
  u.size := 4;
  u.vctr[1] := x;
```

```
 76    u.vctr[2] := y;
 77    u.vctr[3] := z;
 78    u.vctr[4] := 1.0
 79  END {define_3D_vector};
 80
 81  PROCEDURE transform {PUBLIC(u: vector; a: matrix; VAR v: vector)};
 82    {"transform" multiplies a row "vector" by a transformation "matrix"
 83    resulting in a new row "vector". The "size" of the "vector" and "matrix"
 84    must agree (if not, the transformed vector is given a "size" of 1 but none
 85    of the components are defined).}
 86  VAR
 87    i,k : index;
 88    temp: REAL;
 89  BEGIN
 90    v.size := a.size;
 91    IF   a.size = u.size
 92    THEN BEGIN
 93      FOR i := 1 TO a.size-1 DO BEGIN
 94        temp := 0;
 95        FOR k := 1 TO a.size DO
 96          temp := temp + u.vctr[k]*a.mtrx[k,i];
 97        temp := defuzz(temp);
 98        v.vctr[i] := temp
 99      END;
100      u.vctr[a.size] := 1
101    END
102    ELSE BEGIN
103      WRITELN ('GLB01:');
104      WRITELN ('Ignoring attempt to multiply a vector of dimension ',
105               u.size,' by a square matrix of dimension ',a.size,'.');
106      v.size := 1  {signal error by setting dimension of "v" to 1}
107    END
108  END {transform};
109
110  END {global UNIT}.
```

```
{$L- PRINTER:}
{$S+ Put compiler in swapping mode.}
{$R+ Turn range checking on.}
{$I- The compiler will not generate I/O checking code.}

UNIT dotplotter; {UCSD Pascal, Version II.}

  {$C Copyright 1982 by Earl F. Glynn, Manhattan, KS.}
  {Written in Nov 1981 - Jan 1982; last modified on 5 Apr 1982.}

  {The "dotplotter" unit, under user program control, creates (or overlays)
  a logical screen defined as a matrix of dots (pixels). This pixel matrix
  is stored in a diskette file and blocks of it are paged in and out of
  memory as needed. Parameters defining the logical screen are contained
  in a file prefix. The structure of the file follows:

                          +--------------------+
                          |                    |
                          |  Parameter Prefix  |
                          |                    |
                          +--------------------+
                          |                    |
                          |       Pixel        |
                          |       Matrix       |
                          |                    |
                          |                    |
                          |                    |
                          +--------------------+

  User access to the parameters is restricted to procedure calls. The
  paging system used to access different blocks of the pixel matrix is
  entirely transparent to the user. The user controls the maximum size
  of the pixel matrix by defining the maximum size of the logical screen.
  The actual size of the file changes: blocks are dynamically allocated
  (by the operating system from the pool of disk space) when they are
  first accessed. A part of the parameter prefix contains index information
  used by the paging system.

  The size of a memory frame was chosen to correspond to a physical block
  to facilitate random I/O. The dimensions of the frames and I/O blocks
  were parameterized to facilitate implementation on other systems. In
  fact, all constants that possibly could change from one implementation
  to another are treated as UNIT parameters. For example, to change from
  a dot resolution of 84-by-84/inch to 75-by-72/inch, only the constants
  "i_density" and "j_density" need to be changed.

  The "put_plot" procedure is very device dependent and in the current
  implementation supports the high resolution Integral Data Systems
  "Paper Tiger" 560 raster graphics printer.

  The pixels in the current implementation may only be "black" or "white".
  The "pixel" TYPE can be redefined to be any set of colors but certain
  restrictions apply (since pixels are mapped to memory frames which
  in turn are mapped to disk blocks). Using more colors either increases
  the amount of needed disk space for a given size of picture, or reduces
  the maximum size of a picture given a fixed amount of disk space.
  To use the IDS "Prism" printer the "pixel" TYPE must be changed as well
  as the "put_plot" procedure. Most other procedure will require little
  or no change for color graphics.

  With only 64K of memory, the "dot_plotter" UNIT must be divided into
  sections since it is too large to edit as a single file. The "include"
  mechanism is used to put the UNIT together at compile time. Symbol
  table space is also tight with the given 64K of memory when compiling
  this UNIT.}


INTERFACE
USES global;

CONST
  block_size   =  512;  {UCSD disk blocks are 512 bytes long}

TYPE
  disposition  = (keep,delete);
```

```
  pixel        = (white,black);  {each pixel requires only one bit}
  projection   = (orthographic,perspective);

VAR
    {Ideally, access to these VARiables would be made only by procedure
    calls, but limited compiler symbol table space does not allow such
    a luxury.}
  close_printer:  BOOLEAN;  {close/paginate on exiting graphics mode}
  min_space    :  INTEGER;  {space reserved by calling program}
  plt          :  FILE OF PACKED ARRAY[1..block_size] OF CHAR;
  plt_mode     :  (undefined,either,create,overlay);
  plt_name     :  STRING[23];

PROCEDURE begin_dot_plotter_unit;
PROCEDURE end_dot_plotter_unit;
PROCEDURE size (x_len,y_len:  REAL);
PROCEDURE dot_color (color:  pixel);
PROCEDURE fill_color (color:  pixel);
PROCEDURE window (x_min,x_max,y_min,y_max:  REAL);
PROCEDURE view (x_min,x_max,y_min,y_max:  REAL);
PROCEDURE open_plot;
PROCEDURE close_plot (disp:  disposition);
PROCEDURE moveto (u:  vector);
PROCEDURE lineto (u:  vector);
PROCEDURE clear_transform (d:  dimension);
PROCEDURE set_transform (a:  matrix);
PROCEDURE get_transform (d:  dimension; VAR a:  matrix);
PROCEDURE clipping (flag:  BOOLEAN);
PROCEDURE project (u:  vector; VAR v:  vector);
PROCEDURE set_projection_type (prj_type:  projection);
PROCEDURE put_plot (x,y:  REAL; copies:  INTEGER; border:  INTEGER);


IMPLEMENTATION

CONST
  black_byte    =  255;  {hex 'FF'}
  file_tag      = 'DOTplot ';  {tag ensures user does not use non-plot file}
  i_density     =   84;  {dots per inch horizontally}                  {PT 560}
  i_frame_size  =   63;  {pixels (0..63) horizontally in block frame}
  j_density     =   84;  {dots per inch vertically}                    {PT 560}
  j_frame_size  =   63;  {pixels (0..63) vertically in block frame}
  max_frames    =   40;  {max number (1..40) of memory frames}
  max_i_blks    =   17;  {max columns (0..17) in block matrix}
  max_j_blks    =   21;  {max rows (0..21) in block matrix}
  spacedefault  = 1024;  {reserve 1K byte of memory for use by user program}
  white_byte    =    0;  {hex '00'}
                         {derived constants:}
  max_blks      =  396;  {(max_i_blks+1)*(max_j_blks+1)}
  max_i_dots    = 1151;  {(i_frame_size+1)*(max_i_blks+1) - 1}
  max_j_dots    = 1407;  {(j_frame_size+1)*(max_j_blks+1) - 1}
  prm_size      = 1024;  {size of parm prefix to plot file; n * block_size}

TYPE
  blk_no        = 0..max_blks;
  frame_no      = 1..max_frames;
  i_blk_no      = 0..max_i_blks;
  i_dot_no      = 0..max_i_dots;
  i_frame_no    = 0..i_frame_size;
  j_blk_no      = 0..max_j_blks;
  j_dot_no      = 0..max_j_dots;
  j_frame_no    = 0..j_frame_size;
  memory_frame  = PACKED ARRAY[i_frame_no,j_frame_no] of pixel;

VAR
  blk_index     :  ARRAY[frame_no] OF blk_no;
  frame_i_blk   :  ARRAY[frame_no] OF i_blk_no;
  frame_j_blk   :  ARRAY[frame_no] OF j_blk_no;
  frame_ptr     :  ARRAY[frame_no] OF ^memory_frame;
  frames_alloc  :  0..max_frames;   {memory frames allocated:  (0,frame_no)}
  frames_inuse  :  0..max_frames;   {memory frames in use:  (0,frame_no)}
  heap          :  ^INTEGER;    {pointer for MARK/RELEASE}
  i_save        :  INTEGER;
  j_save        :  INTEGER;
  next_out      :  frame_no;
```

```
151     plt_open_flag :  BOOLEAN;
152     prm           :
153       PACKED RECORD
154         CASE INTEGER OF
155           0:  (prm_char  :  PACKED ARRAY[1..prm_size] OF CHAR);
156           1:  (pltfile   :  PACKED ARRAY[1..8] OF CHAR;  {'DOTPLOT '}
157                blk_table :  ARRAY[i_blk_no,j_blk_no] of INTEGER;
158                i_blks    :  i_blk_no;  {columns used in block matrix}
159                i_length  :  i_dot_no;  {dots horizontally}
160                j_blks    :  j_blk_no;  {rows used in block matrix}
161                j_length  :  j_dot_no;  {dots vertically}
162                n_blks    :  blk_no;    {number of blocks in file}
163                vcx,vcy   :  REAL;      {3D projection center-size viewport parms}
164                vsx,vsy   :  REAL;
165                x_dots    :  REAL;      {dots per world unit horizontally}
166                x_east    :  REAL;    {"EAST" clipping value on x-axis }
167                x_first   :  REAL;    {logical origin coordinate [world units]}
168                x_last    :  REAL;
169                x_length  :  REAL;    {maximum size of plot horizontally [inches]}
170                x_west    :  REAL;    {"WEST" clipping value on x-axis }
171                y_dots    :  REAL;
172                y_first   :  REAL;
173                y_last    :  REAL;
174                y_length  :  REAL;
175                y_north   :  REAL;    {"NORTH" clipping value on y-axis }
176                y_south   :  REAL;    {"SOUTH" clipping value on y-axis }
177                xform_2D  :  matrix;  {2D transformation matrix}
178                xform_3D  :  matrix;  {3D transformation matrix}
179                dot_color :  pixel;
180                fill_color:  white_byte..black_byte;
181                prjctntype:  projection;
182                clip_flag :  BOOLEAN);
183       END {prm RECORD};
184     u_save        :  vector;
185
186
187   {$I 4:DOT1.TEXT}
188
189   {$I 4:DOT2.TEXT}
190
191   {$I 4:DOT3.TEXT}
192
193   END {dot_plotter UNIT}.

        -----    -----    -----    -----    -----

194   {FILE:  4:DOT1.TEXT}
195
196   PROCEDURE clear_transform (PUBLIC(d:  dimension));
197     {"clear_transform" sets the "size" of the "two_D" or "three_D"
198     transformation matrix in the parameter prefix to 1.  When the "size" of the
199     matrix is 1, it is not automatically used when "moveto" or "lineto" is
200     invoked.  "clear_transform" removes the effect of the "set_transform"
201     matrix.}
202   BEGIN
203     CASE d OF
204       two_D  :  prm.xform_2D.size := 1;
205       three_D:  prm.xform_3D.size := 1
206     END
207   END {clear_transform};
208
209   PROCEDURE set_transform (PUBLIC(a:  matrix));
210     {"set_transform" establishes a default "two_D" or "three_D" transformation
211     matrix which is used every time "moveto" or "lineto" is invoked.  The
212     transformation matrix can be changed any number of times.  "clear_transform"
213     removes the effect of the matrix.  Alternately, an identity transformation
214     matrix could be specified.}
215   BEGIN
216     CASE a.size OF
217       3:  prm.xform_2D := a;
218       4:  prm.xform_3D := a
219     END
220   END {set_transfrom};
221
222   PROCEDURE get_transform (PUBLIC(d:  dimension; VAR a:  matrix));
```

```
223     {The default "two_D" or "three_D" transformation matrix can be retreived
224     using "get_transform" for inspection or further modification using other
225     matrix operations.}
226   BEGIN
227     CASE d OF
228       two_D  :  a := prm.xform_2D;
229       three_D:  a := prm.xform_3D
230     END
231   END  {get_transform};
232
233   PROCEDURE clipping (PUBLIC{flag:  BOOLEAN});
234     {This procedure sets the clipping flag to a specified value.  When TRUE
235     is specified, the "clip" procedure will be used following any "lineto"
236     calls.}
237   BEGIN
238     prm.clip_flag := flag
239   END {clipping};
240
241   PROCEDURE set_projection_type (PUBLIC{prj_type:  projection});
242   BEGIN
243     prm.prjctntyp := prj_type
244   END {set_projection_type};
245
246   PROCEDURE project (PUBLIC{u:  vector; VAR v:  vector});
247     {A three-dimensional vector is projected into two dimensions with this
248     procedure.  Orthograpic or perspective projections can be specified with
249     the "set_projection_type" procedure.}
250   BEGIN
251     IF   u.size <> 4      {3D vector}
252     THEN BEGIN
253       WRITELN ('non-3D vector passed to "project".');
254       v.size := 1
255     END
256     ELSE WITH prm DO BEGIN
257       CASE prm.prjctntype OF
258         orthographic:  v :=u;
259         perspective:
260           BEGIN
261             v.vctr[1] := vcx + vsx*u.vctr[1]/u.vctr[3];
262             v.vctr[2] := vcy + vsy*u.vctr[2]/u.vctr[3]
263           END
264       END {CASE};
265       v.size := 3;        {now a 2D vector}
266       v.vctr[3] := 1.0    {last component of homogenous coordinates}
267     END
268   END {project};
269
270   PROCEDURE io_check (proc:  STRING; op_type:  STRING);
271     {"io_check" is invoked after I/O calls to trap unexpected disk I/O errors.}
272   VAR
273     rc:  INTEGER;
274   BEGIN
275     rc := IORESULT;
276     IF   rc <> 0
277     THEN BEGIN
278       WRITELN ('DOT01:');
279       WRITE ('Unexpected I/O error ''',rc,''' in routine ''',proc,''' while ');
280       WRITELN ('performing a ',op_type,' operation.');
281       EXIT (PROGRAM)
282     END
283   END {io_check};
284
285   PROCEDURE write_parms;
286     {"write_parms" copies the in-memory parameters to the diskette file.}
287   VAR
288     blk:  blk_no;
289   BEGIN
290     FOR blk := 0 TO prm_size DIV block_size - 1 DO BEGIN
291       SEEK (plt,blk);
292       MOVELEFT (prm.prm_char[1+block_size*blk],plt^,block_size);
293       PUT (plt);
294       io_check ('write_parms','write')
295     END
296   END {write_parms};
297
```

```
298  PROCEDURE read_parms;
299    {"read_parms" reads the parameter file prefix to define the in-memory
300    parameters.}
301  VAR
302    blk:  blk_no;
303  BEGIN
304    FOR blk := 0 TO prm_size DIV block_size - 1 DO BEGIN
305      SEEK (plt,blk);
306      GET (plt);
307      io_check ('read_parms','read');
308      MOVELEFT (pltA,prm.prm_char[1+block_size*blk],block_size)
309    END;
310    IF  prm.pltfile () file_tag
311    THEN BEGIN
312      WRITELN ('DOT02:');
313      WRITELN ('Terminal Error:  File ''',plt_name,''' is not a plot file.');
314      EXIT (PROGRAM)
315    END
316  END (read_parms);
317
318  PROCEDURE vw_parm_error (title:  STRING; x_min,x_max,y_min,y_max:  REAL);
319    {This procedure reports definition errors in setting the "view" or
320    "window".}
321  BEGIN
322    WRITELN (title);
323    WRITELN ('     x_min =',x_min:7:2,', x_max =',x_max:7:2,
324             ', y_min =',y_min:7:2,', y_max =',y_max:7:2);
325  END {vw_parm_error};
326
327  PROCEDURE window (PUBLIC(x_min,x_max,y_min,y_max:  REAL));
328    {This procedure defines the physical minimum and maximum of the logical
329    screen in world coordinates.  Certain internal constants are defined as
330    a result of defining the logical screen.}
331  BEGIN
332    IF   (x_max ( x_min) OR (y_max ( y_min)
333    THEN BEGIN
334      WRITELN ('DOT03:');
335      vw_parm_error ('Window parameter error(s):',x_min,x_max,y_min,y_max);
336      EXIT (window)
337    END;
338    WITH prm DO BEGIN
339      x_first := x_min;
340      x_last  := x_max;
341      y_first := y_min;
342      y_last  := y_max;
343      x_dots := i_density*x_length  / (x_max - x_min);    {dots/world unit}
344      y_dots := j_density*y_length  / (y_max - y_min);
345      view (x_min,x_max,y_min,y_max)        {set clipping parameters}
346    END {WITH}
347  END {window};
348
349  PROCEDURE view (PUBLIC(x_min,x_max,y_min,y_max:  REAL));
350    {The "view" is intended to be a subset of the "window".  This subset is
351    used for clipping.  This definition of a "view" is not the same as used
352    by Newman and Sproull.}
353  BEGIN
354    IF   (x_min ( prm.x_first) OR
355         (x_max ) prm.x_last)  OR
356         (y_min ( prm.y_first) OR
357         (y_max ) prm.y_last)  OR
358         (x_max ( x_min) OR (y_max ( y_min)
359    THEN BEGIN
360      WRITELN ('DOT04:');
361      vw_parm_error ('View parameter error(s):',x_min,x_max,y_min,y_max);
362      WITH prm DO vw_parm_error ('Window parameters:',
363                                 x_first,x_last,y_first,y_last);
364      EXIT (window)
365    END;
366    WITH prm DO BEGIN
367      x_west  := x_min;    {set clipping parameters}
368      x_east  := x_max;
369      y_south := y_min;
370      y_north := y_max;
371      vcx := 0.5 * (x_east  + x_west );   {3D projection parameters}
372      vcy := 0.5 * (y_north + y_south);
```

```
373       vsx := 0.5 * (x_east  - x_west );
374       vsy := 0.5 * (y_north - y_south)
375     END (WITH)
376   END (view);
377
378   PROCEDURE size (PUBLIC(x_len,y_len:  REAL));
379     {"size" defines the physical size of the logical screen in inches.}
380   VAR
381     error:  BOOLEAN;
382     temp :  INTEGER;
383   BEGIN
384     error := FALSE;
385     WITH prm DO BEGIN
386       temp := ROUND(i_density*x_len);
387       IF   temp > max_i_dots
388       THEN BEGIN
389         WRITELN ('DOT05:');
390         WRITE ('Horizontal dimension (',x_len:6:2,' inches) too large:  ');
391         WRITELN (temp,' > ',max_i_dots,' dots.');
392         error := TRUE
393       END
394       ELSE i_length := temp;
395       temp := ROUND(j_density*y_len);
396       IF   temp > max_j_dots
397       THEN BEGIN
398         WRITELN ('DOT05:');
399         WRITE ('Vertical dimension (',y_len:6:2,' inches) too large:  ');
400         WRITELN (temp,' > ',max_j_dots,' dots.');
401         error := TRUE
402       END
403       ELSE j_length := temp;
404       IF   error
405       THEN EXIT(PROGRAM);
406       x_length := x_len;
407       y_length := y_len;
408       i_blks   := i_length DIV (i_frame_size + 1);
409       j_blks   := j_length DIV (j_frame_size + 1)
410     END (WITH);
411     window (0.0,1.0, 0.0,1.0)
412   END (size);
413
414   PROCEDURE dot_color (PUBLIC(color:  pixel));
415     {"dot_color" sets the color of subsequent pixels traced by "lineto".}
416   BEGIN
417     prm.dot_color := color
418   END (dot_color);
419
420   PROCEDURE fill_color (PUBLIC(color:  pixel));
421     {Diskette blocks are mapped to the logical screen but are not allocated
422     until actually referenced.  The virtual pixels represented by the
423     unallocated (undefined) blocks are assigned a "fill_color" value when
424     they are referenced on output to the printer.  That "fill_color" is
425     defined by this procedure which should only be called once for a given
426     logical screen.}
427   BEGIN
428     CASE color OF
429       white:  prm.fill_color := white_byte;
430       black:  prm.fill_color := black_byte
431     END
432   END (fill_color);
433
434   PROCEDURE begin_dot_plotter_unit (PUBLIC);
435     {This procedure is used to guarantee certain VARiables at least have
436     acceptable default values before they are referenced.  A section of code
437     using the "dot_plotter" UNIT must be enclosed with "begin_dot_plotter_unit"
438     and "end_dot_plotter_unit" procedure calls.}
439   BEGIN
440     IF   SIZEOF(memory_frame) <> block_size  {ideally this check would be}
441     THEN BEGIN                               {made by the compiler!}
442       WRITELN ('DOT06:');
443       WRITELN ('ERROR:  Frames must be the same size as I/O blocks.');
444       WRITELN ('A Frame is ',1+i_frame_size,' x ',1+j_frame_size,' bits = ',
445                SIZEOF(memory_frame),' bytes.');
446       WRITELN ('A Block contains ',block_size,' bytes.');
447       EXIT (PROGRAM)
```

```
448     END;
449
450     IF   SIZEOF(prm) () prm_size  {this check should be made by the compiler}
451     THEN BEGIN
452       WRITELN ('DOT07:');
453       WRITELN ('File parameter prefix is ',SIZEOF(prm),' bytes long.');
454       WRITELN ('Internal variable "prm_size" currently has the value ''',
455                prm_size,' and should be adjusted to a multiple of ');
456       WRITELN ('"block_size" (currently ',block_size,') greater than or ',
457                'equal to the size of the parameter prefix.');
458       EXIT (PROGRAM)
459     END;
460
461     FILLCHAR (prm.prm_char,prm_size,0);  {zero parameter record}
462     close_printer := TRUE;        {close/paginate when exiting graphics mode}
463     min_space := spacedefault;
464     plt_mode := undefined;
465     plt_open_flag := FALSE;
466     clipping (TRUE);
467     clear_transform (two_D);
468     clear_transform (three_D);
469     dot_color (black);
470     fill_color (white);
471     set_projection_type (perspective);
472     size (5.0,5.0);
473     REWRITE (prt,'PRINTER:')
474   END {begin_dot_plotter_unit};
475
476   PROCEDURE end_dot_plotter_unit (PUBLIC);
477     {This procedure ends a block of code started with the
478     "begin_dot_plotter_unit".  See comments with that procedure.}
479   BEGIN
480     IF   plt_open_flg
481     THEN close_plot (keep);
482     IF   close_printer
483     THEN BEGIN
484       PAGE (prt);
485       CLOSE (prt)
486     END
487   END {end_dot_plotter_unit};
488
489   PROCEDURE allocate_frames (max:  INTEGER);
490     {"allocate_frames" allocates as many memory frames as possible from
491     available memory.  These frames are used to hold as many diskette blocks
492     in memory as possible for subsequent manipulation.  The user can define
493     the variable "min_space" to reserve memory for other purposes.  The
494     default for "min_space" is set in "begin_dot_plotter_unit" by the constant
495     "spacedefault" defined in the implementation section of this unit.}
496   BEGIN
497     MARK (heap);  {set heap pointer to later deallocate frames}
498     frames_alloc := 0;
499     WHILE (2*MEMAVAIL ) min_space+block_size) AND
500           (frames_alloc ( max) DO BEGIN
501       frames_alloc := SUCC(frames_alloc);
502       NEW(frame_ptr[frames_alloc])
503     END;
504     IF   frames_alloc = 0
505     THEN BEGIN
506       WRITELN ('DOT08:');
507       WRITELN ('Terminal Error:  No memory frames allocated.');
508       EXIT (PROGRAM)
509     END;
510     next_out := 1;
511     frames_inuse := 0     {frames_inuse (= frames_alloc (= max}
512   END {allocate_frames};
513
514   PROCEDURE open_plot (PUBLIC);
515
516     {"open_plot" could be incorporated as part of the "begin_dot_plotter_unit"
517     but it is possible to create/overlay a plot file many times in the
518     same begin..end_dot_plotter_unit section of code and this procedure is
519     needed to control such file accesses.
520
521     The variable "plt_mode" determines whether this procedure prompts the
522     user for a file name or if it uses one directly supplied by the user.
```

```
523     The default value for "plt_mode" is "undefined".  The user can specify
524     a "plt_mode" of "create" or "overlay" and specify a "plt_name" to avoid
525     the interactive prompt.
526
527     This procedure initializes the parameter prefix (if "create") or
528     reads the existing prefix (if "overlay").  Memory frames are allocated
529     and some introductory informational messages are displayed.}
530
531     PROCEDURE init_plot; {LOCAL to open_plot}
532     VAR             {"open_plot" is too large to compile by itself...}
533       i :  i_blk_no;  {...(error 253) but it will compile with ...}
534       j :  j_blk_no;  {..."init_plot" as a local PROCEDURE.}
535       rc:  INTEGER;
536     BEGIN
537       CASE plt_mode OF
538         undefined, either:
539           REPEAT
540             IF  plt_mode = undefined
541             THEN BEGIN
542               WRITELN ('DOT09:');
543               WRITE   ('Enter plot file name ([disk:]name.extension) ');
544               WRITELN ('or ''EXIT'':');
545               READLN (plt_name);
546               IF  (plt_name = 'EXIT') OR (plt_name = 'exit')
547               THEN EXIT (PROGRAM);
548             END;
549             plt_mode := undefined;
550             RESET (plt,plt_name);
551             rc := IORESULT;
552             IF   rc = 0
553             THEN plt_mode := overlay
554             ELSE
555               IF   rc = 10  {no such file on volume}
556               THEN BEGIN
557                 REWRITE (plt,plt_name);
558                 rc := IORESULT;
559                 IF   rc = 0
560                 THEN plt_mode := create
561               END;
562             IF   rc <> 0
563             THEN WRITELN ('DOT10:  Error ''',rc,''' in opening plot file ''',
564                           plt_name,'''.')
565           UNTIL plt_mode <> undefined;
566         create:
567           BEGIN
568             REWRITE (plt,plt_name);
569             rc := IORESULT;
570             IF   rc <> 0
571             THEN BEGIN
572               WRITELN ('DOT10:');
573               WRITELN ('Error ''',rc,''' in opening plot file ''',plt_name,'''.');
574               EXIT (PROGRAM)
575             END
576           END {create CASE};
577         overlay:
578           BEGIN
579             RESET (plt,plt_name);
580             rc := IORESULT;
581             IF   rc <> 0
582             THEN BEGIN
583               WRITELN ('DOT10:');
584               WRITELN ('Error ''',rc,''' in opening plot file ''',plt_name,'''.');
585               EXIT (PROGRAM)
586             END
587         END {overlay CASE}
588       END {CASE plt_mode};
589
590       CASE plt_mode OF
591       create:
592         BEGIN
593           prm.pltfile := file_tag;
594           prm.n_blks := prm_size DIV block_size;
595           write_parms;
596           FOR i := 0 TO max_i_blks DO    {zero all blocks, including those...}
597             FOR j := 0 TO max_j_blks DO  {...which will not be needed}
```

```
598             prm.blk_table[i,j] := 0;
599           WRITELN ('DOT11:');
600           WRITE ('Creating')
601         END {create CASE};
602       overlay:
603         BEGIN
604           read_parms;
605           WRITELN ('DOT11:');
606           WRITE ('Overlaying')
607         END {overlay CASE}
608       END {CASE plt_mode}
609     END {init_plot};
610
611   BEGIN {open_plot}
612     IF   plt_open_flag
613     THEN WRITELN ('DOT12:  Request ignored to open already open plot file.')
614     ELSE BEGIN
615       init_plot;   {procedure divided  --  too big for compiler}
616       WRITELN ({Creating/Overlaying}' plot file ''',plt_name,'''.  ');
617       WITH prm DO
618       WRITELN ('The plot will be ',x_length:5:2,' inches (',i_length+1:4,
619                ' dots) wide by ',y_length:5:2,' inches (',j_length+1:4,
620                ' dots) high.');
621       allocate_frames (max_frames);
622       WRITELN ('There will be ',frames_alloc:3,' in-memory block frames.');
623       plt_open_flag := TRUE
624     END {ELSE}
625   END {open_plot};

       - - - - -    - - - - -    - - - - -    - - - - -    - - - - -

626   {FILE:  4:DOT2.TEXT}
627
628   PROCEDURE close_plot {PUBLIC(disp:  disposition)};
629     {"close_plot" is called to force all in-memory frames to be written to
630     disk and to update the parameter prefix.  After closing the plot file
631     either "open_plot" or "end_dot_plotter_unit" should follow.  This
632     procedure releases the memory used by the in-memory frames.}
633   VAR
634     frame:  frame_no;
635     i     :  i_blk_no;
636     j     :  j_blk_no;
637   BEGIN
638     IF   plt_open_flag
639     THEN BEGIN
640       IF   frames_inuse = 0
641       THEN WRITELN ('DOT13:  Warning:  No frames used.')
642       ELSE
643         FOR frame := 1 TO frames_inuse DO BEGIN  {write in-memory frames to disk}
644           SEEK (plt,blk_index[frame]);
645           MOVELEFT (frame_ptr[frame]^,plt^,block_size);
646           PUT (plt);
647           io_check ('close_plot','write');
648           WITH prm DO      {remove frame reference from blk_table}
649           blk_table[frame_i_blk[frame],frame_j_blk[frame]] := blk_index[frame];
650         END {FOR};
651       write_parms;
652       RELEASE (heap);
653       CASE disp OF
654         keep   :  CLOSE (plt,LOCK);
655         delete:  CLOSE (plt,PURGE)
656       END;
657       plt_open_flag := FALSE
658     END {THEN}
659     ELSE WRITELN ('DOT14:  Request ignored to close plot file which is not open.')
660   END {close_plot};
661
662   PROCEDURE get_blk (i_blk:  i_blk_no;
663                      j_blk:  j_blk_no;
664                      VAR frame:  frame_no);
665     {This procedure guarantees a needed diskette block is in memory for
666     pixel manipulation.  (However, this procedure should not be called for a
667     block which already exists in memory.)  Given a virtual block address as
668     a (i_blk,j_blk) pair, "get_blk" returns the "frame" number where the
669     block was placed.  FIFO block replacement is used to guarantee the file
```

```
670     is initially sequentially created.  The file must be created sequentially
671     or system problems will occur.  After creation, however, random I/O could
672     be used to read/write any blocks.  But since the file could be extended at
673     any time and the blocks must be added sequentially, FIFO replacement is
674     used in all cases.}
675   VAR
676     blk_number:  blk_no;
677   BEGIN
678     IF   frames_inuse < frames_alloc
679     THEN BEGIN
680       frames_inuse := SUCC(frames_inuse);
681       frame := frames_inuse
682     END
683     ELSE BEGIN
684       frame := next_out;
685       next_out := SUCC(next_out MOD frames_alloc);
686       SEEK (plt,blk_index[frame]);
687       MOVELEFT (frame_ptr[frame]^,plt^,block_size);
688       PUT (plt);
689       io_check ('get_blk','write');
690       WITH prm DO
691       blk_table[frame_i_blk[frame],frame_j_blk[frame]] := blk_index[frame]
692     END;
693     blk_number := prm.blk_table[i_blk,j_blk];
694     IF   blk_number = 0
695     THEN BEGIN
696       FILLCHAR (frame_ptr[frame]^,block_size,prm.fill_color);
697       blk_index[frame] := prm.n_blks;
698       prm.n_blks := SUCC(prm.n_blks)
699     END
700     ELSE BEGIN
701       SEEK (plt,blk_number);
702       GET (plt);
703       io_check ('get_blk','read');
704       MOVELEFT (plt^,frame_ptr[frame]^,block_size);
705       blk_index[frame] := blk_number
706     END;
707     prm.blk_table[i_blk,j_blk] := -frame;
708     frame_i_blk[frame] := i_blk;
709     frame_j_blk[frame] := j_blk
710   END {get_blk};
711
712   PROCEDURE dot_tag (i:  i_dot_no; j:  j_dot_no);
713     {"dot_tag" assigns a pixel its color (and possibly any other desired
714     attributes).  This procedures first calculates the virtual address of
715     the block which should contain the pixel.  If the block is not in memory,
716     "get_blk" is called.  Then the address of the pixel within the block is
717     calculated and the desired "pixel" is assigned the desired attributes.}
718   VAR
719     frame:  frame_no;
720     i_dot:  0..i_frame_size;
721     i_blk:  i_blk_no;
722     j_dot:  0..j_frame_size;
723     j_blk:  j_blk_no;
724   BEGIN
725     WITH prm DO BEGIN
726       i_blk := i DIV (i_frame_size+1);
727       j_blk := j DIV (j_frame_size+1);
728       IF   blk_table[i_blk,j_blk] < 0
729       THEN frame := -blk_table[i_blk,j_blk]
730       ELSE get_blk (i_blk,j_blk,frame);
731       i_dot := i MOD (i_frame_size+1);
732       j_dot := j MOD (j_frame_size+1);
733       frame_ptr[frame]^[i_dot,j_dot] := dot_color;
734     END {WITH}
735   END {dot};
736
737   PROCEDURE dot_seg (i1:  i_dot_no; j1:  j_dot_no;
738                      i2:  i_dot_no; j2:  j_dot_no);
739     {Given the end points of a line segment in terms of dot indices, this
740     procedure defines all the intermediate segment pixels.  This procedure
741     recognizes the following special cases:  a single point, horzontal or
742     vertical segments, segements with slope <= 1.0 and segments with slope
743     > 1.0.  No code optimization has been attempted in this first release.
744     Special line-drawing algorithms should eventually be implemented (see
```

```
745     Newman and Sproull, "Principles of Interactive Computer Graphics",
746     Second Edition, pp. 20-26))
747   VAR
748     i    :  i_dot_no;
749     j    :  j_dot_no;
750     step :  -1..1;
751     slope:  REAL;
752   BEGIN
753     IF   i1 = i2
754     THEN IF   j1 = j2
755       THEN dot_tag (i1,j1)    {single point}
756       ELSE BEGIN              {vertical segment}
757         j := j1;
758         IF   j2 > j1
759         THEN step :=  1
760         ELSE step := -1;
761         REPEAT
762           dot_tag (i1,j);
763           j := j + step
764         UNTIL (j = j2);
765         dot_tag (i2,j2)
766       END
767     ELSE IF   j1 = j2         {horizontal segment}
768       THEN BEGIN
769         i := i1;
770         IF   i2 > i1
771         THEN step :=  1
772         ELSE step := -1;
773         REPEAT
774           dot_tag (i,j1);
775           i := i + step;
776         UNTIL (i = i2);
777         dot_tag (i2,j2)
778       END
779       ELSE BEGIN              {non-vertical, non-horizontal segment}
780         dot_tag (i1,j1);      {first dot}
781         slope := (j2-j1)/(i2-i1);
782         IF  ABS(slope) <= 1.00
783         THEN BEGIN
784           IF   i2 > i1
785           THEN step :=  1
786           ELSE step := -1;
787           i := i1 + step;
788           WHILE  i <> i2 DO BEGIN
789             j := j1 + ROUND((i-i1)*slope);
790             dot_tag (i,j);    {middle dot(s), if any, slope <= 1.00}
791             i := i + step
792           END
793         END
794         ELSE BEGIN
795           slope := 1.0/slope;
796           IF   j2 > j1
797           THEN step :=  1
798           ELSE step := -1;
799           j := j1 + step;
800           WHILE  j <> j2 DO BEGIN
801             i := i1 + ROUND((j-j1)*slope);
802             dot_tag (i,j);    {middle dot(s), if any, slope > 1.00}
803             j := j + step
804           END
805         END;
806         dot_tag (i2,j2)       {last dot in segment}
807       END
808   END {dot_seg};
809
810   PROCEDURE world_to_dot (u:  vector; VAR i:  INTEGER; VAR j:  INTEGER);
811     {This procedure converts a "vector" into dot indices.  A three-
812     dimensional vector is projected into two dimensions automatically.
813     The user-defined world coordinates are used in specifying the "vector".
814     The dot indices are determined by the definition of the logical screen.}
815   VAR
816     v:  vector;
817   BEGIN
818     IF   u.size = 4
819     THEN project (u,u);
```

```
820     WITH prm DO BEGIN
821       i := ROUND(x_dots * (u.vctr[1] - x_first));
822       j := ROUND(y_dots * (u.vctr[2] - y_first))
823     END
824   END (world_to_dot);
825
826   PROCEDURE clip (u1,u2:  vector);
827     {"clip" integrates both two- and three-dimensional clipping into a
828     single procedure.  The input parameters are of the type "vector" and
829     internally have their dimensionality defined.  These clippling algorithms
830     were adapted from Newman and Sproull, "Principles of Interactive
831     Computer Graphics", Second Edition, pp. 66-67 for 2D and p. 345 for 3D.
832     The Newman and Sproull internal "code" procedure was replaced by a
833     "region" procedure for better clarity.  For example, their code '1001'
834     is replaced by a set of regions [north west].  See diagram below.
835
836     This "clip" procedure had internal procedures "region", "clip_2D" and
837     "clip_3D" in addition to the code for its definition.
838
839     The clipping boundary is defined by the "view" procedure.  After a
840     line segment has been clipped, it is displayed by procedure "dot_seg"
841     automatically.}
842   TYPE
843     regions    = (north,south,east,west);
844     region_set = SET OF regions;
845   VAR
846     delta_x,delta_y:  REAL;
847     i1,i2,j1,j2    :  INTEGER;
848     reg,reg1,reg2  :  region_set;
849     visible        :  BOOLEAN;
850     u              :  vector;
851
852     PROCEDURE region (u:  vector; VAR reg: region_set);
853       {This procedure is internal to "clip".  "region" defines the regions
854       a given point is in.  If the set of regions is the null set [], then
855       the point is in the viewing area.  If the union of regions from both
856       points is the empty set, the segment is entirely visible. If the
857       intersection of regions from two different points is not the empty
858       set, the segment must lie entirely off the screen.}
859     BEGIN
860       reg := [];
861       CASE u.size OF                  {Picture and surrounding regions:}
862         3:  WITH prm DO BEGIN
863           IF   u.vctr[1] < x_west     {[north west] |  [north]  | [north east]}
864           THEN reg := [west]          {-------------+-----------+-------------}
865           ELSE                        {    [west]   |    []     |   [east]    }
866             IF   u.vctr[1] > x_east   {-------------+-----------+-------------}
867             THEN reg := [east];       {[south west] |  [south]  | [south east]}
868           IF   u.vctr[2] < y_south
869           THEN reg := reg + [south]
870           ELSE
871             IF   u.vctr[2] > y_north
872             THEN reg := reg + [north]
873         END {3};
874         4:  BEGIN
875           IF   u.vctr[1] < -u.vctr[3]
876           THEN reg := [west]
877           ELSE
878             IF   u.vctr[1] > u.vctr[3]
879             THEN reg := [east];
880           IF   u.vctr[2] < -u.vctr[3]
881           THEN reg := reg + [south]
882           ELSE
883             IF   u.vctr[2] > u.vctr[3]
884             THEN reg := reg + [north]
885         END {4}
886       END {CASE}
887     END {region};
888
889     PROCEDURE clip_2D;
890       {This code is used only for clipping two-dimensional vectors.}
891     VAR
892       delta_x, delta_y:  REAL;
893     BEGIN
894       delta_x := u2.vctr[1]-u1.vctr[1];      { x2 - x1 }
```

```
895       delta_y := u2.vctr[2]-u1.vctr[2];      { y2 - y1 }
896       WITH prm DO BEGIN
897         IF   west IN reg          {crosses west edge}
898         THEN define_2D_vector
899                (x_west,                                          {x-component}
900                u1.vctr[2] + delta_y*(x_west-u1.vctr[1])/delta_x, {y-component}
901                u)                                                {vector}
902         ELSE
903           IF   east IN reg        {crosses east edge}
904           THEN define_2D_vector
905                  (x_east,
906                  u1.vctr[2] + delta_y*(x_east-u1.vctr[1])/delta_x,
907                  u)
908           ELSE
909             IF  south IN reg      {crosses south edge}
910             THEN define_2D_vector
911                    (u1.vctr[1] + delta_x*(y_south-u1.vctr[2])/delta_y,
912                    y_south,
913                    u)
914             ELSE
915               IF   north IN reg   {crosses north edge}
916               THEN define_2D_vector
917                      (u1.vctr[1] + delta_x*(y_north-u1.vctr[2])/delta_y,
918                      y_north,
919                      u);
920       END {WITH}
921     END {clip_2D};
922
923     PROCEDURE clip_3D;
924       {This code is used only for clipping three-dimensional vectors.}
925     VAR
926       t,z:  REAL;
927     BEGIN
928       IF   west IN reg          {crosses west edge}
929       THEN BEGIN
930         t := (u1.vctr[3]+u1.vctr[1])/
931              ((u1.vctr[1]-u2.vctr[1])-(u2.vctr[3]-u1.vctr[3]));
932         z := t*(u2.vctr[3]-u1.vctr[3])+u1.vctr[3];
933         define_3D_vector(-z,                                {x-component}
934              t*(u2.vctr[2]-u1.vctr[2])+u1.vctr[2],          {y-component}
935              z,                                             {z-component}
936              u)                                             {vector}
937       END
938       ELSE
939         IF   east IN reg        {crosses east edge}
940         THEN BEGIN
941           t := (u1.vctr[3]-u1.vctr[1])/
942                ((u2.vctr[1]-u1.vctr[1])-(u2.vctr[3]-u1.vctr[3]));
943           z := t*(u2.vctr[3]-u1.vctr[3])+u1.vctr[3];
944           define_3D_vector (z,t*(u2.vctr[2]-u1.vctr[2])+u1.vctr[2],z, u)
945         END
946         ELSE
947           IF  south IN reg      {crosses south edge}
948           THEN BEGIN
949             t := (u1.vctr[3]+u1.vctr[2])/
950                  ((u1.vctr[2]-u2.vctr[2])-(u2.vctr[3]-u1.vctr[3]));
951             z := t*(u2.vctr[3]-u1.vctr[3])+u1.vctr[3];
952             define_3D_vector (t*(u2.vctr[1]-u1.vctr[1])+u1.vctr[1],-z,z, u)
953           END
954           ELSE
955             IF   north IN reg   {crosses north edge}
956             THEN BEGIN
957               t := (u1.vctr[3]-u1.vctr[2])/
958                    ((u2.vctr[2]-u1.vctr[2])-(u2.vctr[3]-u1.vctr[3]));
959               z := t*(u2.vctr[3]-u1.vctr[3])+u1.vctr[3];
960               define_3D_vector (t*(u2.vctr[1]-u1.vctr[1])+u1.vctr[1],z,z, u)
961             END
962     END {clip_3d};
963
964   BEGIN {clip}
965     region (u1,reg1);
966     region (u2,reg2);
967     visible :=  reg1*reg2 = [];
968     WHILE ((reg1 () []) OR (reg2 () [])) AND visible DO BEGIN
969       reg := reg1;
```

```
970       IF   reg = []
971       THEN reg := reg2;
972       CASE u1.size OF
973         3:  clip_2D;
974         4:  clip_3D
975       END;
976       IF   reg = reg1
977       THEN BEGIN
978         u1 := u;
979         region (u,reg1)
980       END
981       ELSE BEGIN
982         u2 := u;
983         region (u,reg2)
984       END;
985       visible :=  reg1*reg2 = []
986     END (WHILE);
987     IF   visible
988     THEN BEGIN (showline)
989       world_to_dot (u1, i1,j1);
990       world_to_dot (u2, i2,j2);
991       dot_seg (i1,j1, i2,j2)
992     END (showline)
993   END (clip);
994
995   PROCEDURE vector_transform (VAR u:  vector);
996     {"vector_transform" is used by "moveto" and "lineto" to automatically
997     multiply a "vector" by a default transformation matrix if one has
998     been defined.}
999   BEGIN
1000    CASE u.size OF
1001      3:  IF   prm.xform_2D.size = 3
1002          THEN transform (u,prm.xform_2D, u);
1003      4:  IF   prm.xform_3D.size = 4
1004          THEN transform (u,prm.xform_3D, u)
1005    END (CASE)
1006  END (vector_transform);
1007
1008  PROCEDURE moveto (PUBLIC(u:  vector));
1009    {"moveto" sets the current screen position.  This position is defined
1010    in user-defined world units.  Transformation of the point automatically
1011    occurs if a transformation matrix was defined for the dimensionality
1012    ("two_D" or "three_D") of the point.}
1013  BEGIN
1014    vector_transform (u);
1015    IF   prm.clip_flag
1016    THEN u_save := u
1017    ELSE world_to_dot (u, i_save,j_save)
1018  END (moveto);
1019
1020  PROCEDURE lineto (PUBLIC(u:  vector));
1021    {"lineto" draws a straight line from the current screen position to
1022    a new point and resets the current screen position.  Transformation
1023    of the point can automatically occur (see note for "moveto" above).
1024    Pixels traced over by the line segement are automatically selected.
1025    Clipping of the line segment to the view boundary can also automatically
1026    occur.}
1027  VAR
1028    i,j:  INTEGER;
1029  BEGIN
1030    vector_transform (u);
1031    IF   prm.clip_flag
1032    THEN BEGIN
1033      clip (u_save,u);
1034      u_save := u
1035    END
1036    ELSE BEGIN
1037      world_to_dot (u, i,j);
1038      dot_seg (i_save,j_save, i,j);
1039      i_save := i;
1040      j_save := j
1041    END
1042  END (lineto);
```

- -

```
1043  {FILE:  4:DOT3.TEXT}
1044  {$R-  Turn range checking off to speed up execution.}
1045
1046  PROCEDURE put_plot (PUBLIC(x,y:  REAL; copies:  INTEGER; border:  INTEGER));
1047    {This "put_plot" procedure was written specifically for the Integral
1048    Data Systems "Paper Tiger" 560 printer.  It is very device dependent.
1049    This procedure takes the logical screen defined as a diskette file
1050    and transfers it to the 560 printer in graphics mode.  For efficiency
1051    this procedure must eventually be written in assembler language or
1052    compiled with a native code generator when UCSD Pascal Version IV
1053    becomes available.  The execution of this procedure is very slow and it
1054    literally takes hours for plots as large as 8-inches by 8-inches.
1055
1056    The (x,y) parameters to this procedure define the position right of
1057    the left margin (the x-coordinate) and down from the top margin (the
1058    y-coordinate) where the upper-left corner of the plot will be positioned
1059    on a page.  Any number of copies of a plot can automatically be requested.
1060    Any number of border dots can be specified to form a picture "frame".
1061
1062    This procedure has the following internal procedures:  "pgc",
1063    "dot_value", and "first_ ", "middle_ " and "last_printer_scan".
1064    The "_printer_scan" procedures seem to be considerable overhead for
1065    implementation of the border dots.
1066
1067    This procedure must be re-written to allow pixels defined to have
1068    more than two colors.
1069
1070    The "ids560" UNIT which controls certain functions of the IDS 560
1071    printer could not be used in this procedure due to symbol table
1072    space limitations.}
1073
1074  CONST
1075    bits_per_scan =  6;
1076    esc           = 27;  {escape ASCII control character}
1077    etx           =  3;  {enter graphics mode; graphics escape character}
1078    ff            = 12;  {form feed}
1079    glf           = 14;  {graphics line feed}
1080    stx           =  2;  {enter normal printer mode}
1081  VAR
1082    border_bottom :  INTEGER;
1083    border_size   :  INTEGER;
1084    border_top    :  INTEGER;
1085    col           :  j_dot_no;
1086    copy          :  INTEGER;
1087    gc            :  CHAR;
1088    j             :  j_dot_no;
1089    max_scan      :  j_dot_no;
1090    plotbyte      :
1091      PACKED RECORD
1092        CASE INTEGER OF
1093          0:  (byte:  CHAR);
1094          1:  (bits:  PACKED ARRAY[0..7] OF pixel)
1095      END;
1096    prior_nulls   :  INTEGER;
1097    row           :  -1..max_j_dots;
1098    scan          :  j_dot_no;
1099    temp          :  INTEGER;
1100
1101    PROCEDURE pgc (c:  CHAR);
1102      {"pgc" (Put Graphics Character) is local to "put_plot".  It checks for
1103      the graphic escape character and eliminates the output of blank
1104      graphic lines -- except for the line return.  WARNING:  UCSD Pascal
1105      will not allow hexadecimal X'10' to reach printer.}
1106    CONST
1107      etx2 = 131;   {X'80' + etx}
1108      nul  =   0;   {graphics space}
1109      nul2 = 128;   {X'80' + nul}
1110    VAR
1111      ordc : INTEGER;
1112    BEGIN
1113      ordc := ORD(c);
1114      IF   (ordc = nul) OR (ordc = nul2)
1115       THEN prior_nulls := SUCC(prior_nulls) {buffer nulls, eliminate blank lines}
1116      ELSE BEGIN
1117        WHILE prior_nulls ) 0 DO BEGIN
```

```
1118          WRITE (prt,CHR(nul));
1119          prior_nulls := PRED(prior_nulls)
1120        END;
1121        IF   (ordc = etx) OR (ordc = etx2)
1122        THEN WRITE (prt,c,c)          {put etx as graphical escape character}
1123        ELSE WRITE (prt,c)            {put any other character}
1124      END
1125    END {pgc};
1126
1127    FUNCTION dot_value (i:  i_dot_no; j:  j_dot_no):  pixel;
1128    VAR                 {LOCAL to "put_plot"}
1129      frame:  frame_no;
1130      i_blk:  i_blk_no;
1131      i_dot:  0..i_frame_size;
1132      j_blk:  j_blk_no;
1133      j_dot:  0..j_frame_size;
1134    BEGIN
1135      WITH prm DO BEGIN
1136        i_blk := i DIV (i_frame_size + 1);
1137        j_blk := j DIV (j_frame_size + 1);
1138        IF   blk_table[i_blk,j_blk]=0
1139        THEN
1140          CASE fill_color OF
1141            white_byte:  dot_value := white;
1142            black_byte:  dot_value := black
1143          END
1144        ELSE BEGIN
1145          IF   blk_table[i_blk,j_blk] < 0
1146          THEN frame := -blk_table[i_blk,j_blk]
1147          ELSE get_blk (i_blk,j_blk,frame);
1148          i_dot := i MOD (i_frame_size+1);
1149          j_dot := j MOD (j_frame_size+1);
1150          dot_value := frame_ptr[frame]^[i_dot,j_dot]
1151        END {ELSE}
1152      END {WITH}
1153    END {dot_value};
1154
1155    PROCEDURE first_printer_scan;  {LOCAL to "put_plot"}
1156    CONST
1157      white_byte = 128; {X'80'; global value is X'00'}
1158    BEGIN
1159      gc := CHR(black_byte);
1160      IF   border_top > bits_per_scan
1161      THEN BEGIN
1162        FOR col := 0 TO prm.i_length + 2*border_size DO
1163          pgc (gc);
1164        border_top := border_top - bits_per_scan - 1
1165      END
1166      ELSE BEGIN
1167        FOR col := 1 TO border_size DO  {"left" window border}
1168          pgc (gc);
1169        plotbyte.byte := CHR(white_byte);
1170        FOR j := 0 TO border_top-1 DO
1171          plotbyte.bits[j] := black;
1172        gc := plotbyte.byte;
1173        FOR col := 0 TO prm.i_length DO BEGIN
1174          plotbyte.byte := gc;
1175          FOR j := 0 TO bits_per_scan - border_top DO
1176            plotbyte.bits[border_top+j] := dot_value (col,row-j);
1177          pgc (plotbyte.byte)   {"top" border; if possible, first dot rows}
1178        END;
1179        gc := CHR(black_byte);
1180        FOR col := 1 TO border_size DO  {"right" window border}
1181          pgc (gc);
1182        row := row - bits_per_scan - 1 + border_top;
1183        border_top := 0
1184      END
1185    END {first_printer_scan};
1186
1187    PROCEDURE middle_printer_scan;  {LOCAL to "put_plot"}
1188    CONST
1189      white_byte = 128; {X'80'; global value is X'00'}
1190    BEGIN
1191      gc := CHR(black_byte);
1192      FOR col := 1 TO border_size DO
```

```
1193        pgc (gc);
1194      FOR col := 0 TO prm.i_length DO BEGIN
1195        plotbyte.byte := CHR(white_byte);
1196        FOR j := 0 TO bits_per_scan DO
1197          plotbyte.bits[j] := dot_value (col,row-j);
1198        pgc (plotbyte.byte)
1199      END;
1200      gc := CHR(black_byte);
1201      FOR col := 1 TO border_size DO
1202        pgc (gc);
1203      row := row - bits_per_scan - 1
1204    END {middle_printer_scan};
1205
1206    PROCEDURE last_printer_scan;  {LOCAL to "put_plot"}
1207    CONST
1208      white_byte = 128; {X'80'; global value is X'00'}
1209    VAR
1210      k          :  j_dot_no;
1211    BEGIN
1212      plotbyte.byte := CHR(white_byte);
1213      IF   row >= 0
1214      THEN BEGIN
1215        k := bits_per_scan - row;
1216        IF   k > border_bottom
1217        THEN k := border_bottom;
1218        FOR j := 0 TO row + k DO
1219          plotbyte.bits[j] := black;
1220        FOR col := 1 TO border_size DO
1221          pgc (plotbyte.byte);
1222        FOR j := 0 TO row DO
1223          plotbyte.bits[j] := white;
1224        gc := plotbyte.byte;
1225        FOR col := 0 TO prm.i_length DO BEGIN
1226          plotbyte.byte := gc;
1227          FOR j := 0 TO row DO
1228            plotbyte.bits[j] := dot_value (col,row-j);
1229          pgc (plotbyte.byte)
1230        END;
1231        FOR j := 0 TO row DO
1232          plotbyte.bits[j] := black;
1233        FOR col := 1 TO border_size DO
1234          pgc (plotbyte.byte);
1235        border_bottom := border_bottom - k;
1236        row := -1
1237      END
1238      ELSE BEGIN
1239        FOR j := 0 TO border_bottom-1 DO
1240          plotbyte.bits[j] := black;
1241        gc := plotbyte.byte;
1242        FOR col := 0 to prm.i_length + 2*border_size DO
1243          pgc (gc)
1244      END
1245    END {last_printer_scan};
1246
1247  BEGIN {"put_plot" divided into "first_", "middle_ and "last_scan" to compile}
1248    IF   NOT plt_open_flag
1249    THEN BEGIN
1250      RESET (plt,plt_name);
1251      temp := IORESULT;
1252      IF   temp <> 0
1253      THEN BEGIN
1254        WRITELN ('DOT10:');
1255        WRITELN ('Error ''',temp,''' in opening plot file ''',plt_name,'''.');
1256        EXIT (PROGRAM)
1257      END;
1258      read_parms;
1259      allocate_frames (2*(max_i_blks+1))
1260    END;
1261    IF   border < 0       {ensure border_size non-negative}
1262    THEN border_size := 0
1263    ELSE border_size := border;
1264    max_scan := (prm.j_length + 2*border_size) DIV (bits_per_scan + 1);
1265    temp := ROUND(120*x);  {left margin in 1/120-inch increments}
1266    WRITE (prt,CHR(esc),'J,',temp,',$');  {reset left margin}
1267    temp := ROUND(48*y);   {top margin in 1/48-inch increments}
```

```
1268    FOR copy := 1 TO copies DO BEGIN;
1269      border_bottom := border_size;
1270      border_top := border_size;
1271      WRITE (prt,CHR(esc),'H,',temp,',$');  {set top margin}
1272      row := prm.j_length;
1273      WRITE (prt,CHR(etx));  {IDS Paper Tiger 560 graphics mode entry}
1274      FOR scan := 0 TO max_scan DO BEGIN
1275        prior_nulls := 0;
1276        IF   border_top > 0
1277        THEN first_printer_scan
1278        ELSE
1279          IF   row > bits_per_scan
1280          THEN middle_printer_scan
1281          ELSE last_printer_scan;
1282        WRITE (prt,CHR(etx),CHR(glf))   {graphics line feed:  1/12th inch}
1283      END {FOR scan};
1284      WRITE (prt,CHR(etx),CHR(stx));    {exit graphics mode}
1285      IF   copy <> copies
1286      THEN PAGE (prt)
1287    END {FOR copy};
1288    WRITE (prt,CHR(esc),'J,0,$');  {reset left margin to zero}
1289    IF   NOT plt_open_flag
1290    THEN BEGIN
1291      RELEASE (heap);
1292      CLOSE (plt)
1293    END
1294  END {put_plot};
```

```
1   {$L- PRINTER:}
2   {$S+ Put compiler in swapping mode.}
3   {$R+ Turn range checking on.}
4   UNIT matrixops;  {UCSD Pascal, Version II.}
5
6     {$C  Copyright (C) 1982 by Earl F. Glynn, Manhattan, KS.}
7     {Written in January-February 1982; last modified 7 April 1982.}
8
9     {The "matrixops" UNIT contains vector/matrix manipulations for two-
10    and three-dimensional geometric transformations of points or lines for
11    subsequent graphical display.  This UNIT manipulates only "vector"
12    and "matrix" TYPEs as defined in the "global' UNIT.  To use these
13    procedures with a more general matrix TYPE, certain dimensioning and
14    indexing variables must be modified.
15
16    The procedure "transform" in the "global" UNIT should be in this UNIT
17    but compiler symbol table space would be exceeded in the "dotplotter"
18    UNIT which uses "transform" but no other matrix operations.  The "transform"
19    procedure is a vector-matrix product which is a new vector.}
20
21
22  INTERFACE
23    USES global;
24
25  TYPE
26    axis        = (x_axis,y_axis,z_axis);
27    coordinates = (cartesian,polar);
28    rotation    = (cw,ccw);  {cw = clockwise, ccw = counterclockwise}
29
30  VAR
31    indent:  INTEGER;
32
33  PROCEDURE matrix_identity (d:  dimension; VAR a:  matrix);
34  PROCEDURE matrix_inverse  (a:  matrix; VAR b:  matrix; VAR determinant:  REAL);
35  PROCEDURE matrix_multiply (a,b:  matrix; VAR c:  matrix);
36  PROCEDURE print_matrix (title:  STRING; a:  matrix);
37  PROCEDURE print_vector (title:  STRING; u:  vector);
38  PROCEDURE rotate_matrix (d:  dimension; xyz:  axis; angle:  REAL;
39                           direction:  rotation; VAR a:  matrix);
40  PROCEDURE scale_matrix  (u:  vector; VAR a:  matrix);
41  PROCEDURE translate_matrix (u:  vector; VAR a:  matrix);
42  PROCEDURE view_transform_matrix (viewtype:  coordinates;
43                                   azimuth, elevation, distance:  REAL;
44                                   screen_x, screen_y, screen_distance:  REAL;
45                                   VAR a:  matrix);
46
47  IMPLEMENTATION
48
49  PROCEDURE matrix_identity (PUBLIC{d:  dimension; VAR a:  matrix)};
50    {"matrix_identity" creates an idenity matrix for the specified dimension-
51    ality.  While this procedure can be accessed by a user program, its
52    purpose was to initialize matrices used by other procedures in this UNIT.}
53  VAR
54    i,j,n:  index;
55  BEGIN
56    CASE d OF
57      two_D:    n := 3;
58      three_D:  n := 4
59    END;
60    FOR i := 1 TO n DO
61      FOR j := 1 TO n DO
62        IF   i = j
63        THEN a.mtrx[i,j] := 1
64        ELSE a.mtrx[i,j] := 0;
65    a.size := n
66  END {matrix_identity};
67
68  PROCEDURE matrix_inverse
69                (PUBLIC{a:  matrix; VAR b:  matrix; VAR determinant:  REAL)};
70    {This procedure inverts a general transformation matrix.  The user need
71    not form an inverse geometric transformation by keeping a product of
72    the inverses of simple geometric transformations:  translations, rotations
73    and scaling.  A determinant of zero indicates no inverse was possible for
74    a singular matrix.}
75  VAR
```

```
  i,i_pivot:  index;
  i_flag   :  PACKED ARRAY[index] OF BOOLEAN;
  j,j_pivot:  index;
  j_flag   :  PACKED ARRAY[index] OF BOOLEAN;
  modulus  :  REAL;
  n        :  index;
  pivot    :  REAL;
  pivot_col:  PACKED ARRAY[index] OF index;
  pivot_row:  PACKED ARRAY[index] OF index;
  temporary:  REAL;
BEGIN                                 {The matrix inversion algorithm used here}
  WITH a DO BEGIN                     {is similar to the "maximum pivot strategy"}
    FOR i := 1 TO size DO BEGIN       {described in "Applied Numerical Methods"}
      i_flag[i] := TRUE;              {by Carnahan, Luther and Wilkes, pp. 282-284.}
      j_flag[i] := TRUE               {The particular algorithm used here was}
    END;                              {presented by the instructor of the Calc III}
    modulus := 1.0;                   {class (Summer 1972 at K-State) whose name}
    FOR n := 1 TO size DO BEGIN       {I cannot remember.}
      pivot := 0.0;
      IF   ABS(modulus) > 0.0
      THEN BEGIN
        FOR i := 1 TO size DO         {This algorithm is particularly well}
          IF   i_flag[i]              {suited for "hand calculations".}
          THEN                        {If the original matrix elements are}
            FOR j := 1 TO size DO     {integers, they remain integers thoughout}
              IF   j_flag[j]          {subsequent calculations until the last}
              THEN                    {step -- division by the determinant.}
                IF   ABS(mtrx[i,j]) > ABS(pivot)
                THEN BEGIN
                  pivot := mtrx[i,j];   {largest value on which to pivot}
                  i_pivot := i;         {indices of pivot element}
                  j_pivot := j
                END;
        IF   defuzz(pivot) = 0        {If pivot is too small, consider}
        THEN modulus := 0             {the matrix to be singular}
        ELSE BEGIN
          pivot_row[n] := i_pivot;
          pivot_col[n] := j_pivot;
          i_flag[i_pivot] := FALSE;
          j_flag[j_pivot] := FALSE;
          FOR i := 1 TO size DO
            IF   i <> i_pivot
            THEN
              FOR j := 1 TO size DO  {pivot column unchanged}
                IF   j <> j_pivot    {for elements not in pivot row or column ...}
                THEN mtrx[i,j] := (mtrx[i,j]*mtrx[i_pivot,j_pivot] -
                                   mtrx[i_pivot,j]*mtrx[i,j_pivot])
                                   / modulus;              {2x2 minor / modulus}
          FOR j := 1 TO size DO
            IF   j <> j_pivot         {change signs of elements in pivot row}
            THEN mtrx[i_pivot,j] := -mtrx[i_pivot,j];
          temporary := modulus;       {exchange pivot element and modulus}
          modulus := mtrx[i_pivot,j_pivot];
          mtrx[i_pivot,j_pivot] := temporary
        END
      END
    END {FOR n}
  END {WITH};
  determinant := defuzz(modulus);
  IF   determinant <> 0
  THEN BEGIN
    b.size := a.size;             {The matrix inverse must be unscrambled}
    FOR i := 1 TO a.size DO       {if pivoting was not along main diagonal.}
      FOR j := 1 TO a.size DO
        b.mtrx[pivot_row[i],pivot_col[j]] := defuzz(a.mtrx[i,j]/determinant)
  END
END {matrix_inverse};

PROCEDURE matrix_multiply (PUBLIC(a,b:  matrix; VAR c:  matrix));
  {Compound geometric transformation matrices can be formed by multiplying
  simple transformation matrices.  This procedure only multiplies together
  matrices for two- or three-dimensional transformations, i.e., 3x3 or 4x4
  matrices.  The multiplier and multiplicand must be of the same dimension.}
VAR
  i,j,k:  index;
```

```
151     temp :  REAL;
152   BEGIN
153     c.size := a.size;
154     IF   a.size = b.size
155     THEN
156       FOR i := 1 TO a.size DO
157         FOR j := 1 TO a.size DO
158           BEGIN
159             temp := 0;
160             FOR k := 1 TO a.size DO
161               temp := temp + a.mtrx[i,k]*b.mtrx[k,j];
162             c.mtrx[i,j] := defuzz(temp)
163           END
164     ELSE BEGIN
165       WRITELN ('MAT01:');
166       WRITELN ('Ignoring attempt to multiply square matrices of ',
167                'different dimensions:  ',a.size, ' and ',b.size,'.');
168       c.size := 1  {signal error by setting dimension of "c" to 1}
169     END
170   END {matrix_multiply};
171
172   PROCEDURE print_matrix {PUBLIC(title:  STRING; a:  matrix)};
173     {"print_matrix" can be used to print a transformation matrix with a title.}
174     {Be sure that the variable "indent" declared in the INTERFACE of this
175     procedure has been defined before using either "print_matrix" or
176     "print_vector".}
177   VAR
178     i,j:  index;
179     k  :  INTEGER;
180   BEGIN
181     WRITELN (prt);
182     WRITELN (prt,title);
183     FOR i := 1 TO a.size DO
184       BEGIN
185         FOR k := 1 TO indent DO
186           WRITE (prt,' ');
187         FOR j := 1 TO a.size DO
188           WRITE (prt,a.mtrx[i,j]:12:5);
189         WRITELN (prt)
190       END
191   END {print_matrix};
192
193   PROCEDURE print_vector {PUBLIC(title:  STRING; u:  vector)};
194     {"print_vector" lists a 2D or 3D vector (point) along with a title.}
195   VAR
196     j:  index;
197     k:  INTEGER;
198   BEGIN
199     WRITELN (prt);
200     WRITELN (prt,title);
201     FOR k := 1 TO indent DO
202       WRITE (prt,' ');
203     FOR j := 1 TO u.size DO
204       WRITE (prt,u.vctr[j]:12:5);
205     WRITELN (prt)
206   END {print_vector};
207
208   PROCEDURE rotate_matrix {PUBLIC(d:  dimension;  (* two_D or three_D *)
209                           xyz:  axis;   (* x_axis, y_axis or z_axis *)
210                           angle :  REAL;          (* degrees *)
211                           direction:  rotation    (* cw or ccw *)
212                           VAR a:  matrix)};
213     {This procedure defines a matrix for a two- or three-dimensional rotation.
214     To avoid possible confusion in the sense of the rotation, "cw" for
215     clockwise or "ccw" for counter-clockwise must always be specified along
216     with the axis of rotation.  Two-dimensional rotations are assumed to
217     be about the z-axis in the x-y plane.
218
219     A rotation about an arbitrary axis can be performed with the following
220     steps:
221       (1) Translate the object into a new coordinate system where (x,y,z)
222           maps into the origin (0,0,0).
223       (2) Perform appropriate rotations about the x and y axes of the
224           coordinate system so that the unit vector (a,b,c) is mapped into
225           the unit vector along the z axis.
```

```
226       (3) Perform the desired rotation about the x-axis of the new
227           coordinate system.
228       (4) Apply the inverse of step (2).
229       (5) Apply the inverse of step (1).)
230   VAR
231     cosx   :  REAL;
232     radians:  REAL;
233     sinx   :  REAL;
234   BEGIN
235     radians := radians_per_degree * angle (degrees);
236     IF   direction = ccw   {ccw is -cw}
237     THEN radians := -radians;
238     cosx := defuzz( COS(radians) );
239     sinx := defuzz( SIN(radians) );
240     CASE d OF
241       two_D:
242         BEGIN
243           matrix_identity (two_D,a);
244           CASE xyz OF
245             x_axis,y_axis:
246               WRITELN ('MAT02:  FOR 2D rotation in x-y plane, specify "z_axis"');
247             z_axis:
248               BEGIN                          { z=angle of rotation }
249                 a.mtrx[1,1] :=  cosx;        { cos(z)  -sin(z)   0 }
250                 a.mtrx[2,2] :=  cosx;        { sin(z)   cos(z)   0 }
251                 a.mtrx[2,1] :=  sinx;        { 0        0        1 }
252                 a.mtrx[1,2] := -sinx
253               END
254           END
255         END;
256       three_D:
257         BEGIN
258           matrix_identity (three_D,a);
259           CASE xyz OF
260             x_axis:
261               BEGIN
262                 a.mtrx[2,2] :=  cosx;        { 1  0       0       0 }
263                 a.mtrx[3,3] :=  cosx;        { 0  cos(z) -sin(z)  0 }
264                 a.mtrx[2,3] := -sinx;        { 0  sin(z)  cos(z)  0 }
265                 a.mtrx[3,2] :=  sinx         { 0  0       0       1 }
266               END;
267             y_axis:
268               BEGIN
269                 a.mtrx[1,1] :=  cosx;        { cos(z)  0  sin(z)  0 }
270                 a.mtrx[3,3] :=  cosx;        { 0       1  0       0 }
271                 a.mtrx[3,1] := -sinx;        {-sin(z)  0  cos(z)  0 }
272                 a.mtrx[1,3] :=  sinx         { 0       0  0       1 }
273               END;
274             z_axis:
275               BEGIN
276                 a.mtrx[1,1] :=  cosx;        { cos(z) -sin(z)  0  0 }
277                 a.mtrx[2,2] :=  cosx;        { sin(z)  cos(z)  0  0 }
278                 a.mtrx[2,1] :=  sinx;        { 0       0       1  0 }
279                 a.mtrx[1,2] := -sinx         { 0       0       0  1 }
280               END
281         END
282       END
283     END
284   END {rotate_matrix};
285
286   PROCEDURE scale_matrix (PUBLIC(u:  vector; VAR a:  matrix));
287     {"scale_matrix" accepts a "vector" containing the scaling factors for
288     each of the dimensions and creates a scaling matrix.  The size
289     of the vector dictates the size of the resulting matrix.}
290   VAR
291     d:  dimension;
292     i:  index;
293   BEGIN                              { 2D: }          { 3D: }
294     CASE u.size OF                   { Sx 0  0 }      { Sx 0  0  0 }
295       3:  d := two_D;                { 0  Sy 0 }      { 0  Sy 0  0 }
296       4:  d := three_D               { 0  0  1 }      { 0  0  Sz 0 }
297     END;                                              { 0  0  0  1 }
298     matrix_identity (d,a);
299     FOR i := 1 TO u.size-1 DO
300       a.mtrx[i,i] := u.vetr[i];
```

```
  a.mtrx[u.size,u.size] := 1.0
END {scale_matrix};

PROCEDURE translate_matrix (PUBLIC(u: vector; VAR a: matrix));
  {"translate_matrix" defines a translation transformation matrix. The
  components of the vector "u" determine the translation components.}
VAR
  d: dimension;
  i: index;
BEGIN                              { 2D: }       { 3D: }
  CASE u.size OF                   { 1  0  0 }   { 1  0  0  0 }
    3: d := two_D;                 { 0  1  0 }   { 0  1  0  0 }
    4: d := three_D                { Tx Ty 1 }   { 0  0  1  0 }
  END;                                           { Tx Ty Tx 1 }
  matrix_identity (d,a);
  FOR i := 1 TO u.size-1 DO
    a.mtrx[u.size,i] := u.vctr[i]
END {translate_matrix};

PROCEDURE view_transform_matrix (PUBLIC(viewtype: coordinates;
          azimuth (*or x*), elevation (*or y*), distance (*or z*): REAL;
          screen_x, screen_y, screen_distance: REAL;
          VAR a: matrix));
  {"view_transform_matrix" creates a transformation matrix for changing
  from world coordinates to eye coordinates. The location of the "eye"
  from the "object" is given in polar (azimuth,elevation,distance)
  coordinates or (x,y,z) cartesian coordinates. The size of the screen
  is "screen_x" units horizontally and "screen_y" units vertically. The
  eye is "screen_distance" units from the viewing screen. A large ratio
  "screen_distance/screen_x (or screen_y)" specifies a narrow aperature
  -- a telephoto view. Conversely, a small ratio specifies a large aperature
  -- a wide-angle viw. This view transform matrix is very useful as the
  default three-dimensional transformation matrix. Once set, all points
  are automatically transformed.}
VAR
  b           : matrix;
  cosm        : REAL;          {COS(-angle)}
  hypotenuse  : REAL;
  rad_azimuth : REAL;
  rad_elevation: REAL;
  sinm        : REAL;          {SIN(-angle)}
  temporary   : REAL;
  u           : vector;
  x,y,z       : REAL;
BEGIN
  CASE viewtype OF
    cartesian:
      BEGIN
        x := azimuth;   {The parameters are renamed to avoid confusion.}
        y := elevation;
        z := distance;
        define_3D_vector (-x,-y,-z, u)
      END;
    polar:
      BEGIN
        rad_azimuth   := radians_per_degree * (azimuth - 90.0);
        rad_elevation := radians_per_degree * elevation;
        temporary := -distance * COS(rad_elevation);
        define_3D_vector (temporary * COS(rad_azimuth),
                          temporary * SIN(rad_azimuth),
                          -distance * SIN(rad_elevation), u);
      END
    END (CASE);
  translate_matrix (u, a);         {translate origin to "eye"}
  rotate_matrix (three_D,x_axis,90.0,cw, b);
  matrix_multiply (a,b, a);
  CASE viewtype OF
    cartesian:
      BEGIN
        temporary := SQR(x) + SQR(y);
        hypotenuse := SQRT(temporary);
        cosm := -y/hypotenuse;
        sinm :=  x/hypotenuse;
        matrix_identity (three_D,b);
        b.mtrx[1,1] :=  cosm;
```

```
376           b.mtrx[3,3] :=  cosm;
377           b.mtrx[3,1] := -sinm;
378           b.mtrx[1,3] :=  sinm;
379           matrix_multiply (a,b, a);
380           cosm := hypotenuse;
381           hypotenuse := SQRT(temporary + SQR(z));
382           cosm := cosm/hypotenuse;
383           sinm := -z/hypotenuse;
384           matrix_identity (three_D,b);
385           b.mtrx[2,2] :=  cosm;
386           b.mtrx[3,3] :=  cosm;
387           b.mtrx[2,3] := -sinm;
388           b.mtrx[3,2] :=  sinm
389         END;
390       polar:
391         BEGIN
392           rotate_matrix (three_D,y_axis,-azimuth,ccw, b);
393           matrix_multiply (a,b, a);
394           rotate_matrix (three_D,z_axis,elevation,ccw, b);
395         END
396     END (CASE);
397     matrix_multiply (a,b, a);
398     define_3D_vector (screen_distance/(0.5*screen_x),
399                       screen_distance/(0.5*screen_y),-1.0, u);
400     scale_matrix (u, b);  {reverse sense of z-axis; screen transformation}
401     matrix_multiply (a,b, a)
402   END (view_transform_matrix);
403
404   END (matrix_ops UNIT).
```

```
1   {$L- PRINTER:}
2   {$S+ Put compiler in swapping mode.}
3   UNIT ids560;  {UCSD Pascal, Version II.}
4
5     {$C  Copyright (C) 1982 by Earl F. Glynn, Manhattan, KS.}
6     {Written in January 1982; last modified on 3 April 1982.}
7
8     {The "ids560" UNIT provides control of the Integral Data Systems "Paper
9     Tiger" 560.  The IDS 560 has other hardware features which are not
10    controlled by procedures in this UNIT.}
11
12  INTERFACE
13
14  USES global;    {for definition of "prt" TEXT FILE}
15
16  TYPE
17    control_char = (null,enhanced_mode,normal_mode,graphics_mode,
18                    just_on,just_off,  {"justify"}
19                    fixed_spacing,
20                    ht,lf,vt,ff,cr,
21                    proportional_spacing,
22                    select_print,deselect_printer,
23                    subscript,superscript,
24                    pitch10,pitch12,pitch16);
25
26  PROCEDURE forms (length,skip:  REAL);
27  PROCEDURE control (code:  control_char);
28  PROCEDURE line (n:  INTEGER);
29  PROCEDURE margins (left,right:  REAL);
30  PROCEDURE position (x,y:  REAL);
31  PROCEDURE tab (n:  INTEGER);
32
33
34  IMPLEMENTATION
35
36  CONST esc = 27;
37
38  PROCEDURE forms (PUBLIC(length,skip:  REAL));
39    {"forms" defines the length in inches of the forms being used.  A skip
40    space must also be specified which is greater than zero but less than
41    the size of the forms.  A zero skip space is treated as 1/48-th inch.
42    Typically, the skip space is 0.5 or 1.0 inch.}
43  VAR
44    l,s:  INTEGER;
45  BEGIN
46    l := ROUND(48.0*length);
47    s := ROUND(48.0*skip);
48    IF   s (= 0
49    THEN s := 1;
50    WRITE (prt,CHR(esc),'L,',l,',',l-s,',$')
51  END {forms};
52
53  PROCEDURE control (PUBLIC(code:  control_char));
54    {The IDS 560 recognizes many special control characters which can be
55    symbolically selected with this "control" procedure.}
56  BEGIN
57    CASE code OF
58      null            :  WRITE (prt,CHR(00));
59      enhanced_mode   :  WRITE (prt,CHR(01));
60      normal_mode     :  WRITE (prt,CHR(02));
61      graphics_mode   :  WRITE (prt,CHR(03));
62      just_on         :  WRITE (prt,CHR(04));  {justify on}
63      just_off        :  WRITE (prt,CHR(05));  {justify off}
64      fixed_spacing   :  WRITE (prt,CHR(06));
65      ht              :  WRITE (prt,CHR(09));  {horizontal tab}
66      lf              :  WRITE (prt,CHR(10));  {line feed}
67      vt              :  WRITE (prt,CHR(11));  {vertical tab}
68      ff              :  WRITE (prt,CHR(12));  {form feed}
69      cr              :  WRITE (prt,CHR(13));  {carriage return}
70      proportional_spacing:  WRITE (prt,CHR(16));
71      select_printer  :  WRITE (prt,CHR(17));
72      deselect_printer:  WRITE (prt,CHR(19));
73      subscript       :  WRITE (prt,CHR(20));
74      superscript     :  WRITE (prt,CHR(25));
75      pitch10         :  WRITE (prt,CHR(29));  {10 characters/inch}
```

```
 76      pitch12         : WRITE (prt,CHR(30));  {12 characters/inch}
 77      pitch16         : WRITE (prt,CHR(31));  {16.8 characters/inch}
 78    END
 79  END {control};
 80
 81  PROCEDURE line (PUBLIC(n:  INTEGER));
 82    {"line" places the print head at line "n" from the top of the form.  The
 83    top line is line 0.  Reverse paper feeding will occur if necessary.}
 84  BEGIN
 85    WRITE (prt,CHR(esc),'M,',n,',$')
 86  END {line};
 87
 88  PROCEDURE margins (PUBLIC(left,right:  REAL));
 89    {"margins" sets the left and right margins of the forms being used.
 90    Default margins at power up are 0.0 to 13.2 inches.}
 91  BEGIN
 92    WRITE (prt,CHR(esc),'J,',ROUND(120.0*left),',',ROUND(120.0*right),',$')
 93  END {margins};
 94
 95  PROCEDURE position (PUBLIC(x,y:  REAL));
 96    {"position" print head "x" inches right of the left margin and "y"
 97    inches down from the top of the current form setting.  Reverse paper
 98    feeding can occur and care should be taken to avoid binding or jamming
 99    of paper.}
100  BEGIN
101    WRITE (prt,CHR(esc),'G,',ROUND(120.0*x),',$',  {horizontal setting}
102               CHR(esc),'H,',ROUND(48.0*y),',$')   {vertical setting}
103  END {position};
104
105  PROCEDURE tab (PUBLIC(n:  INTEGER));
106    {"tab" does not use the horizontal tab programming available on the
107    IDS 560 but rather absolute column positioning.  "tab" advances the
108    print head to the n-th column given the current margins and pitch.
109    The print head can move left or right to the n-th column.}
110  BEGIN
111    WRITE (prt,CHR(esc),'N,',n,',$')
112  END {tab};
113
114  END {ids560 UNIT}.
```

Appendix B. Source Listings of Sample User PROGRAMs

```
{$L- PRINTER:}
{$S+ Put compiler in swapping mode}
PROGRAM map;
  {"map" produces several maps of the State of Kansas.  This program
  demonstrates the use of two-dimensional primitives.  Written in
  January 1982; last modified on 5 April 1982.}
USES global, matrixops, ids560, dotplotter;

VAR
  loop :  INTEGER;
  start:  REAL;

PROCEDURE read_map;
VAR
  ipt:  TEXT;
  n  :  INTEGER;
  u  :  vector;
  x  :  REAL;
  y  :  REAL;
BEGIN
  RESET (ipt,'8:kansas.map.text');
  n := 1;
  READ (ipt,x,y);
  define_2D_vector (x,y, u);
  moveto (u);
  READ (ipt,x,y);
  WHILE NOT EOF(ipt) DO BEGIN
    define_2D_vector (x,y, u);
    lineto (u);
    n := SUCC(n);
    IF   n MOD 8 = 0
    THEN READLN(ipt);
    WRITELN ('Point ',n:4);
    READ (ipt,x,y)
  END;
  CLOSE (ipt)
END (read_map);

PROCEDURE setup_case (i:  INTEGER);
VAR
  a,b:  matrix;
  u  :  vector;
BEGIN
  position (1.50,start);
  control (pitch10);
  CASE i OF
    0:  BEGIN
          WRITELN (prt,'1.1  State of Kansas');
          position (2.00,start+0.125);
          control (pitch16);
          WRITELN (prt,'Demonstrates plotting of unmodified line segments')
        END;
    1:  BEGIN
          WRITELN (prt,'1.1  Local Counties:  Riley, Pottawatomie, et al');
          position (2.00,start+0.125);
          control (pitch16);
          WRITELN (prt,'Demonstrates symmetrical scaling, translation ',
                       'and clipping of line segments');
          define_2D_vector (4.0,4.0, u);
          scale_matrix (u, a);
          define_2D_vector (-21000.0,-10000.0, u);
          translate_matrix (u, b);
          matrix_multiply (a,b, a);
          set_transform (a)
        END;
    2:  BEGIN
          WRITELN (prt,'1.3  "Creative Cartography"');
          position (2.00,start+0.125);
          control (pitch16);
          WRITELN (prt,'Demonstrates asymmetrical scaling, rotation, ',
                       'translation and clipping');
          position (2.00,start+0.250);
          WRITELN (prt,'of line segments; color inversion');
          define_2D_vector (0.75,1.25, u);
          scale_matrix (u, a);
```

```
 76            rotate_matrix (two_D, z_axis, 45.00, cw, b);
 77            matrix_multiply (a,b, a);
 78            define_2D_vector (100.0,4000.0, u);
 79            translate_matrix (u, b);
 80            matrix_multiply (a,b, a);
 81            set_transform (a);
 82            dot_color (white);
 83            fill_color (black)
 84          END
 85      END;
 86    END (setup_case);
 87
 88    BEGIN (map)
 89      begin_dot_plotter_unit;
 90        forms (11.0,8.0);
 91        PAGE (prt);
 92        min_space := 4096 (bytes);  (stack overflow probably caused by procedure)
 93        plt_mode := create;         (calls to "setup_case" and "read_map")
 94        plt_name := '6:KANSAS.DOTS';
 95        size (4.000,2.333);
 96        window (0,8500, 0,5000);
 97        position (1.25,8.50);
 98        control (pitch16);
 99        control (enhanced_mode);
100        WRITE (prt,'Exhibit 1. Cartography Examples');
101        control (normal_mode);
102        FOR loop := 0 TO 2 DO BEGIN   (three maps)
103          start := 0.80 + 3.10*loop;
104          setup_case (loop);
105          open_plot;
106          read_map;
107          put_plot (2.00,start+0.40, 1 (copy), 2 (border dots));
108          close_plot (delete)
109        END;
110      end_dot_plotter_unit
111    END (map).
```

```
 037  752  062 1113  070 1238  110 1957  109 1972  142 2457  150 2569  178 3068
 183 3181  225 3706  221 3800  262 4400  939 4353  909 3757  935 3756  899 3150
 797 3151  767 2540  670 2544  634 1933  660 1929  626 1336  640 1334  637 1209
 409  723  575  722  541  182 1074  151 1103  696 1080  698 1110 1176 1113 1306
1092 1310 1131 1911 1105 1908 1135 2521 1486 2503 1511 3117 1611 3108 1639 3716
1623 3718 1649 4312 2229 4281 2209 3686 2222 3687 2201 3084 2181 2463 2082 2467
2064 1859 2058 1500 2052 1374 2043 1265 2065 1257 2040  761 2165  751 2159  639
2178  636 2149  100 2233  094 2731  073 2754  589 2740  594 2743  719 2763 1087
2768 1212 2758 1217 2760 1345 2769 1587 3005 1573 3006 1459 3356 1449 3006 1459
3005 1573 2769 1587 2786 1820 2770 1832 2775 1960 2788 2427 2775 2432 2795 3053
2813 3653 2792 3657 2806 4259 2807 4255 3292 4234 3396 4237 3385 3628 3406 3626
3391 3029 3376 2411 3361 1923 3363 1804 3486 1798 3477 1441 3356 1449 3356 1194
3353 1061 3336  698 3331  570 3325  058 3836  041 4047  033 4060  560 3937  565
3937  680 3949 1050 3952 1178 3964 1661 3967 1788 3954 1780 3963 2150 3974 2390
3961 2408 3965 2635 3976 3008 3979 3120 3991 3608 4579 3584 3991 3608 3978 3607
3982 4223 4225 4214 4579 4216 4583 3699 5175 3691 4583 3699 4579 3584 4573 3216
4570 3107 4563 2737 4560 2632 4561 2255 4556 2137 4549 1647 4789 1638 4783 1276
4774 1031 4665 1026 4653  659 4650  546 4643  028 5018  017 5340  008 5350  647
5360 1267 5363 1505 5364 1632 5136 1639 5152 2241 5157 2627 5129 2626 5130 2663
5159 2661 5156 2726 5165 2969 5168 3206 5175 3566 5603 3562 5175 3566 5175 3691
5179 4195 5672 4192 5767 4193 5765 3565 6020 3568 6016 3544 5977 3532 5954 3518
5943 3484 5898 3438 5868 3408 5884 3312 5942 3244 5964 3198 5968 3170 6003 3145
6015 3109 6042 3094 6053 3071 6020 3044 6034 3026 6054 3038 6115 3014 6156 3039
6191 3020 6215 3010 6210 2867 6093 2862 6090 2594 6085 2536 6210 2528 6212 2413
6243 2409 6242 2107 6232 1633 6230 1497 6045 1492 6045  830 6044  646 6043  406
6039  009 6618  016 6649  023 6651  009 6656  544 6656  821 6661 1016 6665 1444
6664 1616 6662 1988 6669 2408 6672 2601 6679 2940 6582 2962 6577 3095 7054 3093
6577 3095 6587 3561 6378 3564 6378 3564 6372 4183 6599 4185 6837 4189 6845 3697
6840 3563 6587 3561 6840 3563 6845 3697 7072 3703 7069 3375 7054 3093 7043 2920
7059 2892 7103 2899 7127 2892 7143 2885 7141 2609 7143 2426 7139 2004 7139 1944
7141 1759 7136 1754 7127 1449 7125 1024 7128  547 7137  028 7257  022 7624  024
7614  486 7604  540 7601  944 7602 1030 7609 1447 7613 1953 7612 2432 7607 2768
7591 2765 7582 2751 7556 2740 7519 2757 7482 2758 7481 2851 7471 3375 7731 3388
7471 3375 7069 3375 7072 3703 7314 3705 7554 3704 7314 3705 7303 4204 7324 4212
7357 4199 7412 4144 7439 4103 7491 4066 7540 4016 7628 4027 7650 4057 7676 4063
7698 4087 7730 4013 7788 4005 7798 3956 7789 3906 7753 3894 7817 3849 7776 3840
```

```
7700 3840 7702 3788 7639 3730 7618 3680 7601 3635 7554 3704 7601 3635 7565 3571
7566 3545 7615 3495 7700 3444 7731 3388 7748 3360 7799 3336 7796 3262 7833 3181
7889 3097 7785 3090 7784 2986 7771 2980 7780 2814 7771 2980 7784 2986 7785 3090
7889 3097 7937 3073 7969 3053 8026 3068 8071 3034 8108 3019 8129 3018 8117 2991
8101 2876 8102 2587 8108 2438 8109 2074 8114 1959 8111 1495 8112 1472 8114  959
8113  928 8115  532 8114  493 8116  107 8119  035 7711  028 7624  024 7257  022
7137  028 6841  024 6649  023 6610  016 6039  009 5806  002 5340  008 5018  017
4643  020 4310  026 4047  033 3836  041 3325  058 2817  069 2731  073 2233  094
2149  100 1563  125 1507  668 1563  125 1215  143 1074  151  541  182  037  208
 802  213  037  752  575  722  609  723 1080  698 1103  696 1563  667 1587  668
1563  667 1575  782 2040  761 2165  751 2743  719 2740  594 2754  589 3331  570
3336  698 3937  688 3937  565 4060  560 4650  546 4653  659 5350  647 6044  646
6043  406 6651  409 6656  544 7128  547 7604  540 7616  486 8114  493 8115  532
8113  928 8114  959 7601  944 7602 1030 7125 1024 6661 1016 6656  821 6045  830
6044  646 5350  647 5360 1267 4783 1276 4774 1031 4665 1026 3949 1050 3952 1178
3356 1194 3353 1061 2763 1087 2768 1212 2758 1217 2760 1345 2052 1374 2058 1500
1602 1529 1589 1286 1599 1281 1591 1150 1575  782 1591 1150 1110 1176  637 1209
 870 1238  110 1957  634 1933  660 1929 1105 1908 1131 1911 1593 1890 1619 2493
1593 1890 2064 1859 2770 1832 2775 1960 3361 1923 3363 1804 3486 1798 3954 1780
3967 1788 3964 1661 4225 1660 4226 1637 4301 1637 4309 1648 4549 1647 4789 1638
5136 1639 5364 1632 5363 1505 5692 1502 5702 1713 5710 1714 5719 1888 5731 1897
5733 2113 5731 1897 5719 1888 5710 1714 5702 1713 5692 1502 6045 1492 6230 1497
6232 1633 6290 1637 6316 1623 6361 1622 6664 1616 6665 1444 7127 1449 7609 1447
8112 1472 8111 1495 8114 1959 7613 1953 7139 1944 7139 2004 6662 1988 6669 2408
6243 2409 6242 2107 5733 2113 5634 2107 5630 2231 5634 2516 5678 2515 5673 2590
5678 2515 5634 2516 5630 2231 5152 2241 4561 2255 4556 2137 3963 2158 3974 2390
3961 2400 3376 2411 2788 2427 2775 2432 2181 2463 2082 2467 1619 2493 1486 2503
1135 2521  767 2540  670 2544  150 2569  178 3068  183 3181  797 3151  899 3150
1511 3117 1611 3108 2201 3084 2795 3053 3391 3029 3976 3000 3965 2635 4560 2632
4563 2737 5156 2726 5165 2969 5598 2962 5599 2751 5628 2745 5641 2730 5641 2599
5673 2590 6090 2594 6093 2862 6210 2847 6215 3010 6249 2996 6270 2994 6282 3018
6286 3039 6310 3027 6326 3025 6344 3041 6359 3057 6408 3059 6425 3042 6456 3043
6474 3038 6498 3042 6508 3040 6556 3016 6559 3005 6539 2997 6537 2984 6582 2962
6679 2940 6672 2601 7141 2609 7143 2426 7612 2432 8108 2438 8102 2587 8101 2876
7908 2871 7874 2883 7850 2895 7816 2891 7801 2874 7780 2814 7758 2787 7739 2802
7719 2785 7698 2777 7680 2781 7659 2780 7636 2766 7607 2768 7591 2765 7582 2751
7556 2740 7519 2757 7482 2758 7481 2851 7439 2858 7384 2866 7369 2850 7290 2878
7226 2896 7169 2902 7143 2885 7127 2892 7103 2899 7059 2892 7043 2920 7054 3093
6577 3095 6587 3561 6378 3564 6378 3564 6020 3568 5765 3565 5603 3562 5604 3088
5718 3087 5718 2910 5822 2906 5843 2896 5877 2898 5881 2877 6087 2877 5881 2877
5877 2898 5843 2896 5822 2906 5718 2910 5718 3087 5604 3088 5598 2962 5165 2969
5168 3206 4573 3216 4570 3107 3979 3120 3991 3608 3978 3607 3406 3626 3385 3628
2813 3653 2792 3657 2222 3687 2209 3686 1639 3716 1623 3718  935 3756  909 3757
 825 3786  221 3800  262 4400  939 4353 1034 4349 1620 4319 1649 4312 2229 4281
2806 4259 2807 4255 3292 4234 3396 4237 3769 4225 3982 4223 4225 4214 4579 4216
4717 4206 5179 4195 5672 4192 5767 4193 6141 4186 6372 4183 6599 4185 6837 4189
7303 4204
```

```
{$L- PRINTER:}
{$S+ Put compiler in swapping mode.}
PROGRAM ksu_football_field;
  {This program produces several views -- orthographic and perspective --
  of the K-State football field.  The field is approximated mathematically
  as a flat plane -- the actual crown of the field is ignored.  The
  asymmetrical endzone annotations provide a direction reference.  "K-STATE"
  fills the north endzone; "WILDCATS" fills the south endzone.  The origin
  (0,0,0) is at the southwest corner of the south endzone.  All points are
  on the z=0 plane, except for the goal posts.  Written in January 1982;
  last modified 7 April 1982.}
USES global, matrixops, ids560, dotplotter;

VAR
  a,b    :  matrix;
  border:  INTEGER;
  left   :  REAL;
  loop   :  INTEGER;
  top    :  REAL;
  u      :  vector;

PROCEDURE seg (x1,y1, x2,y2:  REAL);
VAR u:  vector;
BEGIN
  define_3D_vector (x1,y1,0.0, u);
  moveto (u);
  define_3D_vector (x2,y2,0.0, u);
  lineto (u)
END (seg);

PROCEDURE endzone_annotation;
BEGIN
  seg (  5,  5,  20,  5);      {"south" endzone annotation:  WILDCATS}
  seg ( 20,  5,  20, 15);      {upside-down "S"}
  seg ( 20, 15,   5, 15);
  seg (  5, 15,   5, 25);
  seg (  5, 25,  20, 25);

  seg ( 25,  5,  40,  5);      {upside-down "T"}
  seg (32.5, 5, 32.5,25);

  seg ( 45, 25, 52.5, 5);      {upside-down "A"}
  seg (52.5, 5,  60, 25);
  seg ( 49, 15,  56, 15);

  seg ( 65,  5,  80,  5);      {upside-down "C"}
  seg ( 80,  5,  80, 25);
  seg ( 80, 25,  65, 25);

  seg ( 85, 10,  90,  5);      {upside-down "D"}
  seg ( 90,  5, 100,  5);
  seg (100,  5, 100, 25);
  seg (100, 25,  90, 25);
  seg ( 90, 25,  85, 20);
  seg ( 85, 20,  85, 10);

  seg (105, 25, 120, 25);      {upside-down "L"}
  seg (120, 25, 120,  5);

  seg (127.5,5, 127.5,25);     {"I"}

  seg (135,  5, 140, 25);      {upside-down "W"}
  seg (140, 25, 145, 15);
  seg (145, 15, 150, 25);
  seg (150, 25, 155,  5);

  seg (  6,335,   6,355);      {"north" endzone annotation:  K-STATE}
  seg (  6,343,  25,355);      {"K"}
  seg ( 10,345,  25,335);

  seg ( 26,345,  32,345);      {"-"}

  seg ( 39,335,  58,335);      {"S"}
  seg ( 58,335,  58,345);
  seg ( 58,345,  39,345);
```

```
 76     seg ( 39,345,  39,355);
 77     seg ( 39,355,  58,355);
 78
 79     seg ( 63,355,  82,355);      {"T"}
 80     seg (72.5,335,72.5,355);
 81
 82     seg ( 87,335, 96.5,355);     {"A"}
 83     seg (96.5,355, 106,335);
 84     seg ( 92,345, 101,345);
 85
 86     seg (111,355, 130,355);      {"T"}
 87     seg (120.5,355, 120.5,345);
 88
 89     seg (135,335, 154,335);      {"E"}
 90     seg (135,335, 135,355);
 91     seg (135,345, 150,345);
 92     seg (135,355, 154,355)
 93   END {endzone_annotation};
 94
 95   PROCEDURE number_yard_lines;
 96   VAR
 97     i,j :  INTEGER;
 98     feet:  REAL;
 99
100     PROCEDURE number (n:  INTEGER; x,y,size:  REAL);
101     VAR
102       half:  REAL;
103     BEGIN
104       half := 0.50 * size;
105       CASE n OF
106         0:  BEGIN
107               seg (x,     y,      x+size,y);
108               seg (x+size,y,      x+size,y+size);
109               seg (x+size,y+size, x,     y+size);
110               seg (x,     y+size, x,     y)
111             END;
112         1:    seg (x,     y,      x+size,y);
113         2:  BEGIN
114               seg (x,     y,      x,     y+size);
115               seg (x,     y+size, x+half,y+size);
116               seg (x+half,y+size, x+half,y);
117               seg (x+half,y,      x+size,y);
118               seg (x+size,y,      x+size,y+size)
119             END;
120         3:  BEGIN
121               seg (x     ,y,      x,     y+size);
122               seg (x     ,y,      x+size,y);
123               seg (x+half,y,      x+half,y+size);
124               seg (x+size,y,      x+size,y+size)
125             END;
126         4:  BEGIN
127               seg (x,     y,      x+size,y);
128               seg (x+half,y,      x+half,y+size);
129               seg (x+half,y+size, x+size,y+size)
130             END;
131         5:  BEGIN
132               seg (x,     y,      x,     y+size);
133               seg (x,     y,      x+half,y);
134               seg (x+half,y,      x+half,y+size);
135               seg (x+half,y+size, x+size,y+size);
136               seg (x+size,y+size, x+size,y)
137             END
138       END
139     END {number};
140   BEGIN
141     feet := 60;
142     FOR i := 1 TO 9 DO BEGIN
143       IF   i > 5
144       THEN j := 10-i
145       ELSE j := i;
146       number (j, 10,feet+2,+3);
147       number (0, 10,feet-5,+8);
148       number (j,150,feet-2,-3);
149       number (0,150,feet+5,-8);
150       feet := feet+30
```

```
151     END
152   END {number_yard_lines};
153
154   PROCEDURE field_definition;
155   VAR
156     i,j :  INTEGER;
157     feet:  REAL;
158   BEGIN
159     seg (0,0, 160,0);              {"south" endzone boundary}
160
161     feet := 30;
162     FOR i := 0 TO 19 DO BEGIN
163       seg (0,feet, 160,feet);      {yard line}
164       IF   i <> 0 THEN BEGIN       {"vertical" hash lines}
165         seg ( 45,feet-1.5, 45,feet+1.5); {"west"}
166         seg (115,feet-1.5,115,feet+1.5); {"east"}
167       END;
168       FOR j := 1 TO 4 DO BEGIN     {"horizontal" hash lines between yard lines}
169         feet := feet + 3;
170         seg ( 45,feet,  48,feet);        {"west"}
171         seg (115,feet, 112,feet)         {"east"}
172       END;
173       feet := feet + 3
174     END;
175
176     seg (  0,330, 160,330);        {"north" goal line}
177     seg (  0,360, 160,360);        {"north" endzone boundary}
178
179     seg (  0,  0,   0,360);        {"west" sideline}
180     seg (160,  0, 160,360);        {"east" sideline}
181
182     seg ( 77,321,  83,321);        {"north" PAT hash line}
183     seg ( 77, 39,  83, 39);        {"south" PAT hash line}
184
185     endzone_annotation;
186     number_yard_lines
187
188   END {field_definition};
189
190   PROCEDURE goal_posts;
191   VAR
192     i    :  INTEGER;
193     t,x,y:  REAL;
194     u    :  vector;
195   BEGIN
196     FOR i := 1 TO 2 DO BEGIN
197       CASE i OF
198         1:  {south goal post}
199           BEGIN
200             x :=  80.0;
201             y :=   0.0;
202             t :=  -5.0
203           END;
204         2:  {north goal post}
205           BEGIN
206             x :=  80.0;
207             y := 360.0;
208             t :=   5.0
209           END
210       END {CASE};
211       define_3D_vector (x,y+t,0.0, u);
212       moveto (u);
213       define_3D_vector (x,y+t,10.0, u);
214       lineto (u);
215       define_3D_vector (x,y,10.0, u);
216       lineto (u);
217       define_3D_vector (x-10.0,y,20.0, u);
218       moveto (u);
219       define_3D_vector (x-10.0,y,10.0, u);
220       lineto (u);
221       define_3D_vector (x+10.0,y,10.0, u);
222       lineto (u);
223       define_3D_vector (x+10.0,y,20.0, u);
224       lineto (u)
225     END {FOR}
```

```
226   END (goal_posts);
227
228   PROCEDURE case_0;
229   BEGIN
230     PAGE (prt);
231     top := 0.65;
232     left := 2.00;
233     position (1.25,top-0.15);
234     control (pitch16);
235     control (enhanced_mode);
236     WRITE (prt,'Exhibit 2. Orthographic View of KSU Football Field');
237     control (normal_mode);
238     border := 0;
239     size (4.00,9.25);
240     window (0,160.0, -5.0,365.00);
241     clipping (FALSE);
242     set_projection_type (orthographic)
243   END (case_0);
244
245   PROCEDURE case_1;
246   VAR
247     screen_distance:  REAL;
248   BEGIN
249     top := -0.65 + 2.0*loop;
250     IF   loop = 1
251     THEN BEGIN
252       left := 1.50;
253       PAGE (prt);
254       position (1.25,top-0.65);
255       control (pitch16);
256       control (enhanced_mode);
257       WRITE (prt,'Exhibit 3. Perspective Views of KSU Football Field');
258       control (normal_mode);
259       position (1.50,top-0.55);
260       control (pitch10);
261       WRITELN (prt,'3.1  Side View (from the east side)');
262       position (2.00,top-0.425);
263       control (pitch16);
264       WRITELN (prt,'Azimuth = 90 degrees, Elevation = 30 degrees, ',
265                    'Distance = 200 feet from center of field');
266       border := 2;
267       size (5.00,1.50);
268       clipping (TRUE);
269       set_projection_type (perspective)
270     END;
271     define_3D_vector (-80.0,-180.0,0.0, u);
272     translate_matrix (u, a);  {translate origin to center of field}
273     screen_distance :=  4.0 - loop;
274     position (2.00,top-0.175);
275     WRITE (prt,'3.1',CHR(ORD('a')-1+loop),
276           '  Proper perspective when viewed ',ROUND(screen_distance),
277           ' inches from eye');
278     view_transform_matrix (polar, 90.0,30.0,200.0, 5.00,1.50,screen_distance, b);
279     matrix_multiply (a,b, a);
280     set_transform (a)
281   END (case_1);
282
283   PROCEDURE case_2;
284   BEGIN
285     left := 1.50;
286     top := 1.10;
287     PAGE (prt);
288     position (1.25,top-0.60);
289     control (pitch16);
290     control (enhanced_mode);
291     WRITE (prt,'Exhibit 3. Perspective Views of KSU Football Field');
292     control (normal_mode);
293     position (1.50,top-0.30);
294     control (pitch10);
295     WRITELN (prt,'3.2  Corner View (from the southeast corner)');
296     position (2.00,top-0.175);
297     control (pitch16);
298     WRITELN (prt,'Azimuth = 25 degrees, Elevation = 30 degrees, ',
299                  'Distance = 750 feet from center of field');
300     size (5.00,2.50);
```

```
301     define_3D_vector (-80.0,-180.0,0.0, u);
302     translate_matrix (u, a);  {translate origin to center of field}
303     view_transform_matrix (polar, 25.0,30.0,750.0, 5.00,2.50,12.0, b);
304     matrix_multiply (a,b, a);
305     set_transform (a)
306   END {case_2};
307
308   PROCEDURE case_3;
309   BEGIN
310     top := 4.10;
311     position (1.50,top-0.30);
312     control (pitch10);
313     WRITELN (prt,'3.3  Endzone View (from the south endzone)');
314     position (2.00,top-0.175);
315     control (pitch16);
316     WRITELN (prt,'Azimuth = 0 degrees, Elevation = 30 degrees, ',
317                  'Distance = 300 feet from center of field');
318     define_3D_vector (-80.0,-180.0,0.0, u);
319     translate_matrix (u, a);  {translate origin to center of field}
320     view_transform_matrix (polar, 0.0,30.0,300.0, 5.00,2.50,12.0, b);
321     matrix_multiply (a,b, a);
322     set_transform (a)
323   END {case_3};
324
325   PROCEDURE case_4;
326   BEGIN
327     top := 7.10;
328     position (1.50,top-0.30);
329     control (pitch10);
330     WRITELN (prt,'3.4  Aerial View');
331     position (2.00,top-0.175);
332     control (pitch16);
333     WRITELN (prt,'Azimuth = 65 degrees, Elevation = 75 degrees, ',
334                  'Distance = 1500 feet from center of field');
335     define_3D_vector (-80.0,-180.0,0.0, u);
336     translate_matrix (u, a);  {translate origin to center of field}
337     view_transform_matrix (polar, 65.0,75.0,1500.0, 5.00,2.50,12.0, b);
338     matrix_multiply (a,b, a);
339     set_transform (a)
340   END {case_4};
341
342   BEGIN
343     begin_dot_plotter_unit;
344       forms (11.0,0.0);
345       plt_mode := create;
346       plt_name := '6:football.dots';
347       FOR loop := 0 TO 7 DO BEGIN
348         CASE loop OF
349           0:  case_0;       {Five separate procedures are used to circumvent}
350           1,2,3,4:  case_1; {the UCSD Pascal Error 253 -- "Procedure Too Long"}
351           5:  case_2;       {received when case_0..case_4 are placed in a}
352           6:  case_3;       {single "setup_case" procedure.}
353           7:  case_4
354         END {CASE};
355         open_plot;
356         field_definition;
357         goal_posts;
358         put_plot (left,top, 1 {copy}, border {dots});
359         close_plot (delete);
360       END {FOR};
361     end_dot_plotter_unit
362   END {ksu_football_field}.
```

```
{$L- PRINTER:}
{$S+ Put compiler swapping mode}
PROGRAM surface;
  {"surface" produces several perspective plots of a surface described
  mathematically as z = f(x,y).  Hidden lines are not removed.
  Written in February 1982;  last modified on 8 April 1982}
USES global, matrixops, dotplotter, ids560;

CONST
  n      = 40;          {lines 0..n}
  xfirst = -2.0;
  xlast  =  2.0;
  yfirst = -2.0;
  ylast  =  2.0;

VAR
  azimuth  :  REAL;
  a        :  matrix;
  i,j      :  0..n;
  denom    :  REAL;
  distance :  REAL;
  elevation:  REAL;
  loop     :  INTEGER;
  top      :  REAL;
  u        :  vector;
  x,y      :  REAL;
  xinc,yinc:  REAL;
  xsq,ysq  :  REAL;
  z        :  ARRAY[0..n,0..n] OF REAL;

PROCEDURE title;
BEGIN
  PAGE (prt);
  position (1.25,top-0.70);
  control (pitch16);
  control (enhanced_mode);
  WRITE (prt,'Exhibit 4. The Surface Z(X,Y) = XY(X');
  control (superscript);
  WRITE (prt,'2');
  control (subscript);
  WRITE (prt,'-Y');
  control (superscript);
  WRITE (prt,'2');
  control (subscript);
  WRITE (prt,')/(X');
  control (superscript);
  WRITE (prt,'2');
  control (subscript);
  WRITE (prt,'+Y');
  control (superscript);
  WRITE (prt,'2');
  control (subscript);
  WRITE (prt,')');
  control (normal_mode);
  control (pitch12);
  position (2.00,top-0.40);
  WRITELN (prt,'"Hidden" lines are not removed.  x=-2..+2; y=-2..+2.')
END (title);

PROCEDURE surface_points;
BEGIN
  xinc := (xlast-xfirst)/n;
  yinc := (ylast-yfirst)/n;
  {title;                          (* printing of data matrix suppressed *)
   control (pitch12);
   WRITELN (prt);
   WRITELN (prt,'   \ X');
   WRITELN (prt,'    \');
   WRITE (prt,'   Y \  ');
   FOR i := 0 TO n DO BEGIN
     x := xfirst + xinc*i;
     WRITE (prt,x:8:2);
   END;
   WRITELN (prt); WRITELN (prt);}

```

```
 76    FOR j := n DOWNTO 0 DO BEGIN
 77      y := yfirst + yinc*j;
 78      ysq := SQR(y);
 79      {WRITE (prt,y:6:2,'  ');}
 80      FOR i := 0 TO n DO BEGIN
 81        x := xfirst + xinc*i;
 82        xsq := SQR(x);
 83        denom := xsq+ysq;
 84        IF   defuzz(denom) = 0.0
 85        THEN z[i,j] := 0.0        {the limit value}
 86        ELSE z[i,j] := x*y*(xsq-ysq)/denom;
 87        {WRITE (prt,z[i,j]:8:4)}
 88      END;
 89      {WRITELN (prt)}
 90    END;
 91  END {surface_points};
 92
 93  PROCEDURE surface_dots;
 94  BEGIN
 95    FOR i := 0 TO n DO BEGIN
 96      x := xfirst + xinc*i;
 97      FOR j := 0 TO n DO BEGIN
 98        y := yfirst + yinc*j;
 99        define_3D_vector (x,y,z[i,j], u);
100        IF   j = 0
101        THEN moveto (u)
102        ELSE lineto (u)
103      END
104    END;
105
106    FOR j := 0 TO n DO BEGIN
107      y := yfirst + yinc*j;
108      FOR i := 0 TO n DO BEGIN
109        x := xfirst + xinc*i;
110        define_3D_vector (x,y,z[i,j], u);
111        IF   i = 0
112        THEN moveto (u)
113        ELSE lineto (u)
114      END
115    END;
116  END {surface_dots};
117
118  BEGIN
119    begin_dot_plotter_unit;
120      min_space := 4096;
121      forms (11.0,0.0);
122      surface_points;
123      plt_mode := create;
124      plt_name := '4:surface.dots';
125      size (4.0,4.0);
126      FOR loop := 0 TO 3 DO BEGIN
127        open_plot;
128        CASE loop OF
129          0:  BEGIN
130                azimuth := 45.0;
131                elevation := 30.0;
132                distance := 15.0;
133                top := 1.20;
134                title
135              END;
136          1:  BEGIN
137                azimuth := 45.0;
138                elevation := 30.0;
139                distance := 5.0;
140                top := top + 4.5
141              END;
142          2:  BEGIN
143                azimuth := 45.0;
144                elevation := 0.0;
145                distance := 15.0;
146                top := 1.20;
147                title
148              END;
149          3:  BEGIN
150                azimuth := 45.0;
```

```
151                elevation := 90.0;
152                distance := 15.0;
153                top := top + 4.5
154              END
155          END;
156          view_transform_matrix (polar, azimuth,elevation,distance,
157                                 4.0,4.0,10.0, a);
158          set_transform (a);
159          surface_dots;
160          position (1.50,top-0.2);
161          control (pitch10);
162          WRITE (prt,'4.',loop+1,'  ');
163          control (pitch16);
164          WRITELN (prt,'Azimuth = ',ROUND(azimuth),' degrees, ',
165                       'Elevation = ',ROUND(elevation),' degrees, ',
166                       'Distance = ',ROUND(distance),' units');
167          put_plot (2.00,top, 1 (copy), 2 (border dots));
168          close_plot (delete)
169        END;
170      end_dot_plotter_unit
171    END.
```

```
{$L- PRINTER:}
{$S+ Put compiler in swapping mode}
PROGRAM pde;
  {"pde" solves the partial differential equation:
       kx * Uxx + ky * Uyy = c(x,y)
  where "kx" and "ky" are constants, "Uxx" and "Uyy" are the second partial
  derivatives with respect to the spatial coordinates "x" and "y", and
  "c(x,y)" is zero except for a few specific (x,y) points.  The differential
  equation approximates the pressure map of a system of injection/production
  wells.  The non-zero c(x,y) values represent the flow rates at the injection
  and production wells.  The pressure surface is approximated by a 16-by-16
  grid.  The solution involves 256 equations with each unknown representing
  the pressure at a specific (x,y) point.  The difference equation used to
  approximate the differential equation at an arbitrary (x,y) point at
  grid position (i,j) is
       kx[ [u(i-1,j) - 2u(i,j) + u(i+1,j)] / SQR(delta x)] +
       ky[ [u(i,j-1) - 2u(i,j) + u(i,j+1)] / SQR(delta y)] = c(x,y)
  With "delta x" = "delta y", the difference equation reduces to
       -kx[u(i-1,j) + u(i+1,j)] + 2(kx+ky)*u(i,j) - ky[u(i,j-1) + u(i,j+1)]
          = c(x,y)
  The difference equations for boundary points is slightly different.
  Given the different "u" terms, the following shows the coefficients
  for the boundary equations:

    boundary     u(i,j-1)     u(i-1,j)     u(i,j)     u(i+1,j)     u(i,j+1)
      i=1          0            -kx       2(kx+ky)     -kx          -2ky
      i=16        -2ky          -kx       2(kx+ky)     -kx           0
      j=1         -ky            0        2(kx+ky)     -2kx         -ky
      j=16        -ky           -2kx      2(kx+ky)      0           -ky

  Corner points are yet a more special case.  For (i,j)=(1,1), the
  equation would be:  2(kx+ky)*u(i,j) - 2kx*u(i+1,j) - 2ky*u(i,j+1) = c(x,y).
  The other corners have similar equations.

  This program solves the PDE and writes the grid matrix to a diskette
  file for plotting by "pde2".  With 64K memory, both operations could
  not be performed in a single program.  Certain variables were dynamically
  allocated in this program in an unsuccessful attempt to accommodate
  both operations.

  This program was written on 9 April 1982 but was adapted from a homework
  assignment completed in the Fall 1975 Numerical Solutions of PDE course.}
USES global, ids560;

CONST
  n       = 16;
  nsq     = 256;

TYPE
  diag5 = ARRAY[1..n,1..nsq] OF REAL;
  mode_type = (gauss_seidel,optimal_point_sor);
  vector_matrix =
    RECORD
      CASE INTEGER OF
        0:  (vctr:  ARRAY[1..nsq] OF REAL);
        1:  (mtrx:  ARRAY[1..n,1..n] OF REAL)
    END {vector_matrix RECORD};
   ptr_diag5 = ^diag5;
   ptr_vector_matrix = ^vector_matrix;

VAR
  a:  ptr_diag5;          {System of equations:  ax = b}
  x:  ptr_vector_matrix;
  b:  ptr_vector_matrix;

  epsilon   :  REAL;
  heap      :  ^INTEGER;
  i         :  INTEGER;
  iterations:  INTEGER;
  soln      :  TEXT;

PROCEDURE setup_matrix (kx,ky:  REAL);
  {This procedure defines the matrix "a" and the column vectors "x" and
  "b" after allocation in the main section of code.}
VAR
```

```
  i,j,k:  INTEGER;
BEGIN
  kx :=   1.0;
  ky :=   1.0;
  FOR i := 1 TO n DO
    FOR j := 1 TO n DO BEGIN
      k := (i-1)*n+j;
      aA[1,k] := -ky;
      aA[2,k] := -kx;
      aA[3,k] := 2.0*(kx+ky);
      aA[4,k] := -kx;
      aA[5,k] := -ky;
      xA.vctr[k] := 0.0;
      bA.vctr[k] := 0.0;
      IF   i = 1
      THEN BEGIN
        aA[1,k] := 0.0;
        aA[5,k] := -2.0*ky
      END
      ELSE
        IF   i = n
        THEN BEGIN
          aA[1,k] := -2.0*ky;
          aA[5,k] := 0.0
        END;
      IF   j = 1
      THEN BEGIN
        aA[2,k] := 0.0;
        aA[4,k] := -2.0*kx
      END
      ELSE
        IF   j = n
        THEN BEGIN
          aA[2,k] := -2.0*kx;
          aA[4,k] := 0.0
        END
    END (FOR);
  bA.mtrx[ 3, 3] := -1.00;        {vctr[ 35]}
  bA.mtrx[ 4,12] := -0.60;        {vctr[ 60]}
  bA.mtrx[ 9, 9] :=  1.83;        {vctr[137]}
  bA.mtrx[14, 3] := -0.50;        {vctr[211]}
  bA.mtrx[14,14] :=  0.27         {vctr[222]}
END {setup_matrix};

PROCEDURE sor5 (n,nsq:  INTEGER;
                a: ptr_diag5; VAR x: ptr_vector_matrix; b: ptr_vector_matrix;
                mode:  mode_type;
                VAR iterations:  INTEGER; VAR epsilon:  REAL);
  {The system of equations to be solved can be represented simply by ax=b.
  "sor5" solves this system of "nsq" equations with a special "a" matrix
  containing only five non-zero elements.  "a" has a tridiagonal structure
  but also has elements "n" positions left and right of the main diagonal.
  The structure of a full "a" matrix is

                          e f z z . . . z
                          f e f z . . . z
                          z f e f
                          z z f e .
                          . . . . .
                          . .   . f e f z
                          . .     z f e f
                          z z     z z f e

  where each of the "e","f" and "z" elements are matrices of size "n".
  The "z" matrix contains all zero elements.  For the problem being solved,
  the structure of each matrix "e" is

          2(Kx+Ky)  -2Kx      0           .           .
          -Kx       2(Kx+Ky)  -Kx         0           .
          0         -Kx       2(Kx+Ky)    -Kx         .
          0         0         -Kx         2(Kx+Ky)    .
          .         .          .          .           .
          .         .          .          -2Kx        2(Kx+Ky)

  and the structure of each matrix "f" is an identity matrix multiplied by -Ky
```

```
151
152             -Ky       0          .          .
153             0         -Ky        0          .
154             .         .          .          .
155             .         .          0         -Ky
156
157     Note:  Coefficients in the "f" matrix for corner grid points are -2Ky
158            instead of -Ky.
159
160     The solution "mode" can be either "gauss_seidel" or "optimal_point_sor".
161     On entry "iterations" contains the maximum number of iterations allowed;
162     "epsilon" contains the desired criteria for the maximum residual.  On exit
163     "iterations" contains the actual number of loops and "epsilon" contains
164     the maximum residual.)
165   VAR
166     i             :  INTEGER;
167     left          :  REAL;
168     loop          :  INTEGER;
169     max_residual  :  REAL;
170     nsqmn         :  INTEGER;
171     nsqmi         :  INTEGER;
172     npi           :  INTEGER;
173     old_x         :  REAL;
174     old_r2_norm   :  REAL;
175     residual      :  REAL;
176     right         :  REAL;
177     r_norm        :  REAL;
178     r2_norm       :  REAL;
179     temp          :  REAL;
180     w             :  REAL;    {overrelaxation parameter}
181   BEGIN
182     w := 1.00;
183     max_residual := 1.0E10;   {just to get WHILE started}
184     loop := 0;
185     nsqmn := nsq - n;
186     nsqmi := nsq - 1;
187     npi := n+1;
188     WHILE (max_residual ) epsilon) AND (loop ( iterations) DO BEGIN
189       IF   loop MOD 5 = 0
190       THEN WRITELN (prt,'loop = ',loop:3,',  maximum residual = ',max_residual,
191                     ', w = ',w);
192       loop := SUCC(loop);
193       max_residual := 0.0;
194       r_norm := 0.0;
195       FOR i := 1 TO nsq DO BEGIN
196         old_x := xA.vctr[i];
197         IF   i = 1  {first equation}
198         THEN xA.vctr[i] := (bA.vctr[i] - aA[4,i]*xA.vctr[i+1]
199                                        - aA[5,i]*xA.vctr[npi]) / aA[3,i]
200         ELSE
201           IF   i = nsq  {last equation}
202           THEN xA.vctr[i] := (bA.vctr[i] - aA[1,i]*xA.vctr[nsqmn]
203                                          - aA[2,i]*xA.vctr[nsqmi]) / aA[3,i]
204           ELSE BEGIN  {all equations but first or last}
205             IF   i ) n
206             THEN left := xA.vctr[i-n]
207             ELSE left := 0.0;
208             IF   i (= nsqmn
209             THEN right := xA.vctr[i+n]
210             ELSE right := 0.0;
211             xA.vctr[i] := (bA.vctr[i] - aA[1,i]*left
212                                       - aA[2,i]*xA.vctr[i-1]
213                                       - aA[4,i]*xA.vctr[i+1]
214                                       - aA[5,i]*right) / aA[3,i]
215           END;
216         residual := xA.vctr[i] - old_x;
217         IF   ABS(residual) ( 1.0E-6
218         THEN residual := 0.0;     {"defuzz" to avoid underflows}
219         IF   mode = optimal_point_sor
220         THEN BEGIN
221           xA.vctr[i] := old_x + w*residual;
222           r_norm := r_norm + SQR(residual)
223         END;
224         IF   ABS(residual) ) max_residual
225         THEN max_residual := ABS(residual);
```

```
      END {FOR};
      IF   mode = optimal_point_sor
      THEN BEGIN
        r2_norm := SQRT(r_norm);
        IF   loop = 1
        THEN old_r2_norm := r2_norm;
        IF   loop MOD 10 = 0        {Update "w" only every 10 iterations}
        THEN BEGIN                  {because of the considerable overhead}
          temp := r2_norm / old_r2_norm;  {and to help numerical stability}
          w := 2.0 / (1.0 + SQRT( ABS(1.0 - SQR(temp+w-1.0)/(temp*SQR(w)) ) ) );
        END;
        old_r2_norm := r2_norm
      END
    END {WHILE};
    iterations := loop;
    epsilon := max_residual
  END {sor5};

  PROCEDURE put_solution (u:  vector_matrix);
    {"put_solution" prints the pressure map but also writes it to a disk file.}
  VAR
    i,j:  INTEGER;
  BEGIN
    REWRITE (soln,'0:PDEDATA.TEXT');
    WRITELN (soln,n);
    control (pitch10);
    WRITELN (prt,'Iterations = ',iterations:3,'     Maximum Residual = ',
             epsilon);
    WRITELN (prt);
    control (pitch16);
    WRITELN (prt,'    \ i');
    WRITELN (prt,'     \');
    WRITE   (prt,'   j \  ');
    FOR i := 1 TO n DO
      WRITE (prt,i:8);
    WRITELN (prt);
    WRITELN (prt);
    FOR j := n DOWNTO 1 DO BEGIN
      WRITE (prt,j:6,'  ');
      FOR i := 1 TO n DO BEGIN
        WRITE (prt ,u.mtrx[i,j]:8:4);
        WRITE (soln,u.mtrx[i,j]:8:4);
      END;
      WRITELN (prt); WRITELN (soln)
    END;
    CLOSE (soln,LOCK)
  END {put_solution};

  PROCEDURE memory (title:  STRING);
  BEGIN
    WRITELN (prt,title,'  Memory Available = ',2*MEMAVAIL,' bytes.')
  END {memory};

  BEGIN {pde}
    REWRITE (prt,'PRINTER:');
    PAGE (prt);
    control (pitch10);
    memory ('Beginning.    ');
    NEW (x);
    MARK (heap);
    NEW (a);
    NEW (b);
    setup_matrix (2.00,1.00);    {system of equations:  ax=b}
    memory ('After setup.  ');
    iterations := 100;
    epsilon := 1.0E-6;
      {The "a", "x" and "b" pointers must be passed to "sor5".  There is not
      enough memory to pass the arrays as "a^", "x^" and "b^".}
    sor5 (n,nsq, a,x,b, optimal_point_sor, iterations,epsilon);
    RELEASE (heap);  {release "a" and "b" but not the solution vector "x"}
    memory ('After release.');
    put_solution (x^)
  END.
```

- - - - -   - - - - -   - - - - -   - - - - -   - - - - -

```
299  {$L- PRINTER:}
300  {$S+ Put compiler in swapping mode}
301  PROGRAM pde2;
302    {"pde2" is a continuation of the program "pde". Memory restrictions
303    dictated a two-step solution. Written on 10 April 1982.}
304  USES global, matrixops, ids560, dotplotter;
305
306  VAR
307    a,b : matrix;
308    loop: INTEGER;
309    n   : INTEGER;
310    top : REAL;
311    u   : vector;
312    z   : ARRAY[1..16,1..16] OF REAL;
313    azimuth  : REAL;
314    distance : REAL;
315    elevation: REAL;
316
317  PROCEDURE read_data;
318  VAR
319    i,j : INTEGER;
320    soln: TEXT;
321  BEGIN
322    RESET (soln,'8:PDEDATA.TEXT');
323    READLN (soln,n);
324    FOR j := n DOWNTO 1 DO BEGIN
325      FOR i := 1 TO n DO
326        READ (soln,z[i,j]);
327      READLN (soln)
328    END;
329    CLOSE (soln)
330  END {read_data};
331
332  PROCEDURE title;
333  BEGIN
334    PAGE (prt);
335    position (1.25,top-0.70);
336    control (pitch16);
337    control (enhanced_mode);
338    WRITE (prt,'Exhibit 5. ');
339    control (normal_mode);
340    control (pitch12);
341    WRITELN (prt,'Pressure Map of Area with Injection/Production Wells');
342    position (2.00,top-0.55);
343    WRITE (prt,'Partial Differential Equation:  k');
344    control (subscript);   WRITE (prt,'x');
345    control (superscript); WRITE (prt,'U');
346    control (subscript);   WRITE (prt,'xx');
347    control (superscript); WRITE (prt,' + k');
348    control (subscript);   WRITE (prt,'y');
349    control (superscript); WRITE (prt,'U');
350    control (subscript);   WRITE (prt,'yy');
351    control (superscript); WRITELN (prt,' = c(x,y)');
352    position (2.00,top-0.40);
353    WRITE (prt,'k');
354    control (subscript);   WRITE (prt,'x');
355    control (superscript); WRITE (prt,'=2, k');
356    control (subscript);   WRITE (prt,'y');
357    control (superscript); WRITELN (prt,'=1, c(x,y)= constant flow rate');
358  END {title};
359
360  PROCEDURE surface_dots;
361  VAR
362    i,j:  INTEGER;
363    u   : vector;
364  BEGIN
365    FOR i := 1 TO n DO
366      FOR j := 1 TO n DO BEGIN
367        define_3D_vector (i+0.0,j+0.0,z[i,j], u);
368        IF   j = 1
369        THEN moveto (u)
370        ELSE lineto (u)
371      END;
372    FOR j := 1 TO n DO
373      FOR i := 1 TO n DO BEGIN
```

```
374         define_3D_vector (i+0.0,j+0.0,z[i,j], u);
375         IF   i = 1
376         THEN moveto (u)
377         ELSE lineto (u)
378       END;
379   END (surface_dots);
380
381   BEGIN
382     read_data;
383     begin_dot_plotter_unit;
384       forms (11.0,0.0);
385       plt_mode := create;
386       plt_name := '6:pde.dots';
387       size (4.0,4.0);
388       elevation := 30.0;
389       distance := 60.0;
390       FOR loop := 0 TO 1 DO BEGIN
391         open_plot;
392         CASE loop OF
393           0:  BEGIN
394                 azimuth := 0.0;
395                 top := 1.20;
396                 title
397               END;
398           1:  BEGIN
399                 azimuth := 270.0;
400                 top := top + 4.5
401               END
402         END;
403         define_3D_vector (1.0,1.0,5.0, u);  {enhance z values}
404         scale_matrix (u, a);
405         define_3D_vector (-8.0,-8.0,0.0, u);
406         translate_matrix (u,b);
407         matrix_multiply (a,b, a);
408         view_transform_matrix (polar, azimuth,elevation,distance,
409                                       4.0,4.0,12.0, b);
410         matrix_multiply (a,b, a);
411         set_transform (a);
412         surface_dots;
413         position (1.50,top-0.2);
414         control (pitch10);
415         WRITE (prt,'5.',loop+1,'  ');
416         control (pitch16);
417         WRITELN (prt,'Azimuth = ',ROUND(azimuth),' degrees, ',
418                      'Elevation = ',ROUND(elevation),' degrees, ',
419                      'Distance = ',ROUND(distance),' units');
420         put_plot (2.00,top, 1 {copy}, 2 {border dots});
421         close_plot (delete);
422       END {FOR};
423     end_dot_plotter_unit
424   END.
```

Appendix C. System Guide

C. System Guide

C.1 Introduction to Pascal UNITs

A UCSD Pascal UNIT is a group of interdependent PROCEDUREs, FUNCTIONs and associated data structures (CONSTants, TYPEs and VARiables) which perform a specialized task. A UNIT can be compiled separately from a user program. The LIBRARIAN utility allows the user to link separately compiled UNITS into a library file. Whenever a user program needs a UNIT, the program indicates that it USES the UNIT. When the compiler encounters the USES statement it essentially re-compiles the INTERFACE part of the UNIT which is stored in the library file.

A UNIT consists of two sections, the INTERFACE and the IMPLEMENTATION. The INTERFACE declares CONSTants, TYPEs, VARiables, PROCEDUREs and FUNCTIONs that are public and can be used by a host program. The IMPLEMENTATION declares CONSTants, TYPEs, VARiables, PROCEDUREs and FUNCTIONs that are private and used only by the UNIT as well as the body of the FUNCTIONS and PROCEDUREs defined in the INTERFACE. These private variables, etc. used by the UNIT are not available to the host program. The INTERFACE defines how the program will communicate with the UNIT while the IMPLEMENTATION defines how the UNIT will accomplish its task. A UNIT can access another UNIT but the USES must appear in the *INTERFACE section*.

A sample structure of a UNIT is given below:

```
UNIT unit_name;

INTERFACE
  CONST ... ;  (public constants, types and variables)
  TYPE  ... ;
  VAR   ... ;
  PROCEDURE public_one (parms... );
  (other public PROCEDUREs or FUNCTIONs)

IMPLEMENTATION
  CONST ... ;  (local constants, types and variables)
  TYPE  ... ;
  VAR   ... ;
  PROCEDURE local_one (parms... );
  BEGIN
    (body of "local_one")
  END (local_one);
  PROCEDURE public_one; (no parameters here)
  BEGIN
    local_one (parms...);
    (body of "public_one")
  END (public_one);
  (other local or public PROCEDUREs or FUNCTIONs)
END (unit).
```

A UNIT is compiled like a regular Pascal program. The file LIBRARY.CODE contains the utility program which can be executed to add the UNIT to the SYSTEM.LIBRARY file. Once this is done, a user program

USES the UNIT:

```
PROGRAM program_name;
USES unit_name;
  {other declarations}
BEGIN
  public_one (parms ...);
  {body of program}
END.
```

Ideally, a single UNIT would contain all graphic primitives and the supporting routines and data structures. But symbol table space is limited with only 64K of total memory and several UNITs were found to be necessary. A minimum set of entities was placed in a "global" UNIT to be used by all other UNITs. The remaining UNITs, "dotplotter", "matrixops" and "ids560", perform operations which are logically related within each UNIT. The "dotplotter" UNIT contains the bulk of the graphic primitives and the underlying memory management routines to support virtual screen definition. The "matrixops" UNIT contains various matrix operations for manipulating two- and three-dimensional points. The "ids560" UNIT supports control of the Integral Data Systems 560 printer.

C.2 "global" UNIT

The "global" UNIT defines the following public entities:

```
CONSTant:     radians_per_degree
TYPEs:        dimension, index, matrix, vector
VARiable:     prt
FUNCTION:     defuzz
PROCEDUREs:   define_2D_vector, define_3D_vector, transform
```

The CONSTant is self-explanatory. The VARiable "prt" defines the print file (to be used as the dot matrix graphics printer) so that it can be used by the various UNITs which access this "global" UNIT. The function "defuzz(x)" returns the value 0.0 for "x" < "fuzz", "x" otherwise. "defuzz" is used for REAL comparisons and to reduce the possibility of numerical instability by preventing propagation of small values which should be true zeros. "fuzz" is internally set to 1.0E-6. The TYPEs and PROCEDUREs require detailed explanation.

This graphics package attempts to integrate two- and three-dimensional points and operations into a single framework. For example, instead of separate "moveto_2D (x,y)" and "moveto_3D (x,y,z)" primitives, a single "moveto (u)" primitive is used where "u" is of the TYPE "vector" which can be either two- or three-dimensional. (This integration was not caused by insight but rather was an alternative in reducing the required symbol table space given only 64K total memory).

The "index", "vector" and "matrix" TYPEs are defined as follows:

```
TYPE
  index = 1..4;
  matrix=
    RECORD
      size:  index;
      mtrx:  ARRAY[index,index] OF REAL
    END;
  vector=
    RECORD
      size:  index;
      vctr:  ARRAY[index] OF REAL
    END;
```

Now given

```
VAR
  a     :  matrix;
  u,v   :  vector;
  x,y,z:  REAL;
```

the PROCEDURE "define_2D_vector (x,y, u)" defines a row vector containing homogenous coordinates (x,y,1). In algebraic notation, u = [x y 1]. Likewise, "define_3D_vector (x,y,z, u)" defines a row vector containing homogenous coordinates (x,y,z,1). In algebraic notation, u = [x y z 1]. "size" is 3 for a two-dimensional vector; 4 for a three-dimensional vector. A "matrix" must be the same "size" as vectors of corresponding dimensionality. The vector-matrix product v = uA is performed by a PROCEDURE call "transform (u,a,v)". ("transform" logically belongs in the "matrixops" UNIT but it is in "global" since it is needed by "dotplotter"). The "transform" PROCEDURE obtains "size" information from the parameters and performs the appropriate multiplication. Certain operations (e.g., "rotate_matrix" in "matrixops") cannot implicitly determine whether an operation is to be two- or three-dimensional from the "size" of the parameters. A parameter of TYPE "dimension" is used to convey whether "two_D" or "three_D" is intended ("2D" and "3D" are invalid Pascal symbols).

C.3 "dotplotter" UNIT

The "dotplotter" UNIT contains the control and graphic primitives. The plot file and paging system are hidden from the user but are contained in this UNIT. This section will explain the symbols used for definition of PROCEDUREs, TYPEs and VARiables in the INTERFACE. The local IMPLEMENTATION symbols are discussed primarily in sections C.6 and C.7 which address the file structure and paging system.

The control VARiables are:

```
close_printer
min_space
plt
plt_mode
plt_name
```

Even though the user can access any of these VARiables, the "plt" FILE variable should NEVER be accessed by a user program. Due to a UCSD Pascal limitation, FILEs must be placed in the INTERFACE or a compiler error will occur. The user will jeopardize the integrity of the plot file by accessing "plt".

"min_space" reserves a specified number of bytes of memory for use by the UCSD p-system during run time. "min_space" has a default value defined by the "spacedefault" CONSTant in the IMPLEMENTATION but sometimes additional stack space is necessary. (For example, nested PROCEDURE calls sometimes require additional memory.) The user can assign a larger value to "min_space" if the "stack overflow" error is received from the p-system. Reserving too much memory, however, could degrade the performance of the paging system and cause thrashing.

The "close_printer" variable is of TYPE BOOLEAN and has a default TRUE value. The "dotplotter" UNIT performs a page eject when the graphics mode is exited if "close_printer" is TRUE. (This variable is somewhat misnamed and may be changed in future versions).

The "plt_mode" and "plt_name" variables are somewhat related. "plt_mode" can have values of "undefined", "either", "create" or "overlay" ("undefined" is the default). Any user program assigned value of "plt_name" is ignored if "plt_mode" is "undefined". The user will be interactively prompted for a name for the plot file at run time if "plt_mode" is "undefined". The plot file will be created if the file does not exist; if it already exists it will be overlayed by subsequent graphic primitives. The user can assign the "plt_name" for a plot file inside a program if "plt_mode" is assigned a value of "create" or "overlay". The file will be created, or re-created if it already exists when "create" is specified. "overlay" requires that the file must already exist from a previous task -- an error will be displayed if the file cannot be found. If the user for some reason does not care whether "create" or "overlay" is used, then "either" can be specified.

This first version of "dotplotter" has five control PROCEDUREs:

```
begin_dot_plotter_unit
open_plot
close_plot
put_plot
end_dot_plotter_unit
```

The "begin_dot_plotter_unit" PROCEDURE ensures that all necessary variables are assigned an appropriate default value. "dotplotter" cannot function without these initializations. The user can modify most of the default values but such statements must follow the "*begin_dot_plotter_unit*" call. Future versions may incorporate "open_plot" into the "begin_dot_plotter_unit" but for now the two are separate. "open_plot" initializes the parameter prefix if the "plt_mode" is "create" or reads the prefix from disk if "overlay" is specified. "open_plot" also initializes all variables needed by the paging system.

The "close_plot", "put_plot" and "end_dot_plotter_unit" PROCEDUREs are related and they may be eventually incorporated into a single PROCEDURE. "close_plot" flushes out the paging system by forcing in-memory frames to be written to disk. A parameter of TYPE "disposition" must be passed to "close_plot" to "keep" or "delete" the plot file. A call to "put_plot" may precede or follow a "close_plot" call and causes the plot file to be mapped to the graphics printer. Several options must be specified with "put_plot" to control the output process. "put_plot" can be called at any time to output a copy of the current logical screen, or need not be referenced at all if the plot file is being set up for subsequent overlays. "close_plot" actually need not be called since the "end_dot_plotter_unit" performs final cleanup including closing the paging system if necessary. The reason the two procedures are split is that it is possible to perform several "open_plot" and "close_plot" calls inside the same "begin_dot_plotter_unit" and "end_dot_plotter_unit" section of code. Whether this is really a desired feature is unclear at this time. The options may prove too confusing to a user.

The only two graphic primitives are

```
moveto
lineto
```

Both "moveto" and "lineto" accept a single parameter of TYPE "vector". A "vector" internally has a "size" variable to indicate whether it is two- or three-dimensional. "moveto" sets the cursor (or pen) to a specified position but does not draw a line. "lineto" moves the cursor (or pen) from its current position to a new position while drawing a line.

"lineto" also resets the cursor's (or pen's) current position to the new location. With a CRT graphics device "lineto" would trace through the pixels and sets their values to the current default. With a pen plotter "lineto" simply draws a straight line segment having a specified color. Either interpretation of "lineto" can be used to explain its function in "dotplotter" but the CRT analogy is perhaps most appropriate.

The "moveto" and "line" primitives are easily understood for two-dimensional graphics. An implied projection from three to two dimensions is involved for three-dimensional vectors. Projections and vector transformations will be discussed after pixels are introduced below.

The following TYPE and PROCEDURES involve pixel definition:

```
TYPE:        pixel

PROCEDUREs:  dot_color
             fill_color
```

Pixels can have only "black" and "white" colors in this first implementation. The "dot_color" PROCEDURE sets the "pixel" color used by the "lineto" PROCEDURE. The "dot_color" can be changed any number of times. The "fill_color" defines the default picture color and for now should only be defined once -- when the picture is initially created. The disk blocks which are mapped onto the picture do not exist until at least one pixel within a block is referenced. Until a block is created all the virtual pixels are treated by "put_plot" as if they have the "fill_color". When the block is created it is initialized with the current value of the "fill_color". Future versions may allow several fill colors but permit virtual pixels to exist until they are explicitly defined.

The vectors passed to the "moveto" or "lineto" primitives can represent either two- or three-dimensional points. Many times the raw data points must be transformed in some way before they should be plotted. To allow an automatic transformation of all points, default transformation matrices are separately stored in the plot file prefix. These default transformation matrices can be manipulated with the following PROCEDUREs:

```
clear_transform
set_transform
get_transform
```

The "begin_dot_plotter_unit" calls "clear_transform" for both the two- and three-dimensional transformation matrices. The default action of "dotplotter" is to plot points unmodified by any transformation process.

"clear_transform" simply sets the internal "size" of the transformation matrices to 1 (recall a 2D matrix has a "size" of 3; a 3D matrix has size 4) and they are not used. "set_transform" saves a matrix to be used as a default transformation. A transformation matrix is typically created by the PROCEDURES in the "matrixops" UNIT as a product of translation, rotation and scaling matrices. (A particularly useful transformation matrix for three-dimensional objects is defined by "view_transform_matrix" as a product of five matrices: one translation, three rotations and a scaling matrix.) A transformation matrix can be retrieved from the parameter prefix file with "get_transform". The retrieved matrix can then be printed or modified and re-saved.

Three-dimensional vectors after any necessary transformations still must be projected onto a two-dimensional surface for plotting. The following TYPE and PROCEDUREs control such projections:

TYPE: projection

PROCEDUREs: project
set_projection_type

A variable of "projection" TYPE can have values of "orthographic" or "perspective". An "orthographic" projection involves simply ignoring the "z" component and plotting the "x" and "y" vector components. A "perspective" projection can be performed easily but usually requires the appropriate "view_transform_matrix" (see "matrixops" UNIT) be defined and saved with "set_transform". Subsequent "moveto" and "lineto" operations will automatically perform the "perspective" projections.

A user can control the projection process using the "project" PROCEDURE. "project (u,v)" projects the three-dimensional "u" vector into a two-dimensional "v" vector using the current "projection" TYPE. Subsequent "moveto" or "lineto" operations using "v" would involve only two-dimensional operations. Caution: A projected vector would still be affected by any two-dimensional transformation matrix.

Two PROCEDUREs define the relationship between the logical world coordinates and the physical screen coordinates:

size
window

"size" defines the dimensions of the physical screen. The default "size" defined by "begin_dot_plotter_unit" is a 5-inch by 5-inch square area. "size" internally defines some variables which are used by "put_plot". "size" also calls "window" in case the user does not. For two-

dimensional graphics, the "window" defines the logical screen using world coordinate system. The default world coordinates are simply 0.0 to 1.0 for both "x" and "y" dimensions. The "window" simply defines the minimum and maximum values which will be allowed in the "x" and "y" dimensions. These world coordinates are then mapped onto the physical rectangular screen defined by "size". Distortions can occur if the aspect ratio (the ratio of the "x" and "y" dimensions) of the screen is not the same as the window. Such distortions may be desirable. For three-dimensional graphics the window definition is usually not important since a scene can be projected onto a logical screen of any size. The screen coordinate system is independent of the eye coordinate system.

Many times line segments extend outside the area of the logical screen and must be clipped. The PROCEDUREs

```
view
clipping
```

control the clipping process. "view" defines a logical subset of the "window". (This is NOT the "viewport" of Newman and Sproull). A call to "window" automatically sets the "view" to be the same as the "window". The "clipping" parameter must be of TYPE BOOLEAN specifying whether or not clipping should be performed -- clipping does involve considerable overhead. When "clipping" is TRUE lines segments will be clipped to the rectangular "view" area. At present the "clip" PROCEDURE is part of the IMPLEMENTATION and is hidden from user access.

While clipping does involve considerable overhead, failure to clip a line will potentially result in a run time error. Subscripts for each pixel are calculated and failure to clip a line will result in an out-of-range subscript. Clipping should only be disabled for re-runs or when it is absolutely unnecessary.

The vectors passed to "moveto" and "lineto" are first transformed if so specified by the default transformation for the given dimensionality. Clipping then occurs only during a "moveto" operation. Once transformed and clipped, a three-dimensional vector is then projected onto the logical screen.

The "put_plot" PROCEDURE deserves further discussion. "put_plot" internally has PROCEDUREs "pgc", "first_printer_scan", "middle_printer_scan", and "last_printer_scan" as well as FUNCTION "dot_value". The "_printer_scan" PROCEDUREs show the considerable overhead necessary to implement a dot border around a picture. The "pgc"

PROCEDURE 'puts' a graphics character on the printer. "pgc" buffers null characters and traps the graphics escape character recognized by the printer. The "ids560" UNIT could not be used to control the IDS 560 printer because of symbol table space limitations. The FUNCTION "dot_value" returns the characteristics of a given "pixel". "dot_value" traps references to virtual pixels defined only by the "fill_color" value and pages other disk blocks to access the defined pixels.

C.4 "matrixops" UNIT

The matrix operations for transformations of two- and three-dimensional vectors are contained in the "matrixops" UNIT. The TYPEs and PROCEDUREs defined by this UNIT are

```
TYPEs:        axis
              coordinates
              rotation

PROCEDUREs:   matrix_identity
              matrix_inverse
              matrix_multiply
              print_matrix
              print_vector
              rotate_matrix
              scale_matrix
              translate_matrix
              view_transform_matrix
```

The "transform" PROCEDURE in the "global" UNIT logically belongs in this UNIT. It was placed in "global" because of compile-time symbol table space limitations.

The TYPEs are used to define certain parameters for the PROCEDUREs. An "axis" may be "x_axis", "y_axis" or "z_axis". "coordinates" may be either "cartesian" or "polar". A "rotation" may be either "cw" for clockwise or "ccw" for counter-clockwise.

The sense of rotation around one of the "axis" TYPEs is defined to be clockwise when an observer is along the axis looking toward the origin. (The sense would be counter-clockwise from the origin's standpoint). Use of "cw" and "ccw" is intended to clarity the direction of a positive rotation from the standpoint of an observer. Different textbook authors use different conventions and the "cw" and "ccw" TYPE is intended to mix both conventions.

The PROCEDURE "matrix_identity" defines an identity matrix and is used for initializing matrices by many of the other PROCEDUREs. "matrix_inverse" finds the inverse of a given matrix while "matrix_multiply" performs the standard multiplication of two matrices.

(Remember: Matrix multiplication is NOT commutative.) The "print_matrix" and "print_vector" procedures can be used for quick output of a transformation matrix or a row vector. Before using either "print_matrix" or "print_vector" the variable "indent" must be defined to be the number of indentation spaces used to set the numbers apart from the heading margin.

The "rotate_matrix", "scale_matrix" and "translate_matrix" PROCEDUREs define the various operations which can be performed on a row vector representing a point in space. Multiplication of such a vector by a transformation matrix is performed by the "transform" PROCEDURE in the "global" UNIT.

"scale_matrix" and "translate_matrix" require an input vector defining the scaling or translation components for each of the dimensions. The dimensionality of the output matrix is determined implicitly from the "size" of the input "vector".

The "size" of a rotational transformation matrix cannot be determined implicitly but must be explicitly specified by a parameter. "rotate_matrix" also requires specification of the axis, the angle, and the sense ("cw" or "ccw") of rotation. See the comments in the program listing for a rotation about an arbitrary axis. For two-dimensional rotations, only the "z_axis" should be specified since all points are assumed to be in the x-y plane.

The "view_transform_matrix" PROCEDURE combines several transformations to convert from world coordinates to eye coordinates. Also, screen specifications are made since eye coordinates and screen coordinates are independent of each other. The position of the eye is defined in polar (azimuth,elevation,distance) coordinates or cartesian (x,y,z) coordinates. The size of the logical screen and the distance from which it will be viewed by the eye define whether the view will be telephoto or wide-angle. The "view_transform_matrix" is extremely useful as the default three-dimensional transformation matrix passed to "set_transform".

C.5 "ids560" UNIT

Most of the utility of this UNIT is provided by the TYPE "control_char" and the PROCEDURE "control". Passing a "control_char" to the PROCEDURE permits the user to symbolically use the control characters

that the IDS 560 recognizes. The control characters include "null", "enhanced_mode" (double width characters), "normal_mode" (normal width characters), "graphics_mode", "just_on" (justification on), "just_off" (justification off), "fixed_spacing", "proportional_spacing", "ht" (horizontal tab), "lf" (line feed), "vt" (vertical tab), "ff" (form feed), "cr" (carriage return), "select_printer", "deselect_printer", "subscript", "superscript", "pitch10" (10 characters/inch horizontally), "pitch12" and "pitch16" (16.8).

Other IDS 560 features can be used with escape sequences. The following PROCEDUREs use such sequences:

```
forms
line
margins
position
tab
```

"forms" defines the physical page size in inches of the forms being used. Since the IDS 560 uses a vertical increment of 1/48-th inch, conversions are internally made. A "skip space" can be specified to avoid printing on the perforation of continuous forms. "margins" sets the physical left and right margins in inches. Since the IDS 560 uses a horizontal increment of 1/120-th inch, conversions are made internally.

"position", "tab" and "line" are used to position the print head to a specific location. "position" is much like the "dotplotter" "moveto" PROCEDURE except the parameters specify distance from top of page and distance from left margin (i.e., upper left corner of page is origin). "tab" moves the print head to a specific column and "line" moves the print head to a specific line. Caution must be used with "position" and "line" that reverse paper feeding does not cause a paper jam. The IDS 560 does not have a reverse tractor feed mechanism.

Other printer features which are not yet in "ids560" include vertical advance programming, vertical and horizontal tab programming, and intercharacter spacing programming.

C.6 File Structure

The structure of the plot file is completely hidden from a user's direct access. A user can directly define the name of the file but only indirectly define its size. A user can control the name of the file by specifying "plt_name" and a "plt_mode" other than "undefined". The absolute maximum size of the file is defined by "max_blks" which is a

CONSTant in the "dotplotter" IMPLEMENTATION. The "size" PROCEDURE defines the physical dimensions of the logical screen and implicitly defines the number of disk blocks needed to contain the whole plot file. A matrix of disk blocks is mapped onto the logical screen; PROCEDURE "size" defines VARiables "i_blks" and "j_blks" which are the dimensions of that matrix. Since "i_blks" and "j_blks" are defined with a *zero origin*, the product "(i_blks+1)(j_blks+1)" must not exceed "max_blks". Since a block does not exist until it is actually needed, "max_blks" of disk storage need not necessarily be available.

Parameters required to define the logical screen and other operations (e.g., default transformation matrices) are stored in the file prefix. That is, the first few blocks of a plot file simply contain various parameters -- the file prefix. The remaining blocks containing pixels are dynamically added as necessary. The parameters are always kept in memory during operation of the "dotplotter" UNIT but are *written* to disk by IMPLEMENTATION PROCEDURE "write_parms" called by "close_plot". If a plot file is subsequently overlayed, the parameter prefix is read from disk before any other operations are performed.

The VARiables in the prefix are shown in the "dotplotter" listing in Section A.2. "prtfile" is a character string 'DOTplot' used to ensure that an overlay operation will access only a valid plot file.

The "blk_table" ARRAY represents the mapping of the logical screen to disk blocks. Each matrix element of "blk_table" is a pointer to a disk block containing a rectangular array of pixels for a corresponding area of the picture. Since block 0 is the first block of a file and at least the first two blocks of a plot file are used for the prefix, the first block of pixels is block 2. Elements of "blk_table" with positive values less than 2, i.e., 0 and 1, are used to indicate fill colors "white" and "black" instead of actually pointing to a disk block. At present multiple fill colors are not implemented. When a "blk_table" element points to blocks 0 or 1 "fill_color" (see below) is used. (Negative elements should never appear in the disk file. Negative elements are used in memory as special flags but should be removed by "close_plot". See explanation of Paging System.) The "i_blks" and "j_blks" variables define the size of "blk_table" which is currently being used to contain a picture of size "x_length" by "y_length" composed of a pixel matrix of size "i_length" by "j_length". To reduce

computations the variables "x_dots" and "y_dots" define the number of dots per world coordinate unit in the horizontal and vertical directions. "x_dots" and "y_dots" are defined in the "window" PROCEDURE based on the IMPLEMENTATION CONSTants "i_density" and "y_density" which define the dot density for the graphics printer (84 dots/inch for IDS 560). The "plt" FILE contains "n_blks" at any given time.

The "window" parameters "x_first", "x_last", "y_first" and "y_last" are kept in the parameter prefix as well as the "view" clipping parameters "x_west", "x_east", "y_south" and "y_north". The "view" parameters are also converted to center-size variables "vcx", "vsx", "vcy" and "vsy" to aid in perspective projection of three-dimensional vectors.

The variables "xform_2D" and "xform_3D" contain the default two- and three- dimensional transformation matrices. The variable "prjctntype" defines whether projections will be "orthographic" or "perspective". "clip_flag" dictates whether clipping of lines segments will occur. The current default pixel definition is contained in "dot_color". The "fill_color" variable determines the fill color of a newly allocated block of pixels or virtual pixels never explicitly defined.

The "prm" PACKED RECORD is a 'free union' -- a variant record without a tag field. The "prm_char" PACKED ARRAY OF CHAR variant is used only to move a block of the parameter file to (from) a memory frame before it is written to (read from) disk. The other variant defines the variables discussed above.

C.7 Paging System

A logical screen is a rectangular area represented by a matrix of pixels. This pixel matrix is subdivided into smaller rectangular matrices each of which requires a single block of disk storage. Since memory for the pixel matrix easily exceeds that which is available, a demand paging system was developed to maintain as many blocks of pixels as possible in memory frames for manipulation. The "blk_table" ARRAY described in Section C.6 contains one element for each possible disk block. The element contains a pointer to the disk block or a "fill_color" value.

The "begin_dot_plotter_unit" assigns default values for the dimensions of the logical screen. A user can change these defaults by

subsequent "size", "window" or "view" calls. When "open_plot" is called, all dimensional values are known and the "blk_table" is initialized with the "fill_color". The internal "allocate_frames" PROCEDURE allocates as many variables of TYPE "memory_frame" as possible -- all frames are allocated at one time -- and initializes the FIFO replacement control variables. The "frame_ptr" ARRAY contains pointers to the "memory_frame" variables.

Subsequent "lineto" calls first may involve transformations, projection and clipping but ultimately the IMPLEMENTATION PROCEDURE "world_to_dot" is called to convert the segment endpoints into logical pixel addresses. These endpoint addresses are passed to the IMPLEMENTATION PROCEDURE "dot_seg" which selects all the pixels between the endpoints. "dot_seg" calls "dot_tag" as it selects each pixel. "dot_seg" first calculates the logical block address of the pixel. This block address consists of an (i_blk,j_blk) subscript pair for lookup in the "blk_table" matrix. If the pointer from "blk_table" is non-negative, the PROCEDURE "get_blk" is called to bring the desired block into memory. The "get_blk" PROCEDURE returns the "memory_frame" address containing the block. If the pointer from "blk_table" is negative, its absolute value is the frame number containing the block in memory. Once the block is known to exist in memory, the offset address within the block is calculated. This offset is in the form of a (i_dot,j_dot) subscript pair. The pixel's attributes can then be easily assigned.

The operation of "get_blk" should be further explained. The first check made by "get_blk" is to determine if an unused "memory_frame" exists. If one exists it is selected to contain the required disk block. If an unused "memory_frame" is not available, FIFO replacement selects the "oldest" block for replacement. (FIFO replacement is used to ensure the direct access Pascal file is created in a sequential fashion.) Replacement involves transferring the block from the "memory_frame" to the I/O buffer followed by a SEEK and a PUT.

Three ARRAYs store information about a block when it is stored in a "memory_frame": "frame_i_blk", "frame_j_blk" and "blk_index". The (frame_i_blk,frame_j_blk) subscript pair indicates the logical block address while "blk_index" is the pointer to the actual disk block. This information is stored in these ARRAYs since once a block is in a memory frame its normal disk block pointer in the "blk_table" is replaced by its

frame number with a negative sign. When a block is written back to disk, these ARRAYs are used to restore the "blk_table" pointer.

Once a "memory_frame" is available, a check is made to determine if the referenced block ever existed on disk before. If this is the first reference to the block, it is initialized with the "fill_color" pixel value. If the block already exists (indicated by a "blk_table" value) 1) a SEEK and GET are performed to bring the block into the memory buffer. From the buffer it is transferred to the "memory_frame".

In both the GET and PUT I/O operations the block was transferred through an I/O buffer on its way to or from a "memory_frame". The MOVELEFT system intrinsic was used for the move since it did not mind the difference in the definition of the "plt" FILE OF PACKED ARRAY OF CHAR and the "memory_frame" PACKED ARRAY OF "pixel" -- both of which were chosen to be exactly "block_size" bytes long (512 in current implementation). The UCSD system does not allow the user to directly assign a value to a file pointer, "plt^" in this case, to avoid the move operation. This GET-MOVE and MOVE-PUT file access was the only alternative that,would allow a fixed two-block parameter prefix followed by a random file accessed a block at at time. A FILE of variant RECORDs containing the two-block or more prefix as one variant would not have allowed random I/O of a single block as a second RECORD variant.

"close_plot" flushes the in-memory frames to disk. The frame pointers in the "blk_table" are replaced by the disk pointers before the prefix file -- which contains the "blk_table" directory of the plot file -- is re-written. The memory frames are then RELEASEd.

The "put_plot" PROCEDURE re-opens the plot file and re-allocates memory frames if "close_plot" has closed the file and released the frames. If the file is not closed, "put_plot" will use the existing frames. The "dot_value" FUNCTION (local to "put_plot") returns the value of a given pixel. "dot_value" also calls "get_blk" to perform any necessary paging of pixel blocks. "put_plot" leaves the state of the plot file and memory frames in the same state it found them. If the file was open and the frames allocated, they stay that way. If "put_plot" must open the file and allocate frames, it closes the file and releases the frames on exit.

Appendix D. User's Guide

D. User's Guide

D.1 Primitives

The definitions of the symbols in the "global", "dotplotter", "matrixops" and "ids560" UNITs which a user may access are listed below. Certain symbols which a user should not need or should never access are not listed. The three-letter comments "GLB", "DOT", "MTX" and "IDS" refer to the UNIT which contains the entity. The USES statement (discussed in Section D.2) must indicate which UNITs a user PROGRAM needs to access. The TYPE definitions are necessary to discuss in detail the parameters of the various PROCEDUREs. The VARiables and FUNCTION are included primarily for completeness.

NOTE: Only the first eight characters are significant in UCSD Pascal (Version II.0) symbols.

```
TYPE
  axis          =  (x_axis,y_axis,z_axis);                          (MTX)
  control_char  =  (null,enhanced_mode,normal_mode,graphics_mode,   (IDS)
                    just_on,just_off,fixed_spacing,ht,lf,vt,ff,cr,
                    proportional_spacing,select_printer,
                    deselect_printer,subscript,superscript,
                    pitch10,pitch12,pitch16);
  coordinates   =  (cartesian,polar);                               (MTX)
  dimension     =  (two_D,three_D);                                 (GLB)
  disposition   =  (keep,delete);                                   (DOT)
  index         =  1..4;                                            (GLB)
  matrix        =                                                   (GLB)
    RECORD
      size      :  index;
      mtrx      :  ARRAY[index,index] OF REAL
    END;
  pixel         :  (white,black);                                   (DOT)
  projection    :  (orthographic,persepective);                     (DOT)
  rotation      :  (cw,ccw);                                        (MTX)
  vector        =                                                   (GLB)
    RECORD
      size      :  index;
      vctr      :  ARRAY[index] OF REAL
    END;

VAR
  close_printer:  BOOLEAN;                                          (DOT)
  indent        :  INTEGER;                                         (MTX)
  min_space     :  INTEGER;                                         (DOT)
  plt_mode      :  (undefined,either,create,overlay);               (DOT)
  plt_name      :  STRING[23];                                      (DOT)
  prt           :  TEXT;                                            (GLB)

FUNCTION defuzz(x:  REAL):  REAL;                                   (GLB)
```

```
PROCEDURE begin_dot_plotter_unit;                                          (DOT)
PROCEDURE clear_transform (d: dimension);                                  (DOT)
PROCEDURE clipping (flag: BOOLEAN);                                        (DOT)
PROCEDURE close_plot (disp: disposition);                                  (DOT)
PROCEDURE control (code: control_char);                                    (IDS)
PROCEDURE define_2D_vector (x,y: REAL; VAR u: vector);                     (GLB)
PROCEDURE define_3D_vector (x,y,z: REAL; VAR u: vector);                   (GLB)
PROCEDURE define_3D_vector (u: vector; a: matrix; VAR v: vector);          (GLB)
PROCEDURE dot_color (color: pixel);                                        (DOT)
PROCEDURE end_dot_plotter_unit;                                            (DOT)
PROCEDURE fill_color (color: pixel);                                       (DOT)
PROCEDURE forms (length,skip: REAL);                                       (IDS)
PROCEDURE get_transform (d: dimension; VAR a: matrix);                     (DOT)
PROCEDURE line (n: INTEGER);                                               (IDS)
PROCEDURE lineto (u: vector);                                              (DOT)
PROCEDURE margins (left,right: REAL);                                      (IDS)
PROCEDURE matrix_identity (d: dimension; VAR a: matrix);                   (MTX)
PROCEDURE matrix_inverse (a: matrix; VAR b: matrix; VAR det: REAL);        (MTX)
PROCEDURE matrix_multiply (a,b: matrix; VAR c: matrix);                    (MTX)
PROCEDURE moveto (u: vector);                                              (DOT)
PROCEDURE open_plot;                                                       (DOT)
PROCEDURE position (x,y: REAL);                                            (IDS)
PROCEDURE print_matrix (title: STRING; a: matrix);                         (MTX)
PROCEDURE print_vector (title: STRING; a: vector);                         (MTX)
PROCEDURE project (u: vector; VAR v: vector);                              (DOT)
PROCEDURE put_plot (x,y: REAL; copies: INTEGER; border: INTEGER);          (DOT)
PROCEDURE rotate_matrix (d: dimension; xyz: axis; angle: REAL;             (MTX)
                         direction: rotation; VAR a: matrix);
PROCEDURE scale_matrix (u: vector; VAR a: matrix);                         (MTX)
PROCEDURE set_projection_type (prj_type: projection);                      (DOT)
PROCEDURE set_transform (a: matrix);                                       (DOT)
PROCEDURE size (x_len,y_len: REAL);                                        (DOT)
PROCEDURE tab (n: INTEGER);                                                (IDS)
PROCEDURE transform (u: vector; a: matrix; VAR v: vector);                 (GLB)
PROCEDURE translate_matrix (u: vector; VAR a: matrix);                     (MTX)
PROCEDURE view   (x_min,x_max,y_min,y_max: REAL);                          (DOT)
PROCEDURE view_transform_matrix (viewtype: coordinates;                    (MTX)
              azimuth,elevation,distance: REAL; {or x,y,z: REAL}
              screen_x,screen_y,screen_distance: REAL;
              VAR a: matrix);
PROCEDURE window (x_min,x_max,y_min,y_max: REAL);                          (DOT)
```

These PROCEDUREs can be roughly divided into the following categories:

Graphic Control Primitives	Graphic Primitives	Vector/Matrix Primitives	Printer Control
begin_dot_plotter_unit	lineto	clear_transform	control
clipping	moveto	define_2D_vector	forms
close_plot		define_3D_vector	line
dot_color		get_transform	margins
end_dot_plotter_unit		matrix_identity	position
fill_color		matrix_inverse	tab
open_plot		matrix_multiply	
put_plot		print_matrix	
size		print_vector	
view		project	
window		rotate_matrix	
		scale_matrix	
		set_projection_type	
		set_transform	
		transform	
		translate_matrix	
		view_transform_matrix	

D.1.1 Graphic Control Primitives

```
begin_dot_plotter_unit
open_plot
put_plot
close_plot
end_dot_plotter_unit
```

The "begin_dot_plotter_unit" and "end_dot_plotter_unit" should be the first and last statements executed in defining a logical screen. Typically, the "open_plot" will follow "begin_dot_plotter_unit" and "close_plot" will precede "end_dot_plotter_unit":

```
begin_dot_plotter_unit;
  open_plot;

  (user program with other control, graphic or vector/matrix
  primitives)

  close_plot (keep);
end_dot_plotter_unit;
```

Notice that "close_plot" requires a single parameter of TYPE "disposition". A disposition of "keep" is used if the plot file is to be sent to the printer by a separate task or if the plot file is to be overlayed by other programs. A disposition of "delete" indicates the plot file's disk space can be released back to the system. "put_plot" must precede "close_plot" if "delete" is specified. If "delete" is not specified, the position of a "put_plot" with respect to "close_plot" is not important. "put_plot" statements can otherwise appear anywhere after the "begin_dot_plotter_unit".

"put_plot" requires four parameters. The "x" and "y" values give the position of the upper left corner of the plot area on the printed page. The upper left corner of the plot will appear "x" inches from the left margin and "y" inches from the top to the page. Care must be taken to avoid reverse paper feeding. (The IDS 560 needs a reverse paper tractor feed mechanism.) The "copies" parameter of "put_plot" indicates how many copies of the plot are to be printed. Each copy automatically starts on another page. The "border" parameter indicates the size of the border surrounding the picture. The size is indicated in units of dots. Typically, one to three dots provide a nice border but some applications require none. The call

```
put_plot (1.50,2.25,1,2);
```

will start the picture 1.5 inches from the left margin and 2.25 inches from the top of the page; a single copy will be produced with two border dots.

```
dot_color
fill_color
```

"dot_color" is used to specify whether pixels traced over by "lineto" should be "black" or "white", i.e., "dot_color" is the color of the lines. (Future versions for use with the IDS "Prism" printer hopefully will support eight colors.) The "fill_color" is the default color of the picture. The "begin_dot_plotter_unit" sets initial values by issuing calls:

```
dot_color (black);
fill_color (white);
```

The user can freely change the "dot_color". If "black" is to be the fill color, "fill_color (black)" should be included before any "lineto" operations are performed. Once set, "fill_color" should not be changed.

```
size
window
view
clipping
```

"size" defines the physical screen size in inches onto which the logical screen is mapped. For example, "size (8.0,5.0)" defines

that a plot will be 8.0 inches wide by 5.0 inches high. "window" defines the size of the logical screen in world coordinates. For example, "window (1.0,9.0, 0.0,5.0)" indicates the plotting area will have coordinates from 1.0 unit to 9.0 units along the "x" axis, and 0.0 units to 5.0 units along the "y" axis. The aspect ratio of the physical screen and logical screen must be the same for most cases. Distortion of one dimension relative to the other occurs if the aspect ratios are not the same. For the "size" and "window" examples above, the aspect ratio is the same (the ratio of "x" to "y" dimension = 1.6).

When "size" is called a default "window (0.0,1.0, 0.0,1.0)" is set. "size" should only be called once, typically before the "open_plot". The "window" specification could be changed if desired by the user.

The "clipping" PROCEDURE sets a flag to indicate whether or not clipping should be performed. Unless absolutely sure that clipping is not necessary, clipping should not be turned off. "begin_dot_plotter_unit" specifies "clipping (TRUE)" as the default. Run time range errors can occur if clipping is necessary but not requested.

"view" defines the clipping area. (This is not the standard definition of a "viewport".) A "view" is defined in world coordinates just like "window". If fact, the largest "view" possible is specified by "window" without a separate call to "view", i.e., a call to "window" sets an identical "view". For the "window" example above, both "view (1.0,9.0, 0.0,5.0)" and "view (2.0,5.0, 2.0,5.0)" are valid but "view (0.0,10.0, 0.0,5.0)" is not since the "view" cannot extend outside the "x" window dimension.

Setting a "window" or "view" for three-dimensional graphics is rarely necessary. Any three-dimensional picture can be projected onto a logical screen of any size. Setting a different "view" for 3D graphics results in the whole picture being projected into the "view" area instead of the original "window" area. Changing the "window" has no effect since the 'eye' coordinate system and the 'screen' coordinate system are independent. The "view_transform_matrix" really controls what the object will look like and how much clipping will be necessary.

D.1.2 Graphic Primitives

lineto
moveto

The "moveto" primitive establishes the position of the logical screen's cursor. A subsequent "lineto" call moves the cursor from its current position to the specified new position while tracing over and setting pixels to the value established by "dot_color". The "lineto" position becomes the new position of the cursor. Nearly always a call to "define_2D_vector" or "define_3D_vector" precedes a call to either "moveto" or "lineto". The following demonstrates usage of "moveto" and "lineto" to draw a line from point (0,0) to (1,1) to (1,-5):

```
VAR
  u  :  vector;
  x,y:  REAL;
 ...
define_2D_vector (0.0,0.0, u);
moveto (u);
define_2D_vector (1.0,1.0, u);
lineto (u);
define_2D_vector (1.0,-5.0, u);
lineto (u);
 ...
```

Three-dimensional graphics use similar statements. A three-dimensional segment is automatically projected into two dimensions depending on the "set_projection_type" of "orthographic" or "perspective". The following demonstrates usage of "moveto" and "lineto" to draw a line from the point (0,0,0) to (1,1,1) to (1,-1,5):

```
VAR
  u     :  vector;
  x,y,z:  REAL;
  ...
define_3D_vector (0.0,0.0,0.0, u);
moveto (u);
define_3D_vector (1.0,1.0,1.0, u);
lineto (u);
define_3D_vector (1.0,-1.0,5.0, u);
lineto (u);
  ...
```

The vector passed to "moveto" or "lineto" will be multiplied by the default transformation matrix for the appropriate dimensionality (if one exists). "begin_dot_plotter_unit" clears both transform matrices (2D and 3D). The user must use "set_transform" to establish a default transformation matrix.

D.1.3 Vector/Matrix Primitives

define_2D_vector
define_3D_vector

Given the variables "u" and "v" of TYPE vector, and REAL "x", "y" and "z", "define_2D_vector (x,y, u)" defines a two-dimensional vector and "define_3D_vector (x,y,z, v)" defines a three-dimensional vector. Assuming "v" is defined, the statement "u := v" can be used to assign values to vector variables. In an assignment statement, not only are the vector components transferred, but the dimensionality of the vector is also transferred. The user is responsible not to mix vectors and matrices of ditferent dimensionalities.

matrix_identity
matrix_inverse
matrix_multiply

Given "matrix" variable "a", "matrix_identity (two_D,a)" defines a 3 by 3 identify matrix for a 2D identity transformation. This PROCEDURE is available to the user but is really intended for initializing matrices by other PROCEDUREs within the "matrixops" UNIT.

Given "matrix" variables "a", "b", "c" and REAL variable "det", "matrix_inverse (a, b, det)" defines "b" to be the inverse of matrix "a" with "det" the determinant value of "a". If the determinant value is zero, the "b" matrix is undefined and assigned a dimensionality inappropriate for future usage.

"matrix_multiply (a,b, c)" defines matrix "c" to be the product of "a" and "b". Care must be taken that "a" and "b" have the same dimensionality or "c" is undefined. The user is reminded that matrix multiplication is not commutative. The product "ab" is very different from "ba" (unless "a" or "b" is an identity matrix or a zero matrix). When creating a complex transformation matrix, the order of multiplication of the individual transformation matrices is very important.

rotate_matrix
scale_matrix
translate_matrix
view_transform_matrix

"rotate_matrix" defines a rotation transformation matrix. Consider the following examples:

```
rotate_matrix (two_D,z_axis,45.0,cw, a);
rotate_matrix (three_D,y_axis,22.0,ccw, b);
```

The first example is for two-dimensions. Since it assumed that the x-y axis is used for two-dimensional plots, all two-dimensional rotations must be about the "z_axis". The rotation is to be 45.0 degrees clockwise ("cw"). Rotations for three-dimensional plots may be about any of the "x", "y" or "z" axes. The second example shows a 22.0 degree counterclockwise ("ccw") rotation about the "y_axis". The sense of the rotation ("cw" or "ccw") is from a view along the "y_axis" looking toward the origin.

"scale_matrix" and "translate_matrix" require a vector containing the scaling factor or translation component for each dimension. The dimensionality of the resulting matrix is the same as the dimensionality of the vector. Consider these examples:

```
define_3D_vector (0.50,1.50,1.0, u);
scale_matrix (u, a);

define_3D_vector (10.0,20.0,25.0, u);
translate_matrix (u, b);

matrix_multiply (a,b, c);
```

The first two statements define a scaling matrix "a" in which the "x" components will be scaled to be half as large as the original values; "y" components will be scaled to be 1.50 times the original values and "z" components are unchanged. The second two statements define a translation matrix "b" in which 10.0 is added to each "x" component, 20.0 is added to each "y" component, and 25.0 is added to each "z" component. A composite matrix "c" contains both scaling and translation transformations.

The "view_transform_matrix" PROCEDURE defines a special transformation matrix as the product of scaling, rotation and translation matrices. This matrix primitive converts world coordinates to eye coordinates. The eye's point of vision from an object can be specified in either polar coordinates or cartesian coordinates. The eye's intended viewing distance from the screen on which the projection will occur must be specified as well as the size of the screen. If the ratio of the distance to the screen to the size of the screen is large, the view will be telephoto-like; if the ratio is small the view will be a wide-angle view. The following examples are equivalent:

```
view_transform_matrix (polar, 45.0,35.2644,1.7321, 4.0,4.0,10.0, a);
view_transform_matrix (cartesian, 1.0,1.0,1.0,     4.0,4.0,10.0, a);
```

The center of the object being viewed is at the origin (0,0,0) and the observer is at point (1,1,1) which is a 45.0 degree azimuth, 35.2644 degree elevation, 1.7321 world units from the origin. The screen is 4 units wide by 4 units high; the screen is to be viewed from 10 units away. The distance to size ratio of the screen is 2.5 in both dimensions. The screen "x" and "y" points will be scaled by 2.5 before plotting. The view is therefore somewhat telephoto-like. The "view_transform_matrix" is often used as the default 3D transformation matrix established using "set_transform".

clear_transform
set_transform
get_transform
transform

Separate default transformation matrices are stored for both two- and three-dimensional graphics. Once a transformation matrix "a" is created (by using "view_transform_matrix" or a product of other transformation matrices), it is saved using "set_transform (a)". All subsequent "moveto" and "lineto" vectors of the same dimensionality will be multiplied by the transformation matrix. "clear_transform (two_D)" or "clear_transform (three_D)" turns off the use of a transformation matrix for the given dimensionality. The transformation matrix can be retrieved for display or additional modification by using "get_transform (two_D, a)" or "get_transform (three_D, a)".

The "transform" PROCEDURE multiplies a row vector "u" by a square matrix "a": "transform (u,a, v). The resulting row vector "v" is transformed by whatever is dictated by the transformation matrix "a". For most simple applications the user does not directly use "transform". The automatic transformation provided by "set_transform" performs this vector-matrix multiplication.

set_projection_type
project

"orthograpic" and "perspective" are the projection types that can be passed to "set_projection_type". Normally, these projections occur

automatically when 3D vectors are passed to "lineto". "project (u, v)" projects the 3D "u" vector into a 2D "v" using the current projection type.

print_matrix
print_vector
: These PROCEDUREs were added to provide quick display of matrices or vectors. The "ident" variable of the "matrixops" UNIT must be assigned a value before call either PROCEDURE. Output is made to the "prt" FILE defined in the "global" UNIT.

D.1.4 Printer Control

control
forms
line
margins
position
tab
: "control" is used to mnemonically pass special control characters to the IDS 560 printer. "forms" defines the length of the forms being used as well as a 'skip space' to prevent printing on the perforation of continuous forms. "margins" sets the left and right margins in absolute terms [inches]. "line" and "tab" quickly move the print head to any line on the page from the top of the form or any character position from the left margin. "position" is used to move the print head to an absolute position on the page.

D.1.5 Miscellaneous

The user accessible UNIT variables and the "defuzz" FUNCTION need to be explained for completeness.

"defuzz" is used in "matrixops" to remove the "fuzz" present in REAL calculations. The "fuzz" value is set to 1.0E-6. "defuzz" changes all values less than "fuzz" to zero. The FUNCTION is available to the user but probably will be rarely needed.

The "prt" TEXT FILE is used for all printed output -- normal and graphics -- to the IDS 560. "prt" may be used for WRITE/WRITELN statements by the user at any time. Inside "put_plot" the graphics mode is entered and exited. From the user's standpoint, the IDS 560 is always in normal mode.

The "plt_mode" and "plt_name" VARiables are used to name the plot disk file if the user does not want to be prompted for a name. The default "plt_mode" value is "undefined" and the user is prompted for a file name at the time the file is to be opened. If the user specifies a "plt_mode" of "create", "overlay" or "either", the "plt_name" will be used as the name of the plot file. "create" will cause a new file to be created regardless of whether the file already exists. "overlay" will overlay a plot file which must already exist. "either" will create the file if it does not exist, overlay the file if it does.

At the time the plot file is opened, almost all available memory is allocated to memory frames. The "min_space" variable controls the amount of memory not allocated to storage frames. The default value at present is 1024 bytes. If a stack overflow message is received, fewer frames should be allocated by specifying a larger "min_space" value before the "open_plot" call.

When "end_dot_plotter_unit" is issued, the system normally performs a "PAGE (prt)". If this is not desired, set "close_printer" to FALSE. (This variable will be renamed in the future.)

Both "print_matrix" and "print_vector" print a title line followed by the matrix or vector of numbers indented under the title. The number of spaces of this indentation is set by the "indent" VARiable. This VARiable does NOT have a default value so one must be specified before "print_matrix" or "print_vector" is used.

D.2 Sample Setup

Appendix B "Source Listings of Sample User PROGRAMs" should provide invaluable documentation by example of how to use this software package. However, the following skeleton should prove helpful for most one-time runs:

```
PROGRAM user_pgm;
USES global, matrixops, dotplotter, ids560;
CONST ...
TYPE ...
VAR
  a:  matrix;
  u:  vector;
  ...
BEGIN
  REWRITE (prt,'PRINTER:');  (if needed before "open_plot" call)
  ...
  begin_dot_plotter_unit;
    plt_mode := create;
    plt_name := 'disk:name.extension'  (temporary plot file)
    size (x_len (inches wide),y_len (inches high));
    window (x_min,x_max,y_min,y_max);  (not usually needed for 3D)
    open_plot;
    ...
    (define transform matrix "a")
    set_transform (a);
    ...
    moveto (u);
    ...
    lineto (u);
    ...
    put_plot (1.0,1.0, 1 (copy), 2 (border dots));
    close_plot (delete);  (or "keep")
  end_dot_plotter_unit;
  ...
END (user_pgm).
```

D.3 Messages and Errors

The following messages are listed in alphabetical order by code. Each message code consists of a prefix indicating which UNIT contains it and a sequence code. DOT represents "dotplotter", GLB represents "global" and MAT represents the "matrixops" UNIT. The underscored fields indicate where substitutions will occur. A brief explanation follows the text of each message.

DOT01 Unexpected I/O error 'n' in routine 'xxxxxxxx' while performing a [read/write] operation.

Error 'n' is defined in the UCSD Users' Guide, Table 2 "IORESULTS", p. 313. This message will occur when an I/O error occurs paging a block of pixels to or from disk. For example, n=8 for "No room, insufficient space".

DOT02 Terminal Error: File 'disk:name.extension' is not a plot file.

When a user is prompted for a file name in overlaying an existing plot file, it is possible to respond with a valid file name which is not a plot file. Internally, the first eight bytes of each plot file is "DOTplot" to make this check.

DOT03 Window parameter error(s):
x_min = nn.nn, x_max = nn.nn, y_min = nn.nn, y_max = nn.nn

This error occurs when x_max < x_min or y_max < y_min.

DOT04 View parameter error(s):
x_min = nn.nn, x_max = nn.nn, y_min = nn.nn, y_max = nn.nn
Window parameters:
x_min = nn.nn, x_max = nn.nn, y_min = nn.nn, y_max = nn.nn

This error occurs when x_max < x_min or y_max < y_min for the view specification or if the "view" is outside the "window".

DOT05 [Horizontal/Vertical] dimension (nnn.nn inches) too large:
nnn > mmm dots.

Internal limits are imposed on the maximum number of pixels either horizontally or vertically. The horizontal limit is based on the width of the graphics printer. The vertical limit is based on available disk space using the maximum horizontal limit.

DOT06 Error: Frames must be the same size as I/O blocks.
A frame is nn x mm bits = kk bytes. A block contains nnn bytes.

At present the memory frames must be the same size as the disk storage units.

DOT07 File parameter prefix is nnn bytes long. Internal variable "prm_size" currently has the value nnn and should be adjusted to be a multiple of "block_size" (currently mmm) greater than or equal to the size of the parameter prefix.

The current parameter file is two blocks long.

DOT08 Terminal Error: No memory frames allocated.

No space is available for even a single memory frame. Paging block frames is not possible and the run must be aborted.

DOT09 Enter plot file name ([disk:]name.extension) or EXIT:

This prompt occurs if the user does not specify values for "plt_mode" and "plt_name".

DOT10 Error 'n' in opening plot file 'disk:name.extension'.

See IORESULTS in UCSD User's Guide for values of 'n'.

DOT11 [Creating/Overlaying] file 'disk:name.ext'. The plot will be n.nn inches (nnn dots) wide by n.nn inches (nnn dots) high. There will be nn in-memory *block frames*.

These informational messages are displayed on the console screen at the time the plot file is opened.

DOT12 Request ignored to open already open plot file.

This message indicates a probable problem definition error.

DOT13 Warning: No frames used.

This message occurs when a plot file is closed but no memory frames were ever used in paging blocks to/from disk. No pixels were ever defined. The picture is defined by virtual pixels with the default fill color. If a plot file was overlayed, no changes were made.

DOT14 Request ignored to close plot file which is not open.

This message indicates a probable problem definition error.

GLB01 Ignoring attempt to multiply a vector of dimension n by a square matrix of dimension m.

Two- and three-dimensional vectors and matrices *cannot* be intermixed. This message can also occur if the vector or matrix is undefined.

MAT01 Ignoring attempt to multiply square matrices of different dimensions: n and m.

Two- and three-dimensional transformation matrices cannot be intermixed. This message can also occur if either of the matrices is undefined.

MAT02 For 2D rotation in x-y plane, specify "z_axis".

All two-dimensional rotations are assumed to be defined in the x-y plane. If for some reason another plane must be used, use three-dimensional vectors and matrix transformations.

Appendix E. "hexdump" Utility PROGRAM

```
1   {$L- PRINTER:}
2   {$S+ Put compiler in swapping mode.}
3   {$I- The compiler will not generate I/O checking code.}
4   PROGRAM hexdump;  {UCSD Pascal, Version II.}
5
6     {$C Copyright (C) 1981 by Earl F. Glynn, Manhattan, KS.}
7     {Written in November 1981; last modified on 27 March 1982.}
8
9     {"hexdump" prints the blocks of a file in both hexadecimal and ASCII
10    form (unprintable characters are changed to periods).  "hexdump" prints
11    characters in groups of four bytes similar to the format used on IBM
12    370-type machines when a memory dump is produced.  However, "hexdump"
13    allows 16, 32 or 64 bytes to be formatted onto lines which are 64, 116 or
14    220 characters long.  An 80-character title is allowed except for the
15    line which is only 64 characters long.  Block numbers and the address
16    offset are also printed.  At present the user is prompted for the
17    current date.
18
19    Given "n" is the number of 4-byte words (IBM 370 word size) to
20    be formatted per line ("n" must be 4, 8 or 16 for 16, 32 or 64
21    formatted bytes per line), each formatted line contains the following
22    fields:  8 spaces for block number and offset address + 2 spaces +
23    9n-1 hex characters and blank spaces for visual breaks + 1 space
24    and 1 vertical bar + 4*n ASCII characters + 1 vertical bar.}
25
26
27  USES global, ids560;
28    {The "global" UNIT defines a "prt" TEXT FILE.}
29    {The "control" PROCEDURE in "ids560" is used to set the horizontal
30    pitch and the vertical line spacing.  The line spacing must be 8 lines/inch
31    at present.  For the 220-character lines, 16.8 pitch must be used; the
32    other lines can be printed at a user selected pitch of 10, 12 or 16.8.}
33
34  VAR
35    addr        :  INTEGER;          {address relative to beginning of file}
36    blk_count   :  INTEGER;          {number of blocks dumped}
37    blk_number  :  INTEGER;          {block number relative to beginning of file}
38    blocks_read:   INTEGER;          {blocks read by BLOCKREAD intrinsic}
39    buffer      :  PACKED ARRAY[0..511] OF CHAR;
40    buf_idx     :  INTEGER;
41    chr_idx     :  INTEGER;
42    chr_string  :  PACKED ARRAY[0..63] OF CHAR;
43    date        :  STRING[8];        {mm/dd/yy}
44    dump        :  FILE;
45    file_in     :  STRING[23];       {xxxxxxx:xxxxxxxx.xxxxxx}
46    file_out    :  STRING[23];
47    i           :  INTEGER;
48    j           :  INTEGER;
49    n           :  0..16;            {number of 4-byte words formatted per line}
50    option      :  STRING[4];
51    ordx        :  INTEGER;          {ORD(x)}
52    out_type    :  CHAR;
53    page_number:   INTEGER;
54    page_title  :  STRING[80];
55    pitch       :  control_char;     {TYPE "control_char" defined in "ids560"}
56    temp        :  INTEGER;
57    x           :  CHAR;
58
59  FUNCTION hex_digit(index:  INTEGER):  CHAR;
60    {This function returns a hex character '0'..'9' or 'A'..'F'
61    given an integer 0..15; a '?' for integers outside the 0..15 range.}
62  BEGIN
63    IF   index IN [0..9]
64    THEN hex_digit := CHR( index + ORD('0') )
65    ELSE
66      IF   index IN [10..15]
67      THEN hex_digit := CHR( index + ORD('A') - 10 )
68      ELSE hex_digit := '?'
69  END {hex_digit};
70
71  PROCEDURE hex_address (addr:  INTEGER);
72    {Given an unsigned integer 0..65535, this procedure prints the hex
73    equivalent '0000'..'FFFF'.}
74  VAR
75    addr_temp:  INTEGER;
```

```
 76    digit    :  INTEGER;
 77    divisor  :  INTEGER;
 78    i        :  1..4;
 79  BEGIN
 80    addr_temp := addr;
 81    divisor := 4096;
 82    FOR i := 1 TO 4 DO BEGIN
 83      digit := addr_temp DIV divisor;
 84      IF   digit ( 0
 85      THEN digit := digit + 16;
 86      WRITE (prt,hex_digit(digit));
 87      addr_temp := addr_temp MOD divisor;
 88      divisor := divisor DIV 16
 89    END
 90  END {hex_address};
 91
 92  PROCEDURE top_and_bottom_header;
 93  VAR
 94    i     :  0..15;
 95    offset:  INTEGER;
 96  BEGIN
 97    WRITE (prt,'BLK ADDR  ');
 98    FOR i := 0 TO n-1 DO BEGIN
 99      offset := 4*i;
100      WRITE (prt,hex_digit(offset DIV 16),
101                 hex_digit(offset MOD 16),'       ')
102    END;
103    WRITE (prt,' ');
104    FOR i := 0 TO n-1 DO BEGIN
105      offset := 4*i;
106      WRITE (prt,hex_digit(offset DIV 16),
107                 hex_digit(offset MOD 16),'  ')
108    END;
109    WRITELN (prt)
110  END {top_and_bottom_header};
111
112  PROCEDURE inter_block_break;
113  VAR
114    i:  0..155;
115  BEGIN
116    FOR i := 1 TO 11+9*n DO
117      WRITE (prt,' ');
118    FOR i := 0 TO n-1 DO
119      WRITE (prt,':  ');
120    WRITELN (prt)
121  END {inter_block_break};
122
123  PROCEDURE space_fill (start,str_length:  INTEGER);
124    {Pad STRINGs on right with blanks on output instead of right
125    justifying the STRING using 'string:length' output format.}
126  VAR i:  INTEGER;
127  BEGIN
128    i := SUCC(start);
129    WHILE i (= str_length DO BEGIN
130      WRITE (prt,' ');
131      i := SUCC(i)
132    END
133  END {space_fill};
134
135  PROCEDURE heading (blk_count:  INTEGER);
136  VAR i:  1..34;
137  BEGIN
138    IF   blk_count MOD (n DIV 2) = 0      {change test if not 8 lines/inch}
139    THEN BEGIN
140      IF   blk_count ) 0
141      THEN BEGIN
142        inter_block_break;
143        top_and_bottom_header;
144        page_number := SUCC(page_number);
145        IF   out_type = 'A'
146        THEN PAGE(prt)
147      END;
148      WRITE (prt,'Hexadecimal/ASCII Dump of File ',file_in);
149      space_fill (LENGTH(file_in),23);
150      IF   n = 16
```

```
151       THEN BEGIN
152         FOR i := 1 TO 34 DO
153           WRITE (prt,' ');
154         WRITE (prt,page_title);
155         space_fill (LENGTH(page_title),80);
156         FOR i := 1 to 34 DO
157           WRITE (prt,' ');
158       END
159       ELSE
160         IF   n = 8
161         THEN WRITE (prt,'  ',page_title:40,'  ');
162       WRITELN (prt,date:8,'  Page ',page_number:3);
163       WRITELN (prt);
164       top_and_bottom_header
165     END;
166     inter_block_break
167   END (heading);
168
169   PROCEDURE prompts;
170   BEGIN
171     REPEAT
172       WRITELN ('Enter input filename (or EXIT):  ');
173       READLN (file_in);
174       IF   file_in = 'EXIT'
175       THEN EXIT (hexdump);                    {abort}
176       RESET (dump,file_in);
177       temp := IORESULT;
178       IF   temp <> 0
179       THEN
180         BEGIN
181           WRITELN ('File ''',file_in,''' not found.  RC=',temp,'.');
182           WRITELN
183         END
184     UNTIL temp=0;
185
186     out_type := ' ';
187     REPEAT
188       WRITELN ('Enter output type (a) PRINTER, (b) CONSOLE, (c) disk:  ',
189                'a/b/c/EXIT');
190       READLN (option);
191       IF   option = 'EXIT'
192       THEN EXIT (hexdump);
193       IF   (option='a') OR (option='A')
194       THEN BEGIN
195         out_type := 'A';
196         file_out := 'PRINTER:'
197       END
198       ELSE
199         IF   (option='b') OR (option='B')
200         THEN BEGIN
201           out_type := 'B';
202           file_out := 'CONSOLE:'
203         END
204         ELSE
205           IF   (option='c') OR (option='C')
206           THEN BEGIN
207             out_type := 'C';
208             WRITELN ('Enter filename (or EXIT):');
209             READLN (file_out);
210             IF   file_out = 'EXIT'
211             THEN EXIT (hexdump)
212           END;
213     UNTIL out_type <> ' ';
214
215     IF   out_type = 'B'
216     THEN n := 4
217     ELSE BEGIN
218       n := 0;
219       REPEAT
220         WRITELN ('Enter line size (a) 64, (b) 116 or (c) 220 characters:',
221                  '  a/b/c/EXIT');
222         READLN (option);
223         IF   option = 'EXIT'
224         THEN EXIT (hexdump)
225         ELSE
```

```
126           IF   (option='a')  OR  (option='A')
127           THEN n := 4
128           ELSE
129             IF   (option='b')  OR  (option='B')
130             THEN n := 8
131             ELSE
132               IF   (option='c')  OR  (option='C')
133               THEN n := 16
134       UNTIL n ) 0;
135     END;
136
137     IF   out_type = 'A'
138     THEN BEGIN
139       IF   n = 16             {16 four-byte words per formatted line}
140       THEN pitch := pitch16  {16.8 pitch}
141       ELSE BEGIN
142         pitch := null;
143         REPEAT
144           WRITELN ('Enter pitch (a) 10, (b) 12 or (c) 16.8 characters/inch:',
145                    '  a/b/c/EXIT');
146           READLN (option);
147           IF   option = 'EXIT'
148           THEN EXIT (hexdump)
149           ELSE
150             IF   (option='a')  OR  (option='A')
151             THEN pitch := pitch10
152             ELSE
153               IF   (option='b')  OR  (option='B')
154               THEN pitch := pitch12
155               ELSE
156                 IF   (option='c')  OR  (option='C')
157                 THEN pitch := pitch16
158         UNTIL pitch <> null
159       END;
160     END;
161
162     IF   n ) 4
163     THEN BEGIN
164       WRITELN ('Enter page title [up to ',5*n,' characters] (or EXIT):');
165       IF   page_title = 'EXIT'
166       THEN EXIT (hexdump);
167       READLN (page_title)
168     END;
169
170     IF   out_type <> 'B'
171     THEN BEGIN
172       WRITE ('Enter date [mm/dd/yy] (or EXIT):  ');
173       READLN (date);
174       IF   date = 'EXIT'
175       THEN EXIT (hexdump)
176     END
177   END {prompts};
178
179   BEGIN {hexdump}
180     prompts;
181     REWRITE (prt,file_out);
182     IF   out_type = 'A'
183     THEN control (pitch);
184
185     addr := 0;
186     blk_count := 0;
187     blk_number := 0;
188     page_number := 1;
189     blocks_read := BLOCKREAD(dump,buffer,1,blk_number);
190     WHILE (IORESULT=0) AND (blocks_read=1) DO
191     BEGIN
192       IF  out_type = 'B'
193       THEN BEGIN
194         top_and_bottom_header;
195         inter_block_break
196       END
197       ELSE heading (blk_count);
198       buf_idx := 0;
199       FOR i := 0 TO (128 DIV n - 1) DO BEGIN  {8, 16 or 32 lines/block}
300         IF   i = 0
```

```
301         THEN WRITE (prt,blk_number:3,' ')
302         ELSE WRITE (prt,'    ');
303         hex_address (addr);
304         WRITE (prt,' ');
305         chr_idx := 0;
306         FOR j := 0 TO (4*n - 1) DO BEGIN      {16, 32 or 64 bytes/line}
307           x := buffer[buf_idx];
308           ordx := ORD(x);
309           IF   ordx IN [32..126]
310           THEN chr_string[chr_idx] := x       {printable character}
311           ELSE chr_string[chr_idx] := '.';    {unprintable character}
312           IF   j MOD 4 = 0
313           THEN WRITE (prt,' ');               {space every 4 bytes}
314           WRITE (prt,hex_digit(ordx DIV 16));
315           WRITE (prt,hex_digit(ordx MOD 16));
316           chr_idx := SUCC(chr_idx);
317           buf_idx := SUCC(buf_idx)
318         END {FOR j};
319         WRITE (prt,' !');
320         FOR j := 0 TO (4*n - 1) DO
321           WRITE (prt,chr_string[j]);
322         WRITELN (prt,'!');
323         addr := addr + 4*n;
324         IF   (out_type='B') AND ((i+1) MOD 16 = 0)
325         THEN BEGIN
326           inter_block_break;
327           top_and_bottom_header;
328           WRITELN ('Push "RETURN" to continue; enter "EXIT" to terminate:');
329           READLN (option);
330           IF   option = 'EXIT'
331           THEN EXIT (hexdump)
332         END
333       END {FOR i};
334       blk_count := SUCC(blk_count);
335       blk_number := SUCC(blk_number);
336       blocks_read := BLOCKREAD(dump,buffer,1,blk_number)
337     END {WHILE};
338
339     IF   out_type <> 'B'
340     THEN BEGIN
341       inter_block_break;
342       top_and_bottom_header
343     END;
344
345     IF   out_type = 'A'
346     THEN BEGIN
347       control (pitch12);    {reset to user default pitch}
348       PAGE (prt)
349     END;
350     CLOSE (prt,LOCK)
351
352   END {PROGRAM hexdump}.
```

"hexdump" was written as a debugging tool to quickly check the contents of disk files in both ASCII and hexadecimal form. The program was quite helpful in debugging the creation of the "parm" and "plot" files.

HEXDUMP Prompt Sequence:

(1) Enter input filename (or EXIT):

(2) Enter output type (a) PRINTER, (b) CONSOLE, (c) disk: a/b/c/EXIT

(3) Enter line size (a) 64, (b) 116 or (c) 220 characters: a/b/c/EXIT

(4) Enter pitch (a) 10, (b) 12 or (c) 16.8 characters/inch: a/b/c/EXIT

(5) Enter page title [up to nn characters] (or EXIT):

(6) Enter date [mm/dd/yy] (of EXIT):

Note: Option (4) only applies if option (2a) is selected. An additional prompt for a filename will occur if option (2c) is selected. If (2b) is selected, prompt (3) will default automatically to (a). Option (5) only occurs if (3b) or (3c) is selected.

Sample output of "hexdump" showing 16 bytes (line size = 64) formatted per line:

```
Hexadecimal/ASCII Dump of File system.miscinfo    04/03/82  Page   1

BLK ADDR  00       04       08       0C        00  04  08  0C
                                                :   :   :   :
  0 0000  00B60210 B2022EA6 00D7AA0F B20224A6 |..............$.|
    0010  00D7AA0F B2021EA6 00D7AA0F AD000000 |................|
    0020  20D00014 C1B17A00 1AA602C1 00140230 | .....z........0|
    0030  20D00800 C6B17A00 2AA60C00 00001B48 | .....z.*......H|
    0040  4A4B4341 080A6C45 DF001800 50004142 |JKCA..lE....P.AB|
    0050  44430306 0013083F 7F1B1B03 08120F0C |DC.....?........|
    0060  4D00CD00 13B60103 CD0016B9 2BAC4300 |M...........+.C.|
    0070  5400B924 7F010400 04000800 0A007C01 |T..$..........|.|
    0080  0E001000 12001400 16001800 1A001C00 |................|
    0090  97012000 22009901 00A1F2C1 0049EF01 |.. .".........I..|
    00A0  08003500 60010400 0D00FE01 01009604 |..5.'...........|
    00B0  E8042006 6006C206 A805AC07 3807DC07 |.. .'.......8...|
    00C0  00000000 00000000 00000000 00000000 |................|
    00D0  00000000 00000000 00000000 00000000 |................|
    00E0  00000000 00000000 00000000 00000000 |................|
    00F0  00000000 00000000 00000000 00000000 |................|
    0100  00000000 00000000 00000000 00000000 |................|
    0110  00000000 00000000 00000000 00000000 |................|
    0120  00000000 00000000 00000000 00000000 |................|
    0130  00000000 00000000 00000000 00000000 |................|
    0140  00000000 00000000 00000000 00000000 |................|
    0150  00000000 00000000 00000000 00000000 |................|
    0160  00000000 00000000 00000000 00000000 |................|
    0170  00000000 00000000 00000000 00000000 |................|
    0180  00000000 00000000 00000000 00000000 |................|
    0190  00000000 00000000 00000000 00000000 |................|
    01A0  00000000 00000000 00000000 00000000 |................|
    01B0  00000000 00000000 00000000 00000000 |................|
    01C0  00000000 00000000 00000000 00000000 |................|
    01D0  00000000 00000000 00000000 00000000 |................|
    01E0  00000000 00000000 00000000 00000000 |................|
    01F0  00000000 00000000 00000000 00000000 |................|
                                                :   :   :   :
BLK ADDR  00       04       08       0C        00  04  08  0C
```

A MICROCOMPUTER GRAPHICS PACKAGE FOR USE WITH A HIGH-RESOLUTION RASTER-SCAN DOT-MATRIX PRINTER

by

EARL F. GLYNN II

B.S., Kansas State University, 1975

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1982

A general purpose graphics software package was developed for use with a "personal" computer with a high-resolution raster-scan dot-matrix printer. Primitives for both two- and three-dimensions were developed. Only two-color pixels were implemented in this first version. Transformations by rotation, translation and scaling may be specified. Clipping is performed. A user can specify the physical picture size as well as the logical window dimensions. Several examples are explained. Documentation includes source listings, a System Guide and a User's Guide.