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

by

EARL F. CLENN 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:

[Signature]
Major Professor
TABLE OF CONTENTS

1. Introduction
   1.1 Purpose, Motivation and Objectives
   1.2 Background
   1.3 Software Overview

2. A Graphics Tutorial
   2.1 Interactive Versus Passive Computer Graphics
   2.2 Vector Versus Raster Display
   2.3 Two-Dimensional Picture Definition
      2.3.1 World Coordinates and Windows
      2.3.2 Points and Lines
      2.3.3 Clipping
      2.3.4 Viewports
   2.4 Matrix Transformations
      2.4.1 Homogeneous Coordinates
      2.4.2 Translation
      2.4.3 Scaling
      2.4.4 Rotation
      2.4.5 Change in Coordinate Systems
      2.4.6 Concatenation of Simple Transformations
   2.5 Extension to Three-Dimensional Graphics
      2.5.1 Homogeneous Coordinates
      2.5.2 Transformations
      2.5.3 View Transformation
      2.5.4 Projections
      2.5.5 Clipping

3. Examples and Discussion
   3.1 Two-Dimensional Graphics
      3.1.1 Cartography: Map of Kansas
      3.1.2 Orthographic View of a Football Field
   3.2 Three-Dimensional Graphics
      3.2.1 Perspective Views of a Football Field
      3.2.2 Surface Graphics: \( z = f(x,y) \)
      3.2.3 Surface Graphics: Pressure Map of Well Field

4. Design Philosophy and Implementation Strategy
   4.1 System Limitations
      4.1.1 Hardware Configuration
      4.1.2 Operating System/Programming Language
   4.2 Primitives
   4.3 Logical Screens and Plot Files
      4.3.1 World and Screen Coordinate Systems
      4.3.2 Paging System
   4.4 Feasibility Studies
      4.4.1 Hardware/Software Harmony
      4.4.2 Random I/O in UCSD Pascal
      4.4.3 Mapping Pixels to Bits
      4.4.4 Processor Speed
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 [POLE82] are excellent references. The following are helpful and interesting: [ANGE81, GIL078, POSD77, ROGE76, SCOT82, WHIT82].
1.2 Introduction

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
1.3 Introduction

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
A Graphics Tutorial Interactive Vs. Passive

2.1 A Graphics Tutorial Interactive Vs. Passive

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 [POLE82, 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
2.2 A Graphics Tutorial: Vector Vs. Raster

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
2.3.1 A Graphics Tutorial World Coordinates and Windows

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
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 [POLE82, pp. 146-149] for a detailed description of this algorithm. (The Pascal programs in both texts are essentially the same. [POLE82] "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 [POLE82]. 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:
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"
2.3.4 A Graphics Tutorial

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 \(a, b, c, d\) 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 \( \mathbf{T} \) with translation components \( \mathbf{T}_x \) and \( \mathbf{T}_y \), the matrix transformation for translation is

\[
[x' \ y' \ 1] = [x \ y \ 1] \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \mathbf{T}_x & \mathbf{T}_y & 1 \end{bmatrix}
\]

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 \( \mathbf{S} \) with scaling components \( \mathbf{S}_x \) and \( \mathbf{S}_y \), the matrix transformation for scaling is

\[
[x' \ y' \ 1] = [x \ y \ 1] \begin{bmatrix} \mathbf{S}_x & 0 & 0 \\ 0 & \mathbf{S}_y & 0 \\ 0 & 0 & 1 \end{bmatrix}
\]

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 \(\phi\) about the origin of the coordinate system is

\[
[x' \ y' \ 1] = [x \ y \ 1] \begin{bmatrix} \cos \phi & -\sin \phi & 0 \\ \sin \phi & \cos \phi & 0 \\ 0 & 0 & 1 \end{bmatrix}
\]

This transformation matrix can only be used for a rotation about the origin.
2.4.4 A Graphics Tutorial

Rotation

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 $T$ of a point is a translation $-T$ of the coordinate system. A rotation angle $\phi$ of a point is a rotation angle $-\phi$ 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 $\phi$ 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:

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

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 \(4 \times 4\) 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 \(T_x, T_y\) and \(T_z\). The 3D translation matrix is

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
T_x & T_y & T_z & 1
\end{bmatrix}
\]

Given the scaling vector \(S\) with scaling components \(S_x, S_y\) and \(S_z\), the 3D scaling matrix is

\[
\begin{bmatrix}
S_x & 0 & 0 & 0 \\
0 & S_y & 0 & 0 \\
0 & 0 & S_z & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

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 \(\phi\) 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:
Rotation about the x axis is given by:

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

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.
2.5.4 A Graphics Tutorial

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 \( \geq 1 \), 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].
3.1 Examples Two-Dimensional Graphics

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
3.2 Examples  

Three-Dimensional 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.
1. State of Kansas
Demonstrates plotting of unmodified line segments

1.1 Local Counties: Riley, Pottawatomie, et ai
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 3. Perspective Views of KSU Football Field

3.1 Side View (from the east side)
Asimuth = 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
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 = 1.5 units

4.2 Azimuth = 45 degrees, Elevation = 30 degrees, Distance = 1 unit
Exhibit 4. The Surface $Z(X,Y) = XY(X^2 - Y^2)/(X^2 + Y^2)$

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

4.2 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 Design/Implementation System Limitations

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 [NEW79] and are similar to those in [POLE82]. [NEW79] uses the same names for two- and three-dimensional primitives of different dimensionality, e.g., "moveto (x,y)" and "moveto
4.2 Design/Implementation

Primitives

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
4.3 Design/Implementation Logical Screens

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.
4.4 Design/Implementation  Feasibility Studies

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
4.4 Design/Implementation Feasibility Studies

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
5. Future Extensions and Other Applications

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

[MAC82] demonstrates several character sets and special symbols whose coordinates are defined by [WOL76]. 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 WII72]. 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
5. Future Extensions and Other Applications

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
6. Conclusions

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


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


Appendix A. Source Listings of Pascal UNITS

A.1 "global" UNIT
A.2 "dotplotter" UNIT
A.3 "matrixops" UNIT
A.4 "ids560" UNIT
UNIT listings  global UNIT

1 (4L-  PRINTER:)
2 (5S-  Put compiler in swapping mode.)
3 (5R-  Turn range checking on.)
4 UNIT global; (UCSD Pascal, Version II.)
5 (SC Copyright (C) 1982 by Earl F. Glyn, Manhattan, KS.)
6 (Written in January 1982; last modified on 15 February 1982.)
7
8 (The "global" UNIT is used to minimize sharing between the "dotplotter"
9 and other UNITs to provide more symbol table space for compilation
10 of the "dotplotter" UNIT. The TYPE definitions are used extensively in
11 other UNITs and user PROGRAMs.)
12
13 INTERFACE
14
15 CONST
16 radians_per_degree = 1.745329E-2; (3.14159.../180)
17
18 TYPE
19   index = 1..4; (used for "matrix" and "vector" TYPEs)
20   dimension = (two_D,three_D); (two- or three-dimensional TYPE)
21   matrix =   (transformation "matrix")
22   RECORD
23      size : index;
24      matr : ARRAY[index,index] OF REAL
25   END;
26   vector =   ("vector" TYPE used to define points)
27   RECORD
28      size : index;
29      vecr : ARRAY[index] OF REAL
30   END;
31
32 VAR
33   prt : TEXT; (*"prt" is defined here so a UNIT which USES...
34            (another UNIT can compile and reference a common TEXT FILE.)
35
36   FUNCTION defux(x: REAL): REAL;
37   PROCEDURE define_2D_vector (x,y: REAL; VAR u: vector);
38   PROCEDURE define_3D_vector (x,y,z: REAL; VAR u: vector);
39   PROCEDURE transform (u: vector; A: matrix; VAR v: vector);
40
41 IMPLEMENTATION
42
43 CONST
44     (fuzz = 1.0E-6;
45
46   FUNCTION defux (PUBLIC(x: REAL): REAL);
47      (*"defux" is used for comparisons and to avoid propagation of "fuzzy"
48         nearly-zero values. REAL calculations often result in "fuzzy" values.)
49 BEGIN
50     IF ABS(x) < fuzz THEN defux := 0.0
51     ELSE defux := x
52 END (defux);
53
54   PROCEDURE define_2D_vector (PUBLIC(x,y: REAL; VAR u: vector));
55      (*This procedure defines two-dimensional homogeneous coordinates (x,y,1)
56         as a single "vector" data element "u". The "size" of a two-dimensional
57         homogeneous vector is 3.)
58 BEGIN
59     u.size := 3;
60     u.vecr(1) := x;
61     u.vecr(2) := y;
62     u.vecr(3) := 1.0
63 END (define_2D_vector);
64
65   PROCEDURE define_3D_vector (PUBLIC(x,y,z: REAL; VAR u: vector));
66      (*This procedure defines three-dimensional homogeneous coordinates (x,y,z,1)
67         as a single "vector" data element "u". The "size" of a three-dimensional
68         homogeneous vector is 4.)
69 BEGIN
70     u.size := 4;
71     u.vecr[1] := x;
76  u.vecr[2] := y;
77  u.vecr[3] := z;
78  u.vecr[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.vecr[k]*a.mtr[k,i];
97      temp := defun(temp);
98      v.vecr[i] := temp
99    END;
100  u.vecr[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).
C Copyright 1982 by Earl F. Glynk, 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:

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 64-by-64/inch to 75-by-75/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 "dotplotter" 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)
plot : FILE OF PACKED ARRAY1..block_size OF CHAR;
plot_mode : (undefined, either, create, overlay);
plot_name : STRING(123);

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 move_to (u: vector);
PROCEDURE line_to (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 (proj_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 = 80; (dots per inch horizontally) (PT 560)
j_density = 80; (dots per inch vertically) (PT 560)
i_frame_size = 63; (pixels 0..63 horizontally in block frame)
j_frame_size = 63; (pixels 0..63 vertically in block frame)
max_frames = 40; (max number (1..40) of memory frames)
max_i_bkws = 17; (max columns 0..17) in block matrix
max_j_bkws = 31; (max rows 0..31) in block matrix
spacedefault = 1024; (reserve 1K byte of memory for use by user program)
white_byte = 0; (hex '00')

[derived constants:]
max_bkws = 396; ([max_i_bkws+1]*([max_j_bkws+1])
max_i_dots = 1151; ([i_frame_size+1]*([max_i_bkws+1]) - 1
max_j_dots = 1407; ([i_frame_size+1]*([max_j_bkws+1]) - 1
pm_size = 1024; (size of parm prefix to plot file; n * block_size)

TYPE
bfl_no = 0..max_bkws;
frame_no = 1..max_frames;
i_bk_no = 0..max_i_bkws;
i_dots = 0..max_i_dots;
i_frame_no = 0..i_frame_size;
j_bk_no = 0..max_j_bkws;
j_dots = 0..max_j_dots;
j_frame_no = 0..j_frame_size;
memory_frame = PACKED ARRAY1..i_frame_no, j_frame_no) of pixel;

VAR
bfl_index : ARRAY[frame_no] OF bfl_no;
frame_i_bck : ARRAY[frame_no] OF i_bk_no;
frame_i_bck : ARRAY[frame_no] OF j_bk_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))
base : INTEGER; (pointer for MAKE/RELEASE)
frame : INTEGER;
j_save : INTEGER;
nxt_out : frame_no;
BEGIN
CASE 4 OF
  two_D : prm.xform_2D.size := 1;
  three_D : prm.xform_3D.size := 1
END
END (clear_transform);

BEGIN
CASE a.size OF
  3 : prm.xform_2D := a;
  4 : prm.xform_3D := a
END
END (set_transform);

BEGIN
PROCEDURE set_transform (PUBLIC(a: matrix));
("set_transform" establishes a default "two_D" or "three_D" transformation matrix which is used every time "moveto" or "lineto" is invoked. The transformation matrix can be changed any number of times. "clear_transform" removes the effect of the matrix. Alternately, an identity transformation matrix could be specified.)

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

PROCEDURE vw_parm_error (title: STRING; x_min, x_max, y_min, y_max: REAL);
(*This procedure reports definition errors in setting the "view" or
"window".*)
BEGIN
WRITELN (""; title; ");
WRITELN ("'x_min =', x_min:7:2;', x_max =', x_max:7:2;
', y_min =', y_min:7:2;', y_max =', y_max:7:2);";
END (vw_parm_error);

PROCEDURE window (PUBLIC(x_min, x_max, y_min, y_max: REAL));
(*This procedure defines the physical minimum and maximum of the logical
screen in world coordinates. Certain internal constants are defined as
a result of defining the logical screen.*)
BEGIN
IF (x_max < x_min) OR (y_max < y_min)
THEN BEGIN
WRITELN (""; title; ");
vw_parm_error ('Window parameter error(s):', x_min, x_max, y_min, y_max);
ELSE
BEGIN
END (WITH)
END (END (window));

PROCEDURE view (PUBLIC(x_min, x_max, y_min, y_max: REAL));
(*The "view" is intended to be a subset of the "window". This subset is
used for clipping. This definition of a "view" is not the same as used
by Newman and Sproull.*)
BEGIN
IF (x_min < prm.x_first) OR
(x_max > prm.x_last) OR
(y_min < prm.y_first) OR
(y_max > prm.y_last) OR
(x_max < x_min) OR (y_max < y_min)
THEN BEGIN
WRITELN (""; title; ");
vw_parm_error ('View parameter error(s):', x_min, x_max, y_min, y_max);
ELSE
WITH prm DO vw_parm_error ('Window parameters:',
  x_first, x_last, y_first, y_last);
END (EXIT (window));
END (WITH)
ADD xwest := x_min; (*set clipping parameters)
van := 0.5 * (x_east - x_west);
vy := 0.5 * (y_north - y_south)
END (WITH)
END (view);

PROCEDURE size (PUBLIC(x_len, y_len: REAL));
("size" defines the physical size of the logical screen in inches.)
VAR
error: BOOLEAN;
temp: INTEGER;
BEGIN
error := FALSE;
WITH pm DO BEGIN
  temp := ROUND(i_density*x_len);
  IF temp > max_i_dots
    THEN BEGIN
      writeln ('DOTOS:');
      writeln ('Horizontal dimension ',x_len:4:2,' inches too large: ');
      writeln (temp,' ',max_i_dots,' dots.
    error := TRUE
  END (WITH);
  ELSE
    THEN BEGIN
      writeln ('DOTOS:');
      writeln ('Vertical dimension ',y_len:4:2,' inches too large: ');
      writeln (temp,' ',max_i_dots,' dots.
    error := TRUE
  END (WITH);
END (WITH);
window (0.0,1.0, 0.0,1.0)
END (size);

PROCEDURE dot_color (PUBLIC(color: pixel));
("dot_color" sets the color of subsequent pixels traced by "limeo".)
BEGIN
  pm.dot_color := color
END (dot_color);

PROCEDURE fill_color (PUBLIC(color: pixel));
(Diskette blocks are mapped to the logical screen but are not allocated
until actually referenced. The virtual pixels represented by the
unallocated (undefined) blocks are assigned a "fill_color" value when
they are referenced on output to the printer. That "fill_color" is
defined by this procedure which should only be called once for a given
logical screen.)
BEGIN
CASE color OF
  white: pm.fill_color := white_bytes;
  black: pm.fill_color := black_bytes
END (CASE color OF)
END (fill_color);

PROCEDURE begin_dot_plotter_unit (PUBLIC);
(This procedure is used to guarantee certain VARIABLES at least have
acceptable default values before they are referenced. A section of code
using the "dot_plotter" UNIT must be enclosed in "begin_dot_plotter_unit"
and "end_dot_plotter_unit" procedure calls.)
BEGIN
END (begin_dot_plotter_unit);

PROCEDURE begin_dot_plotter_unit (PUBLIC);
BEGIN
IF SIZEOF(memory_frame) () block_size (ideally this check would be)
THEN BEGIN
  writeln ('DOTOS:');
  writeln ('ERROR: Frames must be the same size as I/O blocks.');
  writeln ('A frame is ',.frame_size, x ',.frame_size, bits = ',
  SIZEOF(memory_frame),')
  writeln ('A block contains ',.block_size, '
EXIT (PROGRAM)
END;

IF SIZEOF(prm) () prm_size (this check should be made by the compiler)
THEN BEGIN
WRITE ('DOT07:');
WRITE ('File parameter prefix is ', SIZEOF(prm), ' bytes long.');
WRITE ('Internal variable "prm_size" currently has the value ',
      prm_size, ' and should be adjusted to a multiple of.');
WRITE ('"block_size" (currently ', block_size, ') greater than or ',
       'equal to the size of the parameter prefix.');
EXIT (PROGRAM)
END;

FILLCHAR (prm.prm_char, prm_size, 0); (zero parameter record)
close_printer := TRUE; (close/paginate when exiting graphics mode)
min_space := spacedefault;
plt_mode := undefined;
plt_open_flag := FALSE;
clear_transform (two_D);
clear_transform (three_D);
dot_color (black);
fill_color (white);
set_projection_type (perspective);
size (5.0, 5.0);
REWRITE (prt, 'PRINTER:');
END (begin_dot_plotter_unit);

PROCEDURE end_dot_plotter_unit (PUBLIC);
(This procedure ends a block of code started with the
"begin_dot_plotter_unit". See comments with that procedure.)
BEGIN
IF plt_open_flg THEN close_plot (keep);
IF close_printer THEN BEGIN
PAGE (prt);
CLOSE (prt)
END;
END (end_dot_plotter_unit);

PROCEDURE allocate_frames (max: INTEGER);
("allocate_frames" allocates as many memory frames as possible from
available memory. These frames are used to hold as many diskette blocks
in memory as possible for subsequent manipulation. The user can define
the table "min_space" to reserve memory for other purposes. The
default for "min_space" is set in "begin_dot_plotter_unit" by the constant
"spacedefault" defined in the implementation section of this unit.)
BEGIN
MARK (heap); (set heap pointer to later deallocate frames)
frames_alloc := 0;
WHILE (max < maxavail) min_space+block_size) AND
   AND (frames_alloc < max)
BEGIN
   frames_alloc := succ (frames_alloc);
   NEW (frame_ptr [frames_alloc])
END;
IF frames_alloc = 0 THEN BEGIN
   WRITE ('DOS7:');
   WRITE ('Terminal Error: No memory frames allocated.');
   EXIT (PROGRAM)
END;
next_ent := 1;
frames_inuse := 0 (frames_inuse := frames_alloc := max)
END (allocate_frames);

PROCEDURE open_plot (PUBLIC);
("open_plot" could be incorporated as part of the "begin_dot_plotter_unit"
but it is possible to create/overlay a plot file many times in the
same begin...end_dot_plotter_unit section of code and this procedure is
needed to control such file accesses.

The variable "plt_mode" determines whether this procedure prompts the
user for a file name or if it uses one directly supplied by the user.
The default value for "plt_mode" is "undefined". The user can specify a "plt_mode" of create or overlay and specify a "plt_name" to avoid the interactive prompt.

This procedure initializes the parameter prefix (if "create") or reads the existing prefix (if "overlay"). Memory frames are allocated and some introductory informational messages are displayed.

PROCEDURE init_plot; (LOCAL to open_plot)
VAR
  "open_plot" is too large to compile by itself...
  i : i_blk_no; (...error 253 but it will compile with...)
  j : i_blk_no; (..."init_plot" as a local PROCEDURE.)
  rc: INTEGER;
BEGIN
  TRUE CASE plt_mode OF
    undefined, either:
      REPEAT
        IF plt_mode = undefined
          THEN BEGIN
            WRITELN ('D029:');
            WRITE ('Enter plot file name (Ldisk:name.extension) ');
            WRITELN ('or 'EXIT');
            READLN (plt_name);
            IF (plt_name = 'EXIT') OR (plt_name = 'exit')
              THEN EXIT (PROGRAM);
          END;
        plt_mode := undefined;
      END (undefined);
    reset (plt,plt_name);
    rc := IORESULT;
    IF rc = 0
      THEN plt_mode := overlay
    ELSE
      IF rc = 10 (no such file on volume)
        THEN BEGIN
          REWRITE (plt,plt_name);
          rc := IORESULT;
          IF rc = 0
            THEN plt_mode := create
          END;
        IF rc () 0
          THEN WRITELN ('D010: Error "",rc,"" in opening plot file "","plt_name,""');
    UNTIL plt_mode () undefined;
    create:
      BEGIN
        REWRITE (plt,plt_name);
        rc := IORESULT;
        IF rc () 0
          THEN BEGIN
            WRITELN ('D010:');
            WRITELN ('Error "",rc,"" in opening plot file "","plt_name,""');
          END (create CASE);
      END (create CASE);
    overlay:
      BEGIN
        RESET (plt,plt_name);
        rc := IORESULT;
        IF rc () 0
          THEN BEGIN
            WRITELN ('D010:');
            WRITELN ('Error "",rc,"" in opening plot file "","plt_name,""');
          END (overlay CASE);
      END (overlay CASE);
BEGIN (open_plot)
IF plt_open_flag THEN WRITELN ('DOT12: Request ignored to open already open plot file.');
ELSE BEGIN
init_plot; (*procedure divided -- too big for compiler*)
WRITELN ('Creating/Overlaying' plot file 'plotfile', 'plt_name', ...');
WITH ptr DO
WRITELN ('The plot will be ', x_length:5:2,' inches, ',y_length:5:2,' inches high.');
allocate_frames (max_frames);
WRITELN ('There will be ', frames_alloc:3, ' in-memory block frames.');
plt_open_flag := TRUE
END (ELSE)
END (open_plot);

PROCEDURE close_plot (PUBLIC disp: disposition); (*close_plot is called to force all in-memory frames to be written to disk and to update the parameter prefix. After closing the plot file, either 'open_plot' or 'end_plotter_unit' should follow. This procedure releases the memory used by the in-memory frames.*)
VAR
frame: frame_no;
i: i_bk_no;
j: j_bk_no;
BEGIN
IF plt_open_flag THEN BEGIN
IF frames_idx = 0 THEN WRITELN ('DOT13: Warning: No frames used.');
ELSE BEGIN
FOR frame := 1 TO frames_idx DO BEGIN
seek (plt, blk_index[frame]);
move_left (frame_ptr[frame], plt, block_size);
put (plt);
io_check ('close_plot', 'write');
WITH ptr DO
(blocktable[frame_i_bk[frame], frame_j_bk[frame]] := blk_index[frame]);
END (FOR);
write_parms;
release (heap);
CASK disp DF
heap = close (plt, LOCK);
delete close (plt, FURSE);
END;
plt_open_flag := FALSE
END (THEN)
ELSE WRITELN ('DOT14: Request ignored to close plot file which is not open.');
END (close_plot);

PROCEDURE get_blk (i_blk: i_bk_no;
j_blk: j_bk_no;
VAR frame: frame_no);
(*This procedure guarantees a needed diskette block is in memory for pixel manipulation. However, this procedure should not be called for a block which already exists in memory.*) Given a virtual block address as (i_blk, j_blk) pair, 'get_blk' returns the 'frame' number where the block was placed. FIFO block replacement is used to guarantee the file
is initially sequentially created. The file must be created sequentially
or system problems will occur. After creation, however, random I/O could
be used to read/write any blocks. But since the file could be extended at
any time and the blocks must be added sequentially, FIFO replacement is
used in all cases.)

VAR
 blk_number: blk_no;

BEGIN
 IF frames_inuse = frames_alloc
 THEN BEGIN
 frames_inuse := SUCU(frames_inuse);
 frame := frames_inuse
 END
 ELSE BEGIN
 frame := next_out;
 next_out := SUCU(next_out MOD frames_alloc);
 SEEK (plt.blk_index(frame));
 MOVELEFT (frame_ptr(frame),plt.block_size);
 PUT (plt);
 io_check ('get_blk','write');
 WITH pm DO
 blk_table[frame.i_blk(frame),frame.j_blk(frame)] := blk_index(frame);
 END;
 blk_number := pm.blk_table[i_blk,j_blk];
 IF blk_number = 0
 THEN BEGIN
 FILLCHAR (frame_ptr(frame),block_size,pm.fill_color);
 blk_index(frame) := pm.n_blks;
 pm.n_blks := SUCU(pm.n_blks)
 END
 ELSE BEGIN
 SEEK (plt.blk_number);
 GET (plt);
 io_check ('get_blk','read');
 MOVELEFT (plt.frame_ptr(frame),block_size);
 blk_index(frame) := blk_number
 END;
 pm.blk_table[i_blk,j_blk] := -frame;
 frame.i_blk(frame) := i_blk;
 frame.j_blk(frame) := j_blk
 END (get_blk);

PROCEDURE dot_tag (i: i_dot_no; j: j_dot_no);
 "dot_tag" assigns a pixel its color (and possibly any other desired
attributes). This procedure calculates the virtual address of
the block which should contain the pixel. If the block is not in memory,
"get_blk" is called. Then the address of the pixel within the block is
calculated and the desired "pixel" is assigned the desired attributes.

VAR
 frame: frame_no;
 i_dot: 0...1.frame_size;
 j_dot: 0...1.frame_size;
 i_blk: i_blk_no;
 j_blk: j_blk_no;

BEGIN
 WITH pm DO BEGIN
 i_blk := i DIV (i.frame_size+1);
 j_blk := j DIV (j.frame_size+1);
 IF blk_table[i_blk,j_blk] = 0
 THEN frame := blk_table[i_blk,j_blk]
 ELSE get_blk (i_blk,j_blk,frame);
 END (WITH)

PROCEDURE dot_seg (i1: i_dot_no; j1: j_dot_no;
 i2: i_dot_no; j2: j_dot_no);
 (Given the end points of a line segment in terms of dot indices, this
procedure defines all the intermediate segment pixels. This procedure
recognizes the following special cases: a single point, horizontal or
vertical segments, segments with slopes = 1.0 and segments with slopes
1.0. No code optimization has been attempted in this first release.
Special line-drawing algorithms should eventually be implemented (see
A.2 UNIT listings
dotplotter UNIT -- dot2

745 VAR
746 i : i_dot_no;
747 j : j_dot_no;
748 step : -1..1;
749 slope : REAL;
750 BEGIN
751 IF i = i  
752 THEN IF j1 = j2  
753 THEN dot_tag (i1, j1)  
754 ELSE BEGIN
755 j := j1;
756 IF j1 ) j1  
757 THEN step := 1;
758 ELSE step := -1;
759 REPEAT
760 dot_tag (i1, j1);
761 j := j + step
762 UNTIL (j = j2);
763 dot_tag (i2, j2)
764 END
765 ELSE IF j1 = j2  
766 THEN BEGIN
767 i := i1;
768 IF i2 ) i1  
769 THEN step := 1;
770 ELSE step := -1;
771 REPEAT
772 dot_tag (i, j1);
773 i := i + step;
774 UNTIL (i = i2);
775 dot_tag (i2, j2)
776 END
777 ELSE BEGIN
778 dot_tag (i1, j1);  
779 slopes := (j2-j1)/(/i2-i1);
780 IF ABS(slope) <= 1.00  
781 THEN BEGIN
782 IF i ) i1  
783 THEN step := 1;
784 ELSE step := -1;
785 i := i1 + step;
786 WHILE i ) i2 DO BEGIN
787 j := j1 + ROUND((i-i1)*slope);
788 dot_tag (i, j);  
789 middle dot(s), if any, slope (1.00)
790 i := i + step
791 END
792 END
793 END
794 ELSE BEGIN
795 slopes := 1.00/slope;
796 IF i ) i1  
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);  
803 middle dot(s), if any, slope (1.00)
804 j := j + step
805 END
806 END;
807 END
808 END (dot_seg);
809 PROCEDURE world_to dot (w: vector; VAR i: INTEGER; VAR j: INTEGER);  
810 IF this procedure converts a "vector" into dot indices. A three-
811 dimensional vector is projected into two dimensions automatically.
812 The user-defined world coordinates are used in specifying the "vector",
813 The dot indices are determined by the definition of the logical screen.)
814 VAR
815 v: vector;
816 BEGIN
817 IF v.size = 4
818 THEN project (u,v);
A.2 UNIT listings  
dotplotter UNIT -- dot2  

WITH pm DO BEGIN
  i := ROUND(x_dots * (u.vetr[1] - u_first));
  j := ROUND(y_dots * (u.vetr[2] - y_first));
END
END (world_to_dot);

PROCEDURE clip (u1, u2: vector);
(*"clip" integrates both two- and three-dimensional clipping into a
simple procedure. The input parameters are of the type "vector" and
internally have their dimensionality defined. These clipping algorithms
were adapted from Newman and Sproull, "Principles of Interactive
The Newman and Sproull internal "code" procedure was replaced by a
"region" procedure for better clarity. For example, their code '1001'
is replaced by a set of regions [north west]. See diagram below.

This "clip" procedure had internal procedures "region", "clip_2D" and
"clip_3D" in addition to the code for its definition.

The clipping boundary is defined by the "view" procedure. After a
line segment has been clipped, it is displayed by procedure "dot_reg"
automatically.

TYPE
      regions = (north, south, east, west);
      region_set = SET OF regions;
      VAR
      delta_x, delta_y: REAL;
      i1, i2, j1, j2: INTEGER;
      reg, req1, req2: region_set;
      visible: BOOLEAN;
      u: vector;

PROCEDURE region (u: vector; VAR reg: region_set);
(*This procedure is internal to "clip". "region" defines the regions
a given point is in. If the set of regions is the null set [], then
the point is in the viewing area. If the union of regions from both
points is the empty set, the segment is entirely visible. If the
intersection of regions from two different points is not the empty
set, the segment must lie entirely off the screen.*)
BEGIN
  reg := [ ];
  CASE u.size OF
    3: WITH pm DO BEGIN
      IF u.vetr[1] < u.west THEN reg := [west];
      ELSE IF u.vetr[1] > u_east THEN reg := [east];
      ELSE IF u.vetr[1] < u.south THEN reg := [south];
      ELSE IF u.vetr[1] > u_north THEN reg := [north];
      ELSE reg := reg + [north]
    END (3);
    4: BEGIN
      IF u.vetr[1] < u.west THEN reg := [west];
      ELSE IF u.vetr[1] > u_east THEN reg := [east];
      ELSE IF u.vetr[1] < u.south THEN reg := [south];
      ELSE IF u.vetr[1] > u_north THEN reg := [north];
      ELSE reg := reg + [north]
    END (4)
  END (CASE);
END (region);

PROCEDURE clip_2D;
(*This code is used only for clipping two-dimensional vectors.*)
VAR
  delta_x, delta_y: REAL;
BEGIN
  delta_x := u2.vetr[1] - u1.vetr[1];  (* x2 - x1 *)
delta_y := u1.vptr[2] - u1.vptr[1];  \( \gamma_2 - \gamma_1 \)

WITH prn DO BEG

IF west IN reg  
  THEN define_3D_vector  
  \( (\text{x-component}) \)  
  \( (\text{vector}) \)

ELSE
  IF east IN reg  
  THEN define_3D_vector  
  \( (\text{x-component}) \)  
  \( (\text{vector}) \)

ELSE
  IF south IN reg  
  THEN define_3D_vector  
  \( (\text{x-component}) \)  
  \( (\text{vector}) \)

ELSE
  IF north IN reg  
  THEN define_3D_vector  
  \( (\text{x-component}) \)  
  \( (\text{vector}) \)

END (WITH)

END (clip_2D);

PROCEDURE clip_3D;

(This code is used only for clipping three-dimensional vectors.)

VAR
t, x : REAL;

BEGIN

IF west IN reg  
  THEN BEGIN
    \( (\text{east component}) \)
    \( (\text{vector}) \)
  END;

ELSE

IF east IN reg  
  THEN BEGIN
    \( (\text{east component}) \)
    \( (\text{vector}) \)
  END;

ELSE

IF south IN reg  
  THEN BEGIN
    \( (\text{east component}) \)
    \( (\text{vector}) \)
  END;

ELSE

IF north IN reg  
  THEN BEGIN
    \( (\text{east component}) \)
    \( (\text{vector}) \)
  END;

END (clip_3D);
A.2 UNIT listings  
dotplotter UNIT -- dot2  
53

970  IF  reg = []
971  THEN  reg := reg2;
972  CASE ul.size OF
973      3:  clip_3D;
974      4:  clip_3D
975  END;
976  IF  reg = reg1
977  THEN BEGIN
978      ul := u;
979      region (u,reg1)
980  END;
981  ELSE BEGIN
982      ul := 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 (ul, 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  prn.xform_3D.size = 3
1002            THEN transform (u,prn.xform_3D, u);
1003        4:  IF  prn.xform_3D.size = 4
1004            THEN transform (u,prn.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  prn.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 segment 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  prn.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);
FILE: 4:DOT3.TEX

\(8\text{R- \hspace{1em} Turn range checking off to speed up execution.}\)

\(1043\) PROCEDURE put_plot (PUBLIC(x,y: REAL; copies: INTEGER; border: INTEGER));

\(1044\) (This "put_plot" procedure was written specifically for the Integral Data Systems "Paper Tiger" 560 printer. It is very device dependent.

\(1045\) This procedure takes the logical screen defined as a diskette file

\(1046\) and transfers it to the 560 printer in graphics mode. For efficiency

\(1047\) this procedure must eventually be written in assembler language or

\(1048\) compiled with a native code generator when UCSD Pascal Version IV

\(1049\) becomes available. The execution of this procedure is very slow and it

\(1050\) literally takes hours for plots as large as 8-inches by 8-inches.

\(1051\) The \((x,y)\) parameters to this procedure define the position right of

\(1052\) the left margin (the \(x\)-coordinate) and down from the top margin (the

\(1053\) \(y\)-coordinate) where the upper-left corner of the plot will be positioned

\(1054\) on a page. Any number of copies of a plot can automatically be requested.

\(1055\) Any number of border dots can be specified to form a picture "frame".

\(1056\) This procedure has the following internal procedures: "pgc", "dot_value", and "first", "middle" and "last_printer_scan".

\(1057\) These "printer_scan" procedures seem to be considerable overhead for

\(1058\) implementation of the border dots.

\(1059\) This procedure must be re-written to allow pixels defined to have

\(1060\) more than two colors.

\(1061\) The \"ids560\" UNIT which controls certain functions of the IDS 560

\(1062\) printer could not be used in this procedure due to symbol table

\(1063\) space limitations.)

\(1064\) CONST

\(1065\) bits_per_scan = 6; \hspace{1em} \{escape ASCII control character\}

\(1066\) esc = 17; \hspace{1em} \{escape ASCII control character\}

\(1067\) etx = 3; \hspace{1em} \{enter graphics mode; graphics escape character\}

\(1068\) ff = 12; \hspace{1em} \{form feed\}

\(1069\) gif = 10; \hspace{1em} \{graphics line feed\}

\(1070\) stw = 2; \hspace{1em} \{enter normal printer mode\}

\(1071\) VAR

\(1072\) border_bottom : INTEGER;

\(1073\) border_size : INTEGER;

\(1074\) border_top : INTEGER;

\(1075\) col : INTEGER;

\(1076\) copy : INTEGER;

\(1077\) gc : CHAR;

\(1078\) j : INTEGER;

\(1079\) max_scan : INTEGER;

\(1080\) plbyte : INTEGER;

\(1081\) PACKED RECORD

\(1082\) CASE INTEGER OF

\(1083\) 0: \hspace{1em} \{byte: CHAR\};

\(1084\) 1: \hspace{1em} \{bits: PACKED ARRAY[0..7] OF pixel\}

\(1085\) END;

\(1086\) prior_nulls : INTEGER;

\(1087\) row : INTEGER;

\(1088\) scan : INTEGER;

\(1089\) temp : INTEGER;

\(1090\) PROCEDURE pge (p: CHAR);

\(1091\) ("pgc" - Put Graphic Character) is local to "put_plot". It checks for

\(1092\) the graphic escape character and eliminates the output of blank

\(1093\) graphic lines - except for the line return. WARNING: UCSD Pascal

\(1094\) will not allow hexadecimal \(\text{X}'10'\) to reach printer.)

\(1095\) CONST

\(1096\) etx = 131; \hspace{1em} \{X'8E'\} + etx)

\(1097\) nul = 0; \hspace{1em} \{graphics space\}

\(1098\) nul2 = 128; \hspace{1em} \{X'80' + nul\}

\(1099\) VAR

\(1100\) orde : INTEGER;

\(1101\) BEGIN

\(1102\) orde := ORD(p);

\(1103\) IF (orde = nul) OR (orde = nul2)

\(1104\) THEN prior_nulls := SUCCP(prior_nulls) (buffer nulls, eliminate blank lines)

\(1105\) ELSE BEGIN

\(1106\) WHILE prior_nulls = 0 DO BEGIN


1118        WRITE (prt,CHR(null));
1119        prior_nulls := PRED(prior_nulls)
1120    END;
1121    IF (ordc = etm) OR (ordc = et2)
1122    THEN WRITE (prt,c)  (put etm 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
1129        frame: frame_no;
1130        i_bk:  i_bk_no;
1131        i_dot:  i_frame_size;
1132        j_bk:  j_bk_no;
1133        j_dot:  j_frame_size;
1134    BEGIN
1135    WITH pm DO BEGIN
1136        i_bk := i DIV (i_frame_size + 1);
1137        j_bk := j DIV (j_frame_size + 1);
1138        IF blk_table[i_bk,j_bk] =
1139    THEN
1140            CABF 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_bk,j_bk] = 0
1146    THEN frame := -blk_table[i_bk,j_bk]
1147    ELSE put_bk (i_bk,j_bk,frame);
1148        i_dot := i MOD (i_frame_size+1);
1149        j_dot := j MOD (j_frame_size+1);
1150        dot_value := frame_pix[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'80')
1158    BEGIN
1159    gc := CHR(black_byte);
1160    IF border_top MOD bits_per_scan = 0
1161    THEN BEGIN
1162        FOR col := 0 TO pm.i_length * bits_per_scan DO
1163            pge (go);
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            pge (go);
1169            plotbyte.byte := CHR(white_byte);
1170        FOR j := 0 TO border_top-1 DO
1171            plotbyte.bytes[j] := black;
1172            gc := plotbyte.byte;
1173    FOR col := 0 TO pm.i_length DO BEGIN
1174            plotbyte.byte := gc;
1175        FOR j := 0 TO bits_per_scan - border_top DO
1176            plotbyte.bytes[border_top+j] := dot_value (col,row-j);
1177            pge (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            pge (go);
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'80')
1190    BEGIN
1191    gc := CHR(black_byte);
1192    FOR col := 1 TO border_size DO
BEGIN
FOR col := 0 TO prm_i_length DO BEGIN
plotbyte.byte := CHR(white_byte);
FOR j := 0 TO bits_per_scan DO
plotbyte.bits[j] := dot_value (col, row-j);
pge (plotbyte.byte)
END;
go := CHR(black_byte);
FOR col := 1 TO border_size DO
pge (go);
row := row - bits_per_scan - 1
END (middle_printer_scan);

PROCEDURE last_printer_scan; (LOCAL to "put_plot")
CONST
white_byte = 128; (X'80'; global value is X'00')
VAR
k :
   i_dot_no;
BEGIN
plotbyte.byte := CHR(white_byte);
IF row /= 0 THEN BEGIN
   k := bits_per_scan - row;
   IF k > border_bottom THEN k := border_bottom;
   FOR j := 0 TO row + k DO
      plotbyte.bits[j] := black;
   FOR col := 1 TO border_size DO
      pge (plotbyte.byte);
   FOR j := 0 TO row DO
      plotbyte.bits[j] := white;
go := plotbyte.byte;
   END;
   FOR col := 0 TO prm_i_length DO BEGIN
      plotbyte.byte := go;
      FOR j := 0 TO row DO
         plotbyte.bits[j] := dot_value (col, row-j);
pge (plotbyte.byte)
   END;
   FOR j := 0 TO row DO
      plotbyte.bits[j] := black;
   FOR col := 1 TO border_size DO
      pge (plotbyte.byte);
   border_bottom := border_bottom - k;
   row := -1
END;
ELSE BEGIN
   FOR j := 0 TO border_bottom-1 DO
      plotbyte.bits[j] := black;
go := plotbyte.byte;
   FOR col := 0 TO prm_i_length + 2*border_size DO
      pge (go)
END
END (last_printer_scan);

BEGIN ("put_plot" divided into "first", "middle_and" last_scan" to compile)
IF NOT plt_open_flag THEN BEGIN
   RESET (plt,plt_name);
temp := IORESULT;
IF temp /= 0 THEN BEGIN
   WRITELN ('DOT0:');
   WRITELN ('Error', 'temp', 'in opening plot file', 'plt_name', '...');
   EXIT (PROGRAM)
END;
read_parms;
allocate_frames (2*(max_i_blks+1))
END;
IF border < 0 (ensure border_size non-negative)
THEN border_size := 0
ELSE border_size := border;
max_scan := (prm_i_length + 2*border_size) DIV (bits_per_scan + 1);
temp := ROUND(129*temp); (left margin in 1/120-inch increments)
WRITE (plt,CHR(temp),',',temp,';',temp,';'); (reset left margin)
temp := ROUND(48*temp); (top margin in 1/48-inch increments)
FOR copy := 1 TO copies DO BEGIN;
  border_bottom := border_size;
  border_top := border_size;
  WRITE (prt,CHR(esc),'H','t',temp,'x'); (set top margin)
  row := pm.j_length;
  WRITE (prt,CHR(etx)); (IDs Paper Tiger 560 graphics mode entry)
  FOR scan := 0 TO max_scan DO BEGIN
    prior_nulls := 0;
    IF border_top ) 0
    THEN first_printer_scan
    ELSE
      IF row > bits_per_scan
      THEN middle_printer_scan
      ELSE last_printer_scan;
    WRITE (prt,CHR(elt),CHR(gif)); (graphics line feed: 1/12th inch)
  END (FOR scan);
  WRITE (prt,CHR(ets),CHR(ste)); (exit graphics mode)
  IF copy > copies
  THEN PAGE (prt)
  END (FOR copy);
  WRITE (prt,CHR(esc),'J',0,'x'); (reset left margin to zero)
  IF NOT pit_open_flag
  THEN BEGIN
    RELEASE (heap);
    CLOSE (plot)
  END
  END
END (put_plot);
(SL- PRINTER:\n(SL+ Put compiler in swapping mode.)
(SL+ Turn range checking on.)
UNIT matrixops; 
(UCSF Pascal, Version II.)

(* Copyright (C) 1982 by Earl F. Clynn, Manhattan, KS. 
(Written in January-February 1982; last modified 7 April 1982.)*

(The "matrixops" UNIT contains vector/matrix manipulations for two-
and three-dimensional geometric transformations of points or lines for
subsequent graphical display. This UNIT manipulates only "vector"
and "matrix" TYPEDs as defined in the "global" UNIT. To use these
procedures with a more general matrix TYPED, certain dimensioning and
indexing variables must be modified.

The procedure "transform" in the "global" UNIT should be in this UNIT
but compiler symbol table space would be exceed in the "dotplotter"
UNIT which uses "transform" but no other matrix operations. The "transform"
procedure is a vector-matrix product which is a new vector.)

INTERFACE
USES global;

TYPE
axis = (x_axis,y_axis,z_axis);
coordinates = (cartesian,polar);
rotation = (cw,cw); (cw = clockwise, ccw = counterclockwise)

VAR
indent: INTEGER;

PROCEDURE matrix_identity (d: dimension; VAR a: matrix);
PROCEDURE matrix_inverse (a: matrix; VAR b: matrix; VAR determinant: REAL);
PROCEDURE matrix_multiply (a,b: matrix; VAR c: matrix);
PROCEDURE print_matrix (title: STRING; a: matrix);
PROCEDURE print_vector (title: STRING; v: vector);
PROCEDURE rotate_matrix (d: dimension; axis: axis; angle: REAL;
            direction: rotation; VAR a: matrix);
PROCEDURE scale_matrix (v: vector; VAR a: matrix);
PROCEDURE translate_matrix (v: vector; VAR a: matrix);
PROCEDURE view_transform_matrix (viewtype: coordinates;
            azimuth, elevation, distance: REAL;
            screen_x, screen_y, screen_distance: REAL;
            VAR a: matrix);

IMPLEMENTATION

PROCEDURE matrix_identity (PUBLIC(d: dimension; VAR a: matrix));
("matrix_identity" creates an identity matrix for the specified dimension-
ality. While this procedure can be accessed by a user program, its
purpose was to initialize matrices used by other procedures in this UNIT.)

VAR
i,j,n: index;
BEGIN
CASE d OF
  two_D:  a := 3;
  three_D: a := 4
END;
FOR i := 1 TO a DO
  FOR j := 1 TO a DO
    IF i = j
    THEN a.mtr[i,j] := 1
    ELSE a.mtr[i,j] := 0;
  a.size := a
END (matrix_identity);

PROCEDURE matrix_inverse (PUBLIC(a: matrix; VAR b: matrix; VAR determinant: REAL));
(This procedure inverts a general transformation matrix. The user need
not form an inverse geometric transformation by keeping a product of
the inverses of simple geometric transformations: translations, rotations
and scaling. A determinant of zero indicates no inverse was possible for
a singular matrix.)

VAR
BEGIN WITH a DO BEGIN (The matrix inversion algorithm used here)

FOR i := 1 TO size DO BEGIN (is similar to the "maximunm pivot strategy"

FOR j := 1 TO size DO BEGIN (described in "Applied Numerical Methods"

END; (by Carnahan, Luther and Wilkes, pp. 282-284.)

END; (The particular algorithm used here was)

(presented by the instructor of the CSE III class (Summer 1972 at K-State) whose name)

(I cannot remember.)

END; (I cannot remember.)

IF ABS(modulus) > 0.0

THEN BEGIN

FOR i := 1 TO size DO (This algorithm is particularly well)

IF i_flag[i]

THEN (suited for 'hand calculations'.)

FOR j := 1 TO size DO (if the original matrix elements are)

IF j_flag[j]

THEN (integers, they remain integers throughout)

IF ABS(matrix[i,j]) ABS(pivot)

THEN BEGIN

pivot := matrix[i,j]; (largest value on which to pivot)

i_pivot := i;

j_pivot := j

END;

ELSE BEGIN

pivot_row[n] := i_pivot;

pivot_col[n] := j_pivot;

i_flag[i_pivot] := FALSE;

j_flag[j_pivot] := FALSE;

END;

END;

ELSE BEGIN

IF defus(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;

END;

END;

END; (FOR i)

END;

END; (FOR j)

END; (WITH)

BEGIN (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.matrix[pivot_row[i], pivot_col[j]] := defus(a.matrix[i,j])

END;

END; (matrix_inverse)

PROCEDURE matrix_multiply (PUBLIC(a,b: matrix; VAR c: matrix));

(Compound geometric and 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 multiplicative must be of the same dimension.)

VAR

i, j, k: index;
A.3 UNIT listings

matrixops UNIT

60

temp : REAL;
BEGIN
s.size := a.size;
IF a.size = b.size THEN
FOR i := 1 TO a.size DO
BEGIN
FOR k := 1 TO a.size DO
BEGIN
temp := 0;
FOR k := 1 TO a.size DO
BEGIN
temp := temp + a.mtrix[i,k]*b.mtrix[k,j];
c.mtrix[i,j] := divmod(temp)
END
END
END
ELSE BEGIN
WRITEL ("MAZE: ");
WRITEL ("Ignoring attempt to multiply square matrices of ",
"different dimensions: ", a.size, " and ", b.size, ");
c.size := 1 (signal error by setting dimension of "c" to 1)
END
END (matrix_multiply);

PROCEDURE print_matrix (PUBLIC(title : STRING; a : matrix));
("print_matrix" can be used to print a transformation matrix with a title.)
(Do not that the variable "indent" declared in the INTERFACE of this
procedure has been defined before using either "print_matrix" or
"print_vector".)
VAR
i, j : index;
k : INTEGER;
BEGIN
WRITEL (prt);
WRITEL (prt,title);
FOR i := 1 TO a.size DO
BEGIN
FOR k := 1 TO indent DO
BEGIN
WRITEL (prt, ");
BEGIN
FOR j := 1 TO a.size DO
BEGIN
WRITEL (prt.a.mtrix[i,j]:12:3);
WRITEL (prt)
END
END
END
END (print_matrix);

PROCEDURE print_vector (PUBLIC(title : STRING; u : vector));
("print_vector" lists a 2D or 3D vector (point) along with a title.)
VAR
i : index;
k : INTEGER;
BEGIN
WRITEL (prt);
WRITEL (prt,title);
FOR k := 1 TO indent DO
BEGIN
WRITEL (prt, ");
BEGIN
FOR j := 1 TO u.size DO
BEGIN
WRITEL (prt.u.mtrix[j]:12:5);
WRITEL (prt)
END
END
END (print_vector);

PROCEDURE rotate_matrix (PUBLIC(dimension : # two_D or three_D #)
axis : # x_axis, y_axis or z_axis #)
angle : REAL;
(direction : rotation # cw or ccw #)
VAR a : matrix);
(This procedure defines a matrix for a two- or three-dimensional rotation.
To avoid possible confusion in the sense of the rotation, "cw" for
clockwise or "ccw" for counter-clockwise must always be specified along
with the axis of rotation. Two-dimensional rotations are assumed to
be about the x-axis in the x-y plane.
A rotation about an arbitrary axis can be performed with the following
steps:
(1) Translate the object into a new coordinate system where (x,y,z)
maps into the origin (0,0,0).
(2) Perform appropriate rotations about the x and y axes of the
coordinate system so that the unit vector (a,b,c) is mapped into
the unit vector along the x axis.
(3) Perform the desired rotation about the z-axis of the new
coordinate system.
(4) Apply the inverse of step (2).
(5) Apply the inverse of step (1).

VAR

cosz : REAL;
radians : REAL;
sinz : REAL;
BEGIN
radians := radians_par_degree # angle (degrees);
IF direction = cw THEN cw := -cw;
ELSE cosz := cos(radians);
sinz := sin(radians);
CASE d OF
  two_D:
    BEGIN
      matrix_identity (two_D,a);
      CASE sys OF
        x_axis,y_axis:
          BEGIN
            writeln ("MAT2D: FOR 2D rotation in x-y plane, specify "x_axis");
            BEGIN
              a.mtr[1,1] := cosz;
              a.mtr[2,1] := sinz;
            END;
          END;
        three_D:
          BEGIN
            matrix_identity (three_D,a);
            CASE sys OF
              x_axis:
                BEGIN
                  a.mtr[2,2] := cosz;
                  a.mtr[3,2] := -sinz;
                END;
              y_axis:
                BEGIN
                  a.mtr[1,1] := cosz;
                  a.mtr[3,1] := -sinz;
                END;
              z_axis:
                BEGIN
                  a.mtr[1,1] := cosz;
                  a.mtr[2,1] := sinz;
                END;
            END;
          END;
      END;
    END;
  three_D:
    BEGIN
      matrix_identity (three_D,a);
      CASE sys OF
        x_axis:
          BEGIN
            a.mtr[2,2] := cosz;
            a.mtr[3,2] := -sinz;
          END;
        y_axis:
          BEGIN
            a.mtr[1,1] := cosz;
            a.mtr[3,1] := -sinz;
          END;
        z_axis:
          BEGIN
            a.mtr[1,1] := cosz;
            a.mtr[2,1] := sinz;
          END;
      END;
    END;
  END;
PROCEDURE scale_matrix (PUBLIC(u : vector; VAR a : matrix));
("scale_matrix" accepts a "vector" containing the scaling factors for
each of the dimensions and creates a scaling matrix. The size
of the vector dictates the size of the resulting matrix.)
VAR
  d : dimension;
i : index;
BEGIN
  CASE u.size OF
    2D: ( (2x0) (2x0) )
    5x: ( (5x0 0 0) )
  END;
  BEGIN
    matrix_identity (d,a);
    FOR i := 1 TO u.size-1 DO
      a.mtr[1,1] := u.vet[i];
  END;
END;
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
CASE u.size OF
  3: d := twoD;
  4: d := threeD
END;

matrix_identity (d, a);
FOR i := 1 TO u.size-1 DO
  a_mtrx[i].u.size,i := u.vector[i]
END (translate_matrix);

PROCEDURE view_transform_matrix (PUBLIC(viewtype: coordinates;
  azimuth (for x?), elevation (for y?), distance (for z?): REAL;
  screen_x, screen_y, screen_distance: REAL;
  VAR a: matrix));
(* "view_transform_matrix" creates a transformation matrix for changing
the location of the "eye" 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_distance/screen_y" specifies a narrow aperture
-- a telephoto view. Conversely, a small ratio specifies a large aperture
-- a wide-angle view. This view transform matrix is very useful as the
default three-dimensional transformation matrix. Once set, all points
are automatically transformed.*)
VAR
b : matrix;

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 (threeD, x_axis, 90.0, cw, b);
matrix_multiply (a, b, a);
CASE viewtype OF
cartesian:
  BEGIN
    temporary := SQRT(x) + SQRT(y);
    hypotenuse := SQRT(temporary);
    cosm := -y/hypotenuse;
    sinm := x/hypotenuse;
    matrix_identity (threeD, b);
    b_mtrx(1,1) := cosm
END
b.mtrz[1,1] := cosm;
b.mtrz[3,1] := -sinn;
b.mtrz[1,3] := sinn;
matrix_multiply (a,b, a);
cosm := hypotenuse;
hypotenuse := SQRT(temporary + SQR(z));

cosm := cosm/hypotenuse;
sinn := sin/hypotenuse;
matrix_identity (three_D,b);
b.mtrz[1,2] := cosm;
b.mtrz[3,2] := sinn;
b.mtrz[2,1] := cosm;
b.mtrz[2,3] := sinn;
b.mtrz[3,2] := sinn
END

END polar;
BEGIN
rotates_matrix (three_D,y_axis,-azimuth,cow, b);
matrix_multiply (a,b, a);
rotates_matrix (three_D,x_axis,elevation,cow, b);
END

END (CASE);

matrix_multiply (a,b, a);
define_3D_vector (screen_distance/(0.5*screen_x),
screen_distance/(0.5*screen_y),-1.0, u);
scale_matrix (u, b); (reverse sense of x-axis; screen transformation)
matrix_multiply (a,b, a)
END (view_transform_matrix);
END (matrix_ops UNIT).
INTERFACE

USES global;  {for definition of "ptr" TEST FILE}

TYPE
  control_char = (null, enhanced_mode, normal_mode, graphics_mode,
                  just_on, just_off,  (*"justify"*)
                  fixed_spacing,
                  ht, vt, ff, cr,
                  proportional_spacing,
                  select_printer, deselect_printer,
                  subscript, superscript,
                  pitch10, pitch12, pitch14);

PROCEDURE forms (length, skip: REAL);
PROCEDURE control (code: control_char);
PROCEDURE line (n: INTEGER);
PROCEDURE margins (left, right: REAL);
PROCEDURE position (x, y: REAL);
PROCEDURE tab (n: INTEGER);

IMPLEMENTATION

CONST esc = 27;

PROCEDURE forms (PUBLIC(length, skip: REAL));
("forms" defines the length in inches of the forms being used. A skip
space must also be specified which is greater than zero but less than
the size of the forms. A zero skip space is treated as 1/48-th inch.
Typically, the skip space is 0.5 or 1.0 inch.)

VAR  l, s: INTEGER;
BEGIN
  l := ROUND(48.0*length);
  s := ROUND(48.0*skip);
  IF s = 0
  THEN s := 1;
  WRITE (ptr, CHR(esc),'L','L','L','L-s','L')
END (forms);

PROCEDURE control (PUBLIC(code: control_char));
(The IDS 560 recognizes many special control characters which can be
symbolically selected with this "control" procedure.)
BEGIN
CASE code OF
  null       : WRITE (ptr, CHR(00));
  enhanced_mode       : WRITE (ptr, CHR(01));
  normal_mode       : WRITE (ptr, CHR(02));
  graphics_mode       : WRITE (ptr, CHR(03));
  just_on       : WRITE (ptr, CHR(04));  (*"justify on")
  just_off       : WRITE (ptr, CHR(05));  (*"justify off")
  fixed_spacing       : WRITE (ptr, CHR(06));
  ht       : WRITE (ptr, CHR(09));  (*horizontal tab)
  vt       : WRITE (ptr, CHR(10));  (*line feed)
  ff       : WRITE (ptr, CHR(12));  (*form feed)
  cr       : WRITE (ptr, CHR(13));  (*carriage return)
  proportional_spacing       : WRITE (ptr, CHR(16));
  select_printer       : WRITE (ptr, CHR(17));
  deselect_printer       : WRITE (ptr, CHR(19));
  subscript       : WRITE (ptr, CHR(20));
  superscript       : WRITE (ptr, CHR(23));
  pitch10       : WRITE (ptr, CHR(39));  (*10 characters/inch)
A.4 UNIT listings

PROCEDURE line (PUBLIC(n: INTEGER));
("line" places the print head at line "n" from the top of the form. The
top line is line 0. Reverse paper feeding will occur if necessary.)
BEGIN
  WRITE (prt, CHR(esc), 'M', ',', 'M', ',', 'L', ',')
END (line);

PROCEDURE margins (PUBLIC(left,right: REAL));
("margins" sets the left and right margins of the forms being used.
Default margins at power up are 0.0 to 13.2 inches.)
BEGIN
  WRITE (prt, CHR(esc), 'J', ',', ROUND(120.0*left), ',', ',', ROUND(120.0*right), ',', ',', 'L')
END (margins);

PROCEDURE position (PUBLIC(x,y: REAL));
("position" print head "x" inches right of the left margin and "y"
 inches down from the top of the current form setting. Reverse paper
feeding can occur and care should be taken to avoid binding or jamming
of paper.)
BEGIN
  WRITE (prt, CHR(esc), 'G', ',', ROUND(120.0*x), ',', ',', 'H', ',', ROUND(120.0*y), ',', ',', 'L')
END (position);

PROCEDURE tab (PUBLIC(n: INTEGER));
("tab" does not use the horizontal tab programming available on the
IDS 540 but rather absolute column positioning. "tab" advances the
print head to the n-th column given the current margins and pitch.
The print head can move left or right to the n-th column.)
BEGIN
  WRITE (prt, CHR(esc), 'N', ',', 'M', ',', ',', 'L')
END (tab);

END (ids560 UNIT).
Appendix B. Source Listings of Sample User PROGRAMs

B.1 Cartography
B.2 Football Field
B.3 $z = f(x,y)$
B.4 Pressure Map of Injection/Production Well Field
(SL: PRINTER:)
(IS: 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 3 April 1982.)
USES global, matrixops, ido60, 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_3D_vector (x,y,u);
move_to (u);
READ (ipt,x,y);
WHILE NOT EOF(ipt) DO BEGIN
define_3D_vector (x,y,u);
line_to (u);
n := SUCC(n);
IF n MOD 8 = 0
THEN READLN(ipt);
WHITELN ('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 (1.5,0,start);
control (pitch16);
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 (4.0,0.0, u);
 scale_matrix (u, a);
 define_2D_vector (-1000.0,-1000.0, u);
 translate_matrix (u, b);
 matrix_multiply (u, b);
 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 ',
 'and 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);
[45- PRINTF();
46- (PSF: Put compiler in swapping mode.)
47-]
48- PROGRAM ksu_football_field;
49- (This program produces several views -- orthographic and perspective --
50- of the K-State football field. The field is approximated mathematically
51- as a flat plane -- the actual crown of the field is ignored. The
52- asymmetrical endzone annotations provide a direction reference. "K-STATE"
53- fills the north endzone; "WILDCATS" fills the south endzone. The origin
54- (0,0,0) is at the southwest corner of the south endzone. All points are
55- on the x0 plane, except for the goal posts. Written in January 1980;
56- last modified 7 April 1980.)
57- USES global, matrixops, ido560, dotplotter;
58-
59- VAR
60- a, b : matrix;
61- border : INTEGER;
62- left : REAL;
63- loop : INTEGER;
64- top : REAL;
65- u : vector;
66-
67- PROCEDURE seg (x1,y1, x2,y2: REAL);
68- VAR u: vector;
69- BEGIN
70- define_3D_vector (x1,y1,0,0, u);
71- noveto(u);
72- define_3D_vector (x2,y2,0,0, u);
73- lineto(u)
74- END (seg);
75-
76- PROCEDURE endzone_annotation;
77- BEGIN
78- seg (5, 5, 10, 10); {"south" endzone annotation: WILDCATS}
79- seg (20, 5, 10, 15); {upside-down "H"}
80- seg (20, 15, 10, 15); {upside-down "T"}
81- seg (25, 5, 40, 5); {upside-down "C"}
82- seg (25, 5, 40, 5); {upside-down "K"}
83- seg (30, 2, 30, 8); {upside-down "K"}
84- seg (35, 2, 35, 8); {upside-down "K"}
85- seg (40, 2, 40, 8); {upside-down "K"}
86- seg (45, 2, 45, 8); {upside-down "K"}
87- seg (50, 2, 50, 8); {upside-down "K"}
88- seg (55, 2, 55, 8); {upside-down "K"}
89- seg (60, 2, 60, 8); {upside-down "K"}
90- seg (65, 2, 65, 8); {upside-down "K"}
91- seg (70, 2, 70, 8); {upside-down "K"}
92- seg (75, 2, 75, 8); {upside-down "K"}
93- seg (80, 2, 80, 8); {upside-down "K"}
94- seg (85, 2, 85, 8); {upside-down "K"}
95- seg (90, 2, 90, 8); {upside-down "K"}
96- seg (100, 2, 100, 8); {upside-down "K"}
97- seg (105, 2, 105, 8); {upside-down "K"}
98- seg (110, 2, 110, 8); {upside-down "K"}
99- seg (115, 2, 115, 8); {upside-down "K"}
100- seg (120, 2, 120, 8); {upside-down "K"}
101- seg (125, 2, 125, 8); {upside-down "K"}
102- seg (130, 2, 130, 8); {upside-down "K"}
103- seg (135, 2, 135, 8); {upside-down "K"}
104- seg (140, 2, 140, 8); {upside-down "K"}
105- seg (145, 2, 145, 8); {upside-down "K"}
106- seg (150, 2, 150, 8); {upside-down "K"}
107- seg (155, 2, 155, 8); {upside-down "K"}
108- seg (6,335, 6,335); {"north" endzone annotation: K-STATE}
109- seg (6,335, 12,335); {"K"}
110- seg (10,345, 25,335); {"-"}
111- seg (26,345, 32,345); {"S"}
112- seg (39,335, 50,335); {"S"}
113- seg (58,335, 58,345); {"S"}
114- seg (38,345, 39,345); {"S"}
PROCEDURE number_yard_lines;
VAR
i, j : INTEGER;
feet : REAL;
PROCEDURE number (m : INTEGER; x, y, size : REAL);
VAR
half : REAL;
BEGIN
half := 0.50 * size;
CASE m OF
0: BEGIN
   seq (x, y, x+size, y);
   seq (x+size, y, x+size, y+size);
   seq (x+size, y+size, x, y+size);
   seq (x, y+size, x, y);
END;

1: seq (x, y, x+size, y);

2: BEGIN
   seq (x, y, x, y+size);
   seq (x, y+size, x+half, y);
   seq (x+half, y, x+size, y);
   seq (x+size, y, x+size, y+size)
END;

3: BEGIN
   seq (x, y, x, y+size);
   seq (x, y+size, x+half, y);
   seq (x+half, y, x+size, y);
   seq (x+size, y, x+size, y+size)
END;

4: BEGIN
   seq (x, y, x+size, y);
   seq (x+half, y, x+size, y);
   seq (x+size, y, x+size, y+size)
END;

5: BEGIN
   seq (x, y, x, y+size);
   seq (x, y+size, x+half, y);
   seq (x+half, y, x+size, y);
   seq (x+size, y, x+size, y+size)
END
END (number);
BEGIN
feet := 60;
FOR i := 1 TO 9 DO BEGIN
   IF i < 7 THEN j := 10 - i
   ELSE j := 1;
   number (i, 10, feet + i, 10, feet - i, 0);
   number (0, 10, feet - i + 30, feet + i - 30);
   number (0, 150, feet - i + 30, feet + i - 30);
   feet := feet + 30
END
END
END (number_yard_lines);

PROCEDURE field_definition;
VAR
i, j : INTEGER;
foot: REAL;
BEGIN
  seg (0,0,160,0);  (*south* endzone boundary)
  foot := 30;
  FOR i := 0 TO 19 DO BEGIN
    seg (0,foot,160,foot); (*yard line*)
    IF i = 0 THEN BEGIN (*vertical* hash lines
      seg (45,foot-1.5,45,foot+1.5); (*west*)
      line (115,foot-1.5,115,foot+1.5); (*east*)
    END;
    FOR j := 1 TO 4 DO BEGIN (*horizontal* hash lines between yard lines
      foot := foot + 3;
      seg (45,foot,48,foot); (*west*)
      seg (115,foot,118,foot); (*east*)
    END;
    foot := foot + 3
  END;
  seg (0,330,160,330); (*north* goal line)
  seg (0,360,160,360); (*north* endzone boundary)
  seg (0,0,0,360); (*west* sideline)
  seg (160,0,160,360); (*east* sideline)
  seg (77,321,83,321); (*north* FAT hash line)
  seg (77,39,83,39); (*south* FAT hash line)
endszone_annotation;
number_yard_lines
END (field_definition);

PROCEDURE goal_posts;
VAR
i: INTEGER;
x,y: REAL;
v: vector;
BEGIN
  FOR i := 1 TO 2 DO BEGIN
    CASE i OF
      1: (south goal post)
      BEGIN
        x := 80.0;
        y := 0.0;
        v := -5.0
      END;
      2: (north goal post)
      BEGIN
        x := 80.0;
        y := 360.0;
        v := 5.0
      END;
    END CASE;
  define_3D_vector (x,y,0.0, v);
  moveto (v);
  define_3D_vector (x,y,10.0, v);
  lineto (v);
  define_3D_vector (x,y,10.0, v);
  lineto (v);
  moveto (v);
  define_3D_vector (x-10.0,y,10.0, v);
  lineto (v);
  define_3D_vector (x-10.0,y,10.0, v);
  lineto (v);
  define_3D_vector (x+10.0,y,10.0, v);
  lineto (v);
  define_3D_vector (x+10.0,y,10.0, v);
  lineto (v);
  END (FOR)
END (goal_posts);
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 (0.100,0.0, -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 THEN BEGIN
251          left := 1.50;
252          PAGE (prt);
253          position (1.25, top-4.45);
254          control (pitch16);
255          control (enhanced_mode);
256          WRITE (prt,'Exhibit 3. Perspective Views of KSU Football Field');
257          control (normal_mode);
258          position (1.50,top-0.55);
259          control (pitch16);
260          WRITELN (prt,'3.1 Side View (from the east side)');
261          position (2.00,top-0.425);
262          control (pitch16);
263          WRITELN (prt,'3.1 Side View (from the east side)');
264          border := 2;
265          size (5.00,1.50);
266          clipping (TRUE);
267          set_projection_type (perspective)
268  END;
269
270  PROCEDURE case_2;
271  BEGIN
272      left := 1.50;
273      top := 1.10;
274      PAGE (prt);
275      position (1.25,top-0.60);
276      control (pitch16);
277      control (enhanced_mode);
278      WRITE (prt,'Exhibit 3. Perspective Views of KSU Football Field');
279      control (normal_mode);
280      position (1.50,top-0.30);
281      control (pitch16);
282      WRITELN (prt,'3.2 Corner View (from the southeast corner)');
283      position (2.00,top-0.175);
284      control (pitch16);
285      WRITELN (prt,'3.2 Corner View (from the southeast corner)');
286      border := 2;
287      size (5.00,2.50);
define_3D_vector (-80.0,-100.0,0.0, a);
translate_matrix (u, a); (translate origin to center of field)
view_transform_matrix (polar, 25.0,30.0,75.0, 0.00,2.50,12.0, b);
matrix_multiply (a,b, a);
set_transform (a)
END (case_3);

PROCEDURE case_3;
BEGIN
  top := 4.10;
  position (1.50,top-0.30);
  control (pitch10);
  WRITELN (pnt,'3.3 Endszone View (from the south endzone)');
  position (2.00,top-0.15);
  control (pitch16);
  WRITELN (pnt,'4.3 North = 0 degrees, Elevation = 50 degrees, ');
  'Distance = 300 feet from center of field');
  define_3D_vector (-80.0,-100.0,0.0, u);
  translate_matrix (u, a); (translate origin to center of field)
  view_transform_matrix (polar, 0.0,30.0,300.0, 5.00,2.50,12.0, b);
  matrix_multiply (a,b, a);
  set_transform (a)
END (case_3);

PROCEDURE case_4;
BEGIN
  top := 7.10;
  position (1.30,top-0.30);
  control (pitch10);
  WRITELN (pnt,'3.4 Aerial View');
  position (3.00,top-0.175);
  control (pitch16);
  WRITELN (pnt,'3.4 AAzimuth = 65 degrees, Elevation = 75 degrees, ');
  'Distance = 1500 feet from center of field');
  define_3D_vector (-80.0,-100.0,0.0, u);
  translate_matrix (u, a); (translate origin to center of field)
  view_transform_matrix (polar, 65.0,75.0,1500.0, 5.00,2.50,12.0, b);
  matrix_multiply (a,b, a);
  set_transform (a)
END (case_4);

BEGIN
  begin_plotter_unit;
  forms (11,0.0,0.5);
  plt_mode := create;
  plt_name := '4:football.dots';
  FOR loop := 0 TO 7 DO BEGIN
    CASE loop OF
      0: case_0;
        (Five separate procedures are used to circumvent)
      1,2,3,4: case_1; (the UCSD Pascal Error 253 -- "Procedure Too Long")
      5: case_2;
        (received when case_0..case_4 are placed in a)
      6: case_3;
        (simple "setup_case" procedure.)
      7: case_4
    END (CASE);
  open_plot;
  field_definition;
  goal_posts;
  put_plot (left.top, 1 (copy), border (dots));
  close_plot (delete);
END (FOR);
end_plotter_unit
END (football_field).
B.3 Sample PROGRAM  
\[ z = f(x, y) \]  

Surface

1 (4L- PRINTER:)
2 (JS+ Put compiler swapping mode)
3 PROGRAM surface;
4 (*"surface" produces several perspective plots of a surface described
5 mathematically as \( z = f(x, y) \). Hidden lines are not removed.
6 Written in February 1982; last modified on 0 April 1982)
7 USES global, matrixops, dotplotter, ids560;
8
9 CONST
10 \( m = 40; \)  \( \text{(lines 0..n)} \)
11 xfirst = -2.0;
12 xlast = 2.0;
13 yfirst = -2.0;
14 ylast = 2.0;
15
16 VAR
17 azimuth : REAL;
18 \( a \) : matrix;
19 i, j : 0..n;
20 \( \text{denom} \) : REAL;
21 \( \text{distance} \) : REAL;
22 \( \text{elevation} \) : REAL;
23 \( \text{loop} \) : INTEGER;
24 \( \text{top} \) : REAL;
25 \( u \) : vector;
26 \( x, y \) : REAL;
27 \( \text{xins, yins} \) : REAL;
28 \( \text{xsq, ysq} \) : REAL;
29 \( z \) : \( \text{ARRAY}(0..n, 0..n) \) OF REAL;
30
31 PROCEDURE title;
32 BEG
33 \( \text{PAGE (prt);} \)
34 \( \text{position (1.25, top-0.70);} \)
35 \( \text{control (pitch16);} \)
36 \( \text{control (enhanced_mode);} \)
37 \( \text{WRITE (prt, "Exhibit 4. The Surface } Z(X,Y) = XY(X');} \)
38 \( \text{control (superscript);} \)
39 \( \text{WRITE (prt, 'XY');} \)
40 \( \text{control (subscript);} \)
41 \( \text{WRITE (prt, '-Y');} \)
42 \( \text{control (superscript);} \)
43 \( \text{WRITE (prt, 'Y');} \)
44 \( \text{control (subscript);} \)
45 \( \text{WRITE (prt, 'Y/X');} \)
46 \( \text{control (superscript);} \)
47 \( \text{WRITE (prt, '2');} \)
48 \( \text{control (subscript);} \)
49 \( \text{WRITE (prt, '+'Y');} \)
50 \( \text{control (superscript);} \)
51 \( \text{WRITE (prt, 2');} \)
52 \( \text{control (subscript);} \)
53 \( \text{WRITE (prt, 'Z');} \)
54 \( \text{control (normal_mode);} \)
55 \( \text{control (pitch12);} \)
56 \( \text{position (2.00, top-4.00);} \)
57 \( \text{WRITELN (prt, "Hidden lines are not removed. x=-2..+2; y=-2..+2."}) \)
58 END (title);
59
60 PROCEDURE surface_points;
61 BEG
62 \( \text{zinc} := (xlast-xfirst)/n; \)
63 \( \text{yinc} := (ylast-yfirst)/n; \)
64 \( \text{(title; \qquad \text{(* printing of data matrix suppressed *)}} \)
65 \( \text{control (pitch11);} \)
66 \( \text{WRITELN (prt);} \)
67 \( \text{WRITELN (prt, ' / X');} \)
68 \( \text{WRITELN (prt, ' / Y');} \)
69 \( \text{WRITE (prt, ' Y / ');} \)
70 \( \text{FOR i := 0 TO n \ DO BEGIN} \)
71 \( \quad x := xfirst + zinc*i; \)
72 \( \quad \text{WRITE (prt, x:8:2);} \)
73 \( \text{END;} \)
74 \( \text{WRITELN (prt);} \)
75
FOR j := n DOWNTO 0 DO BEGIN
  y := yfirst + yinc*j;
  ysq := SQR(y);
  (WRITE (prt,y:6:2,' '));
  FOR i := 0 TO n DO BEGIN
    x := xfirst + xinc*i;
    xsq := SQR(x);
    denom := xsq + ysq;
    IF denom = 0.0 THEN z[i,j] := 0.0  (the limit value)
    ELSE z[i,j] := x*y*(xsq-ysq)/denom;
    (WRITE (prt,z[i,j]:8:4))
  END;
  (WRITELN (prt))
END;

BEGIN
FOR i := 0 TO n DO BEGIN
  x := xfirst + xinc*i;
  FOR j := 0 TO n DO BEGIN
    y := yfirst + yinc*j;
    define_3D_vector (x,y,z[i,j], u);
    IF j = 0 THEN move_to (u)
    ELSE line_to (u)
  END;
END;

FOR j := 0 TO n DO BEGIN
  y := yfirst + yinc*j;
  FOR i := 0 TO n DO BEGIN
    x := xfirst + xinc*i;
    define_3D_vector (x,y,z[i,j], u);
    IF i = 0 THEN move_to (u)
    ELSE line_to (u)
  END;
END;

BEGIN
begin_dot_plotter_unit;
min_space := 4646;
forms (11.0,0.0);  
surface_points;
plot_mode := create;
plot_name := '4:surface_dots';
size (4.0,4.0);
FOR loop := 0 TO 3 DO BEGIN
  open_plot;
  CASE loop OF
    0: BEGIN
      azimuth := 45.0;
      elevation := 30.0;
      distance := 15.0;
      top := 1.0;
      title
      END;
    1: BEGIN
      azimuth := 45.0;
      elevation := 30.0;
      distance := 5.0;
      top := top + 4.5
      END;
    2: BEGIN
      azimuth := 45.0;
      elevation := 0.0;
      distance := 15.0;
      top := 1.10;
      title
      END;
    3: BEGIN
      azimuth := 45.0;
      elevation := 0.0;
      distance := 0.0;
      top := 1.00;
      title
      END;
    END;
END;
elevation := 90.0;
distance := 15.0;
top := top + 4.5
END
view_transform_matrix (polar, azimuth, elevation, distance,
4.0, 4.0, 10.0, a);
set_transform (a);
surface_dets;
position (1.50, top-0.2);
control (pitch10);
WRITE (prt,'4.',loop+1,' ');
control (pitch1);
WRITELN (prt,'Azimuth = ',ROUND(azimuth),' degrees,' ,
'Elevation = ',ROUND(elevation),' degrees,' ,
'Distance = ',ROUND(distance),' units');
put_plot (2.00, top, 1 (copy), 2 (border dets));
close_plot (delete)
END;
end_dots_plotter_unit
END.
(5L- PRINTER)  (5L- Put compiler in swapping mode)
PROGRAM pde;
"pde" solves the partial differential equation:
\[ k_x \frac{\partial^2 u}{\partial x^2} + k_y \frac{\partial^2 u}{\partial y^2} = f(x,y) \]
where "k_x" and "k_y" are constants, "Uxx" and "Uyy" are the second partial
derivatives with respect to the spatial coordinates "x" and "y", and
"f(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 f(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

\[ u[i-1,j] - 2u[i,j] + u[i+1,j] / \text{SOR(\text{\text{delta} x}^2)} +
\]
\[ k_y (u[i,j-1] - 2u[i,j] + u[i,j+1]) / \text{SOR(\text{\text{delta} y}^2)} = c(x,y) \]

With "\text{\text{delta} x} = \text{\text{delta} y}" the difference equation reduces to

\[-k_x u[i-1,j] + u[i+1,j] + 2k_x k_y u[i,j] - k_y u[i,j-1] + u[i,j+1] \]
\[ = c(x,y) \]

The difference equations for boundary points is slightly different.
Given the different "\text{\text{u}}" terms, the following shows the coefficients
for the boundary equations:

<table>
<thead>
<tr>
<th>boundary</th>
<th>u(i-1,j)</th>
<th>u(i,j)</th>
<th>u(i+1,j)</th>
<th>u(i,j+1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i=1</td>
<td>0</td>
<td>-k_x</td>
<td>0</td>
<td>-k_y</td>
</tr>
<tr>
<td>i=16</td>
<td>-2k_y</td>
<td>-k_x</td>
<td>-2k_x</td>
<td>0</td>
</tr>
<tr>
<td>j=1</td>
<td>-k_y</td>
<td>0</td>
<td>2k_x</td>
<td>-k_y</td>
</tr>
<tr>
<td>j=16</td>
<td>-k_y</td>
<td>2k_x</td>
<td>-k_y</td>
<td>0</td>
</tr>
</tbody>
</table>

Corner points are yet a more special case. For (i,j)=(1,1), the
equation would be: \(-2k_x k_y u[i,j] - 2k_x k_y u[i+1,j] - 2k_x u[i,j+1] + c(x,y)\).
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 7 April 1982 but was adapted from a homework
assignment completed in the Fall 1975 Numerical Solutions of PDE course.
USES global, ids=56;

CONST
n = 16;
msq = 256;

TYPE
diag5 = ARRAY[1..n] OF REAL;
mode_type = (gauss_seidel, optimal_point_sor);
vector_matrix = RECORD
CASE INTEGER OF
  0: (vcts: ARRAY[1..msq] OF REAL);
  1: (mtrx: ARRAY[1..n, 1..n] OF REAL);
END (vector_matrix RECORD);
ptr_diag5 = Addr5;
ptr_vector_matrix = Addr_vector_matrix;

VAR
a: ptr_diag5;
(System of equations: ax = b)
x: ptr_vector_matrix;
b: ptr_vector_matrix;
epsilon : REAL;
heap : ARRAY[INTEGER, INTEGER, INTEGER, INTEGER] OF REAL;
iterations : INTEGER;
sole : TEXT;
PROCEDURE setup_matrix (k_x, k_y: REAL);

(This procedure defines the matrix "a" and the column vectors "x" and
"b" after allocation in the main section of code.)
VAR
B.4 PROGRAM Injection/Production Well Pressure Map

76 i, j, k: INTEGER;
77 BEGIN
78 kn := 1.0;
79 ky := 1.0;
80 FOR i := 1 TO n DO
81     FOR j := 1 TO n DO BEGIN
82         k := (i-1)*n+j;
83         a[k, k] := -ky;
84         a[k, k] := -kn;
85         a[k, k] := 2.0*(kn+ky);
86         a[k, k] := -kn;
87         a[k, k] := -ky;
88         b[k] := 0.0;
89         END;
90         IF i = 1 THEN BEGIN
91             a[1, k] := 0.0;
92             a[5, k] := -2.0*ky;
93             END;
94             ELSE BEGIN
95                 IF i = 1 THEN BEGIN
97                     a[1, k] := 2.0*ky;
98                     a[5, k] := 0.0;
99                     END;
100             END;
101             IF j = 1 THEN BEGIN
102                 a[k, 1] := 0.0;
103                 a[k, k] := -1.0*kn;
104                 END;
105             ELSE BEGIN
107                 IF j = n THEN BEGIN
109                     a[k, n] := -2.0*kn;
110                     a[n, n] := 0.4;
111                     END;
112             END;
113             END FOR k DO;
114             END FOR i DO;
115             END FOR j DO;

116 PROCEDURE sorS (n, msq: INTEGER;
117     a: ptr_diag5; VAR x: ptr_vector_matrix; b: ptr_vector_matrix;
118     mode: mode_type;
119     VAR iterations: INTEGER; VAR epsilon: REAL);
120     (The system of equations to be solved can be represented simply by a = b.
121     "sorS" solves this system of "msq" equations with a special "a" matrix
122     containing only five non-zero elements. "a" has a tridiagonal structure
123     but also has elements "n" positions left and right of the main diagonal.
124     The structure of a full "a" matrix is
125     
126     e f e f e ....
127     e f e f e ....
128     r f e f r ....
129     r f e f r ....
130     ....
131     e f e f e ....
132     ....
133     ....
134     ....
135     ....
136     ....
137     r f e f r ....
138     r f e f r ....
139     r f e f r ....
140     where each of the "e", "f" and "r" elements are matrices of size "n".
141     The "e" matrix contains all zero elements. For the problem being solved,
142     the structure of each matrix "e" is
143
144     2*(Kn+Kk) -2Kk 0 . . .
145     -Kx 2*(Kn+Kk) -Kx 0 . .
146     0 -Kx 2*(Kn+Kk) -Kx . .
147     0 0 -Kx 2*(Kn+Kk) . .
148     . . . . . . -2Kx 2*(Kn+Kk)
149     and the structure of each matrix "f" is an identity matrix multiplied by -Kx
150     ).
B.4 PROGRAM Injection/Production Well Pressure Map

151
152  -ky  0  0  0
153  0  -ky  0  0
154  0  0  0  -ky
155
156  Note: Coefficients in the "t" matrix for corner grid points are -1*ky
157  instead of -ky.
158
159  The solution "mode" can be either "gauss_seidel" or "optimal_point_sor".
160  On entry "iterations" contains the maximum number of iterations allowed;
161  "epsilon" contains the desired criteria for the maximum residual. On exit
162  "iterations" contains the actual number of loops and "epsilon" contains
163  the maximum residual.
164
165  VAR
166      i    : INTEGER;
167      left  : REAL;
168      max_residual : REAL;
169      nsqnm : INTEGER;
170      nsqml : INTEGER;
171      mpi : INTEGER;
172      old_x : REAL;
173      old_r1_norm : REAL;
174      residual : REAL;
175      right : REAL;
176      r_norm : REAL;
177      r2_norm : REAL;
178      temp : REAL;
179      w  : REAL; 
180              (overrelaxation parameter)
181
182  BEGIN
183      w := 1.00;
184      max_residual := 1.0E10;  { just to get WHILE started }
185      loop := 0;
186      nsqnm := nsq - n;
187      nsqml := nsq - l;
188      mpi := n+1;
189      WHILE (max_residual < epsilon) AND (loop < iterations) DO BEGIN 
190      IF loop MOD 5 = 0 THEN WRITEln (prt, ' , ' , 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. vetr[i];
197      IF i = 1 THEN (first equation) THEN xa. vetr[i] := (ba. vetr[i] - aa[4,i]*xa. vetr[i+1] - 
198          aa[5,i]*xa. vetr[mpi]) / aa[3,i]
199                                    ELSE IF i = nsq THEN (last equation) THEN xa. vetr[i] := (ba. vetr[i] - aa[1,i]*xa. vetr[nsqml] - 
200          aa[2,i]*xa. vetr[nsqml]) / aa[3,i]
201                                    ELSE BEGIN (all equations but first or last) IF i > n THEN left := xa. vetr[i-n] ELSE left := 0.0;
202      IF i < nsq THEN right := xa. vetr[i+n] ELSE right := 0.0;
203      xa. vetr[i] := (ba. vetr[i] - aa[1,i]*left - 
204          aa[2,i]*xa. vetr[i-1] - 
205          aa[4,i]*xa. vetr[i+1] - 
206          aa[5,i]*right) / aa[3,i];
207  END;
208      residual := xa. vetr[i] - old_x;
209      IF ABS(residual) < 1.0E-4 THEN residual := 0.0;  { "defuzz" to avoid underflows }
210      IF mode = optimal_point_sor THEN BEGIN
211      xa. vetr[i] := old_x + w*residual;
212      r_norm := r_norm + SQRT(residual);
213  END;
214      IF ABS(residual) > max_residual THEN max_residual := ABS(residual);
B.4 PROGRAM Injection/Production Well Pressure Map

```
216 END (FOR);
217 IF mode = optimal_point_ser
218 THEN BEGIN
219   r2_nrm := SQRT(r_nrm);
220   IF loop = 1
221   THEN old_r2_nrm := r2_nrm;
222   IF loop MOD 10 = 0  (*Update "w" only every 10 iterations*)
223   THEN BEGIN
224     temp := r2_nrm / old_r2_nrm;  (*and to help numerical stability*
225     w := 2.0 / (1.0 + SQRT(1.0 - SQRT(temp+w-1.0)/((temp+SQRT(w))));
226     END;
227     old_r2_nrm := r2_nrm
228     END
229 END (WHILE);
230 iterations := loop;
231 epsilon := max_residual
232 END (for5);
233
234 PROCEDURE put_solution (u: vector_matrix);
235 (*"put_solution" prints the pressure map but also writes it to a disk file.*)
236 VAR
237 i, j: INTEGER;
238 BEGIN
239   WRITE (sola, 'A:\POEDATA.TEXT');
240   WRITELN (sola,n);
241   control (pitch10);
242   WRITELN (prr, 'Iterations = ', iterations, ', Maximum Residual = ');
243   WRITELN (prr, epsilon);
244   WRITELN (prr);
245   control (pitch16);
246   WRITELN (prr, ' i ');
247   WRITELN (prr, ' j ');
248   WRITELN (prr, '\ ');
249   FOR i := 1 TO n DO
250     WRITE (prr, i, ': ');
251     WRITELN (prr);
252     WRITELN (prr);
253     FOR j := 1 TO n DO BEGIN
254     WRITE (prr, j, ': ');
255     FOR i := 1 TO n DO BEGIN
256     WRITE (prr, m_t[i,j], ', ');  (*system of equations: az=b*
257     END;
258     END;
259     WRITELN (prr);
260     WRITELN (prr);
261     WRITELN (sola, LOCK);
262   END (put_solution);
263   END;
264 PROCEDURE memory (title: STRING);
265 BEGIN
266   WRITELN (prr, 'Memory Available = ', 2*MEMAVAIL, ' bytes.');
267   END (memory);
268 BEGIN (pde)
269   REWRITE (prr, 'PRINTER:' );
270   PAGE (prr);
271   control (pitch10);
272   memory ('Beginning.' );
273   NEW (a);
274   MARK (heap);
275   NEW (a);
276   NEW (b);
277   setup_matrix (2.00,1.00);  (*system of equations: az=b*
278   memory ('After setup.');
279   iterations := 100;
280   epsilon := 1.0E-6;
281   (*The "a", "w" and "b" pointers must be passed to "sort5". There is not*
282   enough memory to pass the arrays as "az", "za" and "bz".*)
283   sort5 (m,nzg, a,za, b,za, optimal_point_ser, iterations, epsilon);
284   RELEASE (heap); (*release "a" and "b" but not the solution vector "z"*)
285   memory ('After release.');
286   put_solution (za)
```
299 (dL = PRINTER:) 
300 (d5 = 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 s : ARRAY[1..14,1..16] OF REAL;
313 azimuth : REAL;
314 distance : REAL;
315 elevation : REAL;

316 PROCEDURE read_data;
317 VAR
318 i, j : INTEGER;
319 so1 : TEXT;
320 BEGIN
321 RESET (so1, 'PDEDATA.TEXT');
322 READLN (so1, n);
323 FOR j := n DOWNTO 1 DO BEGIN
324 FOR i := 1 TO n DO BEGIN
325 READ (so1, s[i, j]);
326 READLN (so1);
327 END;
328 CLOSE (so1);
329 END (read_data);
330
331 PROCEDURE title;
332 BEGIN
333 PAGE (prt);
334 position (1.25, top-4.70);
335 control (pitch14);
336 control (enhanced_mode);
337 WRITE (prt, 'Exhibit 5. ');
338 control (normal_mode);
339 control (pitch12);
340 WRITELN (prt, 'Pressure Map of Area with Injection/Production Wells');
341 position (2.00, top-4.35);
342 WRITE (prt, 'Partial Differential Equation: \kappa');
343 control (subscript);
344 control (superscript);
345 control (subscript);
346 control (subscript);
347 control (superscript);
348 control (subscript);
349 control (subscript);
350 control (subscript);
351 control (subscript);
352 position (2.00, top-4.40);
353 WRITE (prt, 'k');
354 control (subscript);
355 control (superscript);
356 control (subscript);
357 control (superscript);
358 WRITELN (prt, 'c(t,x,y) = constant flow rate');
359 END (title);
360
361 PROCEDURE surface_dots:
362 VAR
363 i, j : INTEGER;
364 u : vector;
365 BEGIN
366 FOR i := 1 TO n DO
367 FOR j := 1 TO n DO BEGIN
368 define_3D_vector (i+0.0, j+0.0, z[i, j], u);
369 IF j = 1
370 THEN moveto (u)
371 ELSE lineto (u)
372 END;
373 FOR j := 1 TO n DO BEGIN
374 FOR i := 1 TO n DO BEGIN
375
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 := 40.0;
390  FOR loo_p := 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 + 0.3
401      END
402    END;
403  define_3D_vector (1.0,1.0,5.0, u);  // enhance w 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,.120, 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 (pr_t,'5.' ,loop+1,' ');
416  control (pitch11);
417  WRITE (pr_t,'Aximuth = ',ROUND(azimuth),' degrees', ');
418  'Elevation = ',ROUND(elevation),' degrees', ';
419  'Distance = ',ROUND(distance),' units';
420  put_plot (2.00,top, 1 topy), 2 (border dots));
421  close_plot (delete);
422 END (FOR);
423 end_dot_plotter_unit
424 END.
Appendix C. System Guide

C.1 Introduction to Pascal UNITS
C.2 "global" UNIT
C.3 "dotplotter" UNIT
C.4 "matrixops" UNIT
C.5 "ids540" UNIT
C.6 File Structure
C.7 Paging System
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 specialised 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
C.1 System Guide

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: prit
FUNCTION: defuzz
PROCEDURES: define_2D_vector, define_3D_vector, transform

The CONSTANT is self-explanatory. The VARIABLE "prit" 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
cau sed by insight but rather was an alternative in reducing the required
symbol table space given only 64K total memory).
C.2 System Guide

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;
    vect: 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 the 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 PROCURCEDes, 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 "space-default" 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 set 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:

```plaintext
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:

```plaintext
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 "matrizops" 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 "matrizops" 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 "pgo", "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 "pgo"
C.3 System Guide  dotplotter UNIT

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 clarify 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 "x_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), "graphios_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_bytes" 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_blnks" and "j_blnks" which are the dimensions of that matrix.
Since "i_blnks" and "j_blnks" are defined with a zero origin, the product
"(i_blnks+1)(j_blnks+1)" must not exceed "max_blnks". Since a block does
not exist until it is actually needed, "max_blnks" 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. "prfile" 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_blnks" and
"j_blnks" 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 550). 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", "vss", "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 "projtype" defines whether projections will be "orthographic" or "perspective". "clip_flag" dictates whether clipping of line 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), 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 a 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.1 Primitives
D.2 Sample Setup
D.3 Messages and Errors
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.

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

VAR
  close_printer: BOOLEAN;  (GLB)
  indent : integer;  (GLB)
  min_space : integer;  (DOT)
  plt_mode : (undefined, either, create, overlay);  (DOT)
  plt_name : string[23];  (DOT)
  ptt : text;  (DOT)
FUNCTION defuzz(z: real): real;  (GLB)
```
These PROCEDURES can be roughly divided into the following categories:

<table>
<thead>
<tr>
<th>Graphic Primitives</th>
<th>Graphic Primitives</th>
<th>Vector/Matrix Primitives</th>
<th>Printer Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin_dot_plotter_unit</td>
<td>dot_color</td>
<td>clear_transform</td>
<td>control</td>
</tr>
<tr>
<td>clipping</td>
<td>move_to</td>
<td>define_2D_vector</td>
<td>forms</td>
</tr>
<tr>
<td>close_plot</td>
<td>size</td>
<td>define_3D_vector</td>
<td>line</td>
</tr>
<tr>
<td>dot_color</td>
<td>view</td>
<td>get_transform</td>
<td>margin</td>
</tr>
<tr>
<td>end_dot_plotter_unit</td>
<td>window</td>
<td>matrix_identity</td>
<td>position</td>
</tr>
<tr>
<td>fill_color</td>
<td></td>
<td>matrix_inverse</td>
<td>tab</td>
</tr>
<tr>
<td>open_plot</td>
<td></td>
<td>matrix_multiply</td>
<td></td>
</tr>
<tr>
<td>put_plot</td>
<td></td>
<td>print_matrix</td>
<td></td>
</tr>
<tr>
<td>scale</td>
<td></td>
<td>print_vector</td>
<td></td>
</tr>
<tr>
<td>transform</td>
<td></td>
<td>project</td>
<td></td>
</tr>
<tr>
<td>turn</td>
<td></td>
<td>rotate_matrix</td>
<td></td>
</tr>
<tr>
<td>view_transform_matrix</td>
<td></td>
<td>scale_matrix</td>
<td></td>
</tr>
<tr>
<td>view_transform_matrix</td>
<td></td>
<td>transform</td>
<td></td>
</tr>
<tr>
<td>window</td>
<td></td>
<td>translate_matrix</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
D.1 User's Guide Graphic Control Primitives

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
two border 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, 0.5)" indicates the plotting area  
will have coordinates from 1.0 unit to 9.0 units along the "x" axis,  
and 0.0 units to 0.5 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_clip_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"  
examples above, both "view (1.0, 9.0, 0.0, 0.5)" and "view (2.0, 5.0,  
0.0, 0.5)" are valid but "view (0.0, 10.0, 0.0, 0.5)" 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 "set_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):

```plaintext
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  
depend 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):
```
D.1 User's Guide

Graphic Primitives

VAR
  u : vector;
  x, y, z : REAL;

define_3D_vector (0.0, 0.0, 0.0, u);
moveo (u);
define_3D_vector (1.0, 0.0, 1.0, u);
lineto (u);
define_3D_vector (1.0, -1.0, 5.0, u);
lineto (u);

The vector passed to "moveo" or "lineto" will be multiplied by the
default transformation matrix for the appropriate dimensionality (if
one exists). "begin dot plots 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

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 different
dimensionalities.

matrix_identity
matrix_inverse
matrix_multiply

Given "matrix" variable "a", "matrix_identity (two_D, a)" defines a 3
by 3 identity 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, c, 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, x_axis, 45.0, ow, a);
  rotate_matrix (three_D, y_axis, 22.0, cow, 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 "x_axis". The rotation is to be 45.0
degrees clockwise ("ow"). 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 ("cow") rotation about the "y_axis".
The sense of the rotation ("ow" or "cow") 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:

```plaintext
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:

```plaintext
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, 1.0, 1.0, 1.0, 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 "x" and "y". 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
glot_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

"orthographic" 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

"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 "plot_mode" and "plot_name" VARIABLES are used to name the plot disk file if the user does not want to be prompted for a name. The default "plot 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 "plot_mode" of "create", "overlay" or "either", the "plot 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.
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, idm560;
CONST ... 
TYPE ... 
VAR 
a: matrix; 
u: vector; 
...
BEGIN 
  REWRITE (prt, 'PRINTER:\'); (if needed before "open_plot" call)
   
   begin_dot_plotter_unit; 
   plot_mods := create; 
   plot_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);  
   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 User's Guide

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", GLE 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 'xxxxxxx' 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.nnn mmm dots.

Internal limits are imposed on the maximum number of pixels either horizontally or vertically. The horizontal limit is based on the widths 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 \( nn \times mm \) 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 overlaid, 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
PROGRAM hexdump;  //UCSD Pascal, Version 11.1

(* "hexdump" prints the blocks of a file in both hexadecimal and ASCII
form (unprintable characters are changed to periods). "hexdump" prints
characters in groups of four bytes similar to the format used on IBM
370-type machines when a memory dump is produced. However, "hexdump"
allows 16, 32 or 64 bytes to be formatted onto lines which are 64, 116 or
210 characters long. An 80-character title is allowed except for the
line which is only 64 characters long. Block numbers and the address
offset are also printed. At present the user is prompted for the
current date.

Given "n" is the number of 4-byte words (IBM 370 word size) to
be formatted per line ("n" must be 4, 8 or 16 for 16, 32 or 64
formatted bytes per line), each formatted line contains the following
fields: 8 spaces for block number and offset address + 2 spaces +
9n-1 hex characters and blank spaces for visual breaks + 1 space
and 4 vertical bar + 4n ASCII characters + 1 vertical bar.)

USES global, ids560;
(The "global" unit defines a "prt" text file.)
(The "control" procedure in "ids560" is used to set the horizontal
and vertical line spacing. The line spacing must be 8 lines/inch
for the 250-character lines. 16.8 pitch must be used; the
other lines can be printed at a user selected pitch of 10, 11 or 16.8.)

VAR
   addr : INTEGER;  //address relative to beginning of file
   blk_count : INTEGER;  //number of blocks dumped
   blk_number : INTEGER;  //block number relative to beginning of file
   blk_read : INTEGER;  //blocks read by BLOCKREAD intrinsic
   buffer : PACKED ARRAY(0..$11) OF CHAR;
   buf_id : INTEGER;
   chr_id : INTEGER;
   chr_string : PACKED ARRAY(0..63) OF CHAR;
   date : STRING(11);  //mm/dd/yy
   dump : FILE;
   file_in : STRING(23);  //xxxxxxxxxxxxxxxxxxxxx
   file_out : STRING(23);
   i, j, n : INTEGER;
   option : STRING(4);
   ordx : INTEGER;  //ORD(x)
   out_type : CHAR;
   page_number : INTEGER;
   page_title : STRING(8);
   pitch : control_char;  //TYPE "control_char" defined in "ids560"
   temp : INTEGER;
   x : CHAR;

FUNCTION hex_digit(index : INTEGER) : CHAR;
   (This function returns a hex character '0'..'9' or 'A'..'F'
given an integer $0..15; a ' ' for integers outside the 0..15 range.)
BEGIN
   IF index IN ($0..9)
   THEN hex_digit := CHR(index + ORD('0'))
   ELSE
      IF index IN ($10..15)
      THEN hex_digit := CHR(index + ORD('A') - 10)
      ELSE hex_digit := ' ';
   END (hex_digit);

PROCEDURE hex_address(addr : INTEGER);
   (Given an unsigned integer 0..65535, this procedure prints the hex
equivalent '0000'..'FFFF'.)
VAR
   addr_temp : INTEGER;
digit : INTEGER;
divisor : INTEGER;
i : 1..4;
BEGIN
addr_temp := addr;
divisor := 4096;
FOR i := 1 TO 4 DO BEGIN
digit := addr_temp DIV divisor;
IF digit = 15 THEN digit := digit + 16;
WRITE (prt.hex_digit(digit));
addr_temp := addr_temp MOD divisor;
divisor := divisor DIV 16
END
END (hex_address);

PROCEDURE top_and_bottom_header;
VAR
i : 0..15;
offset : INTEGER;
BEGIN
WRITE (prt.'BLK ADDR ');
FOR i := 0 TO n-1 DO BEGIN
offset := 4*i;
WRITE (prt.hex_digit(offset DIV 16),
       hex_digit(offset MOD 16),', '
END;
WRITE (prt.' ');
FOR i := 0 TO n-1 DO BEGIN
offset := 4*i;
WRITE (prt.hex_digit(offset DIV 16),
       hex_digit(offset MOD 16),', '
END;
WRITELN (prt);
END (top_and_bottom_header);

PROCEDURE inter_block_break;
VAR
i : 0..155;
BEGIN
FOR i := 1 TO 11+9*n DO
   WRITE (prt.' ');
FOR i := 0 TO n-1 DO
   WRITE (prt.' ');
   WRITELN (prt)
END (inter_block_break);

PROCEDURE space_fill (start,str_length: INTEGER);
   (Pad STRINGS on right with blanks on output instead of right
    justifying the STRING using 'string:length' output format.)
VAR
i: 0..34;
BEGIN
i := SUCC(start);
WHILE i <= str_length DO BEGIN
   WRITE (prt.' ');
i := SUCC(i)
END
END (space_fill);

PROCEDURE heading (blk_count: INTEGER);
VAR
i : 1..34;
BEGIN
IF blk_count MOD (n DIV 2) = 0  (change test if not 8 lines/inch)
THEN BEGIN
   IF blk_count > 0
   THEN BEGIN
      inter_block_break;
top_and_bottom_header;
      page_number := SUCC(page_number);
      IF out_type = 'A'
      THEN PAGE(prt)
END;
WRITE (prt,'Hexadecimal/ASCII Dump of File ',file_in);
space_fill (LENGTH(file_in),13);
IF n = 16

THEN BEGIN
FOR i := 1 TO 34 DO
  WRITE (prt,'.');
WRITE (prt,page_title);
  space_fill (LENGTH(page_title), 40);
  FOR i := 1 to 34 DO
    WRITE (prt,'.');</i
END
ELSE
  IF n = 0
    THEN WRITE (prt,'.','page_title: 40,' .');
  WRITEL (prt, 'date: ', 'Page ', page_number);
WRITEL (prt);
top_and_bottom_header
END;
inter_block_break
END (heading);
BEGIN
PROCEDURE prompts;
REPEAT
  WRITEL ('Enter input filename (or EXIT): ');
READLN (file_in);
  IF file_in = 'EXIT'
    THEN EXIT (hexdump)
      [abort]
RESET (dump,file_in);
temp := 10;RESULT;
  IF temp 0
THEN
BEGIN
  WRITEL ('File ', file_in, ' not found. RC=', temp,'.');</i
  WRITEL
END
UNTIL temp=0;
out_type := ' ';"<i
REPEAT
  WRITEL ('Enter output type (a) PRINTER, (b) CONSOLE, (c) disk: ','a/b/c/EXIT');
READLN (option);
  IF option = 'EXIT'
    THEN EXIT (hexdump);
  IF (option='a') OR (option='b')
    THEN BEGIN
      out_type := 'a';
      file_out := 'PRINTER';
    END
  ELSE
    IF (option='b') OR (option='B')
      THEN BEGIN
        out_type := 'B';
        file_out := 'CONSOLE';
      END
  ELSE IF (option='c') OR (option='C')
      THEN BEGIN
        out_type := 'C';
        WRITEL ('Enter filename (or EXIT): ');"<i
READLN (file_out);
  IF file_out = 'EXIT'
    THEN EXIT (hexdump)
END;
UNTIL out_type = ' ';
IF out_type = 'B'
THEN n := 6
ELSE BEGIN
  n := 0;
REPEAT
  WRITEL ('Enter line size (a) 64, (b) 116 or (c) 220 characters: ','a/b/c/EXIT');
  READLN (option);
  IF option = 'EXIT'
    THEN EXIT (hexdump)
ELSE
IF (option='a') OR (option='A')
THEN n := 4
ELSE
IF (option='b') OR (option='B')
THEN n := 8
ELSE
IF (option='c') OR (option='C')
THEN n := 16
UNTIL n = 0;
END:

IF out_type = 'A'
THEN BEGIN
IF n = 16
 THEN pitch := pitch16 (16.0 pitch)
ELSE BEGIN
 pitch := null;
REPEAT
WHITELN ('Enter pitch (a) 10, (b) 12 or (c) 16.0 characters/inch:',
 'a/b/c/EXIT');
READLN (option);
IF option = 'EXIT'
THEN EXIT (hexdump)
ELSE
IF (option='a') OR (option='A')
THEN pitch := pitch14
ELSE
IF (option='b') OR (option='B')
THEN pitch := pitch11
ELSE
IF (option='c') OR (option='C')
THEN pitch := pitch10
UNTIL pitch () null
END;
END;

IF n > 4
THEN BEGIN
WHITELN ('Enter page title [up to ',5*n, ' characters] (or EXIT):');
IF page_title = 'EXIT'
THEN EXIT (hexdump);
READLN (page_title)
END;

IF out_type () 'B'
THEN BEGIN
WHITELN ('Enter date [mm/dd/yy] (or EXIT): ');
READLN (date);
IF date = 'EXIT'
THEN EXIT (hexdump)
END

BEGIN (hexdump)
prompts;
REWITE (prt, file_out);
IF out_type = 'A'
THEN control (pitch);
addr := 0;
bk_count := 0;
bk_number := 0;
page_number := 1;
blocks_read := BLOCKREAD(dump_buffer,1,bk_number);
WHILE (IORISET=0) AND (blocks_read=1) DO
BEGIN
IF out_type = 'B'
THEN BEGIN
 top_and_bottom_header;
 inter_block_break
END
ELSE heading (blk_count);
buf_size := 0;
FOR i := 0 TO (128 DIV n - 1) DO BEGIN (8, 16 or 32 lines/block)
IF i = 0
THEN WRITE (prt.bnk_number:3,' ',)  
ELSE WRITE (prt,' ');
hex_address (addr);
WRITE (prt,' ');
chr_idx := 0;
FOR j := 0 TO (4*n - 1) DO {16, 32 or 64 bytes/line}
x := buffer[buf_idx];
ordx := ORD(x);
IF ordx IN [32..116] THEN chr_string[chr_idx] := x {printable character}
ELSE chr_string[chr_idx] := '.'; {unprintable character}
IF j MOD 4 = 0 THEN WRITE (prt,' ' ); {space every 4 bytes}
WRITE (prt,hex_digit(ordx DIV 16));
WRITE (prt,hex_digit(ordx MOD 16));
chr_idx := SUC Chr (chr_idx);
buf_idx := SUC Chr(buf_idx)
END (FOR j);
WRITE (prt,' ');
FOR j := 0 TO (4*n - 1) DO 
WRITE (prt,chr_string[j]);
WRITELN (prt,chr_string[]);
WRITELN ('Push "RETURN" to continue; enter "EXIT" to terminate.');
HEADLN (option);
IF option = 'EXIT' THEN EXIT(hexdump)
END (FOR j);
blk_count := SUC Chr(blk_count);
blk_number := SUC Chr(blk_number);
blocks_read := BLOCKREAD (dump,buffer,1,blk_number)
END (WHILE);
IF out_type () = 'B'
THEN BEGIN
inter_block_break;
top_and_bottom_header;
WRITELN ('Push "RETURN" to continue; enter "EXIT" to terminate.');
HEADLN (option);
END (for 1);
END (END FOR 1);
"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) 320 characters: a/b/c/EXIT
4. Enter pitch (a) 10, (b) 12 or (c) 14.8 characters/inch: a/b/c/EXIT
5. Enter page title [up to nn characters] (or EXIT):
6. Enter date [mm/dd/yy] (or EXIT):

Note: Option (4) only applies if option (2a) is selected. An additional prompt for a filename will occur if option (2a) is selected. If (2b) is selected, prompt (3) will default automatically to (a). Option (5) only occurs if (2b) or (3b) 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 00B50210 82012E46 00D73A06 82022EA6 80 80 80 80
0010 00D73A06 82012E46 00D73A06 AD000000 80 80 80 80
0020 20D0000C 181F7A00 1A4C2C1C 00140330 80 80 80 80
0030 20D0000C 181F7A00 2A3A0C00 00001000 80 80 80 80
0040 4A889341 000A6C45 DFB61900 50804192 1E1C1A1E 80 80 80 80
0050 4A889341 000A6C45 7F1B1B63 00130B8C 80 80 80 80
0060 4A889341 13B41403 CD814A09 81AC14A0 80 80 80 80
0070 540A924F 7F110400 00C80000 0A07C601 1E1C1A1E 80 80 80 80
0080 00E91000 12014100 16011000 1A011C00 80 80 80 80
0090 97013040 23055740 D0A1D29C 0049E791 1E1C1A1E 80 80 80 80
00A0 00033500 00140000 0D005F01 01007604 80 80 80 80
00B0 E0842066 0048C206 A05AC7 3807DC07 80 80 80 80
00C0 00000000 00000000 00000000 00000000 80 80 80 80
00D0 00000000 00000000 00000000 00000000 80 80 80 80
00E0 00000000 00000000 00000000 00000000 80 80 80 80
00F0 00000000 00000000 00000000 00000000 80 80 80 80
1000 00000000 00000000 00000000 00000000 80 80 80 80
1010 00000000 00000000 00000000 00000000 80 80 80 80
1020 00000000 00000000 00000000 00000000 80 80 80 80
1030 00000000 00000000 00000000 00000000 80 80 80 80
1040 00000000 00000000 00000000 00000000 80 80 80 80
1050 00000000 00000000 00000000 00000000 80 80 80 80
1060 00000000 00000000 00000000 00000000 80 80 80 80
1070 00000000 00000000 00000000 00000000 80 80 80 80
1080 00000000 00000000 00000000 00000000 80 80 80 80
1090 00000000 00000000 00000000 00000000 80 80 80 80
10A0 00000000 00000000 00000000 00000000 80 80 80 80
10B0 00000000 00000000 00000000 00000000 80 80 80 80
10C0 00000000 00000000 00000000 00000000 80 80 80 80
10D0 00000000 00000000 00000000 00000000 80 80 80 80
10E0 00000000 00000000 00000000 00000000 80 80 80 80
10F0 00000000 00000000 00000000 00000000 80 80 80 80
```

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.