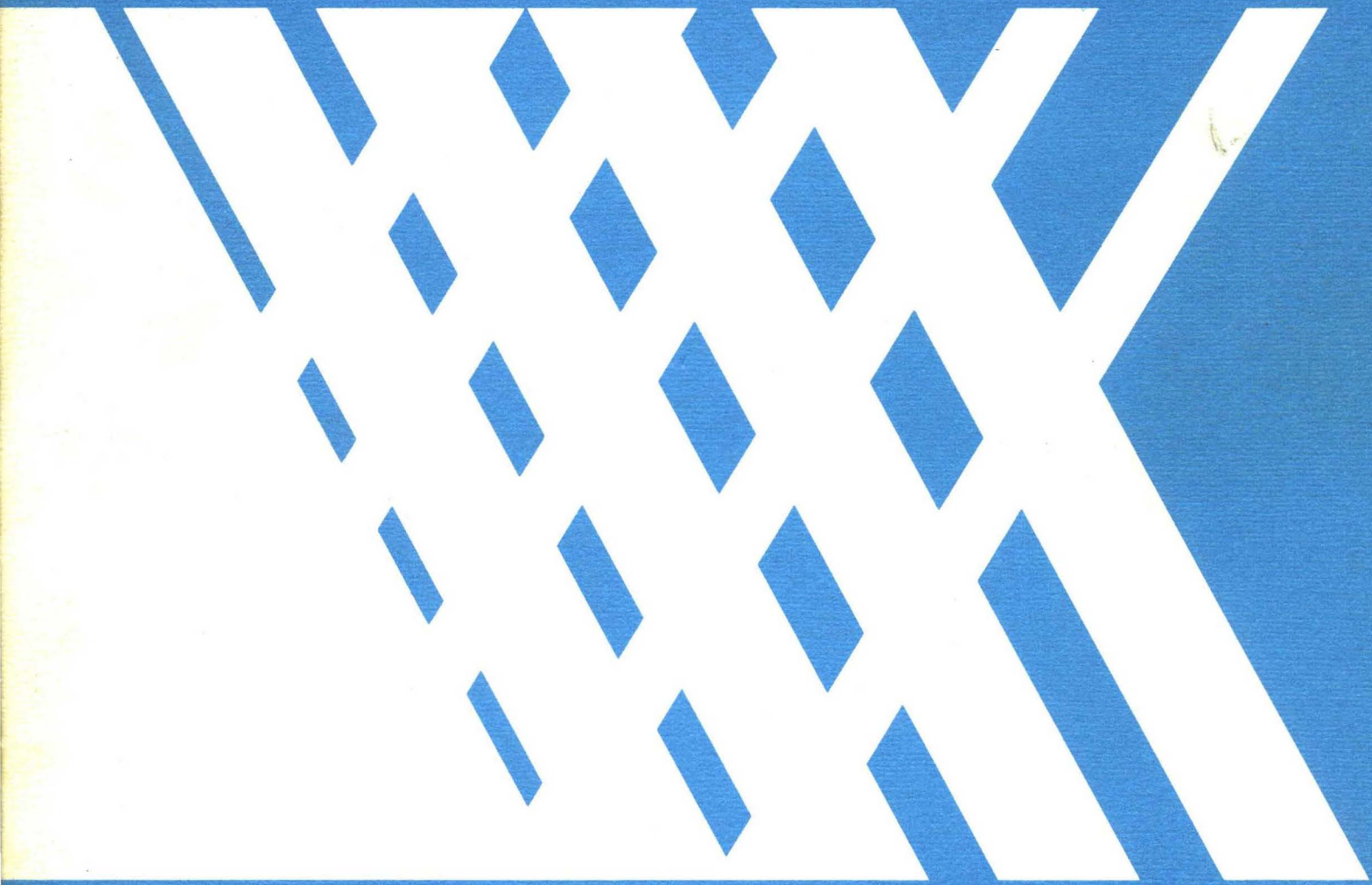
OMNIGRAPH: SIMPLE TERMINAL-INDEPENDENT GRAPHICS SOFTWARE

BY ROBERT F. SPROULL



XEROX
PALO ALTO RESEARCH CENTER

OMNIGRAPH: SIMPLE TERMINAL-INDEPENDENT GRAPHICS SOFTWARE

BY ROBERT F. SPROULL

CSL 73-4 DECEMBER 10, 1973

This paper describes a graphics subroutine package for driving a number of different display devices with any of three different programming languages. The Omnigraah system is designed for routine graphics applications, not for high-performance terminals. The success of the design is largely due to the modest aims of the routines and to the particularly simple framework chosen for the graphics facilities.

The paper cites a number of design errors in the initial Omnigraph routines, and suggests improvements. The Omnigraph Reference Manual is reprinted as an appendix.

XEROX
PALO ALTO RESEARCH CENTER
3180 PORTER DRIVE/PALO ALTO/CALIFORNIA 94304

I. INTRODUCTION

The rapid development of low cost graphics terminals has created a new customer for computer services who wants to view simple drawings or graphs resulting from computer calculations. He is not an accomplished graphics programmer and has probably never heard of Sketchpad; he attaches his terminal to whatever computer resources can be found, preferably timesharing services; his favorite programming language, if he is a programmer, is doubtless FORTRAN. In short, this new user is neither prepared to undertake construction of a graphics programming system nor willing to devote time to issues tangential to his application.

Such a user often relies on "graphics subroutine packages" provided by terminal manufacturers. This software presents special disadvantages both to the user and to the computer facility which must support it.

The user suffers because the software is often poorly designed and documented. None of the software aids in minimizing the number of annoying and time-consuming screen erasures on storage-tube terminals; it often fails to provide even rudimentary graphical operators such as coordinate transformation and windowing; and furthermore, it is frequently riddled with idiosyncratic features, such as curve-drawing capabilities, that are unique to the terminal. Thus, in many ways the design of the software needlessly obscures the basic simplicity of creating drawings.

Even when subroutine packages are thoughtfully designed, they are still troublesome to the computer facility. The staff must support many different subroutine sets, one for each different terminal or different programming language desired by users. Users of different kinds of terminals cannot share programs; programs written for one terminal require alterations to use another. Similarly, the computer facility is hampered in providing graphics services, such as graph-plotting, on many different terminals because a separate program is required for each terminal. For the same reasons, re-programming efforts are required whenever a user decides to discard one terminal and rent another different one.

The Omnigraph display routines, implemented on a PDP-10 timesharing system at the Computer Center of the National Institutes of Health, are designed to solve these problems. They provide modest graphical services for several terminal devices, and can be used with three programming languages (SAIL, LISP 1.6, FORTRAN). From the user's viewpoint, the Omnigraph routines are almost terminal-independent: the programmer is not concerned with the intricacies of driving particular terminals.

The Omnigraph routines drive specific terminals from information provided by the user's terminal-independent subroutine calls. Presently, a DEC 340

refresh display, and Ards, Tektronix 4010 and Computek 400 storage-tube terminals are supported; programs for Adage AGT-30, IMLAC PDS-1 and DEC GT-40 displays are being designed. Many portions of the Omnigraph software are identical for all terminals; less than 20% is actually terminal-dependent.

The routines are deliberately designed to be a less-than-high-performance graphical programming system. They are thus matched to the moderate abilities of low-cost graphics terminals and to the modest graphical tasks undertaken by typical users.

II. OUTLINE OF OMNIGRAPH ROUTINES

The paragraphs below briefly describe the facilities of the Omnigraph routines; more detail is presented in tutorial, reference, and implementation manuals [1]. Several of the concepts presented here are mandatory for a graphics system of this kind; others are necessary to establish a common vocabulary for all terminals.

Classification of the system. The Omnigraph system can best be characterized in terms of the model of a general-purpose graphics system presented in [2] (page 388; also see Figure 1). Omnigraph omits the structured picture definition; the Omnigraph facilities are used to create a 'transformed segmented display file.' All coordinate transformations are performed prior to generation of the display file. The file is used to refresh the display or (in the case of storage-tubes) to send appropriate commands to the terminal in order to generate the display.

Terminal selection. Selection of the display terminal to be driven is delayed until program-execution time. An initialization call loads the device-dependent portion of the Omnigraph routines for the specified display into a reserved space: all subsequent subroutine calls are dispatched to these routines. This feature allows one execution module to be used with any terminal supported by the Omnigraph routines.

The DINI call only loads device-dependent routines; the calls DGET and DREL are used to sieze or relase the display hardware. This permits the program to relinquish the display to others when the user does not need to see the picture, such as during a long computation. All Omnigraph calls still operate correctly, building or modifying pieces of the display file. When the display is next seized, the results of the computation may be viewed.

Segmented display file. A display file of some sort is necessary even for the modest aims of the Omnigraph routines: for refresh displays, the display file is executed by a display processor in order to show the picture; for storage-tube displays, the display file is used to repaint portions of a picture meant to remain after a screen erasure. For example, suppose the display file used to generate Figure 2 were composed of two segments: segment 1 contains a list of vectors required to show the curve, and segment 2 a list to show the coordinate axes. If the user program requests that segment 1 be destroyed, the Omnigraph routines must erase the entire storage-tube screen and then repaint the vectors in segment 2. The display file is thus needed for repainting. (For terminals with selective erasure capabilities, the display file for segment 1 is used to transmit to the terminal a list of vectors to erase.)

The Omnigraph calls for handling segments of the display file are:

DOPEN (n)

Initialize segment n. If a segment numbered n already exists, it is not yet destroyed (the routines automatically double-buffer). All subsequent graphical primitives are added to the new segment.

DCLOSE

This call terminates generation of the currently-open segment.

DKILL (n)

Delete segment n and reclaim the space it occupies.

DPOST (n)

Cause segment n to be displayed. (If segment n is currently open, it is first DCLOSEd.)

DUNPOST (n)

Remove segment n from the set of segments being displayed, but do not destroy it. (When a segment is DKILLed, it is first DUNPOSTed.)

Implementing the display file requires providing space for its storage within the programming language environment; this difficulty may well explain manufacturers' reluctance to provide display files in their subroutine packages. The Omnigraph routines find space in one of two ways: the user can specify in the DINI call a fixed-size array for display-file use, or he may ask that the second segment facility of the PDP-10 operating system be used. The second approach has the advantage that the space can be dynamically expanded to accomodate display files of unexpectedly large sizes.

Erasure Minimization. A general strategy is required to minimize the number of times a storage-tube screen is fully erased. One can identify synchronization points in a program driving such a display -- these are points at which the picture currently visible must accurately show changes requested by the program. A synchronization point is specified with the DDONE call. For example, an interactive program requires synchronization points just before demanding user inputs -- at these points, the user must see an up-to-date picture.

Transmission to a storage-tube terminal occurs only at synchronization points. At these points the Omnigraph routines decide how to make the necessary changes in the visible picture with, at most, one full-screen erasure. If zero or more segments have been DPOSTed since the last synchronization, but no segments have been DUNPOSTed (or DKILLed), the Omnigraph routines avoid erasure altogether, and merely transmit any new segments to the terminal. Otherwise, the screen must be erased, and all currently DPOSTed segments transmitted to the terminal.

Graphical primitives. Subroutine calls are provided for adding vectors, dots and character strings to the open segment of the display file:

DMOVE (x,y)
Move beam to (x,y)

DDRAW (x,y)
Draw a vector from present beam location to (x,y)

DDOT (x,y)
Put a dot at (x,y).

DVECT (x1,y1,x2,y2)
Same as DMOVE(x1,y1); DDRAW(x2,y2)

DTEXT (string)
Display text string starting at present beam location.

The Omnigraph routines apply a coordinate transformation, a windowing operator, and a viewport transformation to each line, dot or character position specified. The coordinate transformation is accomplished by forming the vector [x y 0 1] and multiplying by a 4x4 transformation matrix. This permits any combination of rotations, scalings, and translation to be applied to values passed to the above routines. The windowing operator uses a clipping algorithm to exclude all portions of graphical items outside a specific rectangular window. (Characters that lie wholly or partly outside the window may be excluded as well.) Finally, a viewport transformation is applied to position an image of the rectangular window somewhere on the display screen. These transformations permit the user's coordinate system to be independent of the display terminal coordinate system.

A 'normal viewport coordinate system' is necessary to describe uniformly viewports for all displays. Viewport specifications in terms of inches or display resolution units are not sufficiently terminal-independent. The solution is demonstrated in Figure 3: a unit square is defined to be the largest square that will fit on the screen with its lower left-hand corner coinciding with that of the screen. The casual programmer who specifies viewport limits of x=0, x=1, y=0 and y=1 will thus see a square picture as large as the terminal can draw.

The transformations are controlled as follows:

DAPPLY (array)

Set the transformation matrix from the values of the array. The transformation matrix is initially defaulted to the identity matrix.

DCOMPOSE (array)

Multiply the matrix specified by the array by the current transformation matrix, and replace the current transformation matrix with the result. This is used for concatenating transformations.

DPUSH

Push the contents of the current transformation matrix onto a matrix stack.

DPOP

Pop the matrix on top of the stack into the current transformation matrix.

DWIND (left,right,bottom,top)

Specify the edges of the clipping window. These edges are used for the clipping operation until set again. Initially the window limits are set as if DWIND(-1,1,-1,1) had been executed.

DPORT (left,right,bottom,top)

Specify the edges of the viewport in the 'normal viewport coordinate system.' Initially, the viewport limits are set as if DPORT(0,1,0,1) had been executed.

The Omnigraph routines use hardware character generators to display text strings, although the generators vary greatly among terminals. The occasional user who wishes to position characters carefully may select among character sizes available on the terminal with the DTSCAL call; the routines then provide accurate information about the size of a character on the particular terminal he is using (see Figure 4). (A frustrating idiosyncracy of one terminal is that these measurements are not commensurate with the coordinate system used to draw vectors and dots!)

Three-dimensional facilities. The Omnigraph routines also aid generation of perspective views of three-dimensional scenes. The calls DMOVE3, DDRAW3, DDOT3 and DVECT3 are three-dimensional counterparts of the primitives listed above; the 4x4 transformation matrix permits transformations of three-dimensional coordinates; a three-dimensional windowing algorithm is provided; and a perspective division by depth is performed before the viewport transformation is applied.

Environment inquiry. An "inquiry" subroutine DENQ informs the calling program of many technical details of the terminal: whether the terminal is

a storage-tube or refresh display, what input devices are available, the actual screen size, maximum permissible values of the viewport coordinates, the four dimensions shown in Figure 4 for the currently-selected character size, current window and viewport limits, and various other information. Knowing properties of the terminal permits the application program to select interaction protocols that are appropriate for the terminal. For example, typewritten commands can be used if no input devices are available, or frequently changing images can be avoided on storage tubes (e.g. rotations by small increments).

Accessing special device facilities. Some users are relatively uninterested in device independence and are concerned with using special features of a particlar terminal (e.g. the DEC340 display can display gray-scale images in a special raster-scan mode). To access these facilities, a special terminal-dependent call DCODE is provided that will add specific codes to the display file or send specific control information to the display terminal.

Hard Copies. The Omnigraph routine DPLOT copies the current display file into a disk file that can be used later to drive an off-line plotter or microfilm device. The off-line nature of these plots simplifies adding special effects: overlays, multi-color plots, enlargements, identifying or accounting information, etc. can all be added by the program that translates the display file into plotter commands.

Input Devices. The Omnigraph routines attempt to provide for two-dimensional input information and for common feedback mechanisms such as tracking and inking. The routine DEVENT enables input devices and waits for input "events" to occur. Examples of events are depressing a function button, raising a tablet stylus after it has been depressed, and moving the stylus slightly while it is depressed (for collecting tablet strokes). The routine DOUT controls inking and lights under function buttons. These facilities are only moderately successful -- extremely low bandwidth to the storage-tube terminals necessitates greater device dependencies than are desirable. These are in part due to the rather erratic design of low-cost input hardware.

Errors. There are no fatal errors in the Omnigraph subroutines. Avoiding fatal errors significantly complicates programming the routines, but is absolutely essential for interactive systems -- a user who has invested several hours in a session with the application program cannot afford to lose that effort as a result of some unexpected condition in the graphics routines. The most significant such event is exceeding storage space available for the segmented display file. If, during creation of a segment, the Omnigraph routines exhaust available space, the segment being generated is aborted, a brief message is typed out, and further segment-

generation is inhibited until the next DOPEN command. In an interactive situtation, it is often possible for the user to simplify the displayed picture, thereby deleting segments and then to request that the aborted segment be regenerated.

III. DISCUSSION

The construction and user-acceptance of the Omnigraph routines were remarkably smooth. The eccentricities of character generators and input hardware described above are among the few frustrations. Wide use of the routines is due largely to their incorporation in MLAB [3], an interactive curve-fitting program with rather sophisticated graph-plotting and graphical facilities. Many MLAB users have no graphics or programming experience, but are nevertheless able to make intelligent use of their terminals. The atmosphere of a unifed graphics facility is similar to that created by GINO [4].

The easy construction of these routines is a result of the modest graphical power of the design and the relatively uniform "universe" of displays and users served. A clever programmer would probably be frustrated by the conservative design of the Omnigraph routines. Similarly, exceptional terminal hardware would strain the terminal-independent goal.

The achievement of reasonable device independence in this system is due predominantly toothe simplicity of a "segmented transformed display file." Since the transformations are performed in software, many device dependencies can be included in the transformations as parameters (e.g. screen coordinate addressing and character sizes). The routines for building the display file for a refresh display must of course observe the order code of the specified device. However, storage-tube displays can all use one display file format; the only device-dependent code is about 100 instructions for encoding segments for transmission to the terminal and for some special character processing.

As with any significant software system, hindsight uncovers design flaws of major and minor importance. A minor improvement could be made by supplying utility routines for creating transformation matrices from specification of translation, rotation and scaling. Two major errors in the design of the Omnigraph routines as described in [1] resulted from a greedy attempt to exploit particular device features:

- The input routines, patterned after a high-performance graphics terminal, are too device-dependent. Storage tube terminals with crude input devices (e.g. crosshairs manipulated with thumbwheels) cannot match the 'stylus' and 'interrupt' paradigms of the input routines. Instead, it would be better to provide input routines that attempt to provide an interaction sequence (e.g. pointing, positioning, inking) on the terminal. DENQ could report what interaction techniques can reasonably be used on the specific terminal.

- The dynamic transformation capability of the Adage AGT-30 induced an addition to Omnigraph to permit specifying dynamic transformations. This concept is counter to the Omnigraph philosophy of maintaining transformed display files. The resulting mechanism is in no sense device-independent. Perhaps a compromise would be to allow DCODE to send dynamic transformation matrices to the Adage that could be applied to individual segments of the display file.

The Omnigraph approach offers an attractive alternative to efforts toward display hardware standardization. The cost of construction of the Omnigraph system is substantially less than that of devising and imposing a hardware standard: the Omnigraph system demonstrates that a small amount of software can achieve considerable device independence even among widely varied displays. Wide acceptance of this philosophy would encourage writing programs for use on many different terminals. It would encourage manufacturers to offer more useful software with their terminals. It has implications for design of network protocols for transmitting graphical information and for controlling remote displays. Although it has been proved only on unpretentious graphical applications, the Omnigraph design caters to precisely the modest graphical tasks and low to medium cost display terminals so prevalent today.

ACKNOWLEDGEMENTS

The Omnigraph routines were developed while the author was at the Divison of Computer Research and Technology, National Institutes of Health. Gary Knott, of that group, offered many helpful suggestions on the design and documentation of the Omnigraph routines.

REFERENCES

[1] 'PDP-10 Display Systems,' available from Computer Center Branch, Division of Computer Research and Technology, National Institutes of Health, Bethesda, Maryland 20014. (The Reference Manual is reprinted as a supplement to this report.)

[2] Newman, W.M. and Sproull, R.F., *Principles of Interactive Computer Graphics*, McGraw Hill, 1973.

[3] Knott, G.D. and Reece, D.K., 'Modelab: A civilized curve-fitting system,' *Proceedings ONLINE 72*, Uxbridge, England, September 1972.

[4] Woodsford, P.A., 'The Design and Implementation of the GINO 3D Graphics Software Package,' *Software Practice and Experience*, 1, 4, 335 (October 1971).

APPENDIX -- ERASURE MINIMIZATION FOR STORAGE TUBES

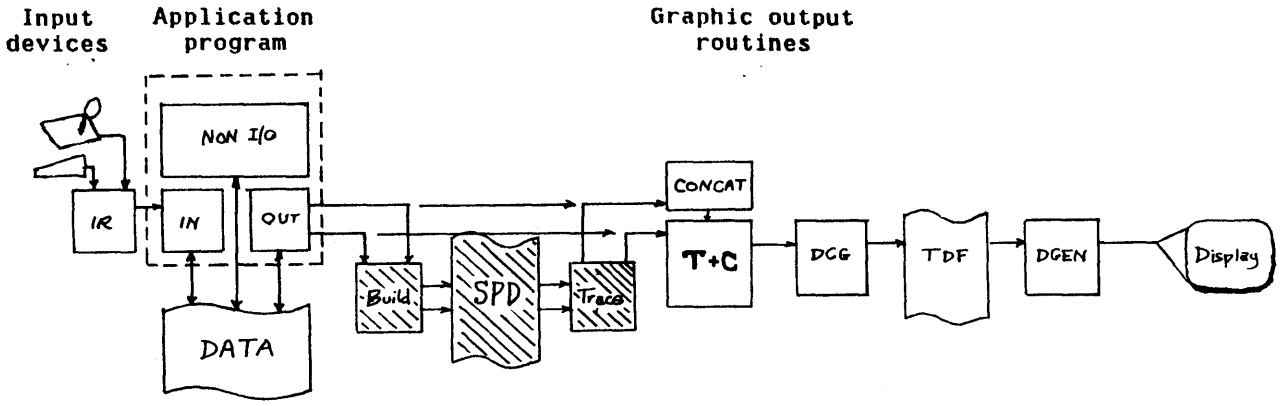
This appendix presents the algorithm used to manage the display file for storage tubes in such a way as to minimize erasures. Each segment of the display file is in one of 5 states, shown in the following table:

	POSTED	UNPOSTED	KILLED
Painted	PP	UP	KP
Unpainted	PU	UU	(free storage)

The table shows the effect of combining the 'logical' status of the segment (POSTed, UNPOSTed, KILLed) with the actual state of the segment on the display screen (if an image of the segment is on the screen, it is 'painted').
When a new segment is generated, it is added to the UU list. Then, if the DPOST call is issued for it, the segment is moved from UU to PU. Thus, the calls for manipulating segments can only cause changes in the horizontal direction in the table; the calls do not directly change the image on the screen.

When the DDONE (synchronization) call is issued, the screen must be updated. If either UP or KP has some segments on it, then a currently-painted segment has been unposted, and a screen erasure will be required. After the erasure (perhaps using selective erasure), those segments on KP can be returned to free storage, and those on UP can be moved to UU. If a full-screen erasure was required, the segments in PP must be retransmitted to the terminal.

Even if no erasure is required, we must consider what new information, if any, must be added to the display. The list PU contains newly created segments that are not yet painted. These are sent to the terminal, and then the PU segments are transferred to PP. Thus, when the DDONE call terminates, segments remain either in PP (posted and painted segments) or UU (unposted and unpainted segments). The other lists are empty.



IR	Interrupt routines
IN	Input routines (application program)
OUT	Output routines (application program)
NON I/O	Non I/O routines (application program)
BUILD	Routines to build SPD
SPD	Structured picture definition
TRACE	Routines to trace SPD
CONCAT	Concatenation routine
T+C	Transformation and clipping routines
DCG	Display code generator
TDF	Transformed display file
DGEN	Display generator

Figure 1

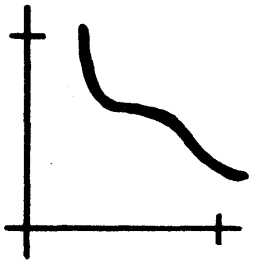


Figure 2

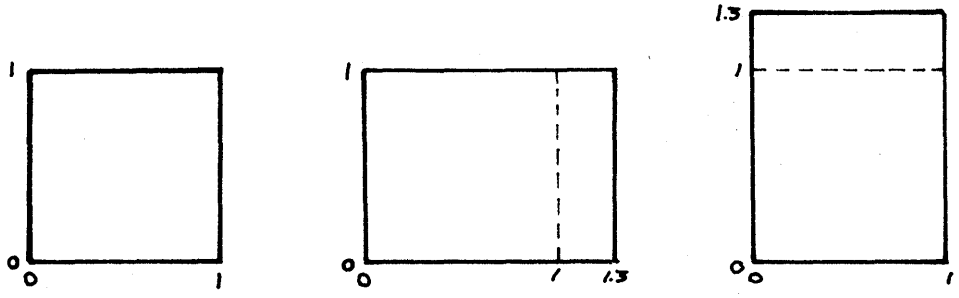


Figure 3

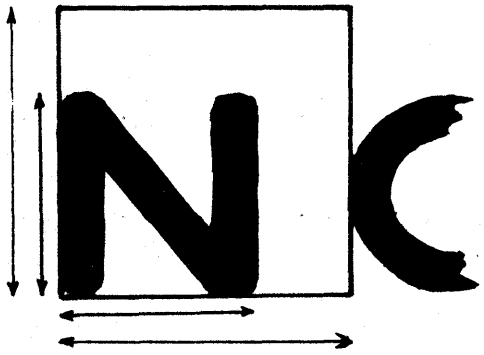


Figure 4

Omnigraph Display Routines

Reference Manual

Table of Contents

This document describes a new set of subroutines available on the PDP-10 for creating graphic displays. These routines are called the "Omnigraph Display Routines", and have been designed so that they may be used with three popular programming languages (SAIL, LISP, FORTRAN), and with a variety of different display terminals (DEC-340, Adage AGT-30, Computek 400, ARDS, Tektronix 4010).

The description of the routines has three distinct sections. The first is concerned with general concepts for creating, modifying, and destroying pictures. These concepts are common to all programming languages and to all terminals served by the routines. The second section deals with language dependencies: the precise usage of the routines in LISP differs from that in SAIL, etc. The third section deals with particular terminal dependencies: certain aspects of the terminals currently supported are documented.

The general approach of these routines is that they should be able to handle all simple graphical chores, and most complicated ones. The desire to use the same general framework to drive display devices of quite different character has necessitated a compromise of some of the graphical power of the fancier terminals. For those who desire to make extensive use of the subtleties, these routines are occasionally inadequate.

In addition, these routines do not impose (yea, permit) any elaborate fixed structure on the picture images. In some graphical applications, the structure of the picture on the display screen can be related to a data structure required by the graphical program (e.g. a circuit diagram probably has a data structure which details connections among components -- these connections are manifested as lines on the display screen). Display subroutines are often useful in these circumstances to save space and to reflect the program data structure in the display file itself. These routines provide no aids to this process, because subroutining interferes with windowing unless the display terminal has windowing hardware.

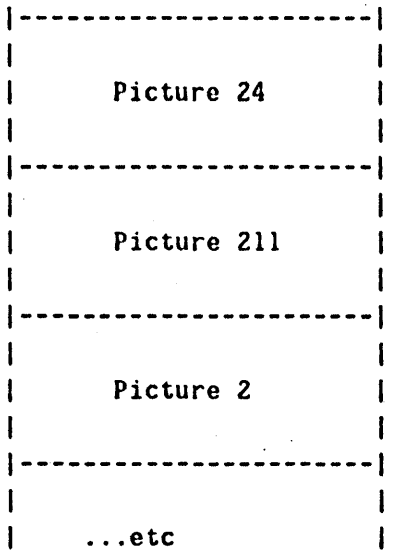
1.0 *The Display-File Compiler*

These routines are properly called a display-file compiler. They interpret subroutine calls from the user's program, and create a file of display instructions. This file is then used to actually create an image on the terminal. For example, in the case of the DEC-340 display, the file is examined by a hardware device, called the *display processor*, which is responsible for actually drawing lines, text characters, etc. on the face of the screen. In the case of the Computek 400, the file is

transmitted to the terminal in the form of graphical orders, and lines and text are drawn in response to these orders.

The Omnigraph routines will interpret a variety of requests to add graphical instructions to a display file. For example, the user may specify coordinates of points and lines in a two-dimensional coordinate system. The coordinates are (1) transformed by an arbitrary matrix transformation specified by the user, (2) checked against the limits of a *window* to see what parts of the line specified should be visible, and finally (3) instructions to draw the visible section of the transformed line are added to the display file. Notice that the transformations are applied before a display file is built. The details of these three operations are described below.

The display-file itself is divided into *pictures* or *segments*. Each picture is probably a logically separate part of the display. Each picture is identified by an integer number supplied by the user program. For example, the display file might look like:



Each picture is a list of display instructions: orders to draw lines, to display points, or to show text characters. The function of the display file compiler is to create and manage these pictures.

The existence of a picture in the display file is not inextricably linked with its image on the display screen. Pictures may be created and not immediately displayed. A picture which is shown is called *posted*; one which is not shown is *unposted*. The idea is that unposted pictures may be

later posted and thus become visible on the display screen. We can thus avoid using the display file compiler to regenerate the picture.

1.1 *Free Storage*

The Omnigraph routines store the display file in a free storage area specified by the programmer. As the number of display instructions in the display file grows, so will the consumption of space in the free storage area. In fact, the demands for space to store picture information may exceed the size of the free storage area. What then? The Omnigraph routines will, under certain circumstances, request more core from the timesharing system in which to continue storing picture data.

The user may request that free-storage be handled in one of two ways:

(1) The user may specify a fixed array of core to be used for all free-storage activities. If the demands for space exceed the size of the array, the Omnigraph routines can take no corrective action (However, see section 4.0).

(2) Alternatively, the user may specify that the Omnigraph routines are to create a *second segment* core area, and use that piece of core exclusively for free-storage. If expansion of that area is required, the Omnigraph routines will issue the appropriate calls to the operating system to request expansion. Under most circumstances, the area can expand, and display-file generation can proceed. If, for any reason, the second segment cannot expand, the Omnigraph routines can take no corrective action (See section 4.0).

Both mechanisms for free-storage management are provided because: (1) some of the display terminals (e.g. the 340) require that a second segment be used to hold display files, and (2) some users may wish to use the second segment facilities of the operating system for other purposes, e.g. sharing a common program or run-time system.

1.2 *Notation*

In the description of the subroutine calls interpreted by the Omnigraph routines, we shall adopt a common notation for describing the subroutine name, its arguments, and its return values, if any. These forms are neither SAIL, nor LISP, nor FORTRAN, but a notation which will yield the correct SAIL, LISP, or FORTRAN calling sequence if interpreted

appropriately. The section on language conventions describes these interpretations.

A subroutine call looks like:

result <- *name* (argument list)

The subroutine called *name* is being described: it requires a list of arguments, and may return a result. For some subroutines, the argument list or the result may be omitted.

The argument list is composed of a list of specifications of the arguments, e.g.

(picture number [integer], size [real])

Each argument is given a name for descriptive purposes, and its type is given inside square brackets ([]). Arguments are separated by commas. All arguments are mandatory (i.e. if an argument is listed, it must be given in the subroutine call).

The types are:

[integer]	integer value
[real]	real value (floating point)
[boolean]	either 'true' or 'false'
[pointer]	address of an entity in core

If a result is returned, it is specified in a fashion similar to an argument (i.e. descriptive name and type).

1.3 *Initialization*

The most useful feature of these routines (and indeed, the hardest feature to implement) is that the exact specification of which display terminal is to be used can be deferred until the program begins execution. A program may be loaded and 'SAVED' without specifying which display terminal is to be used. Then, when the Omnigraph routines are initialized, the routines for driving the particular terminal desired are loaded into your program and executed.

The initialization of the routines is accomplished with the following call:

```
success [boolean] <- DINI ( display-number [integer],
                            i/o-channel-number [integer],
                            free-storage-area [pointer],
                            free-storage-size [integer] )
```

This call initializes the subroutine package for the correct display, and initializes the free-storage area. The display-number argument specifies which display terminal is to be used. The numbers corresponding to each display are listed below with the terminal descriptions. (DINI will also accept the SIXBIT name of the display, as given in section 3, as an argument). The Omnigraph routines *do not initialize the display hardware at this time*, but merely load in the subroutines for driving that terminal. These routines are stored on the system area.

The i/o-channel-number argument specifies the number of an i/o channel (in the sense of I/O channels used by PDP-10 user programs when communicating with the timesharing system; a number between 0 and 15 decimal) which can be used by the display routines if needed. The exact use of this channel varies from terminal to terminal, but it is certainly used when creating plotter output.

The free-storage-area argument points to an array of core which will be used by the display routines as free-storage. The free-storage-size argument is the total length of this array. If the length is 0, then the display routines will create a second segment core area, and use it for free storage, expanding it as required.

If a display file currently exists and (perhaps) contains visible pictures, DINI will, with one exception, destroy the display file and re-initialize the space (see more details in section 1.17).

The value returned by the initialization call specifies whether the initialization was successful. The possible reasons for failure are (1) the free-storage-size was so small that the routines could not adequately initialize the area, or (2) the free-storage-size was 0, but the operating system could not allocate 2048 words of core to the second segment. Either of these circumstances will cause all subsequent subroutine calls to give meaningless results.

The subroutine call which actually seizes the display terminal is:

```
success [boolean] <- DGET
```

The Omnigraph routines attempt to reserve the use of the display terminal requested by the user in the DINI call. If the terminal is available

(i.e. not already seized by some other user), the DGET call returns 'true', otherwise 'false'.

If the display can be seized, then any pictures which are currently in the display file (and POSTed; see below) are shown on the display screen.

The display terminal may be released so that others may seize it with the call:

DREL

This call always succeeds, and leaves the screen blank. It has absolutely no effect on the display file itself, i.e. the lists of display instructions still reside in the free-storage area. If, at some later time, DGET is used to seize the display, any pictures still in the display file will be shown.

For displays which also serve as the user's 'keyboard' terminal, no other user can ever 'seize' the display. In this case, DREL only serves to inhibit picture transmission.

Modifications to the display file may be made with the subroutines listed below even if the display terminal is not seized. Thus, the terminal may need be seized only infrequently in order to view the results of some long computation. It is good practice to design your graphics programs to operate in this fashion. Contention for a particular sharable display terminal such as the DEC-340 is then easily resolved because users are utilizing the display terminals only during the periods when they wish to look at the pictures, and not during the periods when their programs are performing long computations.

1.4 *Generating Pictures*

Each picture in the display file is a separate entity, identified by its *picture number*. The subroutines described below are used for creating, modifying, and destroying pictures. The general strategy is as follows: we shall declare our intention to create a new picture by DOPENing a picture, and specifying a number *n* which will identify that picture. Then, we will issue a series of subroutine calls which request that display instructions be added to the list of instructions for this picture, e.g. lines, text, points. Each subroutine call will request that a particular line, point, or text string be added to the picture. After the last entry is made, we will DCLOSE the picture. Now picture *n* is complete, and becomes part of the display file.

The picture will not appear on the screen during the process of generation. Other Omnigraph routine calls are required in order to post the picture which has been generated.

The actual subroutine calls to accomplish opening and closing of pictures are:

DOPEN (picture-number [integer])

DCLOSE

Subroutine calls which add instructions to the display file are only legal inside a DOPEN-DCLOSE sequence.

The DOPEN-DCLOSE subroutines automatically provide a double-buffering ability. Consider the following sequence:

```
DOPEN (21)
        ...
        ...calls which generate display file
        ...instructions (period A)
        ...
DCLOSE
        ...
        ...other activities (period B)
        ...
DOPEN (21)
        ...
        ...calls which generate display file
        ...instructions (period C)
        ...
DCLOSE
        ...
        ...other activities (period D)
        ...
```

During period A, picture number 21 is being generated. During period B, picture number 21 is part of the display file because it has been fully generated (DOPENed and DCLOSEd). During period C, the first version of picture 21 is still part of the display file, even though a new version of picture 21 is being generated. After the new version is DCLOSEd (period D), the new version becomes part of the display file and the old version is deleted.

The DAPPEND call allows a user to *add* display file instructions to any picture already in existence. The call is:

DAPPEND (picture-number [integer])

For example:

```
DOPEN (21)
        ...
        ...calls which generate display file
        ...instructions (period A)
        ...
DCLOSE
        ...
        ...other activities (period B)
        ...
DAPPEND (21)
        ...
        ...calls which generate display file
        ...instructions to be added to picture
        ...number 21 (period C)
        ...
DCLOSE
        ...
        ...other activities (period D)
        ...
```

During period C, subroutine calls generate a list of display file instructions which will be added to picture 21. When the DCLOSE is given, the additions are actually made. During period D, picture number 21 will show the effects of the instructions generated during period A and during period C.

1.5 *Showing Pictures on the Display Screen*

The DOPEN-DCLOSE sequences listed above are only used to create display file. They do not control the use to which that display file is put. The subroutines DPOST and DUNPOST are used to add and delete pictures from the *display screen*. The calls are:

post-number [integer] <- *DPOST* (picture-number [integer])

post-number [integer] <- *DUNPOST* (picture-number [integer])

The DPOST call adds the specified picture-number to the list of pictures which should be displayed on the screen. The DUNPOST call removes the

specified picture-number from the list of pictures currently displayed on the screen (UNPOSTing an unknown or already unposted picture causes no error message). In either case, the value returned is the number of pictures posted after the call is interpreted.

Note that DPOST and DUNPOST may not have an immediate effect on the displayed picture. (see DDONE, below).

As an added convenience, if DPOST is called for a picture which is currently opened, a DCLOSE will be automatically done. Thus the sequence:

```
DOPEN (21)
        ...
        ...subroutine calls which add
        ...to the display file
        ...
DCLOSE
DPOST (21)
```

can be abbreviated:

```
DOPEN (21)
        ...
        ...subroutine calls which add
        ...to the display file
        ...
DPOST (21)
```

1.6 *Deleting Pictures from the Display File*

When the usefulness of a picture expires, it may be deleted from the display file. The space required to store the display instructions can then be reused by subsequent Omnigraph routine calls. The deletion is accomplished with the call:

DKILL (picture-number [integer])

The appropriate picture will be deleted from the display file. If the picture was POSTed at the time of the DKILL call, it will be UNPOSTed first.

1.7 *Updating the Display Screen*

The operations DPOST, DUNPOST, DKILL, and DAPPEND all may cause changes to the visible display. DPOST adds graphic information; DUNPOST and DKILL remove it. If we DAPPEND to a picture which has previously been DPOSTed, then the effect of the append operation will be to make more graphic items visible.

On some display terminals, the display screen can be changed very rapidly; in this case, the effect of the DPOST, DUNPOST, DKILL or DAPPEND calls will be immediately manifested on the screen. However, for those terminals which use storage-tubes of any variety, updating the screen frequently can consume great quantities of time. It would be preferable to update the screen *once* after each collection of changes (DPOSTs, DUNPOSTs, etc.) is made.

The call

DDONE

requests that the screen be updated. This may cause a preliminary erasure of a storage-tube screen. For example, if the graphics program accepts commands from the user's keyboard and and interprets them, possibly creating a new display, a DDONE call should be performed after the interpretation of every command. If any display information is to be added to or deleted from the display screen, the operations will be done at that time.

Using storage tube terminals interactively is inconvenient at best; the Omnigraph subroutines permit several strategies which should help reduce the absurdly long times required to update the screen with new displays. The idea of the DDONE command is that it should be issued once for every group of display modifications. For example, suppose the user is prompted to type in a command. The interpretation of that command may involve several DOPENings and DCLOSings of pictures, DPOSTs, etc. However, we should delay actually updating the screen (i.e. issuing the DDONE call) until we are sure that the present rush of modifications is terminated. If we issue DDONE too often, many screen erasures and redrawings will be required. The most natural time to issue DDONE is when the interpretation of the user's command is finished and the program is about to prompt him for another.

There is an additional problem on display terminals which are used for both graphical output (pictures, lines, etc.) and for the echoing of text typed at the keyboard. Most of these terminals have 'cursors', which control the position at which text will be added to the screen. It is

important, after each screen update, that the text cursor be positioned so that text echoes will be visible to the typer. Furthermore, the position of the text echo may be of some importance to the programmer. The call

DCURSOR (cursor-x [real],
cursor-y [real])

tells the Omnigraph routines where to position the cursor at the conclusion of the next DDONE operation. The x and y coordinates are measured in the standard viewport coordinate system (see section 1.8.2). The default position for the cursor is equivalent to DCURSOR (0,.95).

The user can request that the cursor be position immediately, rather than at the end of the next DDONE operation, by specifying negative arguments to DCURSOR.

1.8 *Generating Lines*

This section describes subroutine calls used to add instructions to draw lines to the the display file of the currently DOPENed picture. The routines will draw lines from two-dimensional descriptions of the endpoints of the lines, or will draw perspective views of lines from three-dimensional descriptions of the endpoints. (If the display terminal has hardware for showing three-dimensional pictures, as does the Adage AGT-30, the perspective generation can be bypassed. The three-dimensional information is then delivered intact to the terminal, where appropriate views can be generated; see section 1.12).

The processing of a request to draw a line or point goes through three separable operations:

1. Coordinate transformation
2. Windowing (and perspective generation)
3. Generating the display file instruction

These operations are described in the next sections.

1.8.1 *Coordinate Transformation*

Each group of coordinates used to describe a point or an endpoint of a line may be first transformed by according to some parameters given by your program. The transformation is capable of introducing rotation, scaling, and translation to your coordinate values before being displayed. The transformations for two and three dimensions can be thought of as operations on the coordinate vectors:

```
[ x y ]     T   ==> [ x' y' ]

[ x y z ]   T   ==> [ x' y' z' ]
```

In practice, the transformation T is a 4x4 matrix supplied by your program. The transformations may thus be accurately stated:

```
[ x  y  *  1 ]     -  -  -  -     ==> [ x' y'  *  * ]
                   -  -  -  -
                   -  -  -  -
                   -  -  -  -

[ x  y  z  1 ]     -  -  -  -     ==> [ x' y' z'  * ]
                   -  -  -  -
                   -  -  -  -
                   -  -  -  -
```

The values (x,y,z) are those provided by the user; the values (x',y',z') are the transformed counterparts of the user's values. The '*' entry in a vector means that that position in the vector is unused. The array of '-' symbols is the 4x4 matrix.

The 4x4 transformation matrix can be used to express any linear transformation of the x, y and z coordinate values. In the discussion which follows, M[i,j] is an element of the transformation matrix: row i, column j.

The identity transformation is simply M[1,1] = M[2,2] = M[3,3] = M[4,4] = 1.0, with all other elements equal to zero. The effect of the identity transformation is to leave the coordinate values unchanged. When the Omnigraph routines are initialized, the transformation is set to the identity.

Translations can be achieved by modifying the identity matrix with:

```
M[4,1] = x translation
M[4,2] = y translation
M[4,3] = z translation
```

You can easily verify that the transformation schemes given above do actually cause the 'translation' values to be added to the x, y, and z values supplied by the user.

Simple rotations can be specified. For example, the rotation through an angle theta about the origin of a two-dimensional domain is specified by the following modifications to the identity matrix:

```
M[1,1] = M[2,2] = cos (theta)
M[1,2] = sin (theta)
M[2,1] = -sin (theta)
```

Three-dimensional rotations about coordinate axes are similar.

Simple 4x4 matrix transformations can be concatenated to form more complicated transformations by matrix multiplication. For example, if we wish to first rotate an object and then translate it, we could express this sequence of transformations as:

```
[ x1 y1 * 1 ] = [ x0 y0 * 1 ] M
```

followed by

```
[ x' y' * * ] = [ x1 y1 * 1 ] N
```

The matrices M and N are simple rotation and translation matrices respectively. The two operations can be merged into one as follows:

```
[ x' y' * * ] = [ x0 y0 * 1 ] M N
```

The two 4x4 matrices M and N can be multiplied together to form one matrix Q, which has the effect of the combined transformations:

```
[ x' y' * * ] = [ x0 y0 * 1 ] Q
```

The full ramifications of this technique are very useful in graphics applications. References are given below to literature about computer graphics which describes the properties of this approach.

Each point (x,y and possibly z) presented to the Omnigraph routines is first transformed according to the *current* 4x4 transformation matrix. The current matrix is established with the call:

DAPPLY (matrix [pointer], name [integer])

This call provides the Omnigraph routines 16 floating-point numbers which

are loaded into the current transformation matrix. All subsequent points and lines will be subject to this transformation, until the current transformation is changed. The 'name' parameter should be zero for our purposes; it is used for dynamic three-dimensional displays (see section 1.12). If the DAPPLY call is omitted in your display program, the identity transformation will be assumed.

Note: the row/column conventions in the storage of the matrix are identical to those of the programming language used. M[i,j] refers to row i, column j regardless of the programming language used -- the versions of the routines for the different programming languages take account of the differing storage schemes.

The usefulness of the matrix transformations is greatly enchanced by being able to *concatenate* two transformations into one which has the same effect as the sequential application of the two. The call

DCOMPOSE (new-matrix [pointer], name [integer])

causes the current transformation to be replaced by the matrix product of new-matrix and the current transformation matrix. The order of multiplication is <new-matrix> * <current-matrix>.

The 16 values of the current transformation can be 'pushed' and 'popped' from a stack provided by the Omnigraph routines. The call

DPUSH (name [integer])

causes the current 16 values to be copied into a piece of free-storage, and placed at the top of the *transformation stack*. (Again, 'name' should be zero.) The call

DPOP (name [integer])

causes the 16 values on the top of the transformation stack to be stored in the 16 locations of the current matrix; then these 16 values are removed from the top of the stack.

1.8.2 *Windowing*

The windowing operation is applied to each line or point before a display instruction is generated. Windowing is described in detail in several of the references; only a short summary is given here.

In two dimensions, the user may specify a rectangular window which surrounds the area of the two-dimensional plane which should be visible on the display screen. Lines and points which lie partly or wholly within the specified window area will be displayed; lines or portions of lines which do not intersect the window area will not be displayed. In Figure R-1, the line A will be displayed; B will not; a portion of C will be displayed.

The window rectangle is aligned with the x-y coordinate system in which lines and points are specified. It can thus be determined by four numbers: the left, right, bottom, and top of the rectangle as measured in the page coordinate system (the coordinate system used by the user in his calls to have lines and points generated). This coordinate system is Cartesian, but the actual size and position of the system is immaterial: user's coordinates are compared to the window edge coordinates to decide whether a line is visible. The call

DWIND (left-edge [real], right-edge [real],
bottom-edge [real], top-edge [real])

specifies the window to be used when processing all subsequent requests to draw lines, points, or text in two dimensions.

This window area can be mapped onto any rectangular area of the display screen; it need not fill the screen. The image of the window on the screen is called the *viewport*, and is specified in the call

DPORT (left-edge [real], right-edge [real],
bottom-edge [real], top-edge [real])

For the purposes of the DPORT call, we establish a *standard viewport coordinate system* which will be useful for all types of display. The coordinate system is as follows:

x-coordinate

left edge of screen	0.0
right edge of screen	1.0

y-coordinate

bottom edge of screen	0.0
top edge of screen	1.0

The standard viewport coordinate system is an attempt at making most programs work correctly on most displays. However, some displays do not have square screen areas. For these displays, the conventions are as follows: the area of the screen represented by the viewport bounds

(0,1,0,1) will be the largest *square* area that can be located on the screen. The square area will be positioned at the botton left-hand corner of the screen (see Figure R-2). If you wish to refer to areas outside this square area, appropriate values outside the range 0 to 1 are permitted in the DPORT call (see DENQ, below). An empty viewport is perfectly legal.

When the Omnigraph routines are first initialized, the default values of the window and viewport edges are as if the following calls were executed:

```
DWIND ( -1, 1, -1, 1 )
DPORT (  0, 1,  0, 1 )
```

The region of the page bounded by x and y of plus/minus 1.0 will be mapped onto a square viewport on the screen.

1.8.3 *Two-Dimensional Points and Lines*

The following calls specify the generation of points and lines in two dimensions:

```
in [integer] <- DMOVE ( x1 [real], y1 [real] )
in [integer] <- DDRAW ( x2 [real], y2 [real] )
in [integer] <- DVECT ( x1 [real], y1 [real],
                        x2 [real], y2 [real] )
in [integer] <- DDOT  ( x1 [real], y1 [real] )
```

These subroutines can be viewed as manipulating a fictional beam in the page coordinate system. DMOVE moves the fictional beam to the specified point without drawing a line. DDRAW moves the fictional beam from its present position to the specified point; if the fictional beam passes through the window, a line-drawing instruction which will show the visible part of the line will be added to the display file. DVECT (x1,y1,x2,y2) is equivalent to the sequence DMOVE (x1,y1); DDRAW (x2,y2). DDOT causes a dot to be displayed at the specified point, provided it is within the window.

The value returned as a result of any of the 4 calls tells whether the object was within the window, and hence displayed (true return), or did not result in any display (false return).

As an aid to making other decisions about the relation of a display to the current window, the following call

```
code [integer] <- DTEST2 ( x [real], y [real] )
```

computes a code which tells whether the transform of the point (x,y) is within the current window. If the code is 0, the point is within the window. If the code is non-zero, four of the bits of the code specify in which way the point is outside the window:

0001 (octal)	x value is to the left of the window
0010 (octal)	x value is to the right of the window
0100 (octal)	y value is below the bottom of the window
1000 (octal)	y value is above the top of the window

For example, the code 1001 (octal) means that the point is above and to the left of the window.

1.8.4 *Three-Dimensional Points and Lines*

Generation of perspective views of three-dimensional objects can be accomplished with calls very similar to the above two-dimensional calls:

```
in [integer] <- DMOVE3 ( x1 [real], y1 [real], z1 [real] )
in [integer] <- DDRAW3 ( x2 [real], y2 [real], z2 [real] )
in [integer] <- DVECT3 ( x1 [real], y1 [real], z1 [real],
                         x2 [real], y2 [real], z2 [real] )
in [integer] <- DDOT3  ( x1 [real], y1 [real], z1 [real] )
```

The interpretations of the various calls are analogous to those for two dimensions; the fictional beam now moves in a space with a three-dimensional Cartesian coordinate system.

The coordinates are transformed by the current transformation matrix, and then clipped, unless otherwise specified (see section 1.12). The clipping operation for three-dimensional points is different from that for two-dimensional ones: a point must lie within a three-dimensional viewing pyramid in order to be visible. This pyramid is always shaped as shown in Figure R-3. The condition that a point lie within the pyramid is:

$$-z \leq x \leq z$$

and

$$-z \leq y \leq z$$

The transformation matrix can be used to deform the desired pyramid of vision into this 'standard' pyramid used for clipping.

Any visible portion of a line is then subjected to the following computation in order to compute a screen location:

```
screen x =  (x/z) * (viewportright - viewportleft)/2 +
                    (viewportright + viewportleft)/2

screen y =  (y/z) * (viewporttop - viewportbottom)/2 +
                    (viewporttop + viewportbottom)/2
```

The division of x and y by z is the central operation in the generation of a perspective display image. The viewport computations merely position the image on the screen in some desired position.

Notice that the coordinate system used in these computations is a left-handed one: if you face the display screen, the x axis points to the right (as does the x coordinate system on the display), the y axis points up, and the z axis is directed ahead, into the screen. If the three-dimensional coordinates of lines and points are in a right-handed system, the transformation matrix can be used to convert it to a left-handed system (merely setting M[3,3] = -1 will have the correct effect).

A function is available for testing points to see if they are inside the three-dimensional window. This function merely makes the tests described above.

code [integer] <- *DTEST3* (x [real], y [real], z [real])

The value of the code returned is similar to that for the two-dimensional case:

0001 (octal)	x < -z
0010 (octal)	x > z
0100 (octal)	y < -z
1000 (octal)	y > z

1.9 *Text Display*

All of the terminals supported by these subroutines have conventions for displaying text. The Omnigraph routines transform a request for text display into the form required by the particular terminal. Text display will therefore differ from one terminal to another: some will not have lower-case characters; some will not be able to vary the size of characters at all; some will be able to vary the size only in discrete steps.

The call

DTSCAL (height [real])

sets the character size desired. The size is measured in the same units used to specify the viewport. For example, if you desire characters 1/4 inch high on a screen which is 10 inches high, the appropriate size is 1/4 * (1/10) or 0.025. The Omnigraph routines will choose a size available on the terminal you are using which corresponds most closely to the size you have specified. The height setting remains in effect until set with another DTSCAL call. There is no default.

The call

DTEXT (character-string [text])

is used to actually display text. The exact format of this call will vary for different programming languages.

The Omnigraph routines will start displaying text at the position of the fictional beam; the lower left-hand corner of the first character will start where the fictional beam is. Thus, the DTEXT call will usually be preceded by a DMOVE call (or whatever) to position the fictional beam in the desired spot. After each character is drawn, the beam is positioned to be at the lower left-hand corner of the next character. The Omnigraph routines will decide what part of the text to display in one of two ways:

1. Display a character only if the entire character lies within the window.

2. Display a character regardless of whether it lies in the window. It will not be displayed if any part of the character is off the display screen.

The mode is determined by the *sign* of the size parameter last given to DTSCAL: if the size is positive or zero, mode 1 is used; otherwise mode 2. Note that mode 2 may request text characters to be displayed which would exceed the hardware limits of the screen; any such characters are discarded.

The characters may be windowed in the 2D sense or the 3D sense, depending which type of call was last executed before the DTEXT call. In other words, if you are displaying 3D vectors, and call DTEXT, characters will be windowed in three dimensions. The width and height of the character in the three-dimensional object coordinate system are determined at the time of the DTEXT call; this determination uses the current Z coordinate of the fictional beam at the time of the call.

The fictional beam position in two or three dimensions is remembered when DTEXT is called. This position is used as the left margin whenever a carriage-return character is encountered in the text to be displayed.

For those who wish to position characters accurately on the screen, several aids are provided. The DENQ call (see below) and figures listed with each terminal provide information about the actual size of characters on the screen. Four numbers are given: the x and y sizes of the actual character, and the x and y sizes of the 'box' in which that character is drawn (see Figure R-4).

Not all display terminals have the same character repertoires. The Omnigraph routines adopt several conventions to facilitate character display on a variety of terminals: (1) if the terminal has no lower case facility, lower case characters will be converted to upper, (2) if a character cannot be displayed on a terminal, a * is substituted. In addition, the call

status [integer] <- *DCHAR* (character [integer])

can be used to discover whether the terminal in used can faithfully show the ASCII character whose code is passed as an argument to DCHAR. The function will return 0 if there is no equivalent character that can be displayed, -1 if the display can display the character exactly, and 1 if it can display the character by transliteration (e.g. lower case transliterated to upper case).

1.10 *Intensity Control*

For those terminals which can draw lines of various intensities, the call

DINT (intensity [real])

is provided. This will set the intensity value for any subsequent lines, points, and text. The intensity is specified in a standard range: 0 is the lowest intensity, 1 is the highest.

1.11 *Display Subroutines*

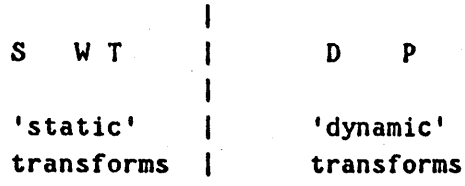
The Omnigraph display routines, as currently written, do not rule out providing subroutine capabilities (e.g. SAIL display routines). These may be added in the future. The complication of display subroutines is that they interfere with windowing: a particular subpicture will appear differently if drawn on different parts of the screen because different parts of the subpicture will be clipped.

1.12 *Dynamic Three-Dimensional Displays*

The Omnigraph routines include special features to take advantage of display terminals which have hardware for performing coordinate transformations, projective transformations, and viewport transformations. The procedure described above for generating perspective displays is called *static*: the transformations are all applied prior to building the display file. Thus the only way to alter such a display is to regenerate segments of the display file, perhaps using different transformations. The *dynamic* process, on the other hand, builds display files which contain x, y and z coordinates *prior* to transformation -- the required transformations are performed by special display hardware. This process acquires a dynamic character because altering the transformations will alter the display without regenerating the display file.

The ideal structure for the display file for dynamic displays would include specifications for lines and points in three dimensions, for applying, composing, pushing and popping transformations, and for viewport limits. The ideal structure is, however, unattainable with any present-day display equipment, largely because of the lack of floating-point hardware; at the very least, we must scale every coordinate so that it lies within the limits of the display's fixed-point number system. The discussion below presents the Omnigraph solution to this problem and to others encountered when designing the driver for an Adage AGT-30 graphics terminal. For additional details, see the Omnigraph Display Routines -- Implementation Manual.

The coordinates (x, y, z) passed via a DMOVE3, DDRAW3, DVECT3 or DDOT3 call to the Omnigraph routines for dynamic display are transformed as follows:



S is called the 'static matrix transformation'; it is expressed as one matrix, the current transformation matrix, just as described above for static perspective generation. W is a 'box clip' step which clips lines and points against a three-dimensional box specified with an Omnigraph call, and T maps the box limits into fixed-point numbers acceptable for the display hardware. These first three transformations are performed before the display file is built. The display hardware applies one more transformation, the combination D P, to the display-file points as the display is refreshed. In the case of the AGT-30, the x,y,z point which results from the transformation is projected orthographically onto the screen, i.e. the displayed point is at (x,y).

The hardware transformation D to be applied is computed as the display is being refreshed. Codes may be placed in the display file which cause the 'current dynamic transformation' to be altered just as Omnigraph calls cause the current static transformation to be altered: applying or composing new matrices; pushing the current matrix onto a stack; popping the stack into the current matrix. In order to alter the display image, we need only alter the values of one of the matrices taking part in an application or composition. We shall permit such 'dynamic matrices' to be given names, integer numbers, which can be used later on to name a matrix to be altered.

Matrices may be altered in two ways. First, the Omnigraph call DSETR will change values in any named dynamic matrix. Thus the graphics program can control the view of a three-dimensional scene without regenerating the display file. Second, the graphics terminal itself may provide commands which cause matrices to be given new values specified by dial readings, keyboard values, or constantly-changing values (thus giving a 'tumbling' effect). The AGT-30 software provides several commands of this sort; in addition some commands affect the entire display, regardless of the user's matrices.

The foregoing discussion represents a somewhat simplified view of the display process. Let us describe in more detail the operation of the Omnigraph routines and the AGT-30. The transformation applied to a point is *precisely*:

$$S_n \ldots S_2\ S_1\ W\ T \mid T'Dm\ T \ldots T'D_2\ T\ T'D_1\ T\ T'PA$$

dynamic transformation

The S_i sequence of transformations is expressed as one matrix, the product of all the S_i. This matrix is called the 'current static transformation'. The box clip has already been described. T is a scaling and translation

transformation which maps the box limits into the AGT coordinate system: coordinate values vary from -1 to +1. The dynamic transform is one matrix, composed from several entities:

1. A dynamic transformation sequence $D_m \ldots D_2 D_1$ modified by the Omnigraph routines to include the effect of the transformation T (T' is the matrix inverse of T). Thus a floating-point matrix D, specified by an Omnigraph call, is actually given to the Adage as T'DT. The user of the Omnigraph routines can thus ignore the fact that T has been applied *before* the D_i; he should think of the transformation of each point as being S W D P A.

   This transformation sequence varies as the display file is executed, i.e. 'apply matrix', 'compose matrix', 'push' and 'pop' commands in the display file will cause this sequence to change.

2. P is a transformation which establishes the viewport. It is multiplied by T' in order to account for the scaling.

3. A is a matrix controlled by the dials on the Adage. If values in A change, the whole picture will appear to rotate, translate, etc. Thus, even if the Omnigraph user provides no dynamic transformation D, the transformation A can still be used to alter the view of the scene.

Each of the matrices $T'D_iT$, T'P, and A is constrained to have values in the range -1 to 1. If the dynamic matrices are used solely to *rotate* the image about the origin, this constraint should not offer difficulties.

The Omnigraph routines for controlling static and dynamic three-dimensional display are listed below:

DBYP (n [integer])

This call is used to specify exactly now each point presented to the Omnigraph routines (DMOVE3, DDRAW3, DVECT3, DDOT3) is to be processed. The default is the static process: static transformation, pyramid clip, and perspective divide, as described in section 1.8. That default can be altered by DBYP calls as follows:

- n =-1 Set 'static processing' mode. Static transformations, pyramid clip, and perspective divide are the default operations. These defaults may be overriden with further calls to DBYP as described below.
- n = 1 Set 'dynamic processing' mode. Static transformations, box clip, the T transformation, and dynamic transformations are the default operations. These defaults can be overriden with further calls to DBYP as described below.

n =-2 Omit the box-clip step.
n = 2 Enable the box-clip step.
n =-3 Disable pyramid clipping and perspective division.
n = 3 Enable pyramid clipping and perspective division. If box clipping is also enabled, it is performed first.
n =-4 Disables the use of T, i.e. sets T to the identity matrix.
n = 4 Enables use of T (see below).

DWIND3 (left [real], right [real], bottom [real], top [real], zmin [real], zmax [real])

This call specifies the limits of the box to be used for the box clipping step. It thus gives minimum and maximum values that the x, y, and z coordinates can assume. If matrix T is enabled, it is set to:

$$\begin{bmatrix} \frac{2}{\text{right-left}} & 0 & 0 & 0 \\ 0 & \frac{2}{\text{top-bottom}} & 0 & 0 \\ 0 & 0 & \frac{2}{\text{zmax-zmin}} & 0 \\ \frac{\text{right+left}}{\text{right-left}} & \frac{\text{top+bottom}}{\text{top-bottom}} & \frac{\text{zmax+zmin}}{\text{zmax-zmin}} & 1 \end{bmatrix}$$

DPORT (left [real], right [real], bottom [real], top [real])

This call is precisely as described in section 1.8.2. It defines the area of the screen in which the picture is to appear. The matrix P is set to:

$$\begin{bmatrix} \text{right-left} & 0 & 0 & 0 \\ 0 & \text{top-bottom} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \text{left} & \text{bottom} & 0 & 1 \end{bmatrix}$$

DAPPLY (new-matrix [pointer], name [integer])

Static processing: If static processing is in effect, or if the terminal has no dynamic transformation capability, this call has the effect described in section 1.8.1: the new-matrix is loaded into the current static transformation.

Dynamic processing: If dynamic processing is in effect, the value of 'name' controls the use of new-matrix: if name =-1, new-matrix replaces the current static transformation, as above. Otherwise, a dynamic 'apply matrix' command, which references a copy of new-matrix, is added to the display file. If name is positive, it is taken as the 'name' of the matrix. (If a matrix of the same name already exists, new-matrix will not supersede it.) If the transformation T is enabled, the matrix which will actually be applied is T'DT, where D is new-matrix. Warning: applying a dynamic matrix nullifies the effect of T'PA; DCOMPOSE is preferred.

DCOMPOSE (new-matrix [pointer], name [integer])

Static: This call has the effect described in section 1.8.1: the product <new-matrix> * <current static matrix> replaces the current static matrix.

Dynamic: If name =-1, the product <new-matrix> * <current static matrix> replaces the current static matrix. Otherwise, a dynamic 'compose matrix' command which references a copy of new-matrix is added to the display file. If name is positive, it defines a name for the matrix. (Again, if matrix T is enabled, the matrix transmitted to the display terminal is really T'DT, where D is new-matrix.)

DPUSH (name [integer]) DPOP (name [integer])

Static: The local stack is manipulated as required: DPUSH pushes a copy of the current top transformation back onto the local stack. DPOP pops the local stack.

Dynamic: If name = -1, DPUSH and DPOP apply to the local stack. Otherwise an appropriate 'push' or 'pop' command is added to the display file. The Adage maintains a transformation stack which is manipulated by the display-file commands 'push' and 'pop', just as the local stack is maintained in the static protocol. The current top of stack is the transformation in force at any time.

The stacks are both initialized with the identity transformation: this initialization is performed in the Omnigraph routines when they are loaded, and by the Adage each time a new picture segment is refreshed. If ever DPOP is applied to an empty stack, an error message is issued and the request is ignored.

DSETR (new-matrix [pointer], name [integer])

Static: this is a no-op.

Dynamic: If name is positive, new-matrix replaces the previous matrix with name 'name'. (Again, if transformation T is enabled, T'DT is the actual matrix used for replacement, where D is new-matrix.) If no such matrix exists, an error message is issued and the DSETR request is ignored.

1.13 *Input Facilities*

Some of the terminals served by the Omnigraph routines have hardware for inputing information to the program. This section describes two calls for utilizing that hardware. Section 3 gives additional details about the input facilities of each terminal.

The input facilities are organized around *events*. The user may enable several different classes of input operations to be reported. The call

DEVENT (device [integer], op [integer], answer [pointer])

is used to control the recording and reporting of events.

Events may be generated by two devices: (1) a function key is depressed by the user, (2) the tablet stylus is raised after having been depressed, or (4) the tablet stylus is depressed and has moved a little bit since the last event. Notice that these devices are numbered as powers of 2, and can therefore be combined. For example, to enable function key hits and tablet 'pushes,' the 'device' parameter would be 3 (=1+2).

The 'op' argument specifies combinations of the following: (1) clear the list of incoming events, (2) enable the 'device' or devices to report events to the list, and (4) wait for an event to happen. Notice that the numbers for these operations are 1, 2 and 4. They may be combined: 5 (=4+1) means clear the current list of events and then wait until an event happens.

The general strategy for initializing input devices is to enable appropriate devices and clear the event buffer. Thereafter, events can be 'read' from the buffer, using the 'wait' option if desired. The enable option should be used sparingly; it is only necessary when *changing* input devices. It is a good practice never to clear the event buffer (except initially); each event in the buffer represents a user action which should be processed in some way by the program.

The DEVENT routine is also used to retrieve events from the event buffer. If an event has arrived, the DEVENT routine fills the 'answer' array with information about the event. Answer[1] contains the number of the 'device' which caused the event (if it is 0, then no event is being reported). The remainder of the 'answer' array depends on the type of event being reported.

If the reported event is a key-hit, Answer[1] is 1, and Answer[2] is the (floating-point) number of the key which was depressed.

If the reported event is due to the raising of the tablet stylus, Answer[1] is 2, and Answer[2] and Answer[3] contain the x and y coordinates of the pen when it was raised. The values for x and y are in the current page coordinate system, i.e. they are in the same coordinate system as is used when passing arguments to the two-dimensional point and line-drawing subroutines.

If the reported event is due to the motion of the tablet stylus, Answer[1] is 4, and Answer[2] and Answer[3] are loaded with the x and y page coordinate values for the new stylus position.

The input devices often have output counterparts, e.g. a key stroke may want to be answered by turning off a light under the key, etc. The following function controls the output effects of the tablet and keys:

DOUT (device [integer], op [integer])

The DOUT call with device = 1 can be used to turn a light on by giving in 'op' the number of the light (same numbering system as that for keys) or to turn a light off by giving the negative number of the light. DOUT (1,0) turns all lights off.

The DOUT call with device = 2 is used to control the tablet. A DOUT (2,x) must be given *before* any tablet events are expected, and is used to initialize the tablet-handling functions. If 'op' is positive, a trail of ink on the display will follow the path of the stylus whenever the stylus is depressed. In addition, a dot will follow the path of the stylus, whether it is depressed or not. If 'op' is zero, the ink is disabled, but the dot will still follow the stylus. If 'op' is -1, ink and dot are disabled. The DOUT (2,x) call may be repeated. In particular, it is used to erase from the screen any ink left over from previous tablet input. A positive value of 'x' passed to DOUT specifies (approximately) how long the maximum ink trace can be (the actual value of x should be established by trial; it corresponds roughly to inches of ink).

The following brief example is a SAIL program which does nothing but allow inking -- a carriage return causes the DOUT (2,x) call to be executed again, thus clearing any existing ink.

```
DINI (0,10,D[1],0); DGET;  "GET 340 DISPLAY"
WHILE TRUE DO BEGIN
        DOUT (2,40);       "INITIALIZE AND CLEAR INK"
        INCHWL;            "WAIT FOR USER CARRIAGE RETURN"
END;
```

1.14 *Plotting*

An off-line plot of any display can be made with the call

DPLOT (buffer [pointer], filename [integer])

This call writes a small disk file which contains the display file instructions for any pictures which are currently DPOSTed. If a non-zero SIXBIT filename is specified, the file will be so named (extension .PLX). Otherwise, the file name will be chosen so as not to conflict with other plot files already on the disk. (Names will be of the form 000000.PLX, 000001.PLX, 000002.PLX, etc.)

The operation of writing the file requires the use of a 128-word core buffer. This buffer cannot, in general, be obtained from the display free-storage, so the user is expected to supply a pointer to a free buffer area. After the DPLOT subroutine returns, the buffer area can be used for other purposes.

The buffer also provides a way of communicating to the plotting package a variety of special parameters related to the plot. At present, the following conventions are established:

buffer [1]	[real] 314159 (the number!)
buffer [2]	[real] x scale value
buffer [3]	[real] y scale value
buffer [4]	[real] x offset value
buffer [5]	[real] y offset value
buffer [6]	[real] paper type: 1 -- standard 2 -- vellum 3 -- grid
buffer [7]	[real] paper width: 11 inches 30 inches
buffer [8]	[real] pen type: 1 -- ball point 2 -- felt tip pen 3 -- wet ink

buffer [9]	[real] pen color: 1 -- black 2 -- read 3 -- green 4 -- blue 5 -- turquoise 6 -- purple 7 -- yellow 8 -- orange 9 -- brown 10 -- pink
buffer [10]	[real] overlay next plot (if 1, will overlay)

If buffer[1] is not 314159, then default values are assumed for the remaining parameters: 1.0, 1.0, 0., 0., 1, 11, 2, 1. If entries 6, 7, 8, or 9 are 0, they are individually defaulted to 1, 11, 2, and 1.

The plotter output will mirror, as successfully as possible, the display image shown at the time of the DPLOT call (The display does not have to be seized, however; DPLOT merely examines the display file). The scale factors and offset values can be used to enlarge the display when plotting. If the scale factors are 1 and the offsets 0, then the plot will be exactly the same size as the display was when it was viewed. The following computation is performed:

```
(plotter x in inches) =
        (x in standard viewport coords) *
        (screen width in inches) * scale + offset
```

along with a similar computation for y. The computation has the effect that 'scale' is an enlargement factor and 'offset' is the offset in inches of the lower left corner of the plot.

A program called PLOTX reads the .PLX files created and actually accomplishes the plotting. This program will produce output for the off-line Calcomp plotter, for the SC 4070 microfilm unit, and for the ZETA plotter. The operation of the program is very simple, and is directed by various promptings.

1.15 *Miscellaneous Subroutines*

The enterprising programmer may want to know several details about the terminal actually being used. The subroutine call

DENQ (array [pointer])

can be used to enquire about many salient characteristics of the terminal, and about the current state of the display routines. The effect of the subroutine is to fill the array with floating-point numbers which describe the terminal:

array[1] *Storage Tube*? This value is non-zero if the terminal does not have a dynamic-refresh display. Otherwise, it is zero.

array[2] *Three-Dimensional Hardware*? This value is non-zero if the terminal has hardware for rotating three-dimensional objects.

array[3] *Tablet*? This value is non-zero if the terminal has some kind of stylus tablet attached to it.

array[4] *Keys*? This value is 1 if the terminal has function keys attached. The value is 2 if the terminal has keys and also has lights under the keys.

array[5] *Points per unit Viewport*. This is the number of resolvable points which corresponds to a viewport value of 1.0.

array[6] *Inches per unit Viewport*. This is the size (in inches) of the screen which corresponds to a unit viewport (value = 1.0).

array[7] *Maximum value of X Viewport*. This is the maximum value which an X viewport value can take, in the DPORT call.

array[8] *Maximum value of Y Viewport*. This is the maximum value which a Y viewport value can take, in the DPORT call.

array[9] *Character Height*. This is the actual height of characters, as set by the last DTSCAL call. The height is measured in page coordinates, as specified in the last DWIND call.

array[10] *Character Width*. This is the width of characters, as set by the last DTSCAL call, as measured in the page coordinate system.

array[11] *Character Box Height.* This is the height, in page coordinates, of the box which surrounds a character.

array[12] *Character Box Width.* This is the width, in page coordinates, of the box which surrounds a character. It can be used to answer questions like: "How many characters can I fit between x = 1 and x = 5500 in the page coordinate system."

array[13] *Current Fictional X Beam.*

array[14] *Current Fictional Y Beam.*

array[15] *Current Fictional Z Beam.*

array[16-19] *Current Viewport Limits.* These four numbers are the left, right, bottom, and top limits of the viewport. These are merely copies of the last arguments passed to DPORT.

array[20-23] *Current Window Limits.* These four numbers are the left, right, bottom and top limits of the window. These are values of the last arguments passed to DWIND.

The display routines have a dynamic storage allocation facility for managing the free-storage area. This facility is used extensively by the various picture-generating routines described above. As a convenience, the two basic subroutines are also made available to the user program. They can be used to reserve and release space within the free-storage area. The calls are:

address [integer] <- *DCORGET* (size [integer])

DCORREL (address [integer])

DCORGET reserves a space of length 'size,' and returns the address of the first word of that space. If no core of that size can be found, 0 is returned. DCORREL releases the space beginning at 'address.' Users should be very careful to stay within the bounds of the piece of core given them; a faulty program can destroy vital information in the Omnigraph routines.

Several of the displays served by the Omnigraph routines have special features which do not fit into the Omnigraph framework. In order to partially circumvent this problem, the call

DCODE (code [integer])

can be used to specify special actions. For most remote terminals, the 'code' is merely transmitted to the terminal. Section 3 explains the function of DCODE for each terminal. Since this feature is terminal-dependent, use of DCODE is not recommended unless absolutely necessary.

1.16 *Sequencing of Omnigraph Routine Calls*

Certain calls may be issued at any time; others can only be issued when a picture has been DOPENed (or is being DAPPENDed to). All calls must follow the DINI initialization call. The classes of calls are as follows:

1. Can appear anywhere:
   DINI
   DGET, DREL, DCURSOR, DDONE, DKILL, DPOST, DUNPOST,
   DOPEN, DAPPEND, DCLOSE, DCHAR
   DAPPLY, DCOMPOSE, DPOP, DPUSH, DSETR, DBYP
   DWIND, DPORT, DWIND3, DTEST2, DTEST3,
   DTSCAL, DINT, DPLOT, DENQ, DEVENT, DOUT.

   (The effect of DTSCAL and DINT when called outside a picture is to remember the parameters as global defaults).

2. Can appear only inside DOPEN-DCLOSE pair:
   DMOVE, DDRAW, DVECT, DDOT
   DMOVE3, DDRAW3, DVECT3, DDOT3
   DTEXT

1.17 *Technical Considerations*

This section describes some particularly complicated uses of the Omnigraph routines and the various caveats which pertain to the complication:

Overlays: If you request that the second segment be used for display data, then *all* internal data in the Omnigraph routines is saved there. Even if the first segment is destroyed, and an entirely new one is read in as an overlay, the second segment data remains untouched. If a second segment exists when DINI is called, it checks to see whether that segment was created by the Omnigraph routines for the same display. If so, the DINI call does not initialize the free-storage area, but leaves all pictures

intact. It is thus possible to create a picture with one overlay, post it with a second, and kill it with a third!

The DINI call is required each time an overlay replaces the place where the display subroutines were loaded. The effect of the call is simply to reload the device-dependent code for your display.

Using several displays: DINI can be called once to read in the code segment for one display, say the DEC340, and then again to read in the code segment for another display, say the ARDS. The usefulness of this strategy is questionable, but the routines do permit the activity. The program should be sure to release (DREL) the first display before reading in code for the second. In addition, when the new routines are initialized, the display file is of necessity destroyed completely. Your program will thus have to regenerate the display.

2.0 *Language Considerations*

This section describes the special behaviour of the Omnigraph routines in each of the three programming languages SAIL, LISP, and FORTRAN.

2.1 *SAIL*

A relocatable copy of the Omnigraph routines is saved as SYS:DISSAI.REL. The SAIL 'REQUIRE' verb can be used to cause this file to be loaded with your program.

The declarations to be included in your SAIL program are:

```
DEFINE EP="EXTERNAL PROCEDURE", EIP="EXTERNAL INTEGER PROCEDURE";

        EIP DINI (INTEGER N,CH; REFERENCE INTEGER AR; INTEGER S);
        EIP DGET;
        EP  DREL;

        EP  DOPEN (INTEGER N);
        EP  DAPPEND (INTEGER N);
        EP  DCLOSE;

        EIP DPOST (INTEGER N);
        EIP DUNPOST (INTEGER N);

        EP  DKILL (INTEGER N);

        EP  DDONE;
        EP  DCURSOR (REAL X,Y);

        EP  DAPPLY (REFERENCE REAL AR; INTEGER NAME);
        EP  DCOMPOSE (REFERENCE REAL AR; INTEGER NAME);
        EP  DPUSH (INTEGER NAME);
        EP  DPOP (INTEGER NAME);
        EP  DSETR (REFERENCE REAL AR; INTEGER NAME);

        EP  DWIND (REAL L,R,B,T);
        EP  DPORT (REAL L,R,B,T);
        EP  DWIND3 (REAL L,R,B,T,ZMIN,ZMAX);

        EIP DMOVE (REAL X1,Y1);
        EIP DDRAW (REAL X2,Y2);
        EIP DVECT (REAL X1,Y1,X2,Y2);
        EIP DDOT (REAL X1,Y1);
```

```
EIP DTEST2 (REAL X,Y);

EIP DMOVE3 (REAL X1,Y1,Z1);
EIP DDRAW3 (REAL X2,Y2,Z2);
EIP DVECT3 (REAL X1,Y1,Z1,X2,Y2,Z2);
EIP DDOT3 (REAL X1,Y1,Z1);
EIP DTEST3 (REAL X,Y,Z);
EP  DBYP (INTEGER CODE);

EP  DTSCAL (REAL S);
EP  DTEXT (STRING S);
EIP DCHAR (INTEGER I);

EP  DINT (REAL I);

EP  DEVENT (INTEGER DEV,OP; REFERENCE REAL ANS);
EP  DOUT (INTEGER DEV,OP);

EP  DPLOT (REFERENCE REAL AR; INTEGER FILE);
EP  DENQ (REFERENCE REAL AR);
EIP DCORGET (INTEGER SIZE);
EP  DCORREL (INTEGER ADDRESS);
EP  DCODE (INTEGER CODE);

REQUIRE "SYS:DISSAI" LOAD!MODULE;
```

2.2 *LISP*

The Omnigraph routines are loaded into LISP in two steps: first the relocatable file is loaded; then a file of S-expressions is loaded which actually defines all the appropriate SUBRs. The relocatable file is SYS:DISLIS.REL; the S-expressions are SYS:DISLIS.LSP.

All of the Omnigraph subroutines become LISP functions after these two files have been read into LISP. Integer and real numbers can be passed to these routines in the usual way:

```
(DWIND 0 1 0 1)
```

The Omnigraph routines will perform any type conversions from integer-to-real or real-to-integer which may be required. Pointers are passed to the Omnigraph routines by means of LISP arrays, e.g.

```
(ARRAY FOO () @(1 . 4) @(1 . 4))
(DAPPLY @FOO 0)
```

Text is passed to the Omnigraph routines via the call

```
(DTEXT ...)
```

which precisely mimics (PRINT ...).

2.3 *FORTRAN*

The Omnigraph routines for FORTRAN are found in the file SYS:DISFOR.REL. You must specify that this file be loaded with the FORTRAN program, e.g.:

```
LOAD MY.F4,SYS:DISFOR
```

The arguments of type real, integer, and pointer may all be passed to Omnigraph routines as you would pass arguments to any FORTRAN subroutine. The Omnigraph routines will convert from integer-to-real and from real-to-integer if necessary. However, when pointers are specified, no conversions are performed (i.e. be sure that arrays passed to DAPPLY, DCOMPOSE and DPLOT are floating-point).

The functions which return values will have to be declared INTEGER, unless you wish their return values to be ignored (e.g. the DMOVE, DDRAW calls, etc.).

Text is passed to FORTRAN in a rather uncomfortable way: the ENCODE statement must be used to create an array of ASCII text to be displayed. Then the call

```
CALL DTEXT ( asciiarray, length )
```

will display text from the ASCII array; 'length' specifies the number of characters to be displayed.

3.0 *Terminal Considerations*

This section describes the idiosyncracies of the various terminals supported by the Omnigraph routines. At the very least, each display terminal is assigned a number which is used in the call to DINI to specify which terminal is to be used.

3.1 *DEC-340 Display*: display number 0, name "DEC340"

The DEC-340 display is one of the higher quality display terminals we have (alas), with the following properties:

Coordinate resolution: 1024 by 1024 points on a 10 inch square screen

Viewport maxima: X: 1.0; Y: 1.0.

Intensity resolution: 8 intensity levels

Character fonts: 64 ASCII characters, not including lower case; text can appear in 4 sizes, corresponding to values of the DTSCAL parameter of .0068, .0127, .0244, and .0479. The character and box dimensions (in inches) are as follows:

DTSCAL	CHARACTER		BOX	
	width	height	width	height
.0068	.0489	.068	.0586	.107
.0127	.0880	.127	.117	.215
.0244	.166	.244	.235	.430
.0479	.323	.479	.469	.860

Capacity: less than 200 inches of vectors

Input facilities: Lights and buttons are numbered 1 through 16. The Graphacon tablet provides inking and tracking facilities.

DCODE operation: refer to Omnigraph Implementation Manual.

The use of this display requires that free-storage come from a second segment. The third and fourth arguments to DINI are thus ignored, and a second segment is always used.

3.2 *Computek 400*: display number 1, name "C400"

This is a direct-view-storage-tube display, connected to the PDP-10 via asynchronous communications lines. The terminal is used both for graphics displays and for echoing text typed in by the user. Essential characteristics are:

Coordinate resolution: 1024 by 780 points on a 8.25 by 6.4 inch screen

Viewport maxima: X: 1.279; Y: 1.0.

Intensity resolution: one intensity level

Character fonts: 96 character ASCII (including lower case) in exactly one size, corresponding to a DTSCAL parameter of .015. The character measures .0645 by .0968 inches, and resides in a box .0968 by .16 inches.

Capacity: extremely large, but the time required to draw compilicated pictures may get quite large

Input facilities: none

DCODE operation: low-order 8 bits of the argument are sent directly to the terminal.

Special features: the ability to draw curves of various kinds

3.3 *Adage AGT-30*: display number 2, name "ADAGE"

The AGT-30 is a high-performance display with analog vector generation and three-dimensional rotation hardware built into the display. The hardware does not produce an accurate perspective view, but it can be used to achieve the kinetic depth effect, that is motions of the object in space serve to communicate spatial relationships.

Coordinate resolution: 16384 by 16384 points on a 10 by 10 inch screen. (An area 12 by 12 inches can in fact be used, although some vectors may be distorted. If you wish to use this larger area, the X and Y viewport parameters may be set to 12/10 = 1.2.)

Viewport maxima: X: 1.0; Y: 1.0 (nominally -- see above).

Intensity resolution: 1000 or more

Character fonts: 96 character ASCII (lower case included), at discrete sizes corresponding to values of the DTSCAL parameter of .015, .030, and .045.

DTSCAL	CHARACTER		BOX	
	width	height	width	height
.015	.1	.15	.125	.3
.030	.2	.30	.250	.6
.045	.3	.45	.375	.9

Capacity: about 15000 inches of vectors

Input facilities: Lights and buttons are numbered 1 through 16. Inking and tracking facilities are provided.

DCODE operation: low-order 16 bits are transmitted to the Adage; refer to 'Adage Picture Transmission Language' for coding information.

Special features: 3-D transformation hardware.

3.4 *ARDS*: display number 3, name "ARDS"

The ARDS display is a storage-tube device. Characteristics:

Coordinate Resolution: 1080 by 1414 points on a 6.375 by 8.25 inch screen.

Viewport maxima: X: 1.0; Y: 1.309.

Intensity resolution: one intensity level.

Character font: 96 character ASCII (including lower case). The character size corresponds to a DTSCAL parameter of .0134; it measures .067 by .086 inches; the box size is .08 by .165 inches.

Capacity: very large, but may take some time to draw pictures.

Input facilities: none

DCODE operation: low-order 8 bits are sent directly to the terminal.

3.5 *Tektronix 4010-1*: display number 4, name "T4010"

The Tektronix terminal is a small, quiet, inexpensive storage-tube terminal. One nice feature of this terminal is that it can be quickly changed to 150, 300, 600, 1200, 2400, 4800 or 9600 baud asynchronous operation. Characteristics:

Coordinate Resolution: 1023 by 780 points on an 8.25 by 6.4 inch screen

Viewport maxima: X: 1.311; Y: 1.0

Intensity resolution: one intensity level.

Character font: 64 character ASCII; no lower case. There is one character size, corresponding to a DTSCAL parameter of .0179. The character size is .09 by .12 inches; the box size is .10 by .16 inches.

Capacity: very large.

Input facilities: The cross-hairs are used for two-dimensional input; they are enabled whenever 'tracking' is requested. No inking is possible. When the cross-hairs are enabled, the coordinates are sent to the PDP-10 (thus causing an 'event') by pressing the 'return' key.

DCODE operation: low order 8 bits are sent directly to the display terminal.

4.0 *Error Reporting*

The Omnigraph routines report errors in a uniform manner. Each error causes a message to be typed out on the user's terminal. If the error is fatal, the terminal is returned to monitor mode; otherwise, the user program is allowed to continue.

Some care has been taken in the construction of the Omnigraph routines to insure that errors which may depend on the state of the time-sharing system or on the exact size of the display file will not be fatal to the entire program. For example, if the Omnigraph routines run out of free storage and cannot acquire more from the operating system, the picture currently being generated is flushed, and all subsequent requests to add to the picture are ignored, up until the picture is CLOSEd. If the graphics program is interactive, perhaps the user could DKILL some pictures he no longer needs, thus reclaiming some free storage space, and could attempt the generation process again.

If the core routines must stop generation of a picture, the message 'No core for display' is typed on the console to warn you that some picture may not be generated correctly.

The error message printed looks like:

```
?DISPLAY ERROR NUMBER 4; ROUTINE DKILL
 CALLED FROM 5107
```

The error can be found in the table of errors given below, together with a more verbose description of the difficulty. For debugging purposes, you can put a DDT breakpoint at the location DERRHL. The breakpoint will stop the program after the error message is printed, but before any action is taken.

Error	Fatal?	Description
1	no	Free storage is exhausted and we cannot get more from the operating system. The generation of the present picture is terminated. At the next DOPEN, however, generation will be restarted.
2	no	The type of a subroutine argument is incorrect. This is usually a programming error of some sort.
3	yes	A subroutine attempted to generate a display file instruction when no picture was currently DOPENed.

4	no	An attempt to DKILL a non-existent picture.
5	no	An attempt to DPOST a non-existent or already posted picture.
6	yes	An attempt to DAPPEND to a non-existent picture.
7	no	One of the parameters to DPORT is out of bounds.
9	yes	Bad core pointer. The dynamic storage allocation routines have discovered an inconsistency in the free-storage mechanism.
10	no	DPLOT cannot write the .PLX disk file.
11	no	DPUSH cannot find a piece of core in which to save the transformation.
12	no	DPOP is trying to 'pop' beyond the top of the stack (i.e. you are popping more than you pushed).
13	no	Function is not implemented for this device.
14	no	Cannot find the code segment for the display you have requested in DINI.
15	no	Cannot find a piece of core for the event buffer, the inking buffer, or some other function related to the DEVENT/DOUT routines.

References

1. *Guide to the PDP-10 Timesharing System.* This document describes the DEC-340 display system in detail, and also the other two brands of display routines available on the PDP-10. (Available from the Technical Information Office).

2. Sproull and Sutherland, *A Clipping Divider*, FJCC 1968. This article describes two-dimensional and three-dimensional clipping and windowing.

3. Sproull, *Notes for Computer Graphics.* These notes for an NIH-CCB seminar on computer graphics describe windowing and clipping algorithms, transformation systems, etc. (Available in the DCRT Library).

4. Omnigraph Display Routines -- Implementation Manual. Available from the DCRT Technical Information Office.

The PLOTX Program

PLOTX is a program which reads specially-formatted disk files and makes off-line plots. The files are created as a result of the DPLOT Omnigraph routine; they are essentially copies of the display files for all posted pictures at the time DPLOT is called. The PLOTX program knows the format of all display files used in the Omnigraph routines, and can decode them and produce plots which mirror, as closely as possible, the display shown on the screen.

PLOTX can also be used to accomplish a number of trivial transformations on the picture as it is plotted. These include enlarging, offsetting, and overlaying.

PLOTX is designed to produce plots on one of three devices:

Off-line Calcomp Plotter
SC 4060 Microfilm device (not implemented 8-72)
ZETA Plotter (not implemented 8-72)

The Calcomp plotter offers a number of additional options: selection of paper type and size, and pen type and color.

The operation of PLOTX is best explained with an example (see below). The dialog used by the program observes several conventions: items listed inside square brackets [ ] are default values or actions that will be obeyed if the question is answered with 'carriage-return'. Items listed inside parentheses ( ) describe admissible answers to the question being posed. Answers to yes or no questions may be abbreviated to Y or N.

The first question is whether the operation is to be 'automatic', or 'manual'. Automatic operation means that all files with extension .PLX on your disk area will be plotted. Any information included in the PLX file about how the plot is to be done (see the description of DPLOT, above) will be obeyed, but the user will not be prompted for additional commands. In 'manual' operation, PLOTX will engage in an initial dialog to permit changes to the plotting parameters associated with each PLX file. The discussion below assumes manual mode is being used.

Next, PLOTX wants to know which plotter you intend to use. The default is 1, the Calcomp.

Now you must provide the names of the PLX files which are to be plotted.

A response of 'carriage-return' will cause all files with extension 'PLX' in your disk directory to be plotted. They will be plotted in the same order that they are listed in the directory. If you wish to specify precisely which files are to be plotted, the names (without extensions) may be typed in. If the name is a 6-digit number generated by DPLOT, then only the number portion need be typed. Examples:

```
12,0,2
     ... will plot 000012.PLX, 000000.PLX, and 000002.PLX
1,nam
     ... will plot 000001.PLX, NAM.PLX
1,1,1
     ... will plot 000001.PLX, 000001.PLX, and 000001.PLX
```

The last example is useful when you wish to change the plotting parameters so that each plot is different (e.g. enlarge one plot, make one in red ink, etc.).

Now PLOTX will ask, for each of the files specified in the previous step, whether you wish to make changes to the plotting parameters. The default parameters are supplied by PLOTX unless the parameter portion of the PLX file specifies valid parameters (see DPLOT, above). The defaults are:

```
X scale factor: 1
Y scale factor: 1
X offset: 0
Y offset: 0
Paper type: 1          (standard)
Paper width: 11        (11-inch)
Pen type: 2            (felt tip)
Pen color: 1           (black)
Overlay: 0             (no overlay)
```

These options are almost self-explanatory. The scale factor is applied before the offset: the x coordinate of a point is x * scale + offset. The offset is measured in inches. The overlay option allows the plotter medium to be 'rewound' before plotting the next PLX file. The origins of the two sucessive plots will thus coincide.

The remaining dialog depends on which kind of plotting medium is selected; we shall assume that the off-line Calcomp has been chosen. In this case, PLOTX must write a magnetic tape to drive the plotter -- it asks for the number of a tape drive on which a scratch tape has been mounted.

It then asks for various accounting information: name, registered initials, account number, and box number. The 'registered initials',

'account number', and 'box number' are assigned to CCB users by the Project Control Office, Bldg. 12A, Room 3013. This information will be scribed on your plot so the Calcomp operator can collect accounting information.

Now PLOTX takes over and begins to write on magnetic tape appropriate codes to plot the files you requested. It will type out the name of each file as it begins processing; it will also type out the text of intervening messages to the Calcomp operator.

When all files have been examined and plots written on the magnetic tape, it is rewound. PLOTX then asks if you would like to have the PLX files deleted from your disk area. If you respond 'yes', it will delete only those which have just been plotted.

Below is a teletype listing from a sample PLOTX run. All underlined characters were typed in by the user; an underline as the last character on a line means that the user pressed the 'return' key at that point.

```
.DIRECTORY *.PLX

000000  PLX     3  <055>   16-Aug-72    DSKA:   [13,16]
000001  PLX     3  <055>   16-Aug-72

  Total of 6 blocks in 2 files on DSKA: [13,16]


.R PLOTX

AUTOMATIC OR MANUAL OPERATION (A OR M)?M
PLOTTER TYPE (1=CALCOMP, 2=COM, 3=ZETA)[1]?1
GIVE PLX FILES, IN ORDER [ALL, IN DIRECTORY ORDER]:
CHANGE PLOTX- OR FILE-SPECIFIED DEFAULTS FOR 000000.PLX?Y
  OMIT PLOTTING THIS FILE? [NO] _
  X SCALE FACTOR [1] _
  Y SCALE FACTOR [1] _
  X OFFSET (INCHES) [0] _
  Y OFFSET (INCHES) [0] _
  PAPER SIZE (11 OR 30 INCHES) [11] _
  PAPER TYPE (1=REGULAR, 2=VELLUM, 3=GRAPH) [1] _
  PEN TYPE (1=BALLPOINT, 2=FELTTIP, 3=WET INK) [2] 3
  COLOR (1=BLACK, 2=RED, 3=GREEN, 4=BLUE, 5=TURQUOISE
         6=PURPLE, 7=YELLOW, 8=ORANGE, 9=BROWN, 10=PINK) [1] _
  OVERLAY NEXT PLOT [NO] _
```

```
CHANGE PLOTX- OR FILE-SPECIFIED DEFAULTS FOR 000001.PLX?Y_
  OMIT PLOTTING THIS FILE? [NO] _
  X SCALE FACTOR [1] _
  Y SCALE FACTOR [1] _
  X OFFSET (INCHES) [0] _
  Y OFFSET (INCHES) [0] _
  PAPER SIZE (11 OR 30 INCHES) [11] _
  PAPER TYPE (1=REGULAR, 2=VELLUM, 3=GRAPH) [1] _
  PEN TYPE (1=BALLPOINT, 2=FELTTIP, 3=WET INK) [2] _
  COLOR (1=BLACK, 2=RED, 3=GREEN, 4=BLUE, 5=TURQUOISE
         6=PURPLE, 7=YELLOW, 8=ORANGE, 9=BROWN, 10=PINK) [1] 2_
  OVERLAY NEXT PLOT [NO] _
MOUNT A MAGNETIC TAPE. WHICH DRIVE (0,1 OR 2)?0_
NAME: SPROULL_
REGISTERED INITIALS: RFS_
ACCOUNT: XXXX_
BOX: YY_
MESSAGE: PDP-10 TAPE - 2 FILES - USE SINGLE PLOT
MESSAGE: NAME: SPROULL INITIALS: RFS ACCOUNT: XXXX BOX: YY
MESSAGE: USE 11 INCH REGULAR PAPER
MESSAGE: USE BLACK WET INK PEN
PLOTTING 000000.PLX
MESSAGE: USE RED FELT TIP PEN
PLOTTING 000001.PLX
FINISHED! DO YOU WANT TO DELETE PLOTTED FILES? [NO] Y_

EXIT
```

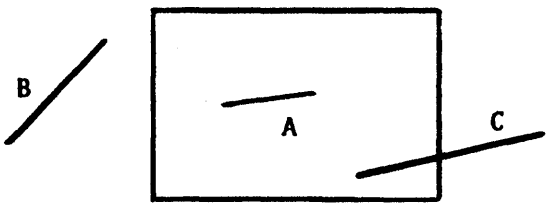


Figure R-1

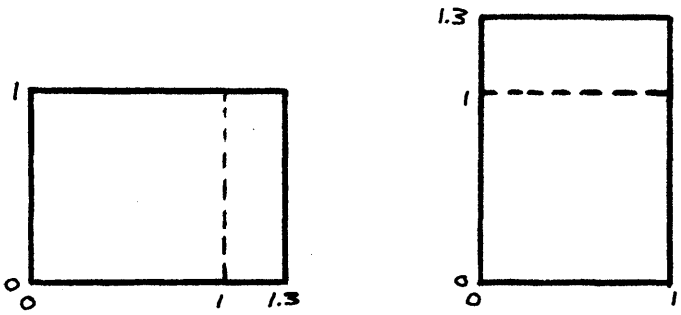


Figure R-2

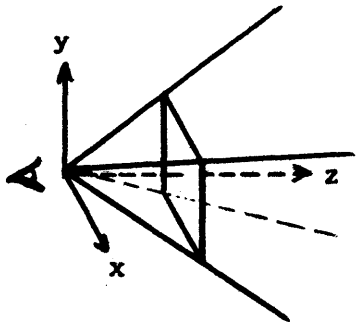


Figure R-3

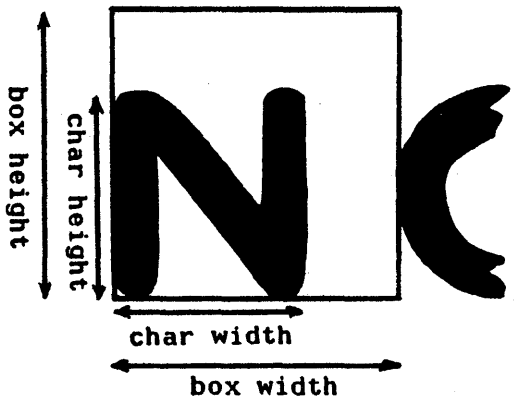


Figure R-4