8-30-1988

Solid modeling interactive interface for design drafting

Edward Maruggi

Rochester Institute of Technology

School of Computer Science and Technology

SOLID MODELING INTERACTIVE INTERFACE FOR DESIGN DRAFTING

by

Edward P. Maruggi

A thesis submitted to:

The Faculty of the School of Computer Science and Technology,

in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science.

August 30, 1988

This thesis entitled:

SOLID MODELING INTERACTIVE INTERFACE FOR DESIGN DRAFTING,

by Edward P. Maruggi,

has been approved by the members of the thesis committee and the Computer Science and Technology Graduate Studies Committee as duly noted by the signatures below.

Guy Johnson

Professor Guy Johnson
Thesis Committee Chair

Donald L. Kreher 9/6/88

Professor Donald L. Kreher
Thesis Committee Member

Peter G. Anderson

Professor Peter G. Anderson
CS&T Graduate Studies Committee Chair

Rochester Institute of Technology

Graduate Computer Science Department

Thesis:

SOLID MODELING INTERACTIVE INTERFACE FOR DESIGN DRAFTING

by Edward P. Maruggi

August 30, 1988

1.3. Abstract.

Providing certain Solid Modeling software systems with Computer-Aided Design Drafting facilities allows these systems to produce industry acceptable engineering drawing displays. A method is put forth to provide these facilities through the design and implemention of a two-dimensional interactive graphics package that interfaces with these systems. This allows other graphical and textual elements to be added to displays produced by the Solid Modeling systems.

1.4. Key Words and Phrases.

Computer Graphics
Solid Modeling or Solid Modelling
PADL-2
Interactive Computer Graphics
Two Dimension or 2 Dimension
Computer-Aided Design
Core System
Computer Graphics Standard

1.5. Computing Review Subject Codes.

Categories and Subject Descriptors: J.6 [**Computer-Aided Engineering**] – *computer-aided design (CAD)*; D.2.2 [**Software Engineering**]: Tools and Techniques – *modules and interfaces*; I.3.M [**Computer Graphics**]: Miscellaneous;

General Term: Design

TABLE OF CONTENTS

Preliminary Information

Section 1 – Introduction

Section 2 – Functional Specification

Section 3 – Architectural Design

Section 4 – Interface Design

Section 5 – Graphic Processor Design

Section 6 – Verification and Validation

Section 7 – Conclusions

Section 8 – Bibliography

Section 9 – Program Listings

Section 10 – User's Manual

1

INTRODUCTION

1.1. Background.

The advent of the computer has spurred many changes in society as mankind has struggled to put to use this powerful, yet sometimes enigmatic invention. This has been particularly evident in industry, where the computer has been employed to improve the quality and increase the quantity of goods and services. The application of Computer-Aided Design and Manufacturing (CAD/CAM) has been instrumental in bringing about these changes.

The ultimate goal of CAD/CAM is to produce goods efficiently by automating the production process. Fully automating the production process requires a means for the computer to understand the geometry of production parts. "The need for powerful computational means for dealing with geometry is widely recognized, and a variety of geometry programs and systems have been or are being developed by research laboratories, industrial users, and commercial vendors." [REQU80]. These have included Solid Modeling (SM) software systems which utilize three-dimensional solid geometry as a basis for describing objects to the computer and displaying them on various output devices.

SM systems have been developed primarily as research tools for mechanical CAD/CAM applications. By creating computer models of objects through the use of solid geometry, complex studies can more easily be performed on mechanical objects. "Models may be manipulated in many ways for component assembly, movement interference and analysis for mass and surface properties and resistance to stress." [SHER88].

While SM systems have proven useful as research tools, they have the potential to be useful for industrial applications as well. In order to realize this potential, SM systems must move beyond merely describing and displaying geometry. "One must abandon the concept of the solid or geometrical modeler and move forward to develop the true product modeler." [PRAT85]. This would allow the process of developing and designing a product to exist entirely within the modeler itself. "It is possible that in the very near future the complete process of designing a detailed part will not need any sketch or drawing produced, neither on a manual nor on an electronic drawing board." [JURI87]. This will only happen if solid modeling continues to advance toward product modeling.

To advance toward becoming true product modelers, SM systems must become more useful in other areas of CAD/CAM. One such area is Computer-Aided Design Drafting (CADD). CADD is used in industrial settings where part specifications must be communicated from the designer to manufacturing through the use of engineering design graphics. "Drafting and CAD exist to convey shapes, sizes, and physical descriptions graphically, so that physical ideas are conveyed unambiguously from one person to another." [OAKE88].

CADD was originally developed to replace traditional drafting tools such as the drafting board, scale, and pencil. For this reason, CADD systems were developed as wireframe systems, because this was the cheapest way to produce industry acceptable documents through the use of the computer. "A wireframe model is constructed of lines, arcs and circles, and is fairly common in CAD software." [OAKE88]. This requires less mathematical computations than constructing a solid representation of an object. SM, because of its use of higher level mathematical computations, uses more memory and processing, and thus is more expensive than wireframe systems.

Because the cost of integrated circuit technology used in computers has declined steadily in recent years, wireframe CADD systems have begun to incorporate more aspects of SM, although these systems are not yet fully automated. "Basic 3D CADD systems with solid modeling capabilities already exist, but it will take several more years to produce industry acceptable systems that will automatically produce hardcopy outputs in the form of engineering drawings." [JURI87]. Nevertheless, by incorporating

aspects of SM, these wireframe systems have become more than simple drafting tools; they have advanced toward becoming product modelers.

1.2. Problem Statement.

The advance of wireframe CADD toward product modeling by incorporating aspects of SM points to the fact that certain SM systems could advance toward product modeling by incorporating aspects of wireframe CADD. The kind of SM systems that could benefit the most from wireframe CADD technology are those that are unable to produce industry acceptable engineering drawings. This includes SM systems that were developed primarily for research in mechanical CAD/CAM. These systems generally have the ability to produce highly sophisticated 2D or 3D shaded or wireframe displays of objects, but are not able to add all of the necessary information to produce engineering drawing displays. This thesis solves this problem by providing an interactive graphics package (IGP) that interfaces with these systems to provide facilities similar to those provided by wireframe CADD systems. This allows for creating complete mechanical drawings of objects.

SM systems created as research tools run on standard computer systems consisting of a central processing unit (CPU), main memory, secondary storage, and an operating system in addition to the the SM software and a display device with vectoring capabilities. In order to support an interactive graphics package, the system must also provide programming language facilities and interaction devices. IGP interfaces with systems that have these characteristics.

1.3. Theoretical and Conceptual Development.

Providing SM systems with wireframe CADD facilities necessary to produce engineering drawing displays was accomplished by designing and implementing a software system that provides these capabilities. The software system is a two-dimensional interactive computer graphics package that can interface with a wide range of SM systems that do not have the ability to quickly and easily produce complete, descriptive, and

unambiguous engineering drawing type of displays. This provides an efficient means through which a user can enhance displays produced by SM systems for design drafting purposes.

The first step used in designing this system was to identify and categorize the requirements of the system. The following requirements were identified:

- From an architectural design standpoint, all the necessary capabilities must be provided without the system being unduly large.
- In the area of graphics processing, all the graphics capabilities that are essential to produce engineering drawing displays from SM system displays must be included.
- The user interface to these graphics capabilities must be user-friendly.
- To satisfy device interfacing requirements, provisions must be made for the control of a wide range of graphic devices in a uniform way.
- The SM interface must allow the system to be portable so that it can be used with more than one SM system.
- Capabilities that use a minimum amount of run-time and memory must be provided while satisfying the requirements of the system.

From each of these requirements a number of design goals that affected the usefulness of the system were formulated to guide the decisions made during the design process. It was discovered that a certain amount of trade-off existed between some of these goals.

1.3.1. Architectural Design Goals.

The architectural design goals were formulated to provide all the necessary capabilities in a structured and well-defined manner without the system being unduly large. These goals were defined in three areas: orthogonality, minimality, and compactness.

A system that satisfies the orthogonality goal is one in which the subroutines are independent of each other or the dependency is structured and well-defined. A system that satisfies the minimality goal is one in which only those subroutines needed at a given level should be provided at that level. A system that satisfies the compactness goal is one in which the desired result is achieved by using the smallest possible number of subroutines and parameters.

1.3.2. Graphic Processing Goals.

The graphic processing goals were formulated to provide all the graphics capabilities that are essential to produce engineering drawing displays from SM system displays. These goals were defined in two areas: completeness, and compatibility.

A system that satisfies the completeness goal is one in which all subroutines needed for a given level of graphic application are included at that level. A system that satisfies the compatibility goal is one in which other graphic standards or commonly accepted rules of practice are followed.

1.3.3. User Interface Goals.

The user interface goals were formulated to allow the system to be user-friendly. These goals were identified in the areas of simplicity, consistency, clarity, and robustness.

A system that satisfies the simplicity goal is one in which the typical user is able to understand and utilize all the capabilities of the system. This means that features that are unnecessary or too complex are omitted. A system that satisfies the consistency goal is one in which the general concepts related to the functionality of the system is easy to follow for the typical user. "Function names, calling sequences, error handling, and coordinate systems all should follow simple and consistent patterns *without exceptions* " [NEWM79].

A system that satisfies the clarity goal is one in which the concepts and functional specifications of the system are easily understandable from the standpoint of both system design and system description. A system that satisfies the robustness goal is one in which error reaction is clearly understandable and informative, and impacts the system as little as possible.

1.3.4. Device Interface Goals.

The device interface goals were formulated to provide for the control of a wide range of graphic devices in a uniform way. Two areas of device interface goals were identified: device independence, and device richness.

A system that satisfies the device independence goal is one which is able to address facilities of differing graphics output and input devices without modification of the application program structure. A system that satisfies the device richness goal is one in which the full capabilities of a wide range of different graphics input and output devices are accessible from the subroutines of the system.

1.3.5. Solid Modeling Interface Goal.

The SM interface goal was formulated in the area of portability. A portable system was found to be one that could be used with more than one SM system.

1.3.6. Functional Goals.

The functional goals were formulated so that capabilities that use a minimum amount of run-time and memory would be provided while satisfying the other system requirements as well. The functional goals were formulated in the areas of performance and efficiency.

"Graphics-system performance is often limited by such factors as

operating-system response and display characteristics, factors beyond the system designer's control" [NEWM79]. Satisfying this goal means that the affect of this on the system is minimized so that a consistent speed of response is provided for all functions.

A system that satisfies the efficiency goal is one in which a function of the system is able to achieve the desired result with a minimum usage of run-time and memory. "Graphics systems should be small and economical..." [NEWM79].

1.4. Equipment Configuration and Implementation Tools.

In order to demonstrate the functioning and usefulness of IGP, it was implemented for use with a specific SM system. This was the PADL-2 system running on the Rochester Institute of Technology Information Systems and Computing Infotron IS4000. This system was comprised of the PADL-2 PP2/1.1 solid modeling software, the VAX/VMS operating system, a cluster of VAX-11/785 super minicomputers with RA-81 and RA-60 hard disk storage, and VT-240 interactive display devices. In addition, the VAX-11 Pascal programming language was used to program the IGP software.

PADL-2 was developed at the University of Rochester Production Automation Project (see [TILO78]). The project was established in 1972 to do research in mechanically oriented CAD/CAM. The PADL-2 system is a direct result of this research. It is a software system that allows programming of solid objects through the use of the PADL-2 language for the purpose of modeling parts and assemblies, computing line drawings, shaded displays and mass properties, and verifying Numerical Control programs. "PADL-2 is a robust and powerful program that forms the basis of several leading mainframe and minicomputer-based solid modeling systems." [FINA88]. This makes it an ideal candidate to use with IGP.

In addition, the system requirements and goals to be followed during the design process were more easily satisfied by choosing an appropriate framework from within which to work. The framework was appropriate in the sense that it must be one that will greatly aided in the development of

IGP. For example, in order to meet the design goal of compatibility, commonly accepted rules of practice were followed, thus a commonly accepted framework, or a framework that uses commonly accepted practices was selected. This also made it easier for the user to understand and utilize the graphics capabilities of the system, which helped satisfy the simplicity goal. Choosing an appropriate framework also insured that IGP was not unduly influenced by the features of the specific hardware system or by highly specialized application requirements. This helped satisfy consistency, and device independence goals as well.

The two most commonly accepted graphics standards in use today are the Graphic Kernel System (GKS) and the Core System. The Core System was followed for this particular application because of the author's familiarity with Core due to his previous work with this standard.

The Core System was first developed in 1977 by the ACM/SIGGRAPH Graphics Standard Planning Committee. "It's functional capabilities include the areas of output primitives, viewing transformations, segmentation, input, and control." [MICV78]. Output primitives are used to describe objects to be displayed, viewing transformations determine how the objects are actually displayed, segmentation allows displays to be modified, input devices allow the user to interact with the system, and control of the general functioning of the system by the user is allowed. "The Core System is a package designed to be rich enough to support devices ranging from plotters, through storage tubes, to high speed refresh displays." [NEWM78]. A more complete explanation of the Core System can be found in [NEWM78], [MICF78], [MICV78], and [BERG78].

1.5. Previous Work.

Much work has been done in the past related to implementing the Core System for particular applications, however, an extensive search has failed to turn up any work related to implementing Core or a subset of Core as part of an interactive interface for a solid modeling software system.

A library search was performed at the Rochester Institute of Technology Wallace Memorial Library. The strategy for the search was as follows: On

the first pass the key phrases *computer graphics and structured programming, PADL-2, interactive computer graphics and 2 dimension or two dimension*, and *solid modeling or solid modelling and interface* were used to uncover relevant work in the areas of computer graphics and solid modeling. On the second pass the key phrase *Core and Computer Graphics and Standard* was used to discover relevant information and examples regarding implementations of the Core System.

The search uncovered two particularly interesting implementations of Core related to the topic of this thesis. [FRIE81] is interesting because both it and this thesis deal with a two-dimensional Core implementation. [NICO81] is helpful because, like this thesis, it uses the Pascal language to implement Core. The search also found the 4 works referenced in section 1.3 as explanations of the Core System.

2

FUNCTIONAL SPECIFICATION

2.1. Functions Performed.

IGP interfaces with a solid modeling software system to produce complete engineering drawing displays. Initially the solid modeler is used to geometrically describe a solid object and generate a two-dimensional or three-dimensional display of the object on the screen. IGP is then used to add additional graphical and textual information needed to create a complete engineering drawing document. The solid modeler and IGP have the ability to store and retrieve files for later use so that the entire document can be redisplayed as many times as desired and so that changes and updates to an existing drawing can easily be made.

2.1.1. The Function of the Solid Modeler.

The solid modeler is used to generate appropriate part displays on various output devices. This involves using whatever method the solid modeler allows to define the geometry of the solid object and how the object is displayed. The modeler displays the image of the object on the screen appropriately and then IGP is used to effectively complete the drawing. This involves using the modeler to draw a 2D or 3D image of the object that best describes the object's geometry for mechanical drafting purposes. Most often this requires a 2D multiview orthographic projection of the object. "Pictoral (three-dimensional) drawings of objects are sometimes used, but the majority of drawings used in mechanical drafting for completely describing an object are multiview drawings as shown in Fig. [2-1]." [JENS85].

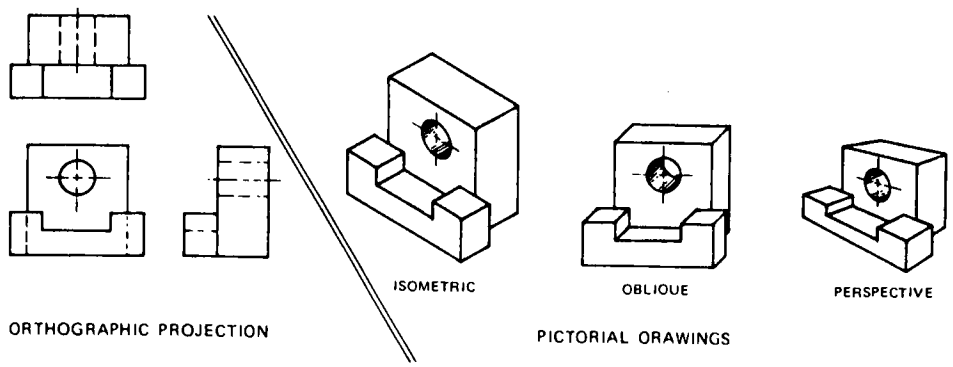


Figure 2-1. Types of projection used in drafting.

The modeler is also used to appropriately set the device display characteristics for the part projection. This includes characteristics such as line color and style, and background color. Certain kinds of engineering drawings may also require that the modeler remove hidden lines from the display.

2.1.2. The Function of IGP.

IGP interacts with the user through an input and an output device, processes graphic information, outputs graphic elements to a display device, and integrates this display information with display information produced by the SM system. IGP satisfies the design goals outlined in section 1.3 while performing these functions.

2.1.2.1. Interaction.

IGP provides an interface to allow the user to interactively create graphic elements for screen display. To satisfy design goals, graphic interaction gives the user easy access to the functions of IGP by using simple polling loops. The user is repeatedly prompted to make a selection until a proper selection has been made. If an improper selection is made, an informative error message is displayed on the screen. An error in selection by the user never results in an abnormal

termination of the program.

The design of the user interface allows for two kinds of interactive input. Input is made solely through the use of the alphanumeric keyboard, or by using the keyboard together with the graphic cursor. Graphic cursor input allows for faster and easier positioning of new items than with the keyboard. IGP provides appropriate feedback to the user through the use of alphanumeric output such as menus and messages, and graphic display of elements created by the user.

2.1.2.2. Graphic Processing.

The graphic processing design goal of compatibility is satisfied by following the basic features outlined in the Core standard. Core provides for five functional areas. They are: graphic primitives, windowing functions, segmenting functions, transformation functions, and control functions. All of the subroutines needed in each of these areas are provided.

Graphic primitives provide the ability to move the cursor and to create simple graphic elements. The graphic elements that can be created are points, lines, and text strings.

The window is the screen area where the display is produced. Windowing functions provide the ability to define the coordinate system, and the boundary of the visible portion of the display.

Segmentation functions allow selective modification of the IGP produced graphic elements by allowing the elements to be partitioned into groups that each comprise an engineering drawing element. Segmentation is necessary to allow deleting of elements.

Transformation functions allow rotation, translation, and scaling of IGP produced graphic elements. This allows easy creation of elements to the correct specifications.

Control functions provide the ability to control the display

characteristics of line style, line color, and text size. These are important considerations for creating engineering drawing displays.

To satisfy device interface design goals, IGP is capable of working on a number of different devices because graphic output goes through a device driving routine that can be rewritten for different devices. Presently this routine is written specifically for a VT-240 device.

2.1.2.3. Integration

To best satisfy SM interface design goals, integration between IGP and the SM system is through the screen. This method does not require modification of the SM software, thus satisfying portability goals.

2.2. Limitations and Restrictions.

IGP does not provide all Core graphic functions. The device driving procedure must be rewritten each time IGP runs on a different device. IGP is a 2D system only; the solid modeling system is responsible for any 3D display geometry.

IGP is used only after a display of an object is produced on the screen by the SM system. Therefore screen erasing and resetting is handled solely by the SM system because this is done prior to an image being produced by the modeler.

The performance of the overall system is limited by the capabilities of the SM system and the type of input and output devices available for use with the system. Although satisfying performance design goals means that the affect of this on IGP is minimized, the affect is not entirely eliminated. IGP is limited by the SM system's ability to produce appropriate part displays, by what type of output devices these displays are produced on, and by what type of input devices can be used interactively with the system.

At RIT, IGP is limited by the abilities of the PADL-2 PP2/1.1 software which was disseminated on June 3, 1983. More recent versions of this

software are able to produce more appropriate displays. Also at RIT, the VT-240's in the Ross User Computer Center are the standard devices used with PADL-2. Therefore, IGP is limited by the capabilities of the VT-240 for both output and interactive input.

Hardcopy is another area where IGP is limited by the lack of equipment available at RIT. The graphics printer for the VT-240's is a line printer that has been converted to produce graphic output through a dump of the VT-240 screen. This produces low-quality, low-resolution output that leaves much to be desired when the aim is to produce engineering drawings.

2.3. User Inputs and Outputs.

IGP is interactive, that is, the user is able to interactively create and manipulate graphics with IGP through the use of the keyboard and other interactive devices. The user must be familiar with engineering drawing practices in order to best utilize IGP as it was intended. The user produces engineering drawing document displays and screen dumps to hardcopy graphic devices if they are available.

2.4. System Files.

Separate files are produced by the PADL-2 portion of the system and the IGP portion. Files produced by PADL-2 are inverse translations of definitions created during a PADL-2 programming session and are automatically named with the .PFI extension. Files produced by IGP are display list data files produced and saved during an IGP interactive session. These files are automatically named with the .DAT extension.

3

ARCHITECTURAL DESIGN

3.1. Overview.

An overview of the design of the complete system that produces engineering drawing displays is shown by the System Organization Chart (Figure 3-1 on page 23). The existing SM system has its own user interface, file storage, and device interface. IGP exists alongside the SM and also has its own user interface, file storage, and device interface.

3.2. Design of IGP.

IGP's user interface is a collection of Pascal procedures that allow the user to interactively create graphic elements. The graphic processor is a group of Pascal procedures that are called on by the interactive routines to convert the interactive input to display processing unit (dpu) commands that can be saved to permanent storage. The device interface is a Pascal procedure that translates dpu commands into appropriate opcodes that are sent to the device for display purposes.

The flow of data through the IGP routines is shown in the Data Flow Diagram (Figure 3-2 on page 24). The data starts out as user input in the user interface. The routines in the user interface determine what kind of data the user has input and converts it to the appropriate data type. These routines then call the appropriate routines in the graphic processor. If required, the graphic processor routines perform a series of manipulations on the data for display purposes. Otherwise, the data goes directly to the device driver for conversion into device specific opcodes and coordinates to be sent to the device.

Data manipulated in the graphic processor that describe coordinates in an x-y coordinate system are multiplied by a current instance transformation for rotation, translation, and scaling of graphic elements. Then the data is multiplied by a viewing transformation that converts coordinates from the user's coordinate system to a conceptual device coordinate system. Coordinate systems used in IGP are discussed further in section 5.2.

The data may be altered by the clipping routine which keeps elements from being displayed outside the area of the screen designated for viewing. The data is then sent as a dpu instruction to a display list for storage or directly to the device driving routine. The device driver converts the data into device specific opcodes and coordinates that are sent to the physical device for display.

Figure 3-3 on page 25 shows the Structure Chart for the user interface routines. Structure Charts of the graphic processor and the device interface are not shown because they are not particularly helpful toward understanding the design of these. The graphic processor routines all exist at the same level except for OUTCODES, REJECTCHECK, ACCEPTCHECK, and SWAP which exist under the CLIPPER routine. The device interface, on the other hand, is comprised of only one routine. Interface design and graphic processor design is discussed in depth in sections 4 and 5.

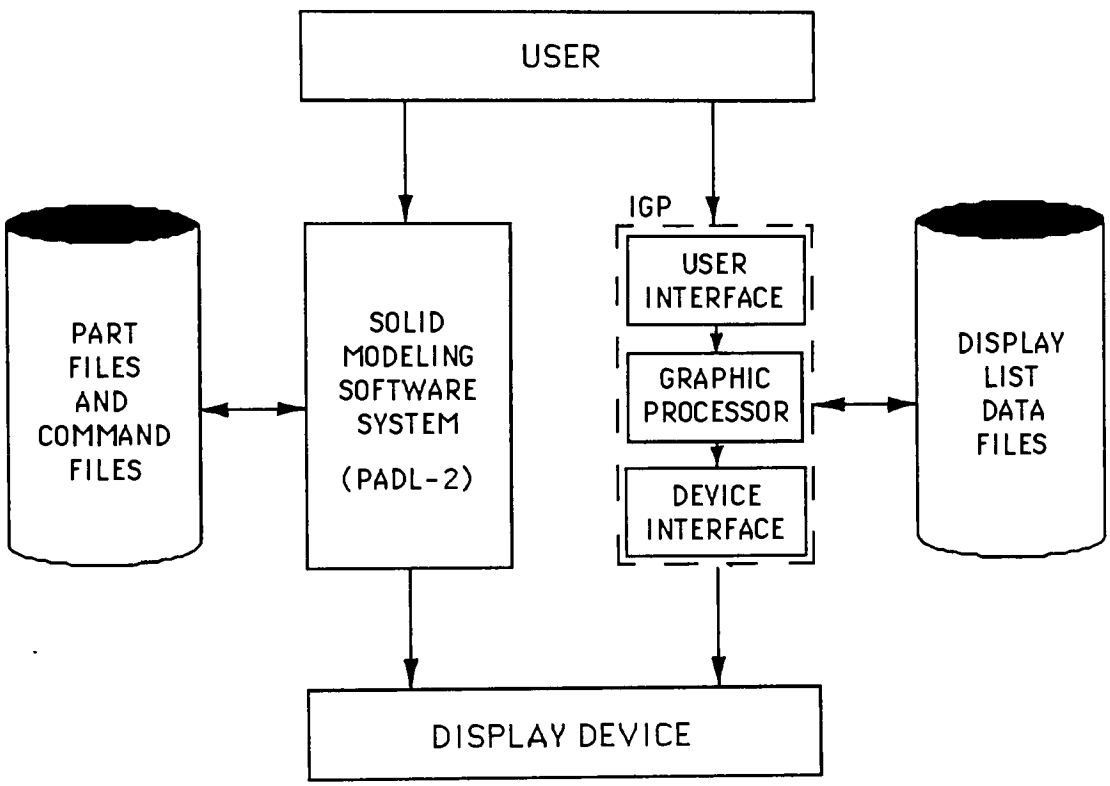


Figure 3-1. System Organization Chart.

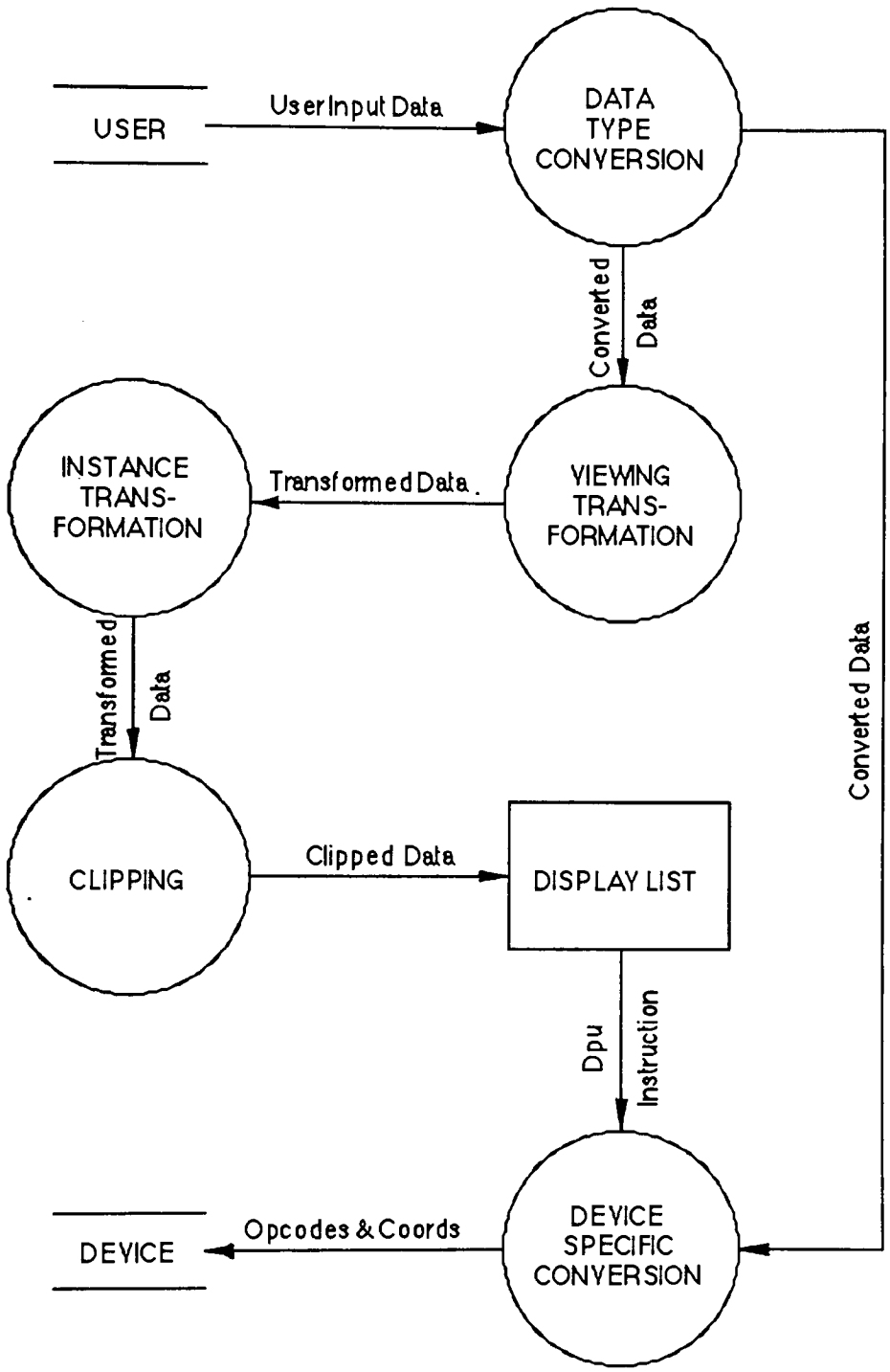


Figure 3-2. Data Flow Diagram.

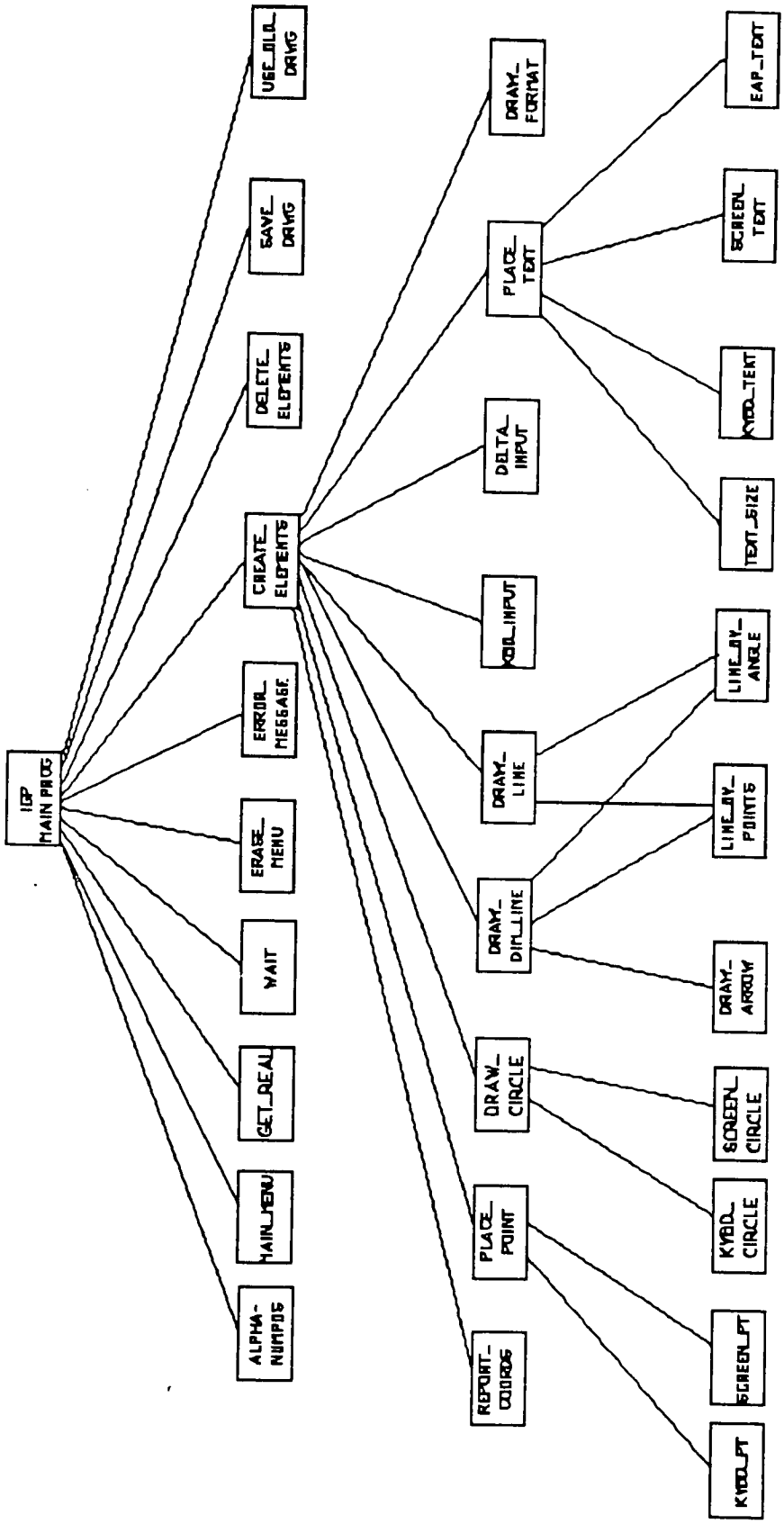


Figure 3-3. Structure Chart of IGP User Interface.

4

INTERFACE DESIGN

4.1. User Interface.

The IGP user interface allows interactive access to the graphic primitives provided by the graphic processor. It has 4 levels of functionality as shown by the structure chart in section 3.5.

4.1.1. Level 1.

Level 1 is the main program which prompts the user for input that defines the world coordinate system of the window. The main program can call any of the level 2 procedures. With PADL-2 it is necessary to erase command lines from screen when entering IGP. This is done by the main program. This section of code may need to be rewritten if IGP is used with other SM systems.

4.1.2. Level 2.

Level 2 has ten procedures. MAIN_MENU is found in the main program code while all other IGP procedures exist in the Pascal module known as coremod. MAIN_MENU erases any previous messages from the screen by calling the ERASE_MENU level 2 procedure, and posts the main menu at the location specified by the ALPHANUMPOS level 2 procedure. The main program then waits for the user to make an appropriate selection from the menu. The program will terminate if the user chooses the exit option.

IGP writes messages in an area that is just underneath an engineering drawing's border. In this way the messages are least likely to interfere with the drawing itself. If it is desirable to change the area where messages are written, the values for the constants alpha and beta may be changed.

If the user makes an inappropriate selection in any of the polling loops, the ERROR_MESSAGE procedure displays an error message and redisplays the main menu until an appropriate selection is made. The WAIT procedure causes a delay to occur whenever a message is displayed to allow time for the user to read it.

Any time the user is asked to input a real number, the GET_REAL procedure reads the input value as a character string, and checks to see if it is valid before converting it to a real number. This must be done because if the value were read as a real number and the user input a character string, the IGP program would abnormally end. Instead, if the user input is not valid, the procedure merely displays an error message and asks for the user to input the correct value.

The DELETE_ELEMENTS procedure allows the user to delete graphic elements that already exist. The user is prompted for the number of the element to be deleted. After an appropriate value is input, the corresponding element is highlighted, and the user is asked if that element is the one that should be deleted. If the answer is "yes", the element is deleted, and the next sequential element is highlighted for deleting. If the user answers "no", the highlighted element will not be deleted, and the delete loop will continue to the next sequential element. The user can escape the delete loop by choosing the exit option from the menu.

Two procedures in the user interface act as an interface to permanent storage. They are SAVE_DRWG and USE_OLD_DRWG. The procedure SAVE_DRWG saves a drawing's display list to the hard disk, and USE_OLD_DRWG copies an existing drawing's display list to the program's linked display list.

The CREATE_ELEMENTS procedure is the gateway to the level 3 procedures. First the create menu is displayed and, after the user makes an appropriate selection, the level 3 procedures are called. The

create menu offers the user 7 choices: creating points, creating lines, creating dimension lines, creating circles, creating text, creating a border, and exiting back to the main menu.

4.1.3. Level 3.

Level 3 procedures are called when the user makes an appropriate selection from the main menu. PLACE_POINT places a point at the specified location on the screen. DRAW_LINE draws a line between the specified end points. DRAW_DIM_LINE draws a dimension line between the specified end points. PLACE_TEXT places specified text at a specified location. These four procedures call on level 4 procedures to carry out their tasks.

A facility for drawing circles is also provided by IGP. The DRAW_CIRCLE routine uses the line drawing facilities provided by the graphic processor to approximate a circle according to the algorithm found on pages 51-2 in [BERG86]. The CIRCLE_DRAW routine accepts a radius and a center point location from the user and draws a circle. It also calls on level 4 procedures.

Other level 3 procedures do not call on level 4 procedures. REPORT_COORDS gets coordinates that are input by the user through the screen cursor. KBD_INPUT accepts absolute coordinates from the keyboard. DELTA_INPUT accepts delta coordinates from the keyboard. DRAW_FORMAT draws a border and title block on the screen.

4.1.4. Level 4.

PLACE_POINT calls two level 4 procedures. KYBD_PT places a point at a location specified by keyboard input and SCREEN_PT places a point at a location specified by screen input.

DRAW_LINE also calls two level 4 procedures. LINE_BY_POINTS accepts only end points as input to drawing lines, and LINE_BY_ANGLE accepts end points and an angle as input to drawing

lines.

DRAW_DIM_LINE calls each of these same line drawing procedures plus an additional procedure called DRAW_ARROW. This procedure takes the first end point of the line and the delta coords to the second end point, calculates the angle used in the transformation, and draws the arrow heads at the first end point.

DRAW_CIRCLE calls two level 4 procedures. KYBD_CIRCLE places a circle at a location specified by keyboard input and SCREEN_CIRCLE places a point at a location specified by screen input.

PLACE_TEXT calls four level 4 procedures. EAP_TEXT lets the user enter the correct textual information and place the text using the KYBD_TEXT and SCREEN_TEXT procedures. KYBD_TEXT accepts the location for the text from keyboard input while SCREEN_TEXT accepts the location for the text from screen input. TEXT_SIZE allows for specifying the size of the text that will be entered and placed.

4.2. Device Interface.

The device interface is the UPDATESCREEN procedure which interprets a dpu instruction and sends the appropriate opcodes and coordinates to the device. Presently this procedure is written specifically for a VT-240 device which interprets ReGIS code to produce graphic output. To utilize a different device, this procedure must be rewritten as well as parts of two other procedures: RESETVARS, and ALPHANUMPOS. This satisfies device interface design goals of device independence and device richness.

5

GRAPHIC PROCESSOR DESIGN

5.1. Graphic Primitives.

The graphic primitives based on the Core standard are the basic drawing facilities that IGP provides. These include four areas: locating the cursor, drawing points, drawing lines, and writing text.

5.1.1. Locating the Cursor.

Core allows for a current position to be maintained at all times. "It takes on values corresponding to the current location of an imaginary stylus in *world coordinate space*" [FOLE82]. The world coordinate space contains the conceptual user coordinate system. For a further discussion of coordinate space and coordinate systems see section 5.2.

Current position is important for the initial placement of graphic output primitives. It is necessary to have a facility to allow the current position to be moved to a specific location before displaying a graphic element. In IGP, this is accomplished through the use of the MOVEABS2, and MOVEREL2 routines. MOVEABS2 allows the current position to be located in absolute world coordinates while MOVEREL2 allows a new current position to be located relative to the old current position.

5.1.2. Drawing Points.

When creating precision drawings it is sometimes necessary to place

a point in world coordinate space for future reference. IGP provides the POINTABS2 and POINTREL2 routines for this purpose. The former accepts absolute positioning coordinates and the latter relative coordinates.

These routines also handle clipping of points, which is a relatively simple process. The coordinates of a point must be checked against the boundaries of the viewport. If the x coordinate falls in the range delineated by the minimum and maximum x viewport values, and the y coordinate falls in the range given by the minimum and maximum y viewport values, no clipping is done. However, if either of these conditions does not hold, then the point is not displayed. Clipping is described in greater detail in section 5.2.2.

5.1.3. Drawing Lines.

Line drawing is the most useful facility provided by IGP. The LINEABS2 and LINEREL2 routines accept absolute and relative coordinates respectively, and draw a line from the current position to the location described by the coordinates. The current position is then updated to the new coordinates.

5.1.4. Writing Text Strings.

The TEXT2 routine causes text to be written on the screen in graphics mode. A subroutine included in this routine is TEXTCLIP, which clips text that starts at a cursor position outside of the boundary of the visible portion of the display. Clipping is described in greater detail in section 5.2.2.

5.2. Windowing.

A discussion of windowing must first start with a definition of the coordinate space associated with the Core graphic standard. The user works in what is called a world coordinate system which is a 2-dimensional Cartesian coordinate plane. This is mapped to the

normalized device coordinate system by the graphic application routines. Normalized device coordinates are necessary in order for IGP to be device independent. This allows a conceptual IGP to map to a conceptual device which is then mapped to the specific device being used by the device driver (see Figure 5-1).

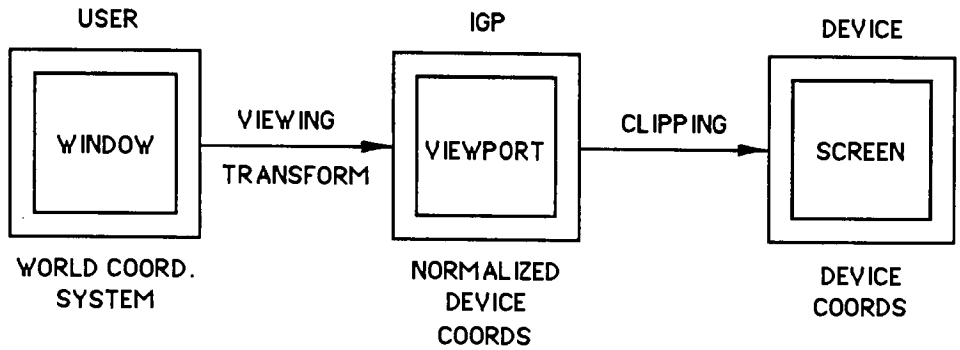


Figure 5-1. Mapping of IGP coordinate systems.

5.2.1. Windows and Viewports.

The WINDOW routine is responsible for defining the coordinate system while the VIEWPORT2 routine is responsible for defining the boundary of the visible display. Elements are clipped at the boundary specified with VIEWPORT2.

The window is set interactively by the user when IGP is first entered. The viewport is set by the application programmer in the RESETVARS routine. This will be different for different devices and SM systems, depending on how much of the screen is used for graphics. If not all of the screen is used, the other part can be used for menus and messages. With PADL-2, the entire screen is needed for graphics so VIEWPORT2 is set to min. x = 0, max. x = 1, min. y = 0, max. y = 1. Also, the menu is placed on the line just below where the border is drawn using the format option on the main menu.

5.2.2. Clipping.

Before an element can be displayed on the screen it must be first be checked and clipped if necessary. This is an important facility to include because if the viewport is defined as being less than the actual screen itself, then it is possible for elements to appear outside of the viewport if they are not clipped. The three kinds of graphic elements that may require clipping are points, lines, and text. Clipping of points and text were discussed earlier.

5.2.2.1. Clipping Lines.

Line clipping is done by the CLIPPER routine which is based on the Cohen-Sutherland clipping algorithm found on pages 148-9 of [FOLE82]. While Foley and Van Dam advocate that the algorithm be used to clip lines at the window, IGP varies from this in that clipping is done at the viewport. This is necessitated for large software packages such as IGP because with viewport clipping the mechanism of object building is easier to manage. Therefore the algorithm used in IGP follows a slightly different format.

After initialization of variables, CLIPPER uses a subroutine called OUTCODES to assign a four-bit outcode to the first endpoint of the line in question. The outcodes are based on the nine regions shown in figure 5-2. Each bit in the outcode is set to 1 (TRUE) if a given relation between the endpoint and viewport is true:

Bit 1—point is above viewport,
Bit 2—point is below viewport,
Bit 3—point is to right of viewport,
Bit 4—point is to left of viewport;

otherwise the bit is set to 0 (FALSE).

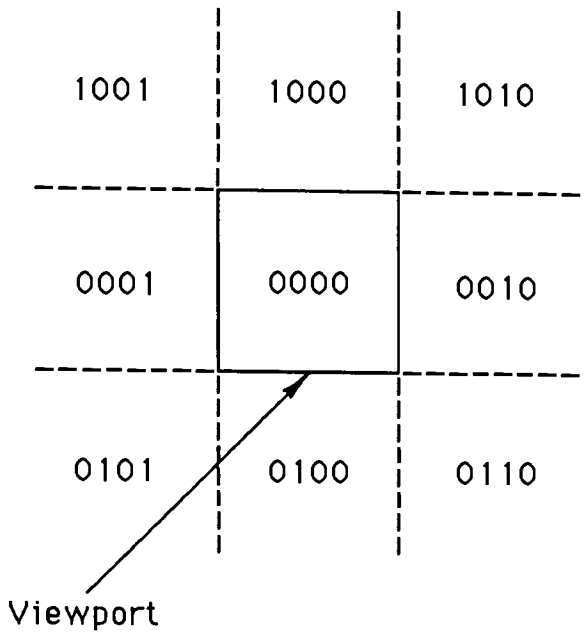


Figure 5-2. End point outcodes.

After an outcode is assigned to the first endpoint, OUTCODES is called upon again and the processes is repeated for the second end point of the line. The next step is to determine if the line can be trivially rejected based on its outcodes. Trivially rejection occurs if the line is completely outside the viewport. In this case, nothing more needs to be done. The function REJECTCHECK provides this facility.

If the line cannot be trivially rejected, then its outcodes are checked by the ACCEPTCHECK function to determine if it can be trivially accepted. Trivial acceptance happens if the line is completely inside the viewport. If this is the case then the line can be drawn in its true form as described by its endpoints.

If the line can be neither trivially rejected or accepted then at most, one endpoint is inside the viewport. If the first endpoint is found to be inside the window, then the subroutine SWAP is used to exchange the two endpoints so that the first endpoint is always outside of the

viewport. This simplifies the remainder of the clipping routine which performs a subdivision and moves the first endpoint to the location where the line intersects the viewport. Now the line can be drawn using the new endpoints.

CLIPPER also keeps track of the current position to insure that it is in the correct location after a clipped line has been drawn. For example, if the first endpoint is inside the viewport and the second endpoint is outside, the current position is placed at the intersection of the line and the viewport in order to keep it inside the viewport (see figure 5-3).

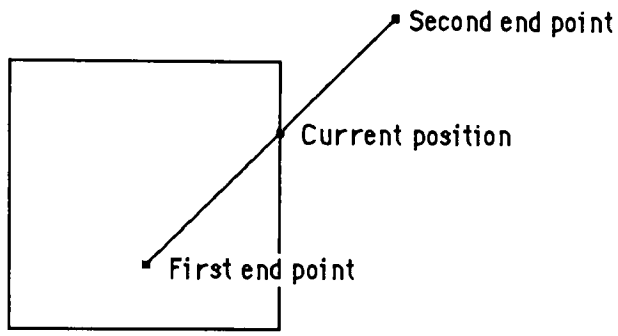


Figure 5-3. Current screen position after clipping.

5.3. Segmenting.

The facility of selective modification is necessary for the operation of IGP. This allows graphic elements to be moved or deleted without disturbing other elements. IGP accomplishes this through the use of its segmenting routines.

These routines allow graphic primitives to be combined in a data structure to form one graphic element called a segment which has a distinct address. To support segment storage a display file is maintained

by IGP. The display file is a linked list that holds all of the segments for a specific drafting file. This allows segments to be easily added and deleted.

The CREATESEG routine "opens" a segment for definition and assigns it a positive integer as its name. If a segment is currently open when this routine is invoked, it is closed first.

The CLOSESEG routine indicates the end of the definition of the currently open segment. The segment is then inserted into the display file in ascending order according the value of its number. If the segment has the same number as an existing segment, the old segment is deleted, and the new one is inserted in its place. If no segment is currently open, this routine does nothing.

DELETESEG deletes the record of the specified segment from the display file and erases the segment's graphic elements from the screen. If the segment does not exist, no action is taken.

POSTSEG searches the display file to find the specified segment and passes the drawing instructions stored there to the device driver for display on the screen. None of the graphic elements found in the segment are displayed on the screen until this routine is invoked. If this routine is invoked for a segment number that does not exist, no action is taken.

UNPOSTSEG searches the display file to find the specified segment and erases its graphic elements from the screen. This is done by setting the drawing color equal to the background color and redrawing the elements by invoking the POSTSEG routine. Following this action, the drawing color is returned to its original value. If the segment is not currently posted or if it does not exist, this routine does nothing.

5.4. Transformations.

Geometric transformations are used by IGP for two purposes: to map coordinates to and from the various coordinate systems (world, normalized, and device), and to allow scaling, rotation, and translation of drawing elements. The transformation routines modify the instance and

viewing transformations that are stored by IGP.

5.4.1. Viewing Transformations.

VIEWTRANSF sets viewport boundaries in normalized device coordinates for use in clipping. It is invoked whenever the window or viewport is changed. XTRANS and YTRANS transform x and y world coordinates to normalized device coordinates. RXTRANS and RYTRANS are the reverse of XTRANS and YTRANS; they transform x and y normalized device coordinates to world coordinates.

5.4.2. Instance Transformations.

Ideally it is desirable to have separate routines to control scaling, rotating, and translating graphic elements, but there are technical difficulties with this approach. In order to combine transformations that require translation, matrix addition is required, however, when combining transformations that require scaling or rotation, matrix multiplication is required. This presents a problem when an attempt is made to combine translation with either scaling or rotation. The way to simplify this problem is to include all three transformation functions in the same routine so that they can be represented by homogeneous coordinates in a three-by-three matrix. "If we express points in *homogeneous coordinates*, all three transformations can be treated as multiplications" [FOLE82].

The method used by IGP to implement this concept is to maintain a current transformation matrix. The world coordinates are multiplied by the current transformation matrix whenever drawing routines are invoked (see data flow diagram). This function is provided by the INSTTRANSF routine.

Other instance transformation routines perform operations on the current transformation matrix. The CLEARTRANSF routine sets the current transformation matrix equal to the identity matrix as shown in figure 5-4. The x-y coordinates do not change when multiplied by the identity matrix.

$$\begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}$$

Figure 5-4. The identity matrix.

The UPDATETRANSF routine premultiplies the current transformation matrix by a transformation consisting of a scaling, followed by a rotation, and then a translation. This produces an updated version of the current transformation matrix. It is important to note that these operations are done with respect to the origin of the world coordinate system. Figure 5-5 shows how this matrix is set up in IGP.

$$\begin{matrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{matrix}$$

where: a = x scaling X cos (angle of rotation)
b = x scaling X sin (angle of rotation)
c = - y scaling X sin (angle of rotation)
d = y scaling X cos (angle of rotation)
e = x translation
f = y translation

Figure 5-5. The current transformation matrix

IGP also maintains a transformation stack to allow transformations to be stored for later use. This makes complex object building easier. PUSHTRANSF pushes the current transformation matrix on the transformation stack, while POPTRANSF pops the top transformation

matrix from the transformation stack and discards it. If the stack is empty, POPTRANSF has no effect.

The RESTORETRANSF routine copies the top transformation matrix from the transformation stack and makes it the current transformation matrix. It performs a CLEARTRANSF if the stack is empty.

5.5. Control.

Controlling the display is an important feature. IGP handles two distinct display modes for this purpose: graphic and alphanumeric. Routines are provided for switching between the two modes so that the transition between interaction and drawing is smooth. The INITIALIZE routine allows IGP to exit alphanumeric mode and enters graphic mode, while the TERMINATE routine allows IGP to exit graphic mode and enter alphanumeric mode.

The RESETVARS routine initializes the variables for IGP. SETLINESTYLE changes the display characteristics for all subsequent lines drawn by IGP. Two line styles are provided: solid, and dashed.

SETCOLOR changes the drawing color for all subsequent graphic output. This routine is base on the availability of a red, green, blue (RGB) color palette to allow a wide range of color capabilities for display devices. The amount of red, green, and blue added on a scale of 0 to 1 determines the color characteristic of the color setting. For example, an RGB palette equal to (1,1,1) is the color white because white contains all of the visible colors of the spectrum. An RGB palette of (0,0,0) is black because black is the absence of color. The palettes (1,0,0), (0,1,0), and (0,0,1) are red, green, and blue, respectively.

5.6. Display List Generation.

Display list record updating is provided by the DIRECTINSTR routine which inserts graphic elements into a segment or sends them directly to the device driver. Inserting graphic elements into a segment means that these graphic elements are part of a larger element.

6

VERIFICATION AND VALIDATION

6.1. Test Plan.

A plan was developed to test the functioning of IGP together with PADL-2. This plan was as follows:

1) Define a complex assembly of parts using the PADL-2 language.

2) Build PADL-2 command files that produce appropriate displays of the assembly and some representative parts found in the assembly.

3) Use the command files to generate the PADL-2 displays and use IGP to add graphical and textual information necessary to produce an engineering drawing display.

4) Produce hardcopy output of the displays if possible.

6.2. PADL-2 Test Procedures.

A program was written using the PADL-2 language to define the assembly named BEARING. This object was chosen because it contained a representative sampling of features found on mechanical parts. Figure 6-2 on page 42 shows the engineering drawing from which the geometry was taken to write the PADL-2 program. Figure 6-3 on page 43 shows two pictoral displays of the completed BEARING produced on a Versetec™ plotter at the University of Rochester. 6-3 (a) is a shaded display and 6-3 (b) is a wireframe display.

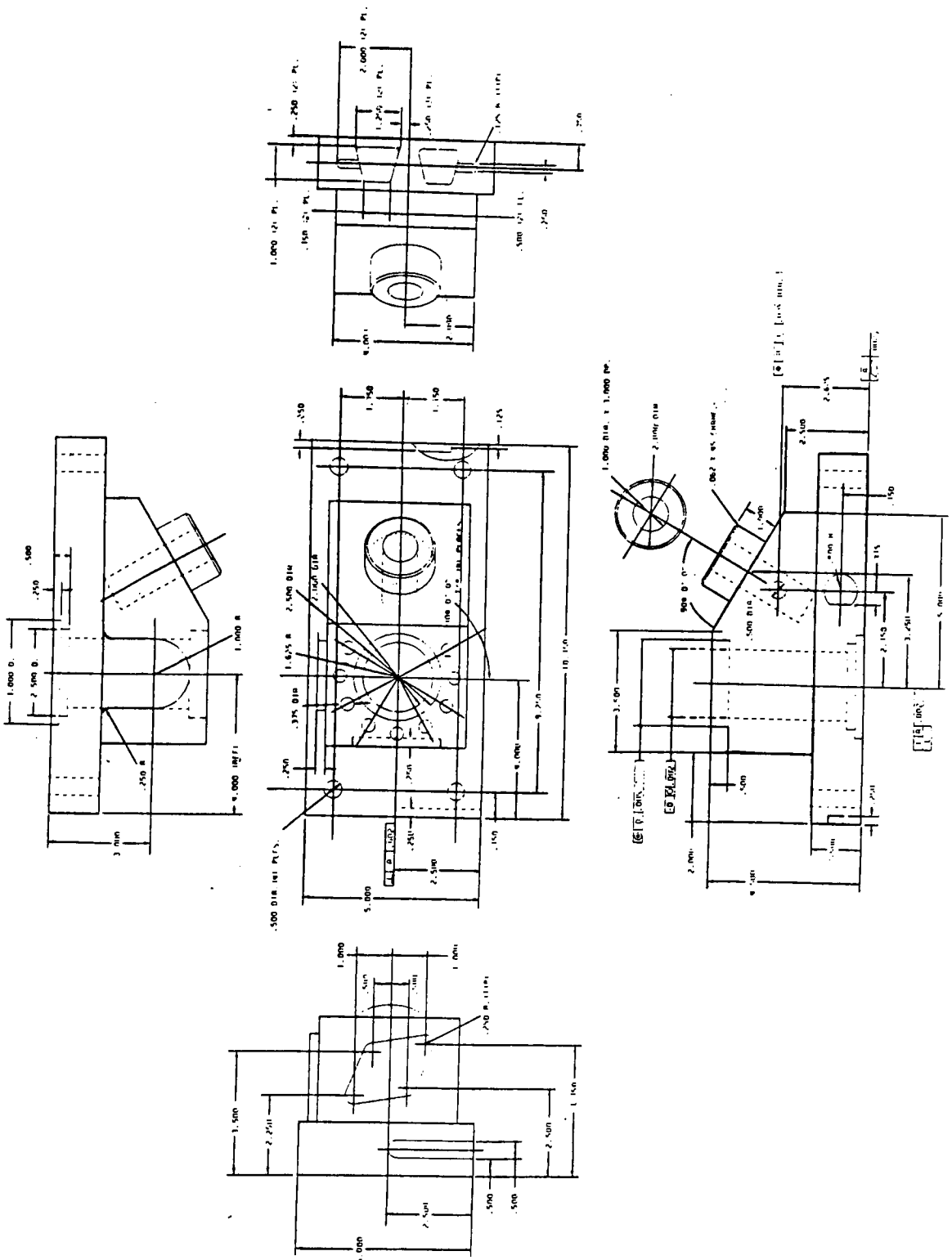


Figure 7-2. Drawing used to define BEARING using PADL-2.

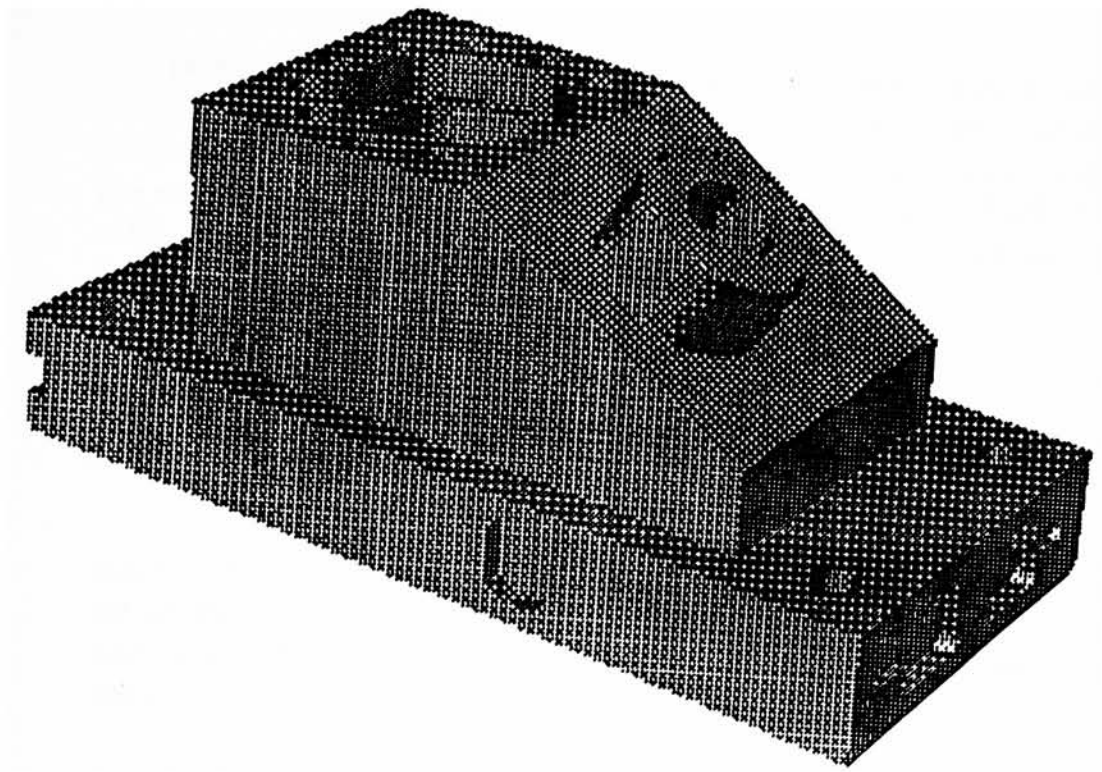
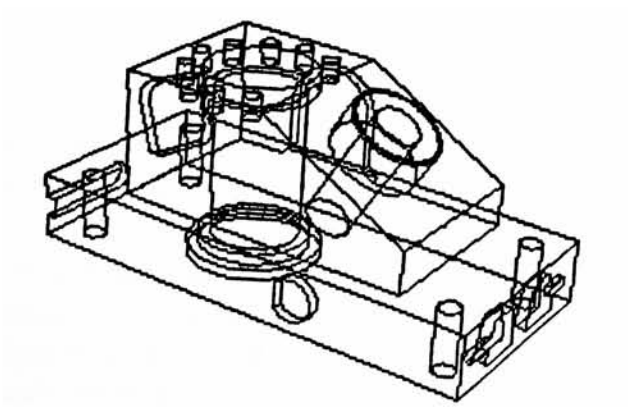


Figure 6-3. (a) Shaded display of BEARING.



(b) Wireframe display of BEARING.

6.2.1. Defining the Object.

Defining the BEARING through the use of the PADL-2 language involved using of four kinds of primitive solids that were combined in various ways to form the object. These primitives, by default, existed at the center of a Cartesian coordinate system modeled by PADL-2. The solid primitives used were: block, wedge, cone, and cylinder. The expressions used to combine solids to get new ones were:

<solid_expression> UN <solid expression>
<solid_expression> INT <solid expression>
<solid_expression> DIF <solid expression>
<solid_expression> ASB <solid expression>

where UN described the set union of two solids, INT described the set intersection of two solids, DIF described the set difference of two solids, and ASB allowed the solids to be associated together under a single name as in an assembly.

It was also necessary to scale, rotate, and translate certain solids before combining them with other solids. Scaling was given by setting x, y, and z in Cartesian coordinates equal to the desired scaling values in the parameter list of the primitive procedure call. For example, creating one particular scaled block was done by using the following definition:

BLO (X=2.5, Y=5, Z=4)

This primitive procedure call created a block that was 2.5 units wide, 5 units high, and 4 units deep.

Rotation and translation were also specified in a parameter list. For rotation in radians, ROTX, ROTY, and ROTZ were used while DEGX, DEGY, and DEGZ were used to define rotation in degrees. For translations, MOVX, MOVY, and MOVZ were used.

Coordinate systems and motions were also used to accomplish rotation and translation. The default coordinate system, named LAB,

existed at the origin, but other coordinate systems were created relative to LAB. Primitives and more complex solids were moved from one coordinate systems *to* another by using TO as a delimiter between the coordinate system names. They were also moved *with respect to* another coordinate system by using WRT as the delimiter. Rotation was accomplished by performing a *transformation* on a coordinate system using XFRM <coordinate system> BY <motion>.

The collection of PADL-2 statements that were used to define the characteristics of each part were brought together under what PADL-2 calls a *generic*. Each generic was assigned a distinct name to differentiate it from other generics. Each PADL-2 part was referred to by the name of its generic.

The PADL-2 definition of BEARING was comprised of six generics: BEARING, BASE, TOP, GUIDE, BLTHOLE, and ROUND. The BEARING generic was an assembly (ASB) of four generics. Three of these generics, BASE, TOP, and GUIDE were definitions of distinct parts while the fourth, BLTHOLE, represented the absence of a solid because it was subtracted from the assembly to produce the completed object.

The ROUND generic was a primitive instance used by the other generics. A primitive instance was found to be any user-defined primitive that can be scaled, rotated, translated, and used by other generics. This was done by passing parameters to it in the same manner as with the solid primitives provided by PADL-2.

The parameters R, A and T were passed to ROUND to describe the exact size. R defined the radius of the round, A was the angle of the corner of the round in radians, and T was the depth or thickness of the round along the z-axis. The output of this primitive was an object which, when differenced (DIF) with the original object allowed a cylinder to be unioned (UN) with the object to produce a rounded corner. This cut object was created at the LAB coordinate system with one of the edges coinciding with the x-axis, and was positioned at the proper corner before differencing.

After programming the object in the PADL-2 language, the PADL-2 system contained a three-dimensional description of the object. The next step was to produce appropriate displays. An appropriate display was defined as being a three view orthographic projection.

6.2.2. Producing Displays.

PADL-2 command files were created and used to produce orthographic projections of the GUIDE, BLOCK, and BEARING. The files were set up to display the required surfaces of each part at the appropriate locations on the screen. This required that each surface for each part be displayed only after the part was rotated so that the surface was parallel to the screen. PADL-2 display commands and view projection parameters were used to accomplish this. Displaying each surface at the appropriate location involved setting display characteristics such as window and viewport sizes and locations. The commands most useful for this purpose were: DISP, UGB, and SET.

6.2.2.1. The DISP Command.

Wireframe representations of the solid objects were displayed using the DISP command. When this command was used, PADL-2 calculated the profile lines of the named object, and displayed them on the previously defined device. For the PADL-2 at RIT software, using the DISP command caused problems with the display that required resetting the terminal under the set-up key in order for the IGP display characteristics to work properly.

6.2.2.2. The UGB Command.

The PADL-2 *graphic box* was found to be a conceptual area that PADL-2 used to display an object on the screen. If a line was drawn at a length too large to fit in the graphic box, PADL-2 automatically enlarged the box to accommodate the line. The UGB command set the center point of the screen equal to the center of the graphic box and set the view size equal to half the diagonal of the graphic box. This was

done so that when the image was displayed, it would be sized properly to take advantage of the full width and height of the screen.

6.2.2.3. The SET Command.

The SET command provided new default values for system parameters, including graphic parameters. The syntax was:

SET <parameter list>;

The set command was combined with DEVICE= to provide a facility for specifying device parameters.

DEVICE = <positive integer> opened the output graphic device, set the background color, and defined the display area to be used. The positive integer was defined as a six digit number, xyXYdD. xy specified the lower left corner of the display, XY specified the upper right corner, and dD indicated the background color and device.

The SET DEVICE= command was invoked before PADL-2 displayed any graphics on the screen. The PADL-2 display area was found to be analogous to the viewport provided by IGP. One difference was found, however: in IGP, a viewport that comprised the entire screen went from x = 0 to x = 1 and y = 0 to y = 1, while in PADL-2 , a full-screen display area went from x = 0 to x = 10, and y = 0 to y = 10 (see Figure 6-1).

For each display, the x and y values, and the X and Y values used with the DEVICE= command were set to a positive integer between 1 and 10. Here the number 0 was used to denote the integer 10 because this was what the PADL-2 format allowed. Also, x was less than X and y was less than Y to allow the viewport size to have a value greater than zero. This was necessary in order for a display to appear.

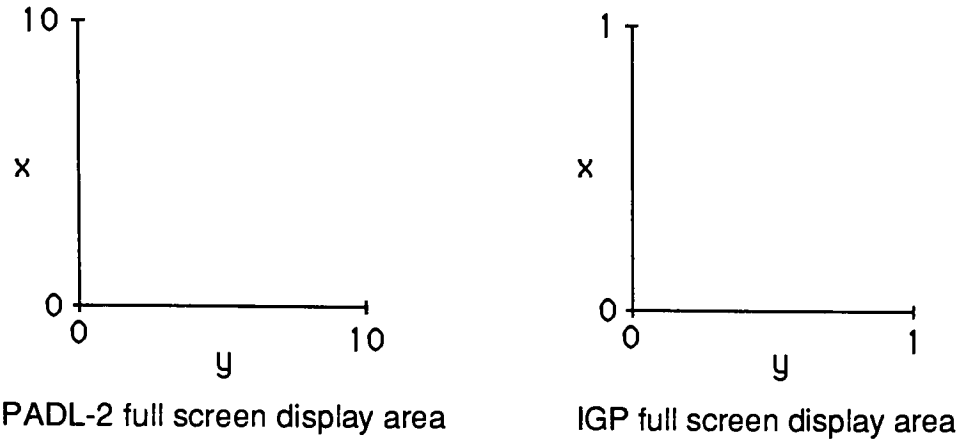


Figure 6-1. Display areas.

Setting the device equal to the VT-240, the device available at RIT for use with PADL-2, was specified in the DEVICE= command by letting the integer 6 be the value of D. According to the PADL-2 manual, the value of d should represent the background color for the particular device being used. However, with the set-up at RIT, d specified the line color, and the background color remained black at all times. d values were tested to find their corresponding line colors and the results are shown in Table 6-1.

set device =	IGP color (before disp)	PADL2 color (after)
6	green	black
16	blue	green
26	green	red
36	green	black
46	red	blue
56	black	blue
66	black	green
76	blue	red

Table 6-1. Line colors.

From the table it can be seen that using a d value of 0 or 3 produced no visible display because the line color was the same as the background color. Also, it was found that setting d = 7 was not appropriate because this caused subsequent graphic elements to be drawn in red. It was found that IGP was not able to highlight red elements for deleting because this same color was used for all highlighting.

Three files were created as PADL-2 command files to produce the appropriate displays. The DISP_GUIDE.PFI file was created to display the GUIDE, the DISP_BASE.PFI file was created to display the BASE and the DISP_BEARING.PFI file was created to display the BEARING. These files used the UGB command which caused PADL-2 to calculate how to neatly fit the object on the screen.

The SET DEVICE = 16 command was used to erase the screen and initialize display and mapping parameters for a VT-240 device. The screen area was divided into three viewports for displaying the top, front, and right side views of the part. The BASE went through a series of rotations and displays using the DISP command to generate the appropriate views.

IGP was entered from PADL-2 through the DCL interface by including the command DCL 'R IGP' each command file. This command spawned a VAX/VMS subprocess that executed the IGP program without leaving PADL-2.

6.3. IGP Test Procedures.

The capabilities of IGP were tested using the displays produced by the three PADL-2 command files. The three examples ranged from simple to highly complex. First a complete engineering drawing display using IGP was generated from a display of the GUIDE as an example of a relatively simple exercise. Then a display of the BASE was used in an attempt to generate a complete engineering drawing for a more complex part. The total BEARING part, consisting of an assembly of the GUIDE, BASE,

TOP, and BLTHOLE parts was then used to determine if IGP could be used to complete an assembly drawing of a highly complex part. The next section explains how IGP was used for these purposes.

6.3.1. Using IGP.

Upon entering IGP the user was reminded to reset the terminal under the set-up key in order for the IGP display characteristics to work properly. The user was then prompted for input values to set up the coordinates of the user's drawing area. Then the main menu was displayed on the screen.

IGP wrote these messages and all other messages in an area that was just underneath where an engineering drawings' top border would go. In this way the messages were least likely to interfere with the drawing itself.

There were six options under the main menu: USE, SAVE, CREATE, DELETE, ERASE MENU, and EXIT IGP. The user chose one of these functions by entering the number that corresponded to the desired option.

6.3.1.1. The USE Option.

The USE option allowed the user to input an existing drawing to the display area. The user was prompted for a file name (without the extension) of a file that resided in the current directory. If an attempt was made to access a file that did not exist or did not reside in the current directory, IGP displayed an error message.

6.3.1.2. The SAVE Option.

The SAVE option allowed the user to save the elements that were created and displayed. The user was prompted for a file name (without the extension). A display file by that name was created in the current directory with the .DAT extension.

6.3.1.3. The CREATE Option.

The CREATE option allowed the user to create new graphic elements and display them on the screen. There were several levels under this option that provided all of the capabilities necessary to create graphic and textual elements. The create menu was displayed on the screen so the user could select more options. The options under this menu were: POINT, LINE, DIM LINE, CIRCLE, TEXT, BORDER, and EXIT.

Points were created and displayed at specified locations by using precision input through the keyboard, or by using the screen crosshairs. Lines were specified by end points or endpoints and an angle. Input was possible through the keyboard or through the use of the screen crosshairs. Also, the style of future lines was set under the line menu.

Dimension line options were the same as line options except that dimension lines were drawn with an arrow at one end or an arrow at both ends, while ordinary lines were not drawn with arrows at either end. Circles were drawn using the circle option, and text was entered and placed, or the size of future text was set using the text option. The text size option did not work for the VT-240.

The border option was used to draw a border around the drawing. By typing in the appropriate selection from the CREATE menu, the border was automatically drawn.

6.3.1.4. The DELETE Option.

The delete option allowed the user to delete specified elements. The user was prompted for the number of the element to be deleted. After an appropriate value was input, the corresponding element was highlighted, and the user was prompted for a response. A "yes" response allowed the element to be deleted, and the next sequential element was then highlighted for deleting. A "no" response meant

that the highlighted element was not deleted, and the delete loop continued to the next sequential element. The user escaped the delete loop by choosing the exit option from the menu.

6.3.1.5. The ERASE MENU Option.

The user erased the menu using the ERASE_MENU option so that only the drawing elements themselves were displayed on the screen. This was done prior to making a hardcopy screen dump to the output device. Screen dumps were made by simultaneously holding down the Shift and Print Screen keys on the VT-240 keyboard for a device that was connected to a graphics printer.

6.3.1.6. The EXIT IGP Option.

Choosing the exit IGP option terminated the IGP program and returned the user to PADL-2. The user exited PADL-2 by using the STOP command.

6.4. Test Results.

To demonstrate the outcome of the tests, hardcopy output of the GUIDE, BASE, and BEARING displays was produced by a screen dump from a VT-240 to a graphics printer in the Ross User Computer Center at RIT. These drawings are shown in Figures 6-4, 6-5, and 6-6 on pages 54, 55, and 56. The GUIDE is an industry acceptable engineering drawing, while the other two are not.

Problems occurred in producing scaled drawings, and in placing the title block information in the correct location. The size of the PADL-2 displays were based on fitting a PADL-2 defined object to the graphic box rather than drawing the object to a specified scale. The was done to allow the display to be as large as possible to maximize the use of the relatively small VT-240 screen. Since the PADL-2 produced display was not drawn to a particular scale, the IGP produced display was not to scale either.

In the case of the title block, fitting the object to the graphic box did not allow room for the title block to be placed in the usual lower right-hand corner of the drawing. If a device with a larger viewing area was used, the display parameters could be set manually to allow the display to be to a particular scale.

A hidden line remover was needed to unclutter the complex BASE and BEARING drawings, but none was available. This was particularly true for the BEARING which was an assembly drawing. Assembly drawings are not supposed to show hidden lines. For the BASE, a facility to allow changing hidden lines from solid to dashed would have been even better than simply removing the hidden lines.

The hardcopy output lacked certain qualities that are desirable for precision drawings. The resolution was poor, there was no ability to produce large drawings, the hardcopy was not a one-to-one translation from the screen to the paper, and there was no facility for producing scaled drawings. This was due to hardware limitations, and was beyond the scope of IGP.

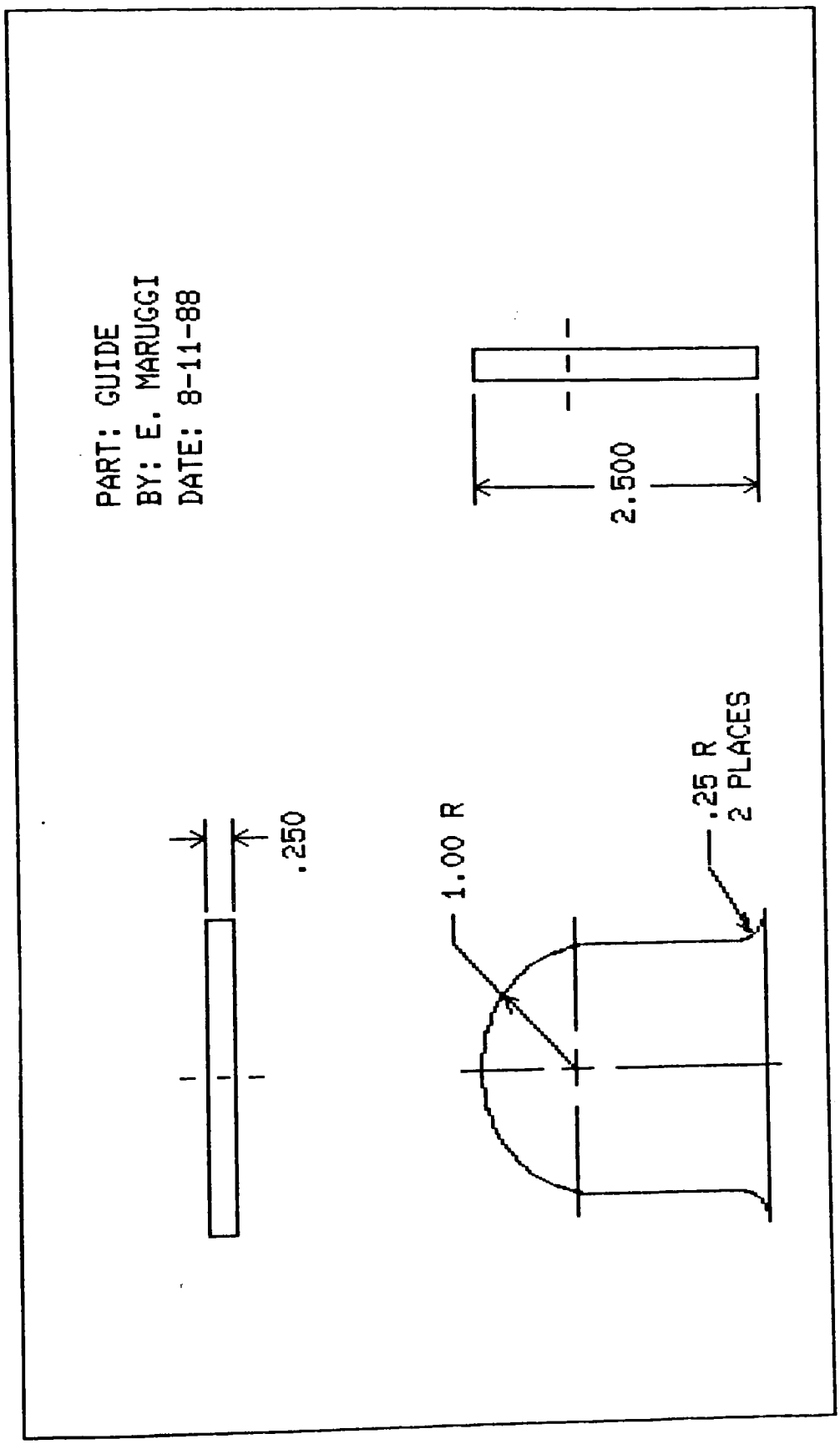


Figure 6-4. Engineering drawing of GUIDE.

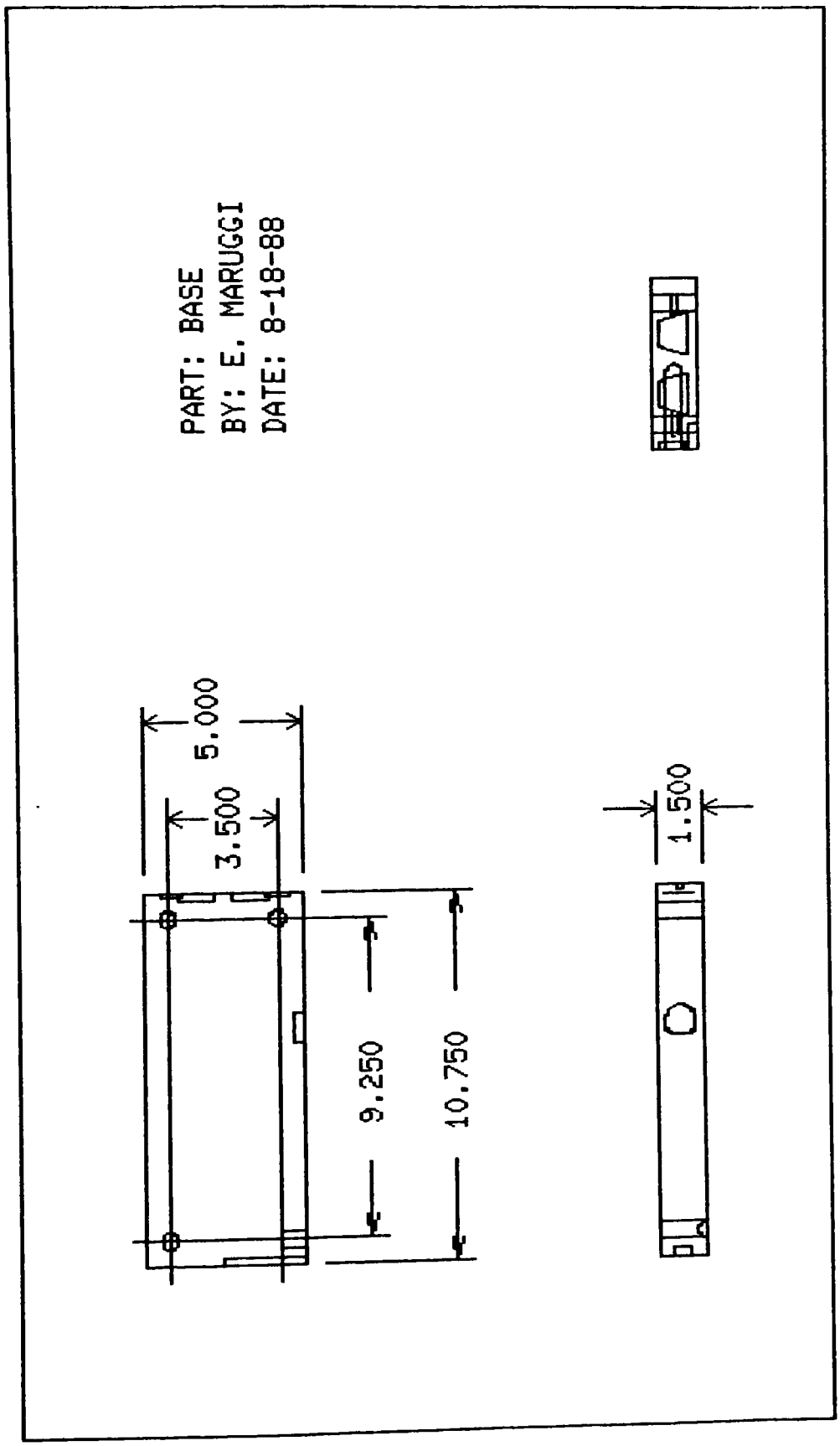


Figure 6-5. Engineering drawing of BASE.

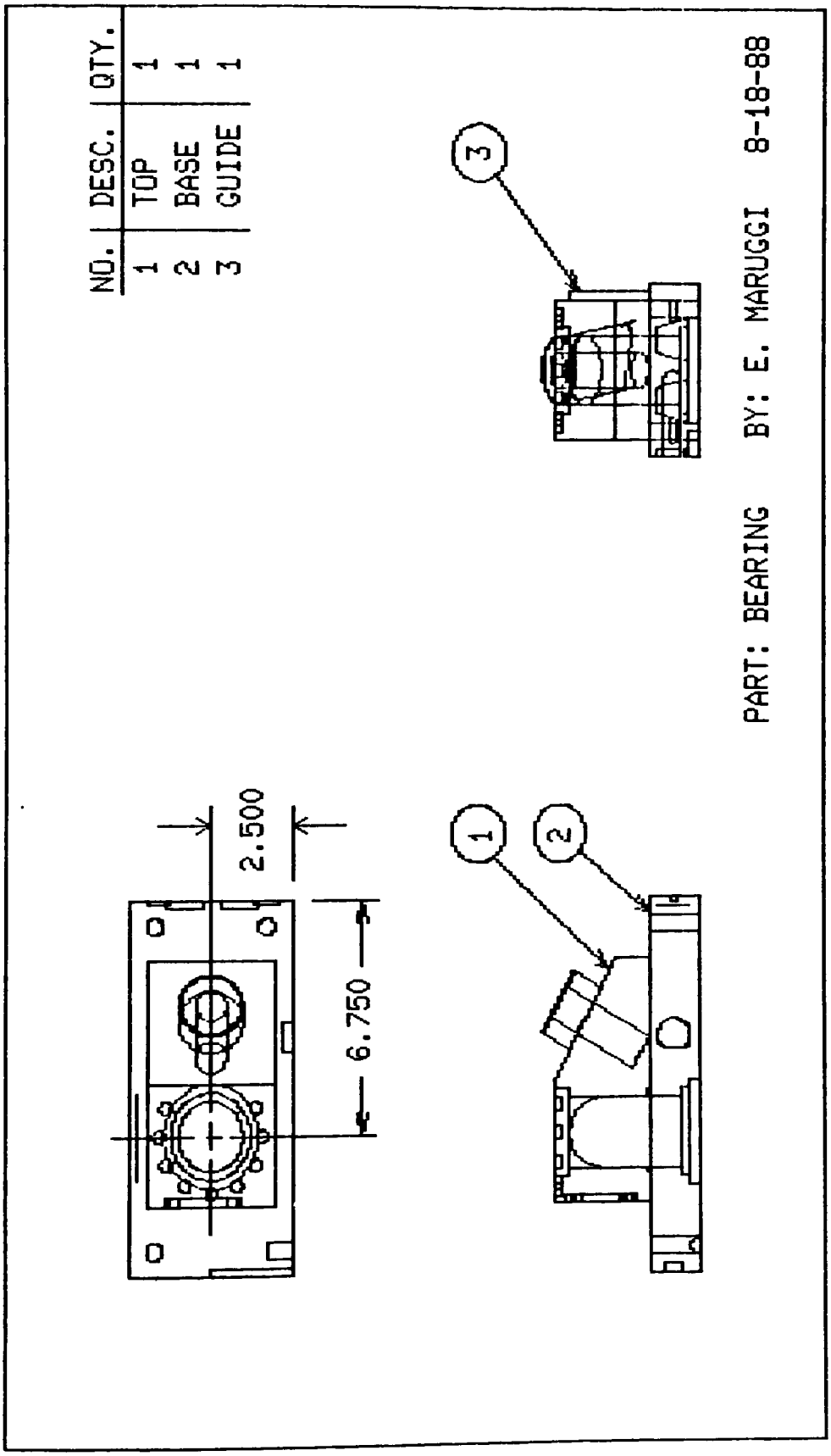


Figure 6-6. Engineering drawing of BEARING.

7

CONCLUSIONS

7.1. Problem Solution.

A two-dimensional interactive graphics package can be designed and implemented to provide certain Solid Modeling software systems with the necessary wireframe Computer-Aided Design Drafting facilities to produce engineering drawing displays. The kind of system best suited for this application is a structured program that allows for the integration of 2D graphic elements together with the image displayed by the Solid Modeler to occur through the video screen of the display device.

Structured programming techniques can be employed to insure that all system requirements are met. By breaking down the system into areas of functional similarity, the design process can more easily be accomplished and the design goals can more easily be met.

7.1.1. Architecture.

Architectural design goals were met by designing IGP with three areas of functionality, with each area having its own subroutines. The three areas are: the Graphics Processor, the User Interface, and the Device Interface. The Graphics processor contains the graphic application and geometry manipulation routines, the User Interface contains the interactive routines, and the Device Interface contains the device driving routine.

To satisfy the orthogonality design goal, the subroutines in each of these areas are independent of each other or the dependency is

structured and well-defined. To satisfy the minimality goal, only those subroutines needed at a given level are provided at that level. To satisfy the compactness goal, the desired result is achieved by using the smallest possible number of subroutines and parameters.

7.1.2. Graphic Processing.

Graphic processing design goals were met by providing graphics capabilities in the areas of graphic primitives, windowing, segmenting, transformations, and control. The design goal of completeness was met by providing all the subroutines needed for the various levels of graphic processing in each of these areas. The design goal of compatibility was satisfied by following the Core Graphics Standard.

7.1.3. User Interface.

The user interface is user-friendly. The design goal of simplicity is satisfied because the typical user is able to understand and utilize all the capabilities of IGP. Features that are unnecessary or too complex have been omitted. The design goal of consistency has been satisfied because the general concepts related to the functionality of IGP are easy to follow for the typical user. The design goal of clarity is satisfied because the concepts and functional specifications of IGP are easily understandable from the standpoint of both system design and system description. The design goal of robustness is met because error reaction is clearly understandable and informative, and impacts the system as little as possible.

7.1.4. Device Interface.

The device interface satisfies the device independence goal because IGP is able to address facilities of differing graphics output and input devices without modification of the application program structure. The design goal of device richness is satisfied because the full capabilities of a wide range of different graphics input and output devices are accessible from the subroutines of IGP.

7.1.5. Solid Modeling Interface.

IGP is presently implemented on the PADL-2 SM system, but can be used with other SM systems as well. This satisfies the portability goal.

7.1.6. Functional Specification.

Capabilities that use a minimum amount of run-time and memory are provided while satisfying the requirements of the system. The performance design goal is satisfied because a consistent speed of response is provided for all functions. The efficiency design goal is satisfied because IGP functions are able to achieve the desired result with a minimum usage of run-time and memory.

7.2. Discrepancies and Shortcomings of the System.

Although IGP meets all of the design goals, there are some shortcomings with this approach. For example, IGP depends on the SM system to draw a "good" representation of the object: an orthographic projection with hidden lines shown as broken lines or hidden lines removed. Also, IGP has a difficult time handling complex objects. The windowing facility needs to be more effective to do more detailed work, however, it is difficult to have good windowing when integration of IGP and the SM system is through the screen.

Other software related problems include the fact that setting the size of text does not work for the VT-240, and there are not enough line styles available for use in IGP. Engineering drawings can require as many as seven different styles of lines, and IGP provides only three: solid, dashed, and dimension.

The system also has hardware related shortcomings. High-quality hardcopy output is essential to any electronic drafting system, and none is available at RIT for this particular implemention of IGP. Also, the system would greatly benefit if a display device with a larger screen and a

more user-friendly interactive input device were available. If a larger screen were available, the title block could be placed in the lower right-hand corner of the drawing.

7.3. Lessons Learned.

The approach taken to provide wireframe capabilities to SM systems was a complex undertaking similar to designing and implementing a complete CADD system from scratch. Solving the problem meant breaking it down into smaller pieces that were functional subsets of the original problem, however it was difficult to decide where and how to break the problem down. One thing that proved particularly useful was studying the methods of those that went before down a similar path.

IGP was designed as an interactive graphics package, so it was helpful to put to use techniques learned in the Computer Graphics course taken in the Graduate Computer Science Department at RIT. It was also helpful to put to use techniques learned in reading through the myriad technical articles and books related to Computer Graphics, Interactive Computer Graphics, Computer-Aided Design, Computer-Aided Design Drafting, and Solid Modeling theory and applications.

CADD systems are largely dependent on hardware to produce high-quality precision output. A well-designed system for producing engineering design graphics is only as good as the display, interactive input, and hardcopy output devices that are available for use with the system. Of course, it is also dependent on the processing and storage capabilities of the CPU and hard disk set-up as well.

7.3.1. Alternative Approaches for an Improved System.

One alternative approach would be to have IGP capture the graphic output produced by the SM system and place it in its own display file. In this way all of the graphic output shown on the screen would be contained in the current IGP display file. If this were the case, the drawing could be redisplayed without leaving the IGP program. This

would be particularly useful for providing windowing capabilities that would allow the user to zoom in close for more detailed work or zoom out to get a better perspective on the overall drawing.

This could also allow for selective modification of the SM produced display as well as the IGP produced elements. This would solve the problem of modelers not producing displays appropriate for engineering drawing purposes because the display could be modified interactively.

The problem with this approach is that some portability would be sacrificed because not all SM systems produce the same or similar graphic output data. It may be possible to use the International Graphics Exchange System (IGES) to allow IGP to interpret graphic output from a wide range of SM systems, however, this would require a more complex IGP that would sacrifice efficiency and compactness.

Another approach would be to provide the CADD capabilities by fitting an existing CADD system to a particular SM system, or by translating graphic output from an SM system to an existing CADD system. Of course, this kind of implementation would lack portability as well.

An alternative approach to satisfying the compatability design goal would be to use the Graphic Kernel System (GKS) instead of the Core System as the graphic standard to be followed. This makes sense because GKS is more widely accepted and more commonly used than Core.

If a device with a larger screen area were available for use, a smaller display could be produced by the SM system thus allowing room for the title block to be placed in the traditional lower right-hand corner of the display. With PADL-2, this would mean using the REL_DISTANCE view projection parameter outlined in section 6.3 of the PADL-2 User's Manual ([HART83]) as opposed to simply using the UGB command to automatically set view projection parameters.

7.3.2. Suggestions for Future Extensions.

The basic premise that wireframe CADD capabilities can be given to SM systems by designing and implementing a 2D interactive graphics package has been verified by IGP. IGP does not provide industry acceptable engineering drawing documents for more complex parts because of hardware limitations. However, IGP is a structured, well-designed system through which implementing additional features will not be difficult.

7.3.2.1. Device Interface Extensions.

The device driver can be rewritten so that IGP can run on a different device than the VT-240. PADL-2 can run on a Tektronics™ 4010 series device. The IGP device driver could be rewritten so that IGP could be used with that particular device instead. Or, IGP could be programmed so that the user could choose from several device drivers that exist simultaneously in the IGP software much the same as in PADL-2. PADL-2 allows the user to choose the appropriate device driver through the SET DEVICE= command (see section 6.1.1.2.3 on page 44).

Another useful extension of the system would provide for high-quality hardcopy output. This can be done by obtaining a suitable plotter such as the Versetec™ V-80, and writing an appropriate driver to interpret the contents of the screen and dump it the the plotter.

7.3.2.2. Solid Modeling Interface Extensions.

IGP could be implemented to run with other SM systems. This would not require much doing because IGP has good portability characteristics.

7.3.2.3. User Interface Extensions.

In the IGP user interface, a facility to allow the user to find out the

coordinates of an existing point or end point of a line would be useful. This would allow the user to be more precise in creating additional elements related to the existing elements. Also, other features common to CADD systems could be implemented to help the user more easily create drawings. Examples of these include graphically selecting elements for deleting, changing the length of existing lines, and automatic dimensioning of part features.

7.3.2.4. Graphic Processing Extensions.

The ability to automatically set up the world coordinate system to the scale of the part that is displayed by the SM system would also allow the user to specify new elements more precisely. Also, a facility to draw arcs used in dimensioning angles would be useful. Better windowing capabilities would also improve the system.

A mechanism for rotating, scaling, and translating elements already exists in IGP, but it is possible to put this to better use. For example, a facility to allow the user to interactively rotate, scale, and translate elements would be useful.

7.3.3. Related Thesis Topics for the Future.

For engineering drawings it is sometimes desirable to produce pictoral representations of mechanical parts. SM systems can produce displays appropriate for these kinds of drawings, but a facility may be needed to allow the addition of graphic and textual information to produce a complete engineering drawing. Providing this capability is similar to what IGP already provides for two-dimensional displays. The difference is that pictoral representations are three-dimensional. Therefore, a related thesis topic for the future could be to design and implement a three-dimensional interactive graphic interface to allow pictoral displays produced by SM systems to easily be completed.

Presently IGP runs on the VT-240 display device, but the design of IGP allows other devices to be used with only minor modifications to the IGP program. In order for this to be true, IGP can only address graphic

facilities that are common to most graphic devices. This means that IGP does not use all of the graphic capabilities of the VT-240. A related thesis topic that would more fully utilize the capabilities of the VT-240 would involve the design and implemention of a device-specific VT-240 drafting package.

Using IGP with PADL-2 allows PADL-2 to be more advanced in its ability to model products for industry. A related thesis topic that would allow PADL-2 to be easier to use and thus improve on its product modeling characteristics would involve designing and developing an interactive graphic PADL-2 programming interface. This would allow a user to interactively combine solids to create PADL-2 objects without using the PADL-2 programming language.

8

BIBLIOGRAPHY

[BERG78] R. D. Bergeron et al, "Graphics Programming Using the Core System", Computing Survey, vol. 10, no. 4, pp. 388-443, Dec. 1978.

[BERG86] M. Berger, Computer Graphics with Pascal. Menlo Park, CA: The Benjamin/Cummings Publishing Co., Inc., 1986.

[BROW82] C. M. Brown, "PADL-2: A technical summary", IEEE Computer Graphics and Applications, vol. 2, no. 2, pp. 69-84, March 1982.

[FINA88] "Announcing AutoSolid: High-Powered, Affordable Solid Modeling", Final Draft: A quarterly newsletter from Autodesk, Inc., vol. 3, no. 1, pp. 2-3, Spring 1988.

[FOLE82] J. D. Foley and A. Van Dam, Fundamentals of Interactive Computer Graphics. Reading, MA: Addison Wesley Publishing Co., 1982.

[FRIE81] A. Frieden, "A two-dimensional level 2 Core system for the Apple II", Computer Graphics, vol.14, no. 4, pp. 127-152, March 1981.

[HART83] E. E. Hartquist and H. A. Marisa, "PADL-2 Users Manual", Doc. No. UM-10/1.2, Production Automation Project, Univ. of Rochester, May 1983.

[JENS85] C. Jensen and J. D. Helsel, Engineering Drawing and Design. New York: McGraw-Hill Book Company, 1985.

[JURI87] D. Juricic and R. E. Barr, "Engineering Graphics and Computer Graphics about their past, present, and future", Engineering Design Graphics Journal, vol. 51, no. 3, pp. 13-16, Autumn 1987.

[MALL83] W. R. Mallgren, Formal Specification of Interactive Graphics Programming Languages. Cambridge, MA: The MIT Press, 1983.

[METZ81] J. Metzger and M. Hopkins, "GOPAK: A graphic output processor package", Doc. No. IOG-10, Production Automation Project, Univ. of Rochester, March 1981.

[MICF78] J. C. Michener and J. D. Foley, "Some major issues in the design of the Core graphics system", Computing Survey, vol. 10, no. 4, pp. 445-463, Dec. 1978.

[MICV78] J. C. Michener and A. Van Dam, "A functional overview of the Core system with glossary", Computing Survey, vol. 10, no. 4, pp. 381-387, Dec. 1978.

[NEWM78] W. M. Newman and A. Van Dam, "Recent efforts toward graphics standardization", Computing Survey, vol. 10, no. 4, p. 365, Dec. 1978.

[NEWM79] W. M. Newman and R. F. Sproull, Principles of Interactive Computer Graphics. New York: McGraw-Hill Book Co., 1979.

[NICO81] C. J. Nicol and A. C. Kilgour, "A Pascal implementation of the GSPC Core graphics package", Computer Graphics, vol. 15, no. 4, pp. 327-335, Dec. 1981.

[OAKE88] J. Oakey, "True 3D Examined", CADalyst, vol. 5, no. 3, p. 41, April 1988.

[PRAT85] M. J. Pratt, "Solid modelling and the CAD/CAM interface", 1985 European Conference on Solid Modelling, London: Oyez Science

and Technology Services, 1985.

[REQU80] A. A. G. Requicha, "Representations for Rigid Solids: Theory, Methods, and Systems", Computing Surveys, vol. 12, no. 4, pp. 437-464, Dec. 1980.

[SHER88] S. Sheridan, "Solids Modeling Theory", CADalyst, vol. 5, no. 3, pp. 31-36, April 1988.

SECTION 9

PROGRAM LISTINGS

IGP MAIN PROGRAM

```
{ File: IGP.PAS }
program igp(input,output);

const alpha = 2; beta = 4;        { menu position }
var choice1: char;
    wminx,wmaxx,wminy,wmaxy: real;        { window min and max values }

%include 'user10:[epm9970.thesis.igprogs]coremod.pro'

procedure MAIN_MENU;              { erases menu and writes new main menu }
begin
ERASE_MENU;
write('MAIN: 1=USE, 2=SAVE, 3=CREATE, 4=DELETE, 5=ERASE MENU, 6=EXIT IGP ')
end;

begin             { main }
RESETVARS;
choice1 := '0';

{ erase top line }
ALPHANUMPOS(1,1);
write(' ':80);

{ erase padl2 dcl command line }
ALPHANUMPOS(3,6);
write(' ':23);

ALPHANUMPOS(alpha,beta);
write('This is IGP: A Solid Modeling Interactive Interface for Design Drafting');
INITIALIZE;
WAIT;
WAIT;
TERMINATE;

ERASE_MENU;
write('Reset terminal with Set-Up key if necessary.');
INITIALIZE;
WAIT;
TERMINATE;


repeat
     ERASE_MENU;
     write(' ENTER MIN X WINDOW COORD: ');
     wminx := GET_REAL;
     ERASE_MENU;
     write(' ENTER MAX X WINDOW COORD: ');
     wmaxx := GET_REAL;
     ERASE_MENU;
     write(' ENTER MIN Y WINDOW COORD: ');
     wminy := GET_REAL;
     ERASE_MENU;
     write(' ENTER MAX Y WINDOW COORD: ');
     wmaxy := GET_REAL;
     ERASE_MENU;
     if (wminx>=wmaxx) or (wminy>=wmaxy) then
          begin
          write('                    Invalid Window, Please Try Again');
          INITIALIZE;
          WAIT;
```

```
        TERMINATE
        end;
until (wminx<wmaxx) and (wminy<wmaxy);

WINDOW(wminx,wmaxx,wminy,wmaxy);

MAIN_MENU;
repeat
    readln(choice1);
    ALPHANUMPOS(alpha,beta);
    case choice1 of
        '1': begin USE_OLD_DRWG;          MAIN_MENU end;
        '2': begin SAVE_DRWG;             MAIN_MENU end;
        '3': begin CREATE_ELEMENTS;       MAIN_MENU end;
        '4': begin DELETE_ELEMENTS;       MAIN_MENU end;
        '5': ERASE_MENU;
        '6': ERASE_MENU;
        otherwise begin ERROR_MESSAGE;    MAIN_MENU end
    end { case }
until (choice1 = '6')

end.
```

IGP EXTERNAL PROCEDURES

```
{ File: COREMOD.PAS }
MODULE igpmod(input,output,fi,fo);
{*************************************************************************}
{ This module is the major portion of an interactive graphics Pascal program
  written by E. P. Maruggi as part of a Master's Thesis in Computer Science
  entitled: "Solid Modeling Interactive Interface for Design Drafting."

  This module is divided into 3 sections.  The first is the device driver
  called UPDATESCREEN, which must be rewritten (as well as part of RESETVARS)
  if a device other than a VT-240 is to be used.  The second section contains
  the graphic processor routines which is a subset of the Core Graphics
  Standard.  The third section contains the user interface routines.

  All procedures in this module are in CAPITAL letters.  Also, the underscore
  (_) character is used to separate words in the names of interactive
  procedures, while in other procedures the words are allowed to run together.}

{***************************** global parameters ***************************}

const maxistore = 300;              { max. no. of instructions allowed in a segment }
      alpha = 2; beta = 4;          { menu position }
      blanks2 = '  ';               { used in setting text size }
      blanks11 = '           '; { used in writing to perm file }
      blanks16 = '                '; { used for text test }

type edge = (left,right,top,bottom);    { window and viewport edges }
     istoreindex = 0..maxistore;        { index into dpulist }
     stringtype = packed array [1..16] of char;
     sizetype = packed array[1..2] of char;

     dpuinstr = record                  { NDC instr }
        ndcx, ndcy: real;               { (x,y) in screen space }
        opcode: char;
        string: stringtype;             { for text }
        tsize: sizetype;                { for text size }
        color: array[0..2] of real;     { color }
        stile: integer                  { line style }
        end;

     ptrtoseg = ^segment;               { linked list of segments }
     segment = record
        name: integer;                  { segment number }
        length: integer;                { seg. length }
        dpulist: array[istoreindex] of dpuinstr;  { seg. primitive }
        link: ptrtoseg                  { pointer }
        end;

     matrixtype = array[1..3,1..3] of real;

     ptrtomatrix = ^matrix;             { linked list of transf. matricies }
     matrix = record
        transf: matrixtype;             { 3 x 3 matrix }
        lnk: ptrtomatrix                { pointer }
        end;

var primitive: dpuinstr;

    index,ind: istoreindex;             { dpulist index }

    viewport:
```

```
    window,
    screenarea: array[edge] of real;     { device dependent parameters }

    screenwidth, screenheight,           { width, height of screen in NDC }

    xportmin,xportmax,yportmin,yportmax,        { viewport boundaries in NDC }

    aratio,                              { aspect ratio for circles }
    xreport,yreport,                     { converted screen to world coords }
    currentx, currenty: real;            { global variables that maintain ccurrent position in NDC }

    start,                               { start of seg. list }
    p,                                   { pointer to new node }
    q,r,                                 { pointers for traversal }
    cptr,endptr,                         { used in reading perm file }
    bptr: ptrtoseg;                      { used in writing to perm file }

    segcount,                            { total no. of segments }
    noln: integer;                       { no. of last seg. name }

    witharrows,                          { used in drawing dim. lines }
    twoarrows,                           { used in drawing dim. lines }
    opened: boolean;                       { flag open segment condition }

    currentit: matrixtype;               { current instance transformation }

    stacktop: ptrtomatrix;               { points to top of matrix stack }

    fi, fo: text;                  { input and output files }

    finame, foname: packed array [1..11] of char;        { file names }

{********************************* device driver ******************************}

[GLOBAL] procedure UPDATESCREEN; { interprets instruction - produces opcodes }
{ This routine is a VT-240 device driver and must be rewritten if a device
  other than a VT-240 is to be used }

var pattern,              { bit pattern }
    scrnx, scrny,         { screen x and screen y }
    gigicolor: integer;   { color # for gigi }
begin

{ convert ndc to screen coords }
scrnx := round(primitive.ndcx * (screenarea[right]-screenarea[left]) + screenarea[left]);
scrny := round((1 - primitive.ndcy) * screenarea[bottom]);

        case primitive.opcode of
            's': begin                        { set line style }
                 if primitive.stile = 1
                        then pattern := 11111111
                        else pattern := 11000011;
                 writeln(' W(P', pattern:8, ')')
                 end;
            'c': begin                        { set line color }
                 gigicolor := round(primitive.color[1]) * 4 + round(primitive.color[0]) * 2 +
                        round(primitive.color[2]);
                 writeln(' W(I', gigicolor:1, ')')
                 end;
            'l': writeln(' V(', scrnx:3, ',', scrny:3, ')');   { draw line }
```

```
            'p': begin                                          { draw point }
                 writeln(' P[', scrnx:3, ',', scrny:3, ']');
                 writeln(' V[ ]')
                 end;
            'm': writeln(' P[', scrnx:3, ',', scrny:3, ']');  { move }
            't': writeln(' T"',primitive.string,'"');    { text }
            'h': writeln(' T(S',primitive.tsize,')');    { for test size }
            'a': begin
                 writeln(' ', chr(27), 'Pp'); {;     initialization to enter graphics }
                 writeln(' S[0,0]')          { set default linestyle to white, }
                                             { screen color to black, scroll off }
                 end;
            'r': writeln(' P[383,240]  R(P(I))');        { puts in report mode }
            'w': writeln(' S(T100)');                    { waits 2 sec }
            'z': writeln(' SM0 ', chr(27), '\')      { to get out of graphics }
            end  { case }
end;  { UPDATESCREEN }

{++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++}
{++++++++++++++  GRAPHIC PROCESSOR ROUTINES (subset of CORE)  +++++++++++++++++}
{++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++}

{****************************** direct instruction ***************************}

[GLOBAL] procedure DIRECTINSTR;  { directs instruction to either the
                                 { device driver or the display file }
begin
if not opened then
    UPDATESCREEN
else begin
    index := index + 1;
    with p^.dpulist[index] do
        begin
        ndcx := primitive.ndcx;
        ndcy := primitive.ndcy;
        opcode := primitive.opcode;
        string := primitive.string;
        tsize := primitive.tsize;
        color := primitive.color;
        stile := primitive.stile
        end  { with }
    end  { else }
end;  { DIRECTINSTR }

{************************** 2-D viewing transformation ***********************}

[GLOBAL] procedure VIEWTRANSF;   { transforms world coords to normalized device coords }
                         { it is called whenever window or viewport is changed }
begin

{ the following sets viewport boundaries in NDC for use in clipping }
xportmin := viewport[left];
xportmax := viewport[right];
yportmax := viewport[top];
yportmin := viewport[bottom]
end;

{********************** 2-D world to ndc transformations *********************}

[GLOBAL] function XTRANS(x: real): real;  { world to ndc transformation for x }
```

```
begin
XTRANS := (x - windw[left]) / (windw[right] - windw[left])
end;

[GLOBAL] function YTRANS(y: real): real;  { world to screen transformation for y }
begin
YTRANS := (y - windw[bottom]) / (windw[top] - windw[bottom])
end;

{************************ Reverse transforms - ndc to world ********************}

[GLOBAL] function RXTRANS(rx: real): real;
begin
RXTRANS := rx * (windw[right] - windw[left]) + windw[left]
end;

[GLOBAL] function RYTRANS(ry: real): real;
begin
RYTRANS := ry * (windw[top] - windw[bottom]) + windw[bottom]
end;

{**** 2-D transformations to manipulate scaling, rotation  and translation ***}

[GLOBAL] procedure INSTTRANSF(var worldx, worldy: real);  { transforms world coords }
var neworldx,neworldy: real;
begin
neworldx := (worldx * currentit[1,1]) + (worldy * currentit[2,1]) + currentit[3,1];
neworldy := (worldx * currentit[1,2]) + (worldy * currentit[2,2]) + currentit[3,2];
worldx := neworldx;
worldy := neworldy
end;  { INSTTRANSF }

[GLOBAL] procedure CLEARTRANSF;  { sets the current instance transformation to the identity matrix }
begin
currentit[1,1] := 1;
currentit[1,2] := 0;
currentit[1,3] := 0;
currentit[2,1] := 0;
currentit[2,2] := 1;
currentit[2,3] := 0;
currentit[3,1] := 0;
currentit[3,2] := 0;
currentit[3,3] := 1
end;  { CLEARTRANSF }

[GLOBAL] procedure UPDATETRANSF(xscale,yscale,angle,xtran,ytran: real);
                { premultiplies the current instance transformation by a transformation }
                { consisting of a scaling, followed by a rotation, and then a translation }
var i,j,k: integer;      { local array counters }
    amat, cmat: matrixtype;      { matrix that is multiplied by CTM to update CTM }
    theta,                { angle in radians }
    wcurrentx, wcurrenty: real;

begin

theta := angle * 3.1416 / 180.0;
amat[1,1] := xscale * cos(theta);
amat[1,2] := xscale * sin(theta);
amat[1,3] := 0;
amat[2,1] := -yscale * sin(theta);
```

```
amat[2,2] := yscale * cos(theta);
amat[2,3] := 0;
amat[3,1] := xtran;
amat[3,2] := ytran;
amat[3,3] := 1;
for i := 1 to 3 do        { multiply CTM by amat to produce new CTM }
    for j := 1 to 3 do
        cmat[i,j] := 0;
for i := 1 to 3 do
    for j := 1 to 3 do
        for k := 1 to 3 do
            cmat[i,j] := amat[i,k] * currentit[k,j] + cmat[i,j];
for i := 1 to 3 do
    for j := 1 to 3 do
        currentit[i,j] := cmat[i,j]
end;   { UPDATETRANSF }

[GLOBAL] procedure PUSHTRANSF;  { pushes the current instance transformation on the transformation stack
var l,m: integer;          { local array counters }
    s: ptrtomatrix;        { points to new node }
begin
new(s);
for l := 1 to 3 do
    for m := 1 to 3 do
        s^.transf[l,m] := currentit[l,m];    { store currentit in matrix node }
if (stacktop = nil) then
    s^.lnk := nil
else s^.lnk := stacktop;
stacktop := s
end;   { PUSHTRANSF }

[GLOBAL] procedure POPTRANSF;  { pops the top transformation from the transf. stack and discards it }
                    { it has no effect if the stack is empty }
begin
if (stacktop <> nil) then
    stacktop := stacktop^.lnk
    end;   { POPTRANSF }

[GLOBAL] procedure RESTORETRANSF;  { Copies the top transformation from the transformation stack and }
                        { makes it the CTM. It performs a CLEARTRANSF if the stack is empty }
var g,h: integer;          { local array counters }
begin
if (stacktop = nil) then
    CLEARTRANSF
else
    for g := 1 to 3 do
        for h := 1 to 3 do
            currentit[g,h] := stacktop^.transf[g,h]
end;   { RESTORETRANSF }

{********************************* Text ***************************************}

[GLOBAL] procedure TEXT2(strng: stringtype);  { places text in primitive }

procedure TEXTCLIP(var s: stringtype);  { clips text }
begin
if (currentx < xportmin) or (currentx > xportmax) or (currenty < yportmin) or (currenty > yportmax) then
s :=
end;   { TEXTCLIP }
```

```
begin  { TEXT }
TEXTCLIP(strng);
with primitive do
     begin
     opcode := 't';
     string := strng;
     end;
DIRECTINSTR
end;   { TEXT }

{********************************** move ****************************************}

[GLOBAL] procedure MOVEABS2(x,y: real);  { moves c.p. to an absolute coord }
begin
INSTTRANSF(x,y);                         { multiply i.t. }
with primitive do
     begin
     opcode := 'm';
     ndcx := XTRANS(x);
     ndcy := YTRANS(y);
     DIRECTINSTR;
     currentx := ndcx;
     currenty := ndcy
     end
end;   { MOVEABS 2 }

[GLOBAL] procedure MOVEREL2(dx,dy: real);  { moves c.p. relative to itself }
var x,y: real;
begin
x := RXTRANS(currentx) + dx;
y := RYTRANS(currenty) + dy;
MOVEABS2(x,y)
end;   { MOVEREL2 }

{********************** 2-D line clipping routine ****************************}

[GLOBAL] procedure CLIPPER(x1,y1,x2,y2,xmin,xmax,ymin,ymax: real);
         { Cohen-Sutherland Algorithm for line P1=(x1,y1) to P2=(x2,y2) }
         { current position remains in viewport at all times }

type outcode = array[1..4] of boolean;
var accept,
     reject,
     switch,
     done:        boolean;
     outcode1,
     outcode2:     outcode;


procedure SWAP;
var temp: real;              { temporary store }
     code: outcode;          { temporary store }
begin
temp := x1;
x1 := x2;
x2 := temp;
temp := y1;
y1 := y2;
y2 := temp;
code := outcode1;
```

```
outcode1 := outcode2;
outcode2 := code
end;   { SWAP }


function REJECTCHECK(oc1,oc2: outcode): boolean;  { check to see if seg. can be trivially rejected }
begin
if (oc1[1] and oc2[1]) or (oc1[2] and oc2[2]) or (oc1[3] and oc2[3])
    or (oc1[4] and oc2[4])
    then REJECTCHECK := true
    else REJECTCHECK := false
end;   { REJECTCHECK }


function ACCEPTCHECK(ocode1,ocode2: outcode): boolean;  { check to see if seg. can be trivially accep
begin
if (not ocode1[1]) and (not ocode1[2]) and (not ocode1[3]) and (not ocode1[4])
   and (not ocode2[1]) and (not ocode2[2]) and (not ocode2[3]) and (not ocode2[4])
    then ACCEPTCHECK := true
    else ACCEPTCHECK := false
end;   { ACCEPTCHECK }


procedure OUTCODES(x,y: real; var oc: outcode);  { calculates outcodes of segment endpoints }
begin
if (y > ymax)    { above viewport }
    then oc[1] := true
    else oc[1] := false;
if (y < ymin)              { below viewport }
    then oc[2] := true
    else oc[2] := false;
if (x > xmax)              { right of viewport }
    then oc[3] := true
    else oc[3] := false;
if (x < xmin)    { left of viewport }
    then oc[4] := true
    else oc[4] := false
end;   { OUTCODES }


procedure DRAW(xone,yone,xtwo,ytwo: real);   { draws line  }
begin
if switch then
    begin
    with primitive do
        begin
        opcode := 'm';
        ndcx := xtwo;
        ndcy := ytwo
        end;   { with }
    DIRECTINSTR
    end;   { if }
with primitive do
    begin
    opcode := 'l';
    ndcx := xone;
    ndcy := yone
    end;   { with }
DIRECTINSTR
end;   { DRAW }
```

```
{################## CLIPPER ###################}

begin { CLIPPER }
accept := false;
reject := false;
done := false;

repeat
    OUTCODES(x1,y1,outcode1);
    OUTCODES(x2,y2,outcode2);
    reject := REJECTCHECK(outcode1,outcode2);    { check trivial reject }
    if reject then done := true
    else
        begin   { possible accept }
        accept := ACCEPTCHECK(outcode1,outcode2); { check trivial accept }
        if accept then done := true
        else { Since at most one endpoint is inside viewport, }
            { now perform a subdivision, move P1 to the intersection point; use }
            { the formulas y = y1 + slope * (x-x1), x = x1 + (1/slope) * (y-y1). }

            begin

            { First, if P1 is inside window, exchange points 1 and 2 and their }
            { outcodes to guarantee that P1 is outside window }

            if not((outcode1[1]) or (outcode1[2]) or (outcode1[3]) or (outcode1[4]))
                then begin
                    SWAP;
                    switch := true
                    end;
            if outcode1[1] and (y1 <> y2) then
                begin                   { divide line at top of window }
                    x1 := x1 + (x2-x1) * (ymax-y1) / (y2-y1);
                    y1 := ymax
                end
            else if outcode1[2] and (y1 <> y2) then
                begin                   { divide line at bottom of window }
                    x1 := x1 + (x2-x1) * (ymin-y1) / (y2-y1);
                    y1 := ymin
                end
            else if outcode1[3] and (x1 <> x2) then
                begin                   { divide line at right edge of window }
                    y1 := y1 + (y2-y1) * (xmax-x1) / (x2-x1);
                    x1 := xmax
                end
            else if outcode1[4] and (x1 <> x2) then
                begin                   { divide line at left edge of window }
                    y1 := y1 + (y2-y1) * (xmin-x1) / (x2-x1);
                    x1 := xmin
                end
            end     { subdivide }
        end     { possible accept }
    until done;
    if accept then
        DRAW(x1,y1,x2,y2)
end;     { CLIPPER }

{********************************* Window ************************************}
```

```
[GLOBAL] procedure WINDOW(minx,maxx,miny,maxy: real);  { sets up world coord system }
begin
if (minx>=maxx) or (miny>=maxy)
        then writeln('invalid window')
        else begin
             windw[left] := minx;
             windw[right] := maxx;
             windw[bottom] := miny;
             windw[top] := maxy
             end;    { if else }
VIEWTRANSF
end;   { WINDOW }

{********************************* Viewport ************************************}

[GLOBAL] procedure VIEWPORT2(minx,maxx,miny,maxy: real);  { sets viewport bounds }
begin
if (minx<0.0) or (maxx>1.0) or (minx>=maxx)
    then writeln('invalid viewport x coord')
    else
        if (miny<0.0) or (maxy>1.0) or (minx>=maxx)
            then writeln('invalid viewport y coord')
            else begin
                 viewport[left] := minx;
                 viewport[right] := maxx;
                 viewport[bottom] := miny;
                 viewport[top] := maxy
                 end;
VIEWTRANSF
end;   { VIEWPORT }

{********************************* Point ***************************************}

[GLOBAL] procedure POINTABS2(x,y: real);  { draws a point at an absolute coord }
begin
INSTTRANSF(x,y);
with primitive do
     begin
     opcode := 'p';
     ndcx := XTRANS(x);
     ndcy := YTRANS(y)
     end;
if not((primitive.ndcx < xportmin) or (primitive.ndcx > xportmax) or
       (primitive.ndcy < yportmin) or (primitive.ndcy > yportmax))
    then DIRECTINSTR;
currentx := primitive.ndcx;
currenty := primitive.ndcy
end;   { POINTABS2 }

[GLOBAL] procedure POINTREL2(dx,dy: real);  { draws a point relative to c.p. }
var x,y: real;
begin
x := RXTRANS(currentx) + dx;
y := RYTRANS(currenty) + dy;
POINTABS2(x,y)
end;   { POINTREL2 }

{********************************* Line ****************************************}

[GLOBAL] procedure LINEABS2(x,y: real);  { draws a line to an absolute coord }
```

```
var transx, transy: real;          { x and y in NDC }
begin
INSTTRANSF(x,y);                   { multiply instance transformation }
transx := XTRANS(x);                   { change W.C. to screen coords }
transy := YTRANS(y);
CLIPPER(transx, transy, currentx, currenty,     { clip if necessary }
        xportmin, xportmax, yportmin, yportmax);
currentx := transx;
currenty := transy
end;  { LINEABS2 }

[GLOBAL] procedure LINEREL2(dx,dy: real);  { draws a line relative to the current position }
var x,y: real;
begin
x := RXTRANS(currentx) + dx;
y := RYTRANS(currenty) + dy;
LINEABS2(x,y)
end;  { LINEREL2 }

{******************************* Set Line Style ********************************}

[GLOBAL] procedure SETLINESTYLE(style: integer);  { sets line style: 1 = solid, 2 = dashed }
begin
if (style < 1) or (style > 2) then
    writeln('linestyle value out of bounds')
else begin
    with primitive do
        begin
        opcode := 's';
        stile := style
        end;
    DIRECTINSTR
    end  { else }
end;  { SETLINESTYLE }

{********************************** Set Color ********************************}

[GLOBAL] procedure SETCOLOR(color0, color1, color2: real);  { sets line color (red,green,blue) }
begin
with primitive do
    begin
    opcode := 'c';
    color[0] := color0;
    color[1] := color1;
    color[2] := color2
    end;
DIRECTINSTR
end;  { SETCOLOR }

{************************** segmentation routines ****************************}

[GLOBAL] procedure CLOSESEG;  { indicates the end of the definition of the currently "open" seg. }
begin
if opened then begin      { if no seg. is "open", this procedure does nothing }
p^.length := index;
index := 0;               { reset index }
q := start;
if (q = nil) then
    begin
    p^.n := p^.name;
```

```
    p^.link := nil;
    start := p
    end
else begin
    while (p^.name > q^.name) and (q^.name < noln) do   { traverse list }
        begin
        r := q;
        q := q^.link
        end;   { while }
    if (p^.name = q^.name) then    { if new seg. has same no. as an existing seg., the existing seg. i
        begin
        p^.link := q^.link;
        r^.link := p
        end
    else if (p^.name < q^.name) then
        begin
        p^.link := q;
        r^.link := p
        end
        else begin
            noln := p^.name;
            p^.link := q^.link;
            q^.link := p
            end
    end;   { else begin }
p := nil;
q := nil;
r := nil;
opened := false;
segcount := segcount + 1
end   { first if }
end;   { CLOSESEG }

[GLOBAL] procedure CREATESEG(segnum: integer);   { "opens" a segment for definition and assigns it a no.
begin
if opened then         { if seg. is open, close it }
    CLOSESEG;
opened := true;
new(p);
p^.link := nil;
p^.name := segnum;
p^.length := 0
end;   { CREATESEG }

[GLOBAL] procedure POSTSEG(snum: integer);   { displays the specific segment on the screen }
var i: integer;          { local index for dpulist }
begin
if (snum <= noln) then       { if the seg. doesn't exist, no action is taken }
    begin
    q := start;
    while (q^.name < snum) do
        begin
        r := q;
        q := q^.link
        end;
    if (q^.name = snum) then
        for i := 1 to q^.length do
            begin
            with primitive do
                begin
```

```
                ndcx := q^.dpulist[i].ndcx;
                ndcy := q^.dpulist[i].ndcy;
                opcode := q^.dpulist[i].opcode;
                string := q^.dpulist[i].string;
                tsize := q^.dpulist[i].tsize;
                color := q^.dpulist[i].color;
                stile := q^.dpulist[i].stile
                end;  { with }
            UPDATESCREEN
            end  { for }
    end  { if }
end;  { POSTSEG }

[GLOBAL] procedure UNPOSTSEG(sn: integer);  { erases the specified segment from the screen }
begin
SETCOLOR(0.0,0.0,0.0);
POSTSEG(sn);
SETCOLOR(1.0,1.0,1.0)
end;  { UNPOSTSEG }

[GLOBAL] procedure DELETESEG(segno: integer);  { deletes the record of the specified segment }
                                               { and erases its image from the screen }
                                               { If  the segment doesn't exist, no action is taken }

begin
UNPOSTSEG(segno);
if (q = start) and (q^.name = segno) then       { seg. is first in list }
    begin
    if (q^.link = nil) then
        noln := 0;
    start := q^.link;
    segcount := segcount - 1
    end
else
    if (q^.name = segno) then
        begin
        if (q^.link = nil) then          { q is last seg. in list }
            noln := r^.name;
        r^.link := q^.link;              { reroute linked list to go around q }
        segcount := segcount - 1
        end
end;  { DELETESEG }

{*****************************  Reset Variables  *********************************}

[GLOBAL] procedure RESETVARS;     { initializations of globals, and enter graphics }
begin

{ The following 4 variables are device dependent }
screenarea[left] := 0.0;
screenarea[right] := 767.0;
screenarea[bottom] := 479.0;
screenarea[top] := 0.0;

aratio := screenarea[right]/screenarea[bottom];
start := nil;
p := nil;
q := nil;
r := nil;
bptr := nil;
cptr := nil;
```

```
endptr := nil;

stacktop := nil;
index := 0;
segcount := 0;
noln := 0;
primitive.tsize := '1';

witharrows := false;
twoarrows := false;
opened := false;

currentx := 0.0;
currenty := 0.0;

viewport[left] := 0.0;
viewport[right] := 1.0;
viewport[bottom] := 0.0;
viewport[top] := 1.0;

windw[left] := 0.0;
windw[right] := 500;      { best for precision input... }
windw[bottom] := 0.0;
windw[top] := 500;        { ...for VT-240 }

VIEWTRANSF;
CLEARTRANSF
end;

{********************************** Initialize **********************************}

[GLOBAL] procedure INITIALIZE;      { initializations of globals, and enter graphics }
begin
primitive.opcode := 'a';
UPDATESCREEN
end;

{********************************** Terminate ***********************************}

[GLOBAL] procedure TERMINATE;     { get out of graphics }
begin
primitive.opcode := 'z';
UPDATESCREEN
end;

{++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++}
{++++++++++++++++++++++++  INTERACTIVE ROUTINES  ++++++++++++++++++++++++++++++++}
{++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++}

{*************************** Alpha Numeric Position ***************************}

[GLOBAL]procedure ALPHANUMPOS(l,c:integer);      { defines cursor absolute position }
begin
if (l>=0) AND (c>=0) then
     if (L<10) then
          if (c<10) then write(' ',chr(27),'[',L:1,';',C:1,'f')
          else write(' ',chr(27),'[',L:1,';',C:2,'f')
     else if (c<10) then write(' ',chr(27),'[',L:2,';',C:1,'f')
          else write(' ',chr(27),'[',L:2,';',C:2,'f')
end;   { ALPHANUMPOS }
```

```
{*********************************  Wait  ************************************}

[GLOBAL]procedure WAIT;          { wait 2 sec (allows a message to be read) }
begin
primitive.opcode := 'w';
UPDATESCREEN
end;

{*******************************  Erase Menu  ********************************}

[GLOBAL] procedure ERASE_MENU;   { Erases the main menu from the screen }
var space: char;
begin
ALPHANUMPOS(alpha,beta);
space := ' ';
writeln(space:76-beta);
ALPHANUMPOS(alpha,beta)
end;

{*****************************  Error Message  *******************************}

[GLOBAL] procedure ERROR_MESSAGE;        { For selection errors }
begin
write('                Error in Selection, Please Try Again.                 ');
INITIALIZE;
WAIT;
TERMINATE
end;

{************************  Convert String to Real Value  *********************}

[GLOBAL] function GET_REAL:real;  { converts string input into value }
var  r,                           { intermediate result for value        }
     p,
     m,                           { multiplying factor for integer part  }
     d:    real;                  { dividing fact for fractionnal part   }
     x1,y1:integer;               { cursor position at beginning }
     sign: integer;               { -1 if '-' is read                    }
     np,                          { true if no '.' encountered so far    }
     fc:   boolean;               { true for first non blanc  character  }
     c:    char;                  { character read and decoded           }
begin
m:=10;                  { until a '.' is entered, then m:=1     }
d:=1;                   { ignored until '.', then d:=10         }
sign:=1;                { positive until '-' entered            }
fc:=true;               { true until '-'                        }
np:=true;               { true until '.'                        }
r:=0;                   { initialize entered value              }
repeat
    read(c);                          { until <cr>          }
    if (c='-') and fc then            { first time '-'      }
        begin
        sign:=-1;                   { negative value      }
        fc:=false                   { not first char      }
        end
    else if (c='.') and np then          {irs  .'      }
        begin                       { -> fractionnal part }
        np:=false;                  { next '.' -> error   }
        fc:=false;                  { not first char      }
```

```
        m:=1;                          { keep integer part   }
        d:=10                          { to divide decimals  }
        end
        else if c in ['0'..'9'] then        { ok digit   }
             begin
             r:=r*m+(ord(c)-ord('0'))/d;
             if not np then d:=d*10;    { decimal part     }
             fc:=false                  { not first char   }
             end
             else if (c<>' ') and (c<>chr(10)) then  { eliminates blancs}
                  begin                                { erroneous character}
                  write(chr(13));
                  ALPHANUMPOS(alpha,beta+20);
                  write(' Enter Correct Value: ');
                  while not eoln do read(c); {ignores rest of line}
                  read(c);                   { because <cr> -> ' '}
                  m:=10;                { to restart                }
                  d:=1;
                  r:=0;                 { forget previous entry     }
                  fc:=true;             { reinirialise first char   }
                  np:=true;             { reinit integer part       }
                  sign:=1;              { reinit positive value     }
                  end;
until eoln;                     { correct entry + eoln     }
GET_REAL := r*sign;                        { in case negative value  }
readln
end;                                  { li= entered value        }

{*****************************  Use Old Drawing  ******************************}

[GLOBAL] procedure USE_OLD_DRWG;  { retrieves an existing file, or writes an
                                    { error message if file does not exist }
var i: integer;
begin
ERASE_MENU;
write('Enter File Name: ');
repeat
    readln(finame);
until (finame <> blanks11);              { gets file name }
ERASE_MENU;
write('File: ',finame);
INITIALIZE;
WAIT;
TERMINATE;
open(fi, finame, old, error := continue);                          { creates new file }
if status(fi) = 0 then           { if file could be opened }
     begin
     reset(fi);                  { prepares file }
     readln(fi,segcount,noln);
     if (segcount <> 0) then     { if segcount=0 then no more reading is done }
          begin
          new(cptr);                                  { "cptr" used to get new seg. for list }
          cptr^.link := nil;
          start := cptr;                         { "start" is marker for top of list }
          readln(fi,cptr^.name,cptr^.length);
          for ind := 1 to cptr^.length do
               begin
               readln(fi,cptr^.dpulist[ind].ndcx, cptr^.dpulist[ind].ndcy,
                         cptr^.dpulist[ind].opcode);
               if (cptr^.dpulist[ind].opcode = 't') then   { if string exists }
```

```
                readln(fi,cptr^.dpulist[ind].string,cptr^.dpulist[ind].tsize);
            readln(fi,cptr^.dpulist[ind].color[0], cptr^.dpulist[ind].color[1],
                    cptr^.dpulist[ind].color[2], cptr^.dpulist[ind].stile)
            end;   { for ind }
        INITIALIZE;
        POSTSEG(cptr^.name);
        for i := 1 to segcount-1 do
            begin
            endptr := cptr;
            new(cptr);
            cptr^.link := nil;
            readln(fi,cptr^.name,cptr^.length);
            for ind := 1 to cptr^.length do
                begin
                readln(fi,cptr^.dpulist[ind].ndcx, cptr^.dpulist[ind].ndcy,
                                cptr^.dpulist[ind].opcode);
                if (cptr^.dpulist[ind].opcode = 't') then       { if string exists }
                    readln(fi,cptr^.dpulist[ind].string,cptr^.dpulist[ind].tsize);
                readln(fi,cptr^.dpulist[ind].color[0], cptr^.dpulist[ind].color[1],
                        cptr^.dpulist[ind].color[2], cptr^.dpulist[ind].stile)
                end;   { for ind }
            endptr^.link := cptr;
            POSTSEG(cptr^.name)
            end;            { for i }
        TERMINATE;
        close(fi)           { closes file }
        end     { if segcount }
    else begin
        ERASE_MENU;
        write('File contains no elements ');
        WAIT
        end     { end segcount else }
    end         { end if status }
else begin      { if invalid file }
    ERASE_MENU;
    write('                     File NOT Found or Unavailable
    INITIALIZE;
    WAIT;
    TERMINATE
    end;        { else }
cptr := nil;
endptr := nil
end;    { use_old_drwg }

{*******************************  Save Drawing  *********************************}

[GLOBAL] procedure SAVE_DRWG;    { Writes the existing drawing elements to }
                                 { permanent storage                       }
begin
ERASE_MENU;
write('Save Drawing As:  ');
repeat
    readln(foname);
until (foname <> blanks11);              { gets file name }
open(fo,foname );                      { creates new file }
rewrite(fo);                     { prepares file }
bptr := start;                   { put bptr at top of seg list }
if (bptr <> nil) then
    begin
    writeln(fo,segcount,nnln);
```

```
    repeat
        writeln(fo,bptr^.name,bptr^.length);
        for ind := 1 to bptr^.length do
            begin
            writeln(fo,bptr^.dpulist[ind].ndcx, bptr^.dpulist[ind].ndcy,
                bptr^.dpulist[ind].opcode);
            if (bptr^.dpulist[ind].opcode = 't') then   { if string exists }
                writeln(fo,bptr^.dpulist[ind].string,bptr^.dpulist[ind].tsize);
            writeln(fo,bptr^.dpulist[ind].color[0], bptr^.dpulist[ind].color[1],
                bptr^.dpulist[ind].color[2], bptr^.dpulist[ind].stile)
            end;   { for }
        bptr := bptr^.link;
    until (bptr = nil)
    end
else begin
    ALPHANUMPOS(alpha, beta);
    write('                Drawing has no elements. ');
    INITIALIZE;
    WAIT;
    TERMINATE
    end;     { else begin }
close(fo)
end;     { save_drwg }

{*******************************  Delete Elements  ******************************}

[GLOBAL] procedure DELETE_ELEMENTS;      { Deletes the specified element from }
                                         { the drawing                        }
var elnum: integer;
    answer: char;
begin
ERASE_MENU;
if (noln <> 0) then begin
write('Enter Element Number ( <=',noln:3,' ) or 0 to Exit ');
elnum := TRUNC(GET_REAL);
if (elnum <> 0) then
    begin
    if (elnum > noln) or (elnum < 1) then
        begin
        ERASE_MENU;
        write('                    Element Number Out of Bounds. ');
        INITIALIZE;
        WAIT;
        TERMINATE
        end;
    repeat
        if (elnum > noln) or (elnum < 1) then
            elnum := 1;
        INITIALIZE;
        SETCOLOR(1.0,0.0,0.0);
        POSTSEG(elnum);
        SETCOLOR(1.0,1.0,1.0);
        TERMINATE;
        ERASE_MENU;
        write('Delete Element No. ',elnum:3,'? 1 = YES, 2 = NO, 3 = EXIT ');
        readln(answer);
        case answer of
            '1':begin
                INITIALIZE;
                DELETESEG(elnum);
```

```
                TERMINATE
                end;
        '2','3':begin
                INITIALIZE;
                POSTSEG(elnum);
                TERMINATE
                end;
        otherwise ERROR_MESSAGE
        end;    { case }
    elnum := elnum + 1;
    until (answer = '3')
    end { if }
end  { if }
else write('                    Drawing Contains No Elements. ')
end;    { DELETE_ELEMENTS }

{ ***************************** Create Elements  ******************************** }

[GLOBAL] procedure CREATE_ELEMENTS;      { For interactive creation of graphic }
                                         { elements.  Has several subroutines. }
var choice2: char;
    savex, savey: real;          { used in transformations, esp. for drwng arrow }
    eta: real;  { angle value }


procedure REPORT_COORDS(var xw,yw: real);        { *** embed. in CREATE_ELEMENTS }
                                                 { Gets coords from screen }
const lent = 20;
type stringer = packed array[1..lent] of char;
var x,y,xnorm,ynorm: real;
    st: stringer;
    i:  integer;
    c:  char;

begin
primitive.opcode := 'r';         { puts in report mode }
UPDATESCREEN;
readln(st);                      { reads in string }

c := ' ';
x := 0.0;
y := 0.0;
i := 1;

while (st[i] <> '[') and (i <= lent-1) do
    begin
    c := st[i];          {'c' becomes first letter before '[' of returned string}
    i := i + 1
    end;

if i = lent then         { 'c' = st[19] and haven't reached '[' yet }
    begin
    readln(st);          { read in new line of string }
    c := ' ';
    i := 1;              { reset vars }
    while (st[i] <> '[') and (i <= lent-1) do
        begin
        c := st[i];      { try to get good value for 'c' from second string }
        i := i + 1
        end
```

```
    end;    { if }          { now we should have a good value for 'c' }
i := i + 1;                 { advance i to after '[' }

while (i < lent) and (st[i] in ['0'..'9']) do
    begin
    x := x * 10.0 + (ord(st[i]) - ord('0'));    { convert x char to real }
    i := i + 1
    end;

i := i + 1;

while (i<lent) and (st[i] in ['0'..'9']) do
    begin
    y := y * 10.0 + (ord(st[i]) - ord('0'));
    i := i + 1
    end;

{ convert screen coords to ndc coords }
xnorm := (x - screenarea[left])/(screenarea[right]-screenarea[left]);
ynorm := 1 - (y/screenarea[bottom]);

{ convert ndc to world coords }
xw := RXTRANS(xnorm);
yw := RYTRANS(ynorm)
end;                        { REPORT_COORDS }


procedure KBD_INPUT(var xkbd, ykbd: real);      { *** embed. in CREATE_ELEMENTS }
                                                { Accepts coords from the keyboard }
begin
ERASE_MENU;
write('ENTER KBD X COORD                ');
ALPHANUMPOS(alpha,beta+20);
xkbd := GET_REAL;
ERASE_MENU;
write('ENTER KBD Y COORD                ');
ALPHANUMPOS(alpha,beta+20);
ykbd := GET_REAL
end;

procedure DELTA_INPUT(var dx, dy: real);        { *** embed. in CREATE_ELEMENTS }
                                                { Accepts delta coords from kybd }
begin
ERASE_MENU;
write('ENTER DELTA X COORD              ');
ALPHANUMPOS(alpha,beta+20);
dx := GET_REAL;
ERASE_MENU;
write('ENTER DELTA Y COORD              ');
ALPHANUMPOS(alpha,beta+20);
dy := GET_REAL
end;

procedure PLACE_POINT;  { *** procedure embedded in CREATE_ELEMENTS *** }
                        { Places point at specified location on the screen }
var choice3: char;

procedure KYBD_PT;      { * 2 procedures embedded in PLACE_POINT * }
                        { Places point at location specified by kybd input }
```

```
var xcoord, ycoord: real;
begin
KBD_INPUT(xcoord,ycoord);
INITIALIZE;
CREATESEG(noln+1);
POINTABS2(xcoord,ycoord);
CLOSESEG;
POSTSEG(noln);
TERMINATE
end;

procedure SCREEN_PT;     { Places point at location specified by screen input }
begin
ERASE_MENU;
write('                  Position Crosshairs and Press Any Key. ');
ALPHANUMPOS(alpha,beta);
INITIALIZE;
REPORT_COORDS(xreport,yreport);
CREATESEG(noln+1);
POINTABS2(xreport,yreport);
CLOSESEG;
POSTSEG(noln);
TERMINATE
end;

begin                                { beginning of PLACE_POINT }
choice3 := '0';
repeat
     ERASE_MENU;
     write('POINT: 1 = SCREEN, 2 = KBD, 3=EXIT ');
     readln(choice3);
     ALPHANUMPOS(alpha,beta);
     case choice3 of
          '1': SCREEN_PT;
          '2': KYBD_PT;
          '3': begin end;
          otherwise ERROR_MESSAGE
     end;          { case }
until choice3 = '3'
end;     { PLACE_POINT }


procedure DRAW_ARROW(deltax,deltay,oldx,oldy:real);{embedded in CREATE_ELEMENTS}

{ Takes the first endpoint of the line and the delta coords to the second
  endpont, calculates the angle used in the transformation, and draws the
  arrowheads at the first endpoint. }

var quad: char;
begin
if (deltax = 0) and (deltay = 0) then
     begin
     TERMINATE;
     ERASE_MENU;
     write('                          Error in drawing dim. lines ');
     INITIALIZE;
     WAIT
     end
else if (deltax > 0) and (deltay >= 0) then     { which quad is dx,dy in? }
     quad := '1'
```

```
    else if (deltax <= 0) and (deltay > 0) then
        quad := '2'
        else if (deltax < 0) and (deltay <= 0) then
            quad := '3'
            else if (deltax >= 0) and (deltay < 0) then
                quad := '4';
case quad of
    '1': eta := ARCTAN(deltay/deltax)*57.3;
    '2': eta := ARCTAN(ABS(deltax)/deltay)*57.3 + 90;
    '3': eta := ARCTAN(ABS(deltay)/ABS(deltax))*57.3 + 180;
    '4': eta := ARCTAN(deltax/ABS(deltay))*57.3 + 270
end;    { case }
UPDATETRANSF(1,1,eta,oldx,oldy);
MOVEABS2(0.0,0.0);
PUSHTRANSF;
UPDATETRANSF(1,1,20,0,0);
LINEABS2((windw[right]-windw[left])/60,0.0);
RESTORETRANSF;
MOVEABS2(0.0,0.0);
UPDATETRANSF(1,1,-20,0,0);
LINEABS2((windw[right]-windw[left])/60,0.0);
POPTRANSF;
CLEARTRANSF
end;                { DRAW_ARROW }

procedure LINE_BY_POINTS;          { * procedure embedded in CREATE_ELEMENTS * }
                                   { Accepts endpoints only }
var choice5,choice6: char;
    xcoord,ycoord: real;
begin
choice5 := '0';
choice6 := '0';
repeat
    ERASE_MENU;
    write('FIRST POINT: 1 = SCREEN, 2 = KBD, 3 = EXIT ');
    readln(choice5);
    ALPHANUMPOS(alpha,beta);
    case choice5 of
    '1': begin
         ERASE_MENU;
         write('                    Position Crosshairs and Press Any Key. ');
         ALPHANUMPOS(alpha,beta);
         INITIALIZE;
         REPORT_COORDS(xreport,yreport);
         CREATESEG(noln+1);
         MOVEABS2(xreport,yreport);
         if witharrows then
            begin
            savex := xreport;
            savey := yreport
            end;
         TERMINATE
         end;
    '2': begin
         KBD_INPUT(xcoord,ycoord);
         INITIALIZE;
         CREATESEG(noln+1);
         MOVEABS2(xcoord,ycoord);
         if witharrows then
            begin
```

```
                savex := xcoord;
                savey := ycoord
                end;
            TERMINATE
            end;
        '3': begin end;
        otherwise ERROR_MESSAGE
        end; { case }
until (choice5 = '1') or (choice5 = '2') or (choice5 = '3');
if (choice5 <> '3') then
repeat
        ERASE_MENU;
        write('SECOND POINT:  1 = SCREEN,  2 = KBD,  3 = EXIT ');
        readln(choice6);
        ALPHANUMPOS(alpha,beta);
        case choice6 of
        '1': begin
             ERASE_MENU;
             write('                    Position Crosshairs and Press Any Key.  ');
             ALPHANUMPOS(alpha,beta);
             INITIALIZE;
             REPORT_COORDS(xreport,yreport);
             LINEABS2(xreport,yreport);
             if witharrows then
                begin
                DRAW_ARROW(xreport-savex,yreport-savey,savex,savey); {1st e.p.}
                if (twoarrows = true) then
                   DRAW_ARROW(savex-xreport,savey-yreport,xreport,yreport) {2nd ep}
                end;
             CLOSESEG;
             POSTSEG(noln);
             TERMINATE
             end;
        '2': begin
             DELTA_INPUT(xcoord,ycoord);
             INITIALIZE;
             LINEREL2(xcoord,ycoord);
             if witharrows then
                begin
                DRAW_ARROW(xcoord,ycoord,savex,savey);       { at first end point }
                if (twoarrows = true) then
                   DRAW_ARROW(-xcoord,-ycoord,savex+xcoord,savey+ycoord) { 2nd e.p. }
                end;
             CLOSESEG;
             POSTSEG(noln);
             TERMINATE
             end;
        '3': begin
             INITIALIZE;
             CLOSESEG;
             DELETESEG(noln);
             TERMINATE
             end;
        otherwise ERROR_MESSAGE
        end;            { case }
until (choice6 ='1') or (choice6 = '2') or (choice6 = '3')
end;                { LINE_BY_POINTS }

procedure LINE_BY_ANGLE;                { * procedure embedded in CREATE_ELEMENTS * }
                                        { Accepts endpoints and angle }
```

```
var choice7,choice8: char;
    xcoord,ycoord,          { from screen input }
    ang: real;              { angle of rotation }
begin
choice7 := '0';
choice8 := '0';
repeat
    ERASE_MENU;
    write('FIRST POINT:  1 = SCREEN,  2 = KBD,  3 = EXIT ');
    readln(choice7);
    ALPHANUMPOS(alpha,beta);
    case choice7 of
    '1': begin
         ERASE_MENU;
         write('                    Position Crosshairs and Press Any Key.  ');
         ALPHANUMPOS(alpha,beta);
         INITIALIZE;
         REPORT_COORDS(xreport,yreport);
         CREATESEG(noln+1);
         MOVEABS2(xreport,yreport);
         savex := xreport;
         savey := yreport;
         TERMINATE
         end;
    '2': begin
         KBD_INPUT(xcoord,ycoord);
         INITIALIZE;
         CREATESEG(noln+1);
         MOVEABS2(xcoord,ycoord);
         savex := xcoord;
         savey := ycoord;
         TERMINATE
         end;
    '3': begin end;
    otherwise ERROR_MESSAGE
    end;            { case }
until (choice7 = '1') or (choice7 = '2') or (choice7 = '3');
if (choice7 <> '3') then begin
ERASE_MENU;
write('ENTER ANGLE ');
ang := GET_REAL;
repeat
    ERASE_MENU;
    write('LENGTH:  1 = SCREEN,  2 = KBD,  3 = EXIT ');
    readln(choice8);
    ALPHANUMPOS(alpha,beta);
    case choice8 of
    '1': begin
         ERASE_MENU;
         write('                    Position Crosshairs and Press Any Key.  ');
         ALPHANUMPOS(alpha,beta);
         INITIALIZE;
         REPORT_COORDS(xreport,yreport);
         UPDATETRANSF(1,1,ang,savex,savey);
         LINEABS2(xreport-savex,yreport-savey);
         CLEARTRANSF;
         if witharrows then
            begin
            DRAW_ARROW(xreport-savex,yreport-savey,savex,savey); {1st e.p.}
            if (twoarrows = true) then
```

```
                DRAW_ARROW(savex-xreport,savey-yreport,xreport,yreport) {2nd ep}
            end;
        CLOSESEG;
        POSTSEG(noln);
        TERMINATE
        end;
    '2': begin
        DELTA_INPUT(xcoord,ycoord);
        INITIALIZE;
        UPDATETRANSF(1,1,ang,savex,savey);
        LINEABS2(xcoord,ycoord);
        CLEARTRANSF;
        if witharrows then
            begin
            DRAW_ARROW(xcoord,ycoord,savex,savey);       { at first end point }
            if (twoarrows = true) then
                DRAW_ARROW(-xcoord,-ycoord,savex+xcoord,savey+ycoord) { 2nd e.p. }
            end;
        CLOSESEG;
        POSTSEG(noln);
        TERMINATE
        end;
    '3': begin
        INITIALIZE;
        CLOSESEG;
        DELETESEG(noln);
        TERMINATE
        end;
    otherwise ERROR_MESSAGE
    end;            { case }
until (choice8 = '1') or (choice8 = '2') or (choice8 = '3');
CLEARTRANSF
end {if }
end;            { LINE_BY_ANGLE }

procedure DRAW_LINE;     { *** procedure embedded in CREATE_ELEMENTS *** }
                  { Draws line between specified end points }
var choice4: char;

procedure SET_STYLE;     { * procedure embedded in DRAW_LINE * }
                         { sets line style for all subsequent lines }
var choice14: char;
begin
choice14 := '0';
repeat
    ERASE_MENU;
    write('STYLE: 1 = SOLID, 2 = BROKEN ');
    readln(choice14);
    ALPHANUMPOS(alpha,beta);
    case choice14 of
        '1': begin
            INITIALIZE;
            CREATESEG(noln+1); { save command in display file }
            SETLINESTYLE(1);
            CLOSESEG;
            POSTSEG(noln);
            TERMINATE
            end;
        '2': begin
            INITIALIZE;
```

```
                CREATESEG(noln+1); { save command in display file }
                SETLINESTYLE(2);
                CLOSESEG;
                POSTSEG(noln);
                TERMINATE
                end;
          otherwise ERROR_MESSAGE
      end;          { case }
until (choice14 = '1') or (choice14 = '2')
end;                { SET_STYLE }

begin                       { begin DRAW_LINE }
choice4 := '0';
repeat
     ERASE_MENU;
     write('LINE: 1= BY POINTS, 2= BY POINTS & ANGLE, 3= SET STYLE, 4= EXIT ');
     readln(choice4);
     ALPHANUMPOS(alpha,beta);
     case choice4 of
          '1': LINE_BY_POINTS;
          '2': LINE_BY_ANGLE;
          '3': SET_STYLE;
          '4': begin end;
          otherwise ERROR_MESSAGE
     end;           { case }
until choice4 = '4'
end;                { DRAW_LINE }

procedure DRAW_DIM_LINE;        { *** procedure embedded in CREATE_ELEMENTS *** }
                                { Draws dimension line between specified end points }
var choice11,choice12: char;

begin                        { begin DRAW_DIM_LINE }
choice11 := '0';
choice12 := '0';
repeat
     ERASE_MENU;
     write('DIM LINE: 1 = ONE ARROW, 2 = TWO ARROWS, 3 = EXIT ');
     readln(choice12);
     ALPHANUMPOS(alpha,beta);
     case choice12 of
          '1': begin
               twoarrows := false;
               write('DIM LINE: 1= BY POINTS, 2= BY POINTS & ANGLE, 3= EXIT ');
               readln(choice11);
               ALPHANUMPOS(alpha,beta);
               case choice11 of
                    '1': LINE_BY_POINTS;
                    '2': LINE_BY_ANGLE;
                    '3': begin end;
                    otherwise ERROR_MESSAGE
               end          { case }
               end;         { '1' }
          '2':   begin
               twoarrows := true;
               write('DIM LINE: 1= BY POINTS, 2= BY POINTS & ANGLE, 3= EXIT ');
               readln(choice11);
               ALPHANUMPOS(alpha,beta);
               case choice11 of
                    '1': LINE_BY_POINTS;
```

```
                '2': LINE_BY_ANGLE;
                '3': begin end;
                otherwise ERROR_MESSAGE
            end          { case }
            end;         { '2' }
        '3': begin end;
        otherwise ERROR_MESSAGE
    end;   { case }
until (choice11 = '3') or (choice12 = '3');
end;            { DRAW_DIM_LINE }

procedure PLACE_TEXT;    { *** procedure embedded in CREATE_ELEMENTS *** }
                         { Place specified text at specified location on screen }
var choice9: char;
    words: packed array[1..16] of char;    { inputted text }

procedure KYBD_TEXT;     { * 4 procedures embedded in PLACE_TEXT * }
                         { Accepts location of text from the keyboard }
var xcoord, ycoord: real;
begin
KBD_INPUT(xcoord,ycoord);
INITIALIZE;
CREATESEG(noln+1);
MOVEABS2(xcoord,ycoord);
TEXT2(words);
CLOSESEG;
POSTSEG(noln);
TERMINATE
end;

procedure SCREEN_TEXT;   { Accepts location of text from the screen }
begin
INITIALIZE;
REPORT_COORDS(xreport,yreport);
CREATESEG(noln+1);
MOVEABS2(xreport,yreport);
TEXT2(words);
CLOSESEG;
POSTSEG(noln);
TERMINATE
end;

procedure EAP_TEXT;      { Allows for the Entering And Placing of text }
var choice10: char;
    ans: char;
begin
repeat
    ERASE_MENU;
    write('Enter Text (<17 chars): ');
    repeat
        readln(words);
    until (words <> blanks16);          { gets text }
    ERASE_MENU;
    write('Text OK? (Y?N): ',words);
    readln(ans);
until (ans = 'Y') or (ans = 'y');
repeat
    ERASE_MENU;
    write('PLACE TEXT: 1=SCREEN, 2=KBD, 3=EXIT ');
    readln(choice10);
```

```
    ALPHANUMPOS(alpha,beta);
    case choice10 of
         '1': SCREEN_TEXT;
         '2': KYBD_TEXT;
         '3': begin end;
         otherwise ERROR_MESSAGE
    end;              { case }
until choice10 = '3'
end;     { EAP_TEXT }

procedure TEXT_SIZE;      { embedded in PLACE_TEXT }
                          { Allows the size of subsequent text to be specified }
begin
ERASE_MENU;
write('Enter Text Size (0-16):  ');
repeat
    readln(primitive.tsize);
until (primitive.tsize <> blanks2);                { gets text size }
primitive.opcode := 'h';
INITIALIZE;
DIRECTINSTR;
TERMINATE
end;              { TEXT_SIZE }

begin                                  { begin PLACE_TEXT }
choice9 := '0';
repeat
    ERASE_MENU;
    write('TEXT:  1= ENTER & PLACE,  2= SET SIZE,  3= EXIT ');
    readln(choice9);
    ALPHANUMPOS(alpha,beta);
    case choice9 of
         '1': EAP_TEXT;
         '2': TEXT_SIZE;
         '3': begin end;
         otherwise ERROR_MESSAGE
    end;              { case }
until choice9 = '3'
end;              { PLACE_TEXT }

procedure DRAW_FORMAT;    { *** procedure embedded in CREATE_ELEMENTS *** }
                          { Draws a border and title block on the screen }
begin
INITIALIZE;
CREATESEG(noln+1);
MOVEABS2(windw[left],windw[bottom]);
LINEABS2(windw[right],windw[bottom]);
LINEABS2(windw[right],windw[top]);
LINEABS2(windw[left],windw[top]);
LINEABS2(windw[left],windw[bottom]);
{MOVEABS2(0.6*windw[right],windw[bottom]);
LINEABS2(0.6*windw[right],0.15*windw[top]);
LINEABS2(windw[right],0.15*windw[top]);}
CLOSESEG;
POSTSEG(noln);
TERMINATE
end;              { DRAW_FORMAT }

procedure DRAW_CIRCLE;
```

```
const twopi = 6.28318;

var radius,dtheta,ct,st,x,y,xtemp,xc,yc: real;
count: integer;

procedure  CIRCLE_POINT(var xcd, ycd: real);
                        { *** procedure embedded in DRAW_CIRCLE *** }
                        { moves to specified location on the screen }
var choice3: char;

procedure KYBD_CIRCLE(var xcoord,ycoord: real);
                        { * 2 procedures embedded in CIRCLE_POINT * }
begin                   { moves to location specified by kybd input }
KBD_INPUT(xcoord,ycoord);
INITIALIZE;
CREATESEG(noln+1);
MOVEABS2(xcoord+x,ycoord+y*aratio)
end;

procedure SCREEN_CIRCLE(var xreport,yreport: real);
                        { moves to location specified by screen input }
begin
ERASE_MENU;
write('                 Position Crosshairs and Press Any Key.  ');
ALPHANUMPOS(alpha,beta);
INITIALIZE;
REPORT_COORDS(xreport,yreport);
CREATESEG(noln+1);
MOVEABS2(xreport+x,yreport+y*aratio)
end;

begin                                   { beginning of CIRCLE_POINT }
choice3 := '0';
repeat
     ERASE_MENU;
     write('CENTER POINT:  1 = SCREEN,  2 = KBD,  3=EXIT ');
     readln(choice3);
     ALPHANUMPOS(alpha,beta);
     case choice3 of
          '1': SCREEN_CIRCLE(xcd,ycd);
          '2': KYBD_CIRCLE(xcd,ycd);
          '3': begin end;
          otherwise ERROR_MESSAGE
     end;          { case }
until (choice3 = '1') or (choice3 = '2') or (choice3 = '3')
end;     { CIRCLE_POINT }

begin
ERASE_MENU;
write('ENTER RADIUS ');
ALPHANUMPOS(alpha,beta+15);
radius := GET_REAL;
dtheta := twopi/16*radius;
ct := cos(dtheta);
st := sin(dtheta);
x := 0;
y := radius;
CIRCLE_POINT(xc,yc);
if (radius < 1) then radius := 1/radius;
for count := 1 to round(16*radius) do
```

```
     begin
     xtemp := x;
     x := (x*ct-y*st);
     y := (y*ct+xtemp*st);
     LINEABS2(xc+x,yc+y*aratio)
     end;
CLOSESEG;
POSTSEG(noln);
TERMINATE
end;     { DRAW_CIRCLE }

begin                          { begin CREATE_ELEMENTS }
choice2 := '0';
repeat
     ERASE_MENU;
     write('CREATE: 1=POINT, 2=LINE, 3=DIM LINE, 4=CIRCLE, 5=TEXT, 6=BORDER, 7=EXIT');
     readln(choice2);
     ALPHANUMPOS(alpha,beta);
     case choice2 of
          '1': PLACE_POINT;
          '2': DRAW_LINE;
          '3': begin
               witharrows := true;
               DRAW_DIM_LINE;
               witharrows := false
               end;
          '4': DRAW_CIRCLE;
          '5': PLACE_TEXT;
          '6': DRAW_FORMAT;
          '7': ERASE_MENU;
          otherwise ERROR_MESSAGE
     end; { case }
until choice2 = '7'
end;

end.
```

LIST OF IGP

EXTERNAL PROCEDURES

```
{ File COREMOD.PRO, a list of external IGP procedures found in COREMOD.PAS }
type stringtype = packed array [1..16] of char;
[EXTERNAL] procedure UPDATESCREEN; EXTERN;
[EXTERNAL] procedure DIRECTINSTR;  EXTERN;
[EXTERNAL] procedure VIEWTRANSF;   EXTERN;
[EXTERNAL] function XTRANS(x: real): real; EXTERN;
[EXTERNAL] function YTRANS(y: real): real; EXTERN;
[EXTERNAL] function RXTRANS(rx: real): real; EXTERN;
[EXTERNAL] function RYTRANS(ry: real): real; EXTERN;
[EXTERNAL] procedure INSTTRANSF(var worldx,worldy: real); EXTERN;
[EXTERNAL] procedure CLEARTRANSF;   EXTERN;
[EXTERNAL] procedure UPDATETRANSF(xscale,yscale,angle,xtran,ytran: real); EXTERN;
[EXTERNAL] procedure PUSHTRANSF; EXTERN;
[EXTERNAL] procedure POPTRANSF;  EXTERN;
[EXTERNAL] procedure RESTORETRANSF; EXTERN;
[EXTERNAL] procedure TEXT(strng: stringtype); EXTERN;
[EXTERNAL] procedure MOVEABS2(x,y: real); EXTERN;
[EXTERNAL] procedure MOVEREL2(dx,dy: real); EXTERN;
[EXTERNAL] procedure CLIPPER(x1,y1,x2,y2,xmin,xmax,ymin,ymax: real); EXTERN;
[EXTERNAL] procedure WINDOW(minx,maxx,miny,maxy: real); EXTERN;
[EXTERNAL] procedure VIEWPORT2(minx,maxx,miny,maxy: real); EXTERN;
[EXTERNAL] procedure POINTABS2(x,y: real); EXTERN;
[EXTERNAL] procedure POINTREL2(dx,dy: real); EXTERN;
[EXTERNAL] procedure LINEABS2(x,y: real); EXTERN;
[EXTERNAL] procedure LINEREL2(dx,dy: real); EXTERN;
[EXTERNAL] procedure SETLINESTYLE(style: integer); EXTERN;
[EXTERNAL] procedure SETCOLOR(color0,color1,color2: real); EXTERN;
[EXTERNAL] procedure CLOSESEG;   EXTERN;
[EXTERNAL] procedure CREATESEG(segnum: integer); EXTERN;
[EXTERNAL] procedure POSTSEG(snum: integer); EXTERN;
[EXTERNAL] procedure UNPOSTSEG(sn: integer); EXTERN;
[EXTERNAL] procedure DELETESEG(segno: integer); EXTERN;
[EXTERNAL] procedure RESETVARS;  EXTERN;
[EXTERNAL] procedure INITIALIZE;  EXTERN;
[EXTERNAL] procedure TERMINATE;   EXTERN;
[EXTERNAL] procedure ALPHANUMPOS(l,c: integer); EXTERN;
[EXTERNAL] procedure WAIT;                EXTERN;
[EXTERNAL] procedure ERASE_MENU;          EXTERN;
[EXTERNAL] procedure ERROR_MESSAGE;       EXTERN;
[EXTERNAL] function  GET_REAL:real;       EXTERN;
[EXTERNAL] procedure USE_OLD_DRWG;        EXTERN;
[EXTERNAL] procedure SAVE_DRWG;           EXTERN;
[EXTERNAL] procedure DELETE_ELEMENTS;     EXTERN;
[EXTERNAL] procedure CREATE_ELEMENTS;     EXTERN;
```

PADL-2 BEARING

PART FILE

```
\ THE FOLLOWING IS A PADL-2 PART DEFINITION OF "BEARING" WRITTEN BY ED MARUGGI.
\
\ BEARING IS COMPRISED OF 6 GENERICS: BEARING, BASE, TOP, GUIDE, BLTHOLE, & ROUND.
\ THIS PART WAS CHOSEN BECAUSE IT INCORPORATES A GREAT DEAL OF PADL-2'S
\ ABILITY TO MODEL SOLID OBJECTS, AND IN THIS WAY DEMONSTRATES MOST SOLID
\ MODELING CONCEPTS USED BY PADL-2.
\
\
GENERIC BEARING(BEARING);                    { MAIN GENERIC - FORMS TOTAL PART }
\
BEARING = BASE( ) UN TOP( ) UN GUIDE( ) DIF BLTHOLE( );
```

PADL-2 BASE

PART FILE

```
GENERIC BASE(BASE);                    { BASE OF BEARING }
\
Y = 1.5;
A = -90.0;
\
BASEHOLE = CYL(R=.25, H=Y);                    { CREATE BOLT HOLES }
CS1 = DEGX=A, MOVX=.75, MOVZ=.75;
CS2 = CEGX=A, MOVX=.75, MOVZ=4.25;
CS3 = DEGX=A, MOVX=10, MOVZ=4.25;
CS4 = DEGX=A, MOVX=10, MOVZ=.75;
BASEHOLE1 = BASEHOLE MOVEDBY CS1;
BASEHOLE2 = BASEHOLE MOVEDBY CS2;
BASEHOLE3 = BASEHOLE MOVEDBY CS3;
BASEHOLE4 = BASEHOLE MOVEDBY CS4;
\
R = .25;
SLOTBLK = BLO(X=0.25, Y=0.5, Z=2.25);    { CREATE SLOT }
SLOTCYL = CYL(R=R, H=R) MOVEDBY DEGY=-A, MOVY=R;
SLOT = (SLOTBLK UN SLOTCYL) MOVEDBY MOVY=0.5, MOVZ=2.75;
\
BASEBLK = BLO(X=10.75, Y=Y, Z=5.0);      { CREATE BASE BLOCK }
\
SIDECYL = CYL(R=.5, H=R) MOVEDBY (MOVY=.75, MOVX=.375)\ { CREATE SIDE MOUNT }
INT BASEBLK;
SIDEMOUNT = SIDECYL MOVEDBY MOVX=6.375, MOVZ=5.0-R;
\
RAD = 0.125;                                   { CREATE FRONT MOUNTING BLOCK }
T = 0.25;
A1 = ATAN(4);
A2 = 3.14159-ATAN(4);
MOUNTBLOCK = BLO(X=0.625, Y=1, Z=T);
CUTBLOCK = BLO(X=1.5, Y=1.0, Z=T) MOVEDBY ROTZ=A1;
CUT1 = ROUND(R=RAD, A=A1, T=T);
CS5 = ROTZ=-A2, MOVX=0.25, MOVY=1.0;
CUT2 = ROUND(R=RAD, A=A2, T=T) MOVEDBY CS5;
MNTBLOCK = MOUNTBLOCK DIF CUTBLOCK DIF CUT1 DIF CUT2;
MNBLOCK = MNTBLOCK MOVEDBY DEGY=180, MOVZ=T, MOVX=1.25;
MBLOCK = MNBLOCK UN MNTBLOCK;
FMOUNT1 = MBLOCK MOVEDBY DEGY=90, MOVX=10.5, MOVZ=2.25, MOVY=0.25;
FMOUNT2 = MBLOCK MOVEDBY DEGY=90, MOVX=10.5, MOVZ=4.0, MOVY=0.25;
\
SBLK = BLO(X=0.125, Y=0.25, Z=0.75);                    { CREATE FRONT SLOTS }
SCYL = CYL(R=RAD, H=RAD) MOVEDBY DEGY=-A, MOVY=RAD;
SLT = (SBLK UN SCYL) MOVEDBY MOVY=0.625, MOVX=10.625;
SLT1 = SLT MOVEDBY MOVZ=0.625;
SLT2 = SLT MOVEDBY DEGY=180, MOVX=21.375, MOVZ=4.5;
\
BASE = BASEBLK DIF BASEHOLE1 DIF BASEHOLE2\  { REMOVE SOLIDS FORM BASE }
DIF BASEHOLE3 DIF BASEHOLE4 DIF SLOT DIF SIDEMOUNT\
DIF FMOUNT1 DIF FMOUNT2 DIF SLT1 DIF SLT2;
```

PADL-2 TOP

PART FILE

```
GENERIC TOP(TOP);                                   { THIS IS THE TOP BLOCK FOR BEARING }
\
CS = DEGX=90, MOVY=3, MOVX=2, MOVZ=2;    { CREATE BOLT HOLES ON BOLT CIRCLE }
BCYL = CYL(R=0.1875, H=0.25);
BCR = 1.625;                    { BOLT CIRCLE RADIUS }
BH1 = BCYL MOVEDBY MOVY=-BCR, CS;
BH2 = BCYL MOVEDBY MOVY=-BCR, DEGZ=-30, CS;
BH3 = BCYL MOVEDBY MOVY=-BCR, DEGZ=-60, CS;
BH4 = BCYL MOVEDBY MOVX=-BCR, CS;
BH5 = BCYL MOVEDBY MOVX=-BCR, DEGZ=-30, CS;
BH6 = BCYL MOVEDBY MOVX=-BCR, DEGZ=-60, CS;
BH7 = BCYL MOVEDBY MOVY=BCR, CS;
BH8 = BCYL MOVEDBY MOVY=BCR, DEGZ=-30, CS;
BH9 = BCYL MOVEDBY MOVY=-BCR, DEGZ=30, CS;
\
\                                                   { CREATE TOP BLOCK }
X = 7;              { LEGNTH OF BLOCK }
TOPBLOCK = BLO(X=X, Y=3, Z=4);
\
CUTANGL = ATAN(3.5/2);                              { CREATE CUT BLOCK }
CUTTER = BLO(X=2.5, Y=5, Z=4) MOVEDBY ROTZ=CUTANGL, MOVX=X, MOVY=1;
\
BEARCYL = CYL(R=1,H=1);                             { CREATE BEARING }
CONE1 = CON(R=1, H=1) MOVEDBY MOVZ=0.5;
CONE2 = CON(R=1, H=1.5);
CHMF = (CONE2 DIF CONE1) MOVEDBY DEGY=180, MOVZ=2.5-0.062;
BEARCHMF = BEARCYL DIF CHMF;
YDIST = 0.5*SIN(CUTANGL)/COS(CUTANGL)+1.125;
CSBEAR = MOVX=X-1.75, MOVY=YDIST, MOVZ=2;
BEARING = BEARCHMF MOVEDBY DEGX=-90, ROTZ=CUTANGL-3.14159/2 MOVEDBY \
CSBEAR;
\
\                                                   { CREATE BEARING SHAFT HOLE }
W = (YDIST-0.5*COS(CUTANGL))/SIN(CUTANGL);          { LENGTH OF HOLE IN TOP }
SHAFTHOLE = CYL(R=.5, H=W+1) MOVEDBY MOVZ=-W, DEGX=-90, ROTZ=CUTANGL\
-3.14159/2 MOVEDBY CSBEAR;
\
\                                                   { CREATE BACK MOUNT HOLE }
THICK = .25;                    { DEPTH OF MOUNT HOLE }
RADIUS = .25;                   { RADIUS OF ROUND }
CSTIP = ROTX=-ATAN(0.25/1.5);                       { ANGLE BY WHICH BLOCK IS TIPPED }
BACKMOUNT = BLO(X=THICK, Y=1.75, Z=2) MOVEDBY CSTIP;
WEDGE = WED(X=THICK, Y=1.75, Z=0.7) MOVEDBY CSTIP;
WEDGE1 = WEDGE MOVEDBY ((DEGZ = 180, MOVX=THICK, MOVY=1.75) WRT CSTIP);
WEDGE2 = WEDGE1 MOVEDBY MOVZ=2 WRT CSTIP;
RND1 = ROUND(R=RADIUS, A=ATAN(1.25/0.5)-ATAN(0.25/1.5), T=THICK);
RND2 = ROUND(R=RADIUS, A=3.14159-ATAN(1.25/0.5)-ATAN(0.25/1.5), T=THICK);
RD1 = RND1 MOVEDBY DEGY=-90, MOVX=THICK, CSTIP;
RD2 = RD1 MOVEDBY DEGY=180, ROTX=-ATAN(1.75/.7), MOVX=THICK, MOVY=1.75, \
MOVZ=2.7 WRT CSTIP;
RD3 = RND2 MOVEDBY CSTIP, DEGY=90, MOVZ=2 WRT CSTIP;
RD4 = RD3 MOVEDBY MOVZ=-2, DEGX=180, MOVZ=.7, MOVY=1.75 WRT CSTIP;
BCKMOUNT = BACKMOUNT DIF WEDGE1 UN WEDGE2 DIF RD1 DIF RD2 DIF RD3 \
DIF RD4;
BMOUNT = BCKMOUNT MOVEDBY MOVZ=.562, MOVY=.428;
TPBLOCK = TOPBLOCK DIF CUTTER UN BEARING DIF SHAFTHOLE DIF BMOUNT \
DIF BH1 DIF BH2 DIF BH3 DIF BH4 DIF BH5 DIF BH6 DIF BH7 DIF BH8 DIF BH9;
TOP = TPBLOCK MOVEDBY MOVX=2, MOVY=1.5, MOVZ=.5;
```

PADL-2 BLTHOLE

PART FILE

```
GENERIC BLTHOLE(BLTHOLE);                    { CREATE BOLT HOLE }
\
BOLTCYL1 = CYL(R=1.5, H=.25);
BOLTCYL2 = CYL(R=1.25, H=.25) MOVEDBY MOVZ=.25;
BOLTCYL3 = CYL(R=1, H=3.5) MOVEDBY MOVZ=.5;
BOLTCYL4 = CYL(R=1.25, H=.5) MOVEDBY MOVZ=4;
BOLTHOLE = BOLTCYL1 UN BOLTCYL2 UN BOLTCYL3 UN BOLTCYL4;
BLTHOLE = BOLTHOLE MOVEDBY DEGX=-90, MOVX=4, MOVZ=2.5;
```

PADL-2 GUIDE

PART FILE

```
GENERIC GUIDE(GUIDE);    \ CREATES SIDE GUIDE
GUIDECYL = CYL(R=1, H=.25) MOVEDBY MOVY=1.5, MOVX=1;
GUIDEBLOCK = BLO(X=2, Y=1.5, Z=.25);
FBLOCK = WED(X=.25, Y=.25, Z=.25) MOVEDBY DEGY=-90;
FILLET = FBLOCK DIF (CYL(R=.25,H=.25) MOVEDBY MOVX=-.25,MOVY=.25);
FILLET1 = FILLET MOVEDBY MOVX=3.0, MOVY=1.5, MOVZ=.25;
FILLET2 = FILLET MOVEDBY DEGY=180, MOVX=5.0, MOVY=1.5, MOVZ=.5;
GUIDECB = (GUIDECYL UN GUIDEBLOCK) MOVEDBY MOVX=3.0, MOVY=1.5, MOVZ=.25;
GUIDE = GUIDECB UN FILLET1 UN FILLET2;
```

PADL-2 ROUND

PART FILE

```
GENERIC ROUND(ROUND);
\
\  INPUT R = RADIUS OF ROUND
\        A = ANGLE OF CORNER IN RADIANS
\        T = THICKNESS OF OBJECT (Z DIRECTION)
\
\  OUTPUT IS AN OBJECT WHICH WHEN DIFFERENCED WITH THE ORIGINAL OBJECT WILL
\        ALLOW A CYLINDER TO BE UNIONED WITH THE OBJECT TO PROCUCE A ROUNDED
\        CORNER.
\
\  THE CUT OBJECT IS CREATED AT LAB WITH ONE OF THE EDGES COINCIDING WITH
\        THE X-AXIS, AND MUST BE POSITIONED AT THE PROPER CORNER BEFORE
\        DIFFERENCING.
\
{ SET DEFAULT PARAMETERS }
A = ATAN(4);
R = 0.125;
T = 0.25;
\
D = R/(SIN(A/2)/COS(A/2));        { DISTANCE FROM CORNER TO TANGENT POINT }
\
{ BUILD CUT WEDGE (2 UNIONED WEDGES) }
\
W1 = WED(X=T, Y=R, Z=D) MOVEDBY MOVZ=-D, ROTX=(3.14159-A)/2;
W2 = W1 MOVEDBY ROTY=3.14159, MOVX=T;
WEDGE = (W1 UN W2) MOVEDBY ROTZ=-3.14159/2, ROTX=3.14159/2, MOVZ=T;
\
HYP = R/SIN(A/2);                 { HYPOTENUSE }
\
C1 = CYL(R=R, H=T) MOVEDBY MOVX=HYP;      { CYLINDER }
\
ROUND = (WEDGE DIF C1) MOVEDBY ROTZ=A/2;          { FINAL OBJECT }
```

PADL-2 BEARING COMMAND FILE

```
\ Filename: DISP_BEARING.PFI
\ This file displays 3 orthographic views of the generic named BEARING found
\ in the BEARING.PFI file.
\
use bearing;
disp bearing;
ugb;              \ to neatly fit object on the screen
set device=16;    \ entire screen is initialized and mapped to VT-240
\
.toparea = 055000;        \ upper left screen display area
.frontarea = 005500;      \ bottom left screen display area
.rightarea = 500500;      \ bottom right screen display area
\
.topview = (view_type = 0,direction = degx = -90);
.frontview = (view_type = 0,direction = [AB);
.rightview = (view_type = 0,direction = degy = 90);
\
set device = .toparea,.topview;
disp bearing;
set device = .frontarea,.frontview;
disp bearing;
set device = .rightarea,.rightview;
disp bearing;
dcl 'r igp'
```

PADL-2 BASE

COMMAND FILE

```
\ Filename: BASEORTHO.PFI
\ This file displays 3 orthographic views of the generic named BASE found
\ in the BASE.PFI file and then enters the IGP program.
\
use base;
disp base;
ugb;                \ to neatly fit object on the screen
set device=16;      \ entire screen is initialized and mapped to VT-240
\
.toparea = 055000;          \ upper left screen display area
.frontarea = 005500;        \ bottom left screen display area
.rightarea = 500500;        \ bottom right screen display area
\
.topview = (view_type = 0,direction = degx = -90);
.frontview = (view_type = 0,direction = [AB);
.rightview = (view_type = 0,direction = degy = 90);
\
set device = .toparea,.topview;
disp base;
set device = .frontarea,.frontview;
disp base;
set device = .rightarea,.rightview;
disp base;
dcl 'r igp'
```

PADL-2 GUIDE

COMMAND FILE

```
\ Filename: DISP_GUIDE.PFI
\ This file displays 3 orthographic views of the generic named GUIDE found
\ in the GUIDE.PFI file.
\
use guide;
disp guide;
ugb;                \ to neatly fit object on the screen
set device=16;      \ entire screen is initialized and mapped to VT-240
\
.toparea = 055000;          \ upper left screen display area
.frontarea = 005500;        \ bottom left screen display area
.rightarea = 500500;        \ bottom right screen display area
\
.topview = (view_type = 0,direction = degx = -90);
.frontview = (view_type = 0,direction = [AB);
.rightview = (view_type = 0,direction = degy = 90);
\
set device = .toparea,.topview;
disp guide;
set device = .frontarea,.frontview;
disp guide;
set device = .rightarea,.rightview;
disp guide;
dcl 'r igp'
```

10

USER'S MANUAL

10.1. Overview of IGP.

IGP is a Solid Modeling (SM) interactive interface for design drafting that allows a user to create complete engineering drawing document displays from displays produced by SM systems. Although IGP was designed to have the ability to be used with more than one SM system and more than one graphic device, presently it can only be used with the PADL-2 SM system running on the VAX/VMS operating system with a VT-240 or compatible device.

IGP writes messages in an area that would be just underneath an engineering drawing's border. In this way the messages are least likely to interfere with the drawing itself.

10.2. Preliminary Set-up.

The user must first generate an appropriate part display using the SM system before IGP can be used to complete the display. With PADL-2 the user must create a part description and use display commands or a command file to display the part (see PADL-2 user's manual [HART83]). IGP can be entered from PADL-2 through the DCL interface by the command DCL 'R IGP'.

For the PADL-2 at RIT software, using the DISP command causes problems with the display that requires resetting the terminal under the set-up key in order for the IGP display characteristics to work properly.

Resetting the terminal should occur just after entering IGP.

10.3. Defining the Window.

Upon entering IGP the user is prompted for input values to set up the coordinates of the user's drawing area. Then the main menu is displayed on the screen.

10.4. The Main Menu.

There are six options under the main menu: USE, SAVE, CREATE, DELETE, ERASE MENU, and EXIT IGP. The user can choose one of these functions by entering the number that corresponds to the desired option.

10.4.1. USE.

This allows the user to input an existing drawing to the display area. The user is prompted for file name (without the extension). The file must reside in the current directory or an error message is given.

10.4.2. SAVE.

This allows the user to save the elements that have been created and displayed. The user is prompted for a file name (without the extension). A display file by that name is created in the current directory with the .DAT extension.

10.4.3. CREATE.

This option allows the user to create new graphic elements and display them on the screen. There are several levels under this option that provide all of the capabilities necessary to create graphic and textual elements. The create menu is displayed on the screen so the user may select more options. The options under this menu are:

POINT, LINE, DIM LINE, CIRCLE, TEXT, BORDER, and EXIT.

10.4.3.1. POINT.

A point can be created and displayed at a specified location by using precision input through the keyboard, or by using the screen crosshairs.

10.4.3.2. LINE.

A line can be specified by end points or endpoints and an angle. Input can be through the keyboard or through the use of the screen crosshairs. Also, the style of future lines can be set under the line menu.

10.4.3.3. DIM LINE.

Dimension line options are the same as line options. The only difference is that dimension lines can be drawn with an arrow at one end or an arrow at both ends.

10.4.3.4. CIRCLE.

A circle can be specified by the radius and the location. The radius is entered with the keyboard while the location can be entered with the keyboard or through the use of the screen crosshairs.

10.4.3.5. TEXT.

Text can be entered and placed, or the size of future text can be set. The text size option does not currently work for the VT-240.

10.4.3.6. BORDER.

Selecting this option draws a border for the drawing. This is done automatically by IGP.

10.4.4. DELETE.

This option allows the user to delete specified elements. The user is prompted for the number of the element to be deleted. After an appropriate value is input, the corresponding element is highlighted, and the user is asked if that element is the one that should be deleted. If the answer is "yes", the element is deleted, and the next sequential element is highlighted for deleting. If the user answers "no", the highlighted element will not be deleted, and the delete loop will continue to the next sequential element. The user can escape the delete loop by choosing the exit option from the menu.

10.4.5. ERASE MENU.

The user can erase the menu so that only the drawing elements themselves are displayed. This is useful if a hard copy screen dump is made to an output device.

10.4.6. EXIT IGP.

This terminates the IGP program and returns the user to PADL-2. The STOP command is used to exit PADL-2.

```
12-56 IS4000

PLEASE ENTER DESTINATION, "HELP", "STATUS" OR "DIR" (DIRECTORY):
vax
PAUSE

CONNECTED TO 02-03

Username: EPM9970
Password:

Welcome to VAX/VMS Version 4.7 on RIT VAXCluster node VAXA

    Last interactive login on Thursday, 11-AUG-1988 15:46
    Last non-interactive login on Saturday,  8-NOV-1986 16:56

[Ed] cd thesis.igfiles
[Ed] padl2



  -- If your terminal is TEXTRONIX 4010 compatible, type

     SET DEVICE=4

  -- If your terminal is a VT-240 or VT-241, type

     SET DEVICE=16

  -- For further help, type

     HELP

           Initializing PADL2.  Be back in a few seconds.

     0.00 P2>disp_guide
    Last non-interactive login on Saturday,  8-NOV-1986 16:56
    Last non-interactive login on Saturday,  8-NOV-1986 16:56

   This is IGP: A Solid Modeling Interactive Interface for Design Drafting

   Reset terminal with Set-Up key if necessary.

    ENTER MIN X WINDOW COORD: 0

    ENTER MAX X WINDOW COORD: 10

    ENTER MIN Y WINDOW COORD: 0

    ENTER MAX Y WINDOW COORD: 10

   MAIN: 1=USE, 2=SAVE, 3=CREATE, 4=DELETE, 5=ERASE MENU, 6=EXIT IGP 1
```

Figure 10-1. Beginning an interactive session.