This paper discusses a system designed for interactive problem solving by use of a graphic display console. Existing application programs can readily be modified for use with a graphic display device, and the graphics programming can usually be done in a higher-level language. Based on intermediate results, the order of execution of application modules can be controlled from the console.

The system description emphasizes the structure and generation of display formats for displaying output, for accepting user-defined commands, and for accepting data that is made accessible to the application modules. Also described is a generalized data structure and a set of experimental routines designed to adapt the structure to particular needs.

INTERACTIVE GRAPHICS IN DATA PROCESSING

A system for implementing interactive applications

by F. C. Chen and R. L. Dougherty

Computer-aided design has evolved from batch processing through keyboard-type conversational-mode operation to display-type man-machine interaction.[1] So far, however, use of display console interaction has been inhibited by the complexity of graphics programming and the difficulty in some programming languages of executing program segments in an arbitrary order.

problem definition

The programming required to create a graphics interface for a new or existing application program has been simplified by general-purpose graphics support programs, such as that described by Rully in this issue. However, creating the graphics interface requires conventional programming, and the coding is specialized for a particular application. In addition, display subroutines have to be coded for the particular system (for example, IBM 1130 versus SYSTEM/360) on which they will be run. Thus, graphics programming would be easier if basic display images and interactive controls could be specified in high-level, user-oriented statements. Moreover, the coding of graphics interfaces would be simpler if a program written for one system could be made acceptable to another system with only slight modifications.

In interactive problem solving, unanticipated situations frequently arise that make complete preplanning difficult or impossible. It is therefore necessary to be able to perform functions as the need for them becomes apparent during the problem-solving process. Thus, execution-time control over the order of execution of program modules is needed for interactive problem solving, because

unanticipated computations must be performed. This capability is provided, for example, in assembler language as implemented in the IBM SYSTEM/360 Operating System, a language unfamiliar to most graphics users. However, it is not provided in some higher-level programming languages. The ability to dynamically load program modules is also desirable to conserve main storage space.

problem solution

The approach to computer-aided design discussed here facilitates the use of graphics in several ways. The computational and the graphics portions of application programs are separated, so that libraries of existing programs can be modified and recompiled for graphic interactive problem solving. Furthermore, standard display program modules can be interfaced with application programs by means of problem-oriented language statements. This approach also facilitates the division of the computational portions of application programs into segments or modules, even if they are coded in programming languages that do not provide for dynamic loading of program modules. Such modules can then be called and executed in any order by the console operator.

The program discussed in this paper is called PLAN Graphics Support (PGS), which in turn is supported by the PLAN system.[2] PLAN provides the capability for creating, interpreting, and executing problem-oriented statements for requesting execution of modules, manipulating data, etc. Instead of problem-oriented statements, the primary communication media for PGS are display formats created at a particular installation for the IBM 2250. This paper is focused on those aspects of PLAN that enhance graphics and does not discuss its capabilities for interactive computer-aided design using nongraphics terminals.

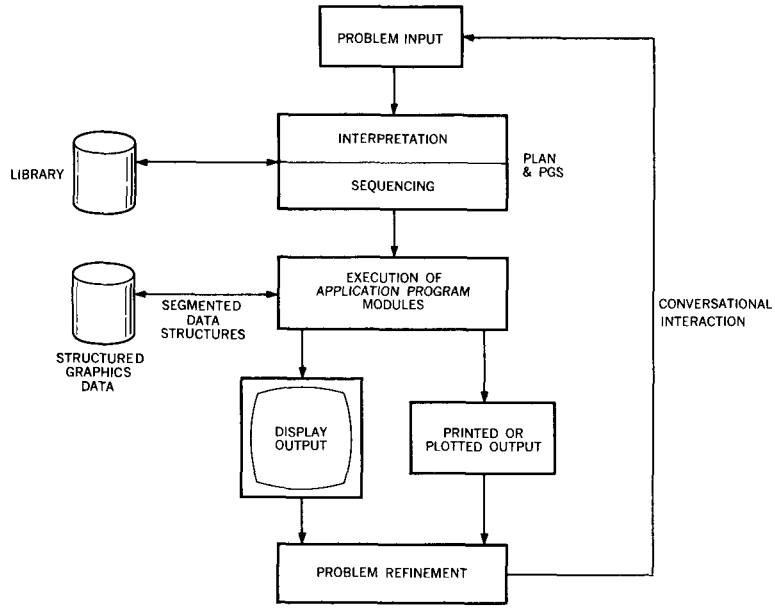
PLAN and PGS are executed under the SYSTEM/360 Operating System and under the IBM 1130 Monitor, Version 2, using the facilities of this programming support whenever appropriate. The graphics subroutine package (GSP), described by Rully in this issue, is also used, although the functions performed by GSP are usually requested at a higher level not requiring conventional programming.

This paper describes the overall design of PLAN-PGS, including detailed descriptions of its major features. PLAN-PGS is designed to provide a consistent user interface to the SYSTEM/360 and the 1130; however,where internal differences exist, the implementation used in SYSTEM/360 is described. Data structures are also discussed, including one possible approach for associating properties with data entities, for model segmentation, and for dynamic loading of data segments.

System design

Interactive problem solving using PLAN-PGS is illustrated by the closed-loop system in Figure 1. Problem input is interpreted, and the execution sequence of the application program modules is determined. After the appropriate modules have been executed,

Figure 1 Computer-aided design environment



intermediate results are made available for review and possible refinement. At this time, the console operator may decide that other computations are needed or that different data should be displayed. If the computation modules have been previously coded and exist in his library and if appropriate provisions for display and interaction have been defined, he can proceed in this interactive fashion to his problem solution.

dynamic program control

PLAN provides a dynamic loading capability that eliminates most of the programming effort connected with preplanning the sequences for executing program modules. For example, FORTRAN source program modules designed to perform discrete functions can be written in conformance with established conventions. Application modules written for use under PLAN make extensive use of a common data area, to enable communication among separately executed application modules. After each module has been compiled and processed for loading, the modules can be loaded and executed in any meaningful sequence by the console operator without help from a programmer.

In this environment, the scope of an application program is not rigidly defined; thus it does not have to be redefined for every addition or change of logic. Modules can be added to libraries without requiring an existing program to be rewritten, recompiled, and reprocessed for loading.

To gain access to these functional modules, dictionary entries are created that name library modules, supply default data, and define a language devised at the installation and based on the

nature of the applications. For any given application, the sequence of execution of program modules may be predefined or it may be specified during the course of the analysis.

The display formats of PGS are similar to the problem-oriented language of PLAN. Under PLAN, each computation step definition includes a phrase, a data list, and a program sequence. A *phrase* is defined as a unique group of words that identifies a command. A *data list* defines data items, values, and output. A *program sequence* identifies one or more program modules to be executed when the defined phrase is specified.

PLAN graphics support

PGS supplements PLAN for graphics applications, primarily by facilitating development of the graphics portions of interactive graphics applications, usually without conventional programming. The flow of operation of PLAN-PGS is shown in Figure 2. Several conditions are required for operation of the system.

The state of the entire 2250 console must have been predefined. Thus, one condition for operation of the system is that the data needed to control the state of the 2250 must be available either on *specification files* (i.e., in a form independent of display devices) or on *panel files* (i.e., in 2250 graphics order format). Another requirement is that the appropriate compiled program modules needed to perform computation functions are available in a library.

To initiate an application, PLAN itself must be loaded, of course. For example, in the SYSTEM/360 Operating System environment, the job control statements needed to describe PLAN as a job are required. Because the system resources needed for application programs run under PLAN are allocated at this time, the console operator need not concern himself with job control statements for each application.

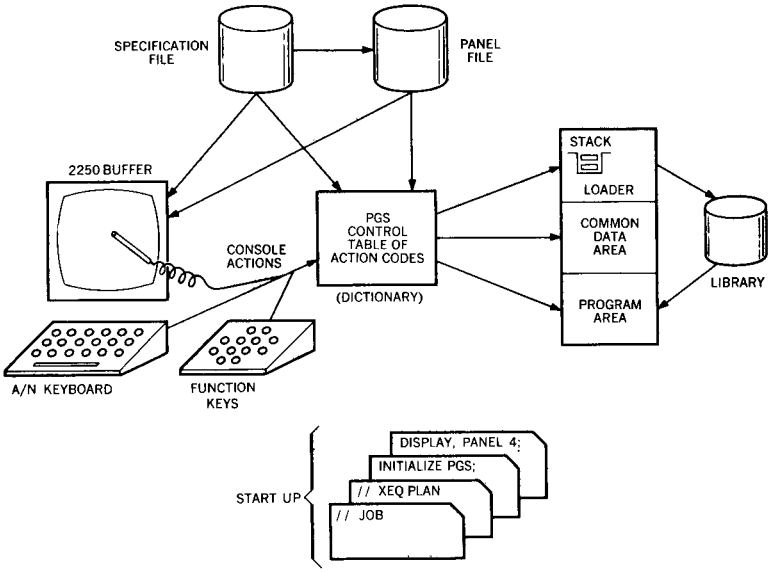
PGS is then initialized to run under PLAN. PGS sets up a control table in main storage. Data stored in this table is used to interpret light pen and keyboard actions to satisfy such requests as: to transfer control to a program module already in main storage, to place a program name in a last-in-first-out stack so that the program can be dynamically loaded by PLAN, or to place input data in a common data area so that it is accessible to the application program.

panels

The state of the 2250 at a given time is called a *panel.* This state includes more than the image on the 2250 screen; it includes, for example, the significance of the function keys at the time the image is being displayed. Panels fulfill the basic functions of accepting data from the user and of displaying data resulting from application module execution. Depending on the alternatives, a panel may provide the medium for accepting a command, which is passed to PLAN for execution; a panel may also be designed to accept data, which is stored in a common data area accessible to application modules; and a panel may be the means of displaying data.

The provisions in PGS for designing panels enable panel contents to be described in terms of text and graphics. Panels also allow the

Figure 2 PLAN-PGS program flow



specification of system response to user interaction, such as panel switching, data entry, and data display.

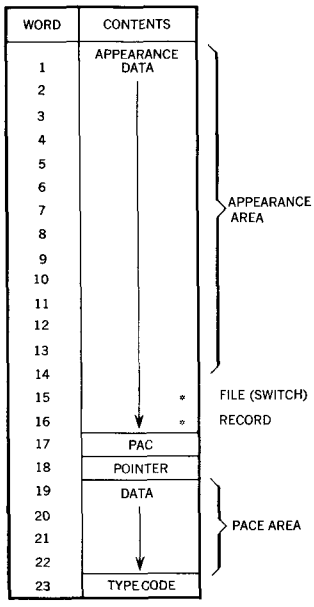
panel switching

As long as the console operator is entering data on the same panel, that panel remains on the screen after each console action, and the data is placed in the common area shown in Figure 2. However, a problem arose when we considered how to switch to a new panel as a result of a console action. By storing panel data in graphics order format, panels can be switched rapidly, avoiding uncomfortable delays for the console operator and facilitating animated displays. In contrast, device-independent data is more compact, more flexible, and requires less external storage space. In the final design, we decided to allow data to be stored in either form so that the compromise between response time and storage space could be made at the installation. Display information can be kept in graphics order format in the panel file; raw data needed to construct panels can be kept in the specification file.

When curves or graphs are to be displayed using data produced by application program execution, the output data is normally stored in an array on a disk file. These arrays can then be referred to in a panel description on the specification file so that output data can be displayed within the framework of predefined panels.

The arrows from the specification and panel files in Figure 2 show the various ways that display output can be created on the 2250. Note that the arrows also show information flowing into the control table; this is necessary since there must be an "action code" reference for each display item, so that it can be detected by the light pen or associated with a function key. (Action codes are discussed in greater detail later.)

Figure 3 Specification record



In addition to panel and specification files, display files are needed for the actual display of an image. A 2250 display results from graphics orders and must be continually regenerated. To identify the elements of the image (for light-pen detection, for example), control data is also required. In PGS, this control data is kept in control tables in main storage. The regenerating orders together with these control tables constitute an *activated display file*. When the 2250 is attached to the 1130, the display file is an in-storage array. For the SYSTEM/360, the graphics orders for regeneration are kept in the display unit buffer and the control tables in main storage. For the 1130, the panel file is a disk file copy of the display file. For the SYSTEM/360, the panel file is kept in main storage in the form of an equivalent graphics data set plus control tables. There may be as many as fifty of these panel files, each with a unique identification number from one to fifty.

Both panel and display files can be created from specification files. Specifications include such information as the actual text for a display, canonical-form data of graphics items, display screen coordinates, light pen and function key action codes, and identification data.

records

Specifications for each panel are contained in individual 23-word records, as shown in Figure 3. Records may be formed from each command statement pertaining to the creation of a panel, and for every panel there may be as many as desired. Records are grouped in a uniquely identified specification file.

program action codes

Associated with every record is a program action code (PAC). The PAC determines program flow and function. Each PAC specifies control information to be associated with display specifications or attention signal sources (program function keys and the END key on the alphanumeric keyboard). Program action codes are classified according to the functions with which they are associated:

- PLAN monitor functions
- Panel switching and display control functions
- Logical functions
- Functions related to the entry, manipulation, and display of data values

Each program action code is a four-digit number. The first digit specifies when the action is to be performed, which may be at display generation time, at console attention signal time, or at both times. The second digit specifies the type of function, and the last two digits specify the particular program action. The program action code is also designed to allow an alternative type of function at display generation time. If certain digits are specified as the first digit, such a digit replaces the second digit at display generation time.

A program action code extension (PACE) is a field added to a PAC and used to specify additional information that may be required with some program action codes. A PACE is divided into

a pointer used to specify a particular common data location and a subfield used to specify the information.

A record also contains an appearance code, which specifies the information to be displayed on the screen. Appearance specifications may include text, lines or line sets, points or point sets, and geometric entities such as circles, arcs, ellipses, hyperbolas, and parabolas. The display item specification is identified by a type code.

appearance codes

In some cases, the area allocated for appearance data within the specification record may not be adequate. For this reason, a pointer is provided (words 15 and 16 in Figure 3), in which is specified an external disk file and record identification. Thus, the appearance data can be obtained when the panel associated with the record is to be displayed.

Whenever a specification file is to be activated, i.e., either to be displayed or to be made into a panel file, the PGS monitor processes each record and generates graphics orders that are placed into the specified graphics data set. The PAC/PACE information is retained in a control table for the particular panel. (The PAC is also analyzed to determine whether any activity is to be done at this time.)

When a panel is being displayed, the PAC/PACE table for the panel is interrogated by the PGS interruption analyzer, and the appropriate actions are taken. (At present, about one hundred PAC's have been coded into PGS.)

A set of command statements is provided so that a panel designer can specify, generate, and maintain panels. These statements conform with the rules for PLAN commands (mentioned earlier). The general form of these statements is:

generating panels

COMMAND NAME, DATA NAME AND VALUE, DATA NAME AND VALUE, . . . ;

After the command name, several data names may be specified, separated by commas. The statement ends with a semicolon. Data names and values may be entered in any order.

Table 1 shows, by category, a representative set of commands. The data associated with a particular command varies depending on the function desired. In the case of specification statements, the data can be of two types:

Appearance, which may include text, points, and other graphic entities with their locations. Lines and conics may be specified along with line types and smoothness factors.

Function, in which program control may be indicated via a PAC and its PACE. For example, a PAC might request a program to be called as a result of a light-pen detect on an entity in the appearance section, and the PACE might provide the name of the program to be called. The system contains a predefined set of PAC's for performing various control and data management functions. Each of the com-

Table 1 Sample panel commands

Category	*Command name*
Initialization and termination	INITIALIZE PGS
	TERMINATE PGS
Specification statements	MENU ITEM
	TEXT
	COMMAND
	DATA NAME
	VARIABLE NAME
	POINT
	PLOT CHARACTER
	POSITION BEAM
	VECTOR
	POLYSTRING
	LINE
	LINE SET
	CIRCLE
	ARC
	HYPERBOLA
	PARABOLA
	ELLIPSE
	FUNCTION KEY
	END KEY
File generation	READ SPECIFICATION CARDS
	BEGIN SPECIFICATION STATEMENTS
	END SPECIFICATION STATEMENTS
	CREATE
	CREATE HIGH SPEED
	DISPLAY HIGH SPEED
	DISPLAY
	READ COORDINATE SET
Maintenance	LIST SPECIFICATIONS
	DELETE SPECIFICATIONS
	ADD SPECIFICATIONS
	DELETE PANEL

mands grouped under specification statements in Table 1 causes the system to create a specification record. All of the specification records created for an entire panel are grouped into a specification file.

An example of a specification statement with its associated data is:

MENU ITEM, LETTERS 'PROGRAM A', LOCATION 300, 400, CHARSIZE LARGE, ACTION CODE 313, DATA 'PROGA';

This statement, as interpreted by PGS, would display in large characters

PROGRAM A

on the screen commencing at location (x,y) = (300,400) when the associated panel is displayed. The word DATA indicates that

Table 2 Sample commands

```
INITIALIZE PGS
BEGIN SPECIFICATION STATEMENTS, NUMBER 6;
TEXT, LOCATION 18, 32, LETTERS 'Z COMPUTATION',
VARIABLE NAME, LETTERS 'A =', LOCATION 7, 26,
    LARGE, ACTION CODE 4422, DATA 0,
    POINTER 51;
VARIABLE NAME, LETTERS 'B =', LOCATION 7, 22,
    LARGE, ACTION CODE 4422, DATA 0,
    POINTER 52,
VARIABLE NAME, LETTERS 'C =', LOCATION 7, 18,
    LARGE, ACTION CODE 4422, DATA 0,
    POINTER 53;
VARIABLE NAME, LETTERS 'Z =', LOCATION 12, 13,
    LARGE, ACTION CODE 4602, DATA 0,
    POINTER 60;
MENU ITEM, LETTERS 'COMPUTE Z', LOCATION 5, 4,
    LARGE, ACTION CODE 0313,
    DATA 'ZCMP';
MENU ITEM, LETTERS 'CONTINUE', LOCATION 34, 4
    LARGE, ACTION CODE 0318;
END SPECIFICATION STATEMENTS, NUMBER 6;
CREATE, PANEL 8, FROM FILE 6, WITH BOUNDS 0, 0, 49, 35;
DISPLAY, PANEL 8;
```

PROGA is the PACE associated with the action code (PAC). The PAC of 313 specifies that when a light-pen detect is made on the menu item PROGRAM A, the system will call the application program named PROGA. When PROGA has been executed, control is to be returned to the panel to wait for another action (interruption) by the console operator. PROGA is a computational module coded in accordance with PLAN rules. Table 2 shows an example in which PGS commands are used.

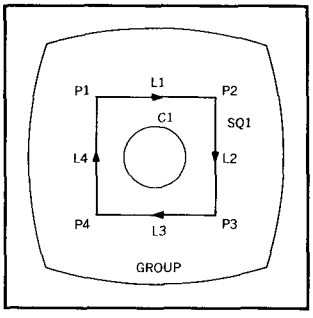
Graphics data structure

The need for a dynamic and flexible data structure arises when a console operator creates or modifies the shape and description of a geometric object on the display screen. All geometric relationships of surfaces, lines, and curves must be preserved as the structure is rapidly updated in response to light-pen actions at the display console. A complete set of routines was designed experimentally to permit the programmer to readily organize a particular data structure and tailor the elements within the structure to individual needs. These routines are based upon a dynamic storage allocation scheme (discussed later) so that the actual amount of data may outgrow the size of main storage.

Some of the requirements for a viable data structure are best illustrated by a simple picture drawn on a graphic display. A square, for example, has different features of interest: for some, the coordinates of the square may be significant; for others, the connectivity of the lines or the area may be of interest. The data structure should be general enough to take all these individual requirements into account.

When a console operator points to, or *detects*, an object on the display screen with the light pen, an interruption occurs in the program, and the program is given the address of the data that generated the detected graphic object. From this information, the program must deduce various properties of the object. For example, if a line has been selected, not only the identity of the line and its location must be made available, but also the fact that the line may be part of a square, and that a circle may be associated with the square, as indicated in Figure 4. Objects can be associated by grouping. In Figure 4, the combination of the square and circle are considered a *group*.

Figure 4 Group of objects



This example illustrates a few problems that a generalized structure must be able to resolve: (1) an individual graphic object must be identifiable, (2) relationships, hierarchical or otherwise, between objects must be established, (3) properties must be shared by different objects, and objects must be allowed to have multiple properties. Since drawings may be modified, deleted, or expanded in interactive problem solving, data structures must allow for dynamic growth and dynamic association.

An implementation that meets this basic objective includes pointers with the data (such as in a list structure), and ties together —in a closed *ring* structure—objects sharing a common property. The use of multiple address pointers within a block of data allows many properties to be associated with the block. The power of the multiple ring structure was demonstrated by Ivan Sutherland.[3,4]

block format

Physically, a *block* is a set of contiguous words. A typical block contains the following items:

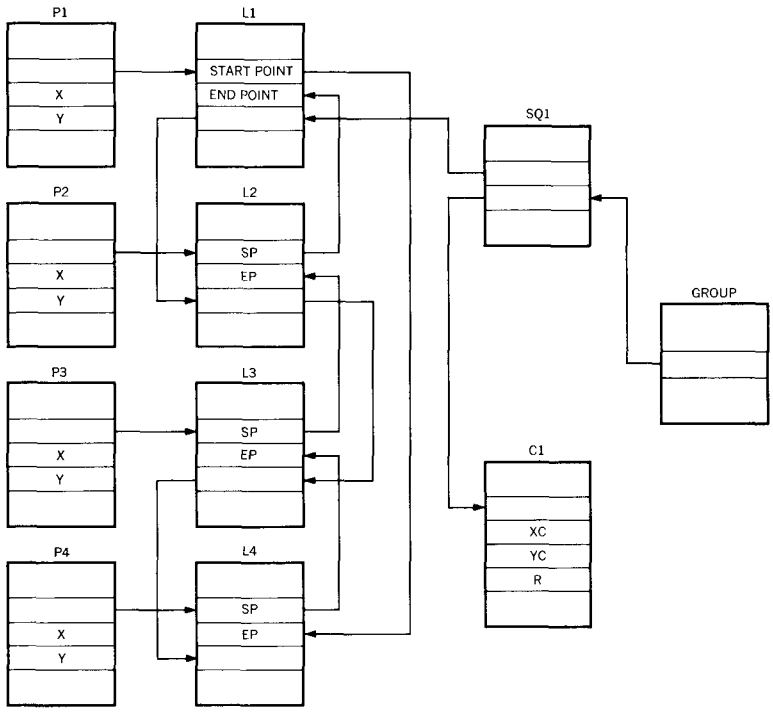
- Link area
- TYPE word
- Data area

Link areas contain address pointers that tie together blocks and by which blocks can be related. The TYPE word is actually a pointer to a generic block, a block that contains information relevant to all the blocks of a particular type. The data area may contain any information about the element that the block represents. For example, data words can contain coded information (flags, formats), values of variables, or text. Normally, blocks are fixed in size, but the system does support blocks having variable-size data areas.

rings and links

Rings are closed chains that string together all the data elements that are associated with each other. The ring pointers are called *links*. A link in one block points to a link in another, and

Figure 5 Ring structure of a model



so on until the last link points back to the first. Every ring has a ring start called *master-link*; the rest of the members of the ring are *slave-links*. Rings are formed according to certain rules:

- A ring must have one and only one master link, but it may have any number of slave links.
- A link cannot be a member of more than one ring.
- A block may be a part of as many rings as it has links.
- A block may have any combination of master links and slave links.

In Figure 5, a data representation (or *model*) for the square (SQ1) and circle (C1) of Figure 4 is shown using the ring structure. A block, called "group," associates the square and the circle. For easier readability, the rings in Figure 5 are not connected back to the starting point. All blocks may be reached from any given block. The end points of a line are defined in separate point blocks (P1 through P4). The lines (L1 through L4) are actually subordinate to these points. Thus, if a point is relocated, all connected lines move with it. The location and radius of the circle are described in block C1.

data structure system

The system in which this data structure is used contains a set of subroutines that allows the programmer to build a model, retrieve information by name from it, modify it, and process this information with an application program. This set of data structure subroutines is designed specifically for the FORTRAN programmer. They

can be used to implement any kind of *element blocks* (programmer-defined entities), provided structural guidelines are followed.

model segmentation

The system approach discussed in this paper allows the segmenting of data and provides a programmed paging technique to automatically transfer segments between main storage (in a SYSTEM/360 or 1130) and secondary storage. This feature is required to handle large quantities of data, which places demands on main storage far exceeding that available in any general computer system.

Address pointers or links must necessarily be symbolic when the model structure grows in size so that portions of the model are forced to reside in secondary storage. Under such circumstances, the model structure may be divided into fixed-size segments. Under the control of the system, one or several segments from different areas of the model may be in main storage simultaneously.

A "virtual memory" concept[5] for addressing within the model structure is used, and access to information is made via simple address translation using an index register. This concept, in a SYSTEM/360, for example, using 24 bits as an address pointer, permits a model to be as large as 16,777,216 bytes, regardless of actual main storage size.

To support a large data base using the virtual memory concept outlined, disks and drums must be allocated for that purpose at system initialization time. At that time, the size of a segment is also specified (or the data structure is segmented by the system automatically by default).

The segmentation of the model follows these rules:

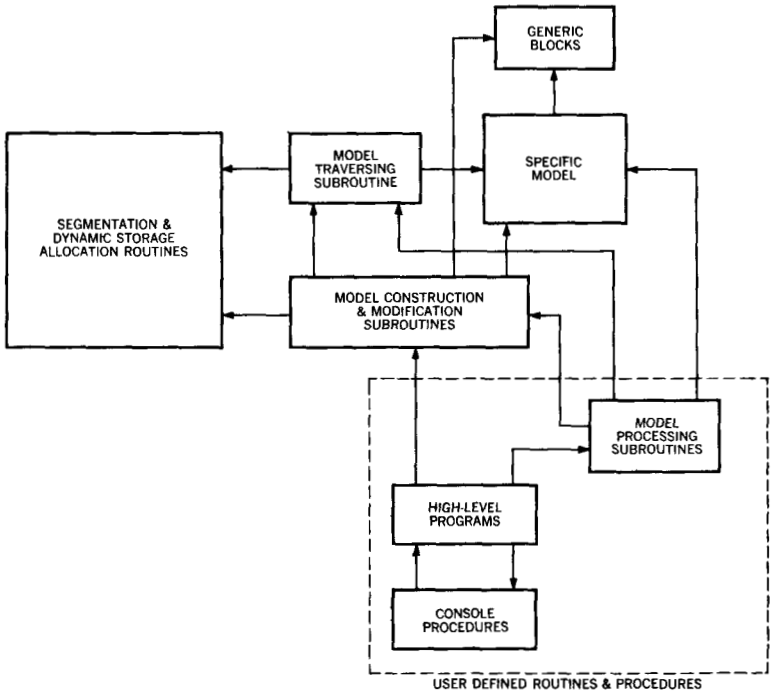
- The data set (or a model structure) resides in a virtual memory.
- A data set may not refer to any data outside of this virtual memory.
- At any time, main storage contains at least one segment.
- All segments within one model are of the same size.
- A segment is relocatable to any part of working main storage.

A block is read into main storage on demand, together with all other blocks within the same segment. After use, a segment may be overlaid by another segment. Before the segment is overlaid, it is written back into secondary storage, so that the copy in secondary storage is always an updated version.

The virtual memory is normally "transparent." However, some control over the segment to which blocks are allocated and the placement of blocks in the segment is available through subroutine parameters, so the experienced programmer can increase the efficiency of the structure.

The use of the data structure and dynamic storage allocation system relative to geometrical data is now apparent. However it must be emphasized that the system may be used for other applications where a complicated file storage and retrieval mechanism is required. For example, within the PLAN graphics support, panel files and specification files could be treated as blocks that are con-

Figure 6 Data structure system



nected via links. Given the identification of a particular panel file, the file could then be retrieved together with any number of its associated specification files. A specification file could be part of several panels, and a panel could contain several specification files.

system overview

Figure 6 shows, at an overview level, the functions of the system and their relationship to application-dependent programs. The programmer wants to build a specific model, modify it, retrieve information from it, and process particular elements or sets of elements in accordance with his graphics application program. The model-processing subroutines depend on the application and the kind of data being placed into the structure. Model-traversing subroutines can be considered totally independent of block formats and contents. Model-building subroutines are almost independent. A construction subroutine can automatically access a generic block, find a suitable place in storage, and initialize the specific block. However, higher-level programs must provide the information about the relationships among blocks, so that automatic ties between hierarchical levels can be embedded in the blocks. These higher-level programs should also interface with console procedure programs when any action at the graphic display console affects the model structure.

Summary

A system intended for interactive problem solving (PLAN) with added support for graphics applications (PGS) may overcome some

of the fundamental problems confronting graphics users. The inherent separation of the graphics portion of application programs simplifies the adapting of existing application programs for use with graphics devices. The modularizing of the computational portions of application programs encourages development of open-ended user libraries of modules that can be executed dynamically as required. In most cases, graphics programming is done at a high level, and previously coded display formats can be used as needed.

An experimental set of routines was designed to enable a programmer to organize a generalized data structure to suit particular needs. Such a system would allow a model to be segmented, and a paging scheme could be used to load the segments as they are needed.

ACKNOWLEDGMENT

The authors wish to acknowledge the helpful contributions, comments, and suggestions by C. B. Morrill and J. G. Sams during the preparation of the manuscript.

CITED REFERENCES AND FOOTNOTE

1. D. Parker, "Solving design problems in graphical dialogue," *On-Line Computer Systems*, edited by W. J. Karplus, McGraw-Hill Book Company, New York, New York (1966).
2. PLAN (Program Language Analyzer) and PGS have been announced as TYPE II programs with full IBM maintenance and support.
3. I. E. Sutherland, "SKETCHPAD: A man-machine graphical communication system," *Proceedings of the Spring Joint Computer Conference* **23,** 329–346 (1963).
4. J. C. Gray, "Compound data structure for computer-aided design—a survey," *Proceedings—1967 ACM National Conference,* 355–365 (1967).
5. L. A. Belady, "A study of replacement algorithms for a virtual storage computer," *IBM Systems Journal* **5,** No. 2, 78–101 (1966).